DESIGN AND ANALYSIS OF UPDATE-BASED CACHE COHERENCE PROTOCOLS FOR SCALABLE SHARED-MEMORY MULTIPROCESSORS

David Brian Glasco

Technical Report No. CSL-TR-95-670

June 1995

This research was Supported by Digital Equipment Corporation and Sun Microsystems, Inc.

DESIGN AND ANALYSIS OF UPDATE-BASED CACHE COHERENCE PROTOCOLS FOR SCALABLE SHARED-MEMORY MULTIPROCESSORS

by

David Brian Glasco

Technical Report No. CSL-TR-95-670
June 1995

Computer Systems Laboratory
Department of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract

This dissertation examines the performance difference between invalidate-based and update-based cache coherence protocols for scalable shared-memory multiprocessors. The first portion of the dissertation reviews cache coherence. First, chapter 1 describes the cache coherence problem and identifies the two classes of cache coherence protocols, invalidate-based and update-based. The chapter also reviews bus-based protocols and reviews the additional requirements placed on the protocols to extend them to scalable systems. Next, chapter 2 reviews two latency tolerating techniques, relaxed memory consistency models and software-controlled data prefetch, and examines their impact on the cache coherence protocols. Finally, chapter 3 reviews the details of three invalidate-based protocols defined in the literature and defines two new update-based protocols.

The second portion of this dissertation examines the performance differences between invalidate-based and update-based protocols. First, chapter 4 presents the methodology used to examine the performance of the protocols. This presentation includes a discussion of the simulation environment, the simulated architecture and the scientific applications. Next, chapter 5 describes and analyzes the performance of two enhancements to the update-based cache coherence protocols. The first enhancement, a fine-grain or word based synchronization scheme, combines data synchronization with the data. This allows the system to take advantage of the fine-grain data updates which result from the update-based protocols. The second enhancement, a write grouping scheme, is necessary to reduce the network traffic generated by the update-based protocols. Next, chapter 6 presents and discusses the simulated results that demonstrate that update-based protocols, with the two enhancements, can significantly improve the performance of the fine-grain scientific applications examined compared to invalidate-based protocols. Chapter 7 examines the sensitivity of the protocols to changes in the architectural parameters and to migratory data. Finally chapter 8 discusses how the choice of protocols affect the correctness, cost and efficiency of the cache coherence mechanism.

Overall, this work demonstrates that update-based protocols can be used not only as a coherence mechanism, but also as a latency reducing and tolerating technique to improve the performance of a set of fine-grain scientific applications. But as with other latency reducing techniques, such as data prefetch, the technique must be used with an understanding of its consequences.

Acknowledgements

First, I would like to thank my wife Reem for without her love and support, especially her hugs, none of this work would have been possible. I would also like to thank my parents who instilled in me the desire and ambition required to complete such an undertaking.

Next, I would like to thank my advisors, Bruce Delagi and Michael Flynn, who gave me excellent guidance throughout this endeavor. I especially would like to thank Bruce for seeing my potential and offering me support early in my academic career.

I would like to thank Manu Thapar for helping me understand the cache coherence problem and some of the possible solutions and Nakul Saraiya for answering my endless questions about the simulator and LISP.

Finally, I would like to thank Sun Microsystems for providing me with office space and more machines than I could ever use.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Parallel Processors

As the computational requirements for computer applications increase, parallel processors are becoming more popular. Parallel processors attempt to improve the execution time of an application by executing portions of it on separate processors. There are two basic types of parallel processor systems: distributed-memory multicomputers and shared-memory multiprocessors. In distributed-memory multicomputers, independent computer nodes are connected together through a message-passing network. To communicate, nodes send explicit messages between themselves. In shared-memory multiprocessors, tightly-coupled processors share a global memory space. Communications between processors is implicit in accesses to shared variables. This programming model is preferred by many over the message-passing programming model of the distributed-memory multicomputers as it more closely resembles the programming model used for uniprocessor systems [68].

1.2 Shared-Memory Multiprocessors

Shared-memory multiprocessors may be implemented using one of three models: the uniform memory access (UMA) model, the non-uniform memory access (NUMA) model, or the cache-only memory (COMA) architecture [40, 47, 25]. Discussion of the COMA architectures is beyond the scope of this dissertation.

In the UMA model, each processor has an uniform memory access latency for each word of memory. The processors may be connected to the memory by a shared bus, crossbar switch or multistage interconnect network, as shown in figure 1.1. One disadvantage of the UMA model is that as the size of the system is scaled up, the memory access latency of *all* memory increases.

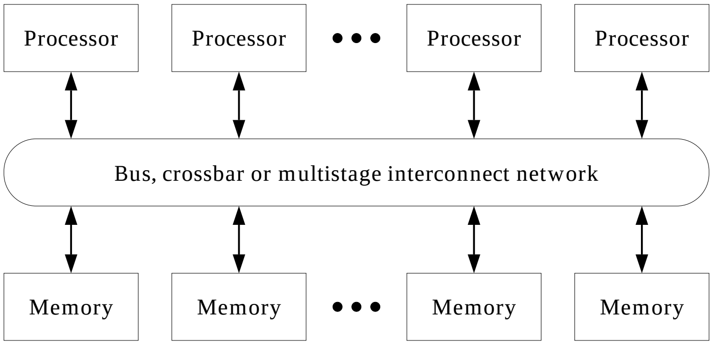


Figure 1.1: Uniform Memory Access (UMA) Model

The NUMA model addresses the scalability issue by physically distributing memory among the processors, as shown in figure 1.2. In this case, the latency of a memory access is dependent on the physical location of memory. The local memory access latency is constant regardless of the size of the system, but the non-local or remote memory access latency varies based on the distance from the requesting processor to the remote memory. Several techniques have been presented to tolerate or reduce this memory access latency, which may become significant for remote memory accesses. These include processor caches [64], relaxed memory consistency models [1, 22, 38, 33] and software-controlled data prefetch [55]. The remainder of this chapter describes the problem introduced by processor caches in shared-memory multiprocessors, and chapter 2 describes the relaxed memory consistency models and software-controlled data prefetch in greater detail.

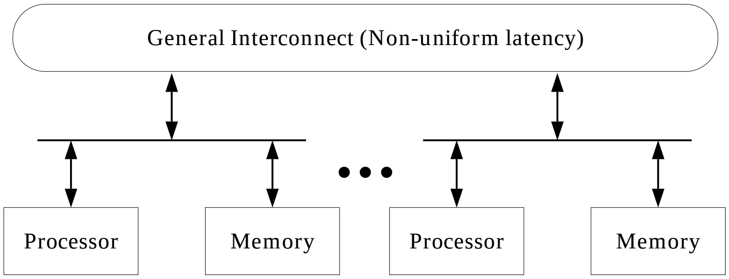


Figure 1.2: Non-Uniform Memory Access (NUMA) Model

1.3 Cache Coherence

A processor cache is a small, fast memory placed near the processor that holds the most recently accessed data [64]. When the desired data is not found in the cache, a cache miss occurs and the data is fetched from the memory and placed in the cache. The addition of these caches to shared-memory multiprocessors introduces a data consistency problem known as the cache coherence problem[1].

The problem is that a given memory line may be present in any of the processor caches. To execute programs correctly[2], the copies of this memory line must remain consistent. Therefore, when a processor modifies its cached copy of the line, other caches that have a copy of the line must be notified so that their copy may be made consistent.

For example, in figure 1.3 the shared variable A is being shared by multiple processors. When processor 1 modifies its cached copy of A, the other remote cached copies must be made consistent and memory's copy may need to be made consistent.

[1] The terms coherence and consistency tend to be used interchangeably in the literature.

[2] Some iterative programs may still converge even if memory is not always coherent.

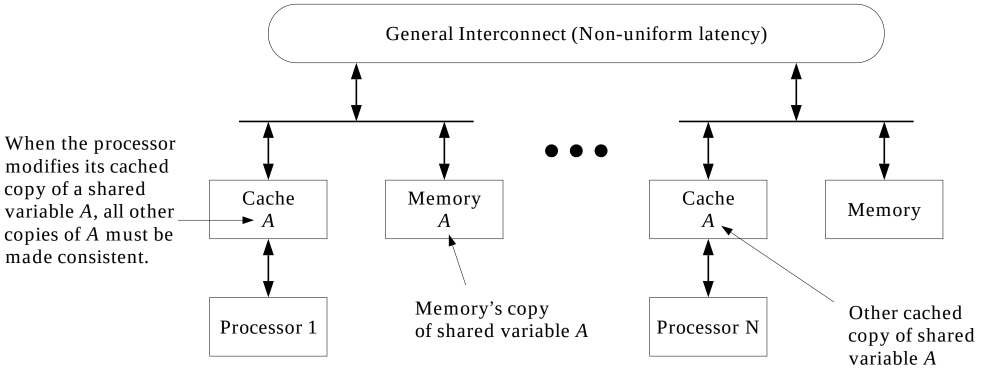

General Interconnect (Non-uniform latency)
When the processor modifies its cached copy of a shared variable *A*, all other copies of *A* must be made consistent.
Cache *A*
Memory *A*
Cache *A*
Memory
Processor 1
Memory's copy of shared variable *A*
Processor N
Other cached copy of shared variable *A*

Figure 1.3: Cache Coherence Problem

The two methods for maintaining consistency on a write are invalidating or updating the remote copies of the memory line. Invalidating purges the copies of the line from the other remote caches which results in a single copy of the line in the writing processor's cache, and updating forwards the write value to the other remote caches, after which all caches are consistent. This consistency of data is maintained through what is known as a cache coherence protocol.

Cache coherence protocols define what actions are required when a processor modifies a cached copy of a memory line. The protocols must guarantee that after the actions triggered by the write are completed, all caches in the system are consistent with each other. Figure 1.4 summarizes the possibilities that result from either invalidating or updating the remote caches and possibly updating memory. These four possibilities result in two classes of protocols: update-based (UP) or invalidate-based (INV). For update-based protocols, remote caches are updated on a write, and memory may be updated on every write. For invalidate-based protocols, remote caches are invalidated and memory may be updated when a writing cache releases exclusive ownership of the line. All protocols update memory's copy of a line if memory is not consistent when the last cached copy of the line is replaced.

		Remote Caches	
		Invalidated	Updated
Memory	Not Updated	Invalidate	Update
	Updated	Invalidate	Update

Figure 1.4: Protocol Classes

The details of the cache coherence protocols are dependent on the structure of the underlying interconnect. If all caches and memories are connected to a common bus, a broadcast-based protocol may be used, but if a general interconnect is used in which caches cannot observe all memory transactions, a directory-based protocol is necessary [66].

1.3.1 Broadcast-Based Protocols

Broadcast-based protocols rely on the ability of each cache to observe or “snoop” all memory transactions on the shared bus and change the state of the cache lines appropriately [8]. Each cache operates independently of all other caches. If a cache observes a write to a memory line it has a copy of, the cache coherence protocol will either invalidate or update the cached copy depending on the type of protocol used.

Several broadcast-based protocols have been specified in the literature. These include both invalidate-based and update-based protocols. The invalidate-based protocols include Goodman’s write-once [37], the Synapse [30], the Illinois [58] and the Berkeley [29] protocols. The update-based protocols include the Firefly [67] and the Dragon [54] protocols. These protocols have been well studied [46, 26, 8, 7, 54, 67].

The next two sections give a brief description of how the Berkeley invalidate-based protocol and the Dragon update-based broadcast protocols operate for typical processor reads and writes.

Berkeley Invalidate-Based Protocol

In the Berkeley invalidate-based protocol, a cache line may be in one of four states:

- Invalid - Cached copy of line is not valid,
- Unmodified-Shared - Read only copy of line and memory is consistent,
- Modified-Shared - Read only copy of line and memory is not consistent, or
- Dirty - Read/Write copy of line and memory is not consistent.

Figure 1.5 shows the state diagrams for the protocol. The state of a cache line may change as a result of a local processor request or an observed memory transaction by another processor, as shown in figure 1.5a and 1.5b respectively.

On a processor read miss, the state of the cache line is set to UnModified-Shared, and if another cache has a copy of the line, then it supplies the data and changes its local state appropriately.

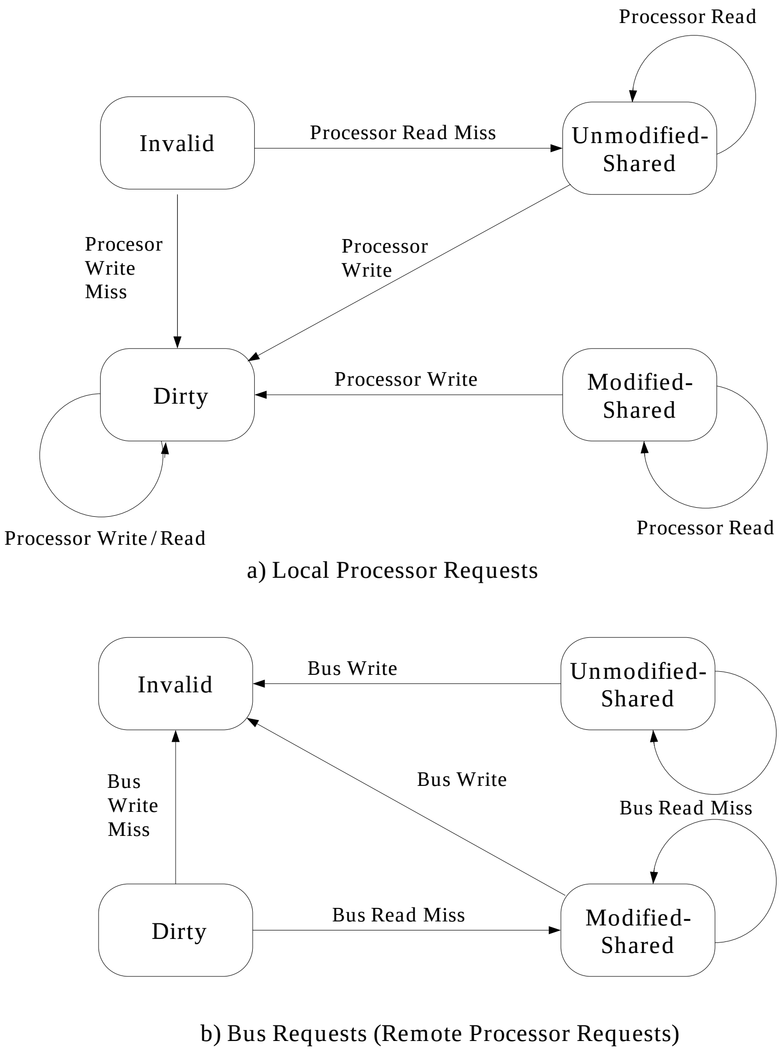
On a processor write miss or a write hit to a read-only line, the state is set to Dirty. All other copies of the line are invalidated, as the other caches observe the write on the bus. If the line was owned by another cache, the owning cache supplies the data and invalidates it copy.

The protocol has several important characteristics. First, exclusive ownership of the line (Dirty state) is required before the line can be modified. Second, cache-to-cache transfers are used when possible to satisfy miss requests. This reduces the need for expensive accesses to main memory, and third, main memory is only updated when a dirty cache line is replaced by the owning cache.

Dragon Update-Based Protocol

In the Dragon update-based protocol a cache line may be in one of five states:

- Invalid - Cached copy of line is not valid,
- Read-Private - Only copy of line and memory is consistent,
- Private-Dirty - Only copy of line and memory is not consistent,



a) Local Processor Requests

b) Bus Requests (Remote Processor Requests)

Figure 1.5: Berkeley Invalidate-Based Cache Coherence Broadcast Protocol

- Shared-Clean - Multiple copies of line and memory is consistent, or
- Shared-Dirty - Multiple copies of line and memory is not consistent.

Figure 1.6 shows the state diagrams for the Dragon update-based cache coherence protocol. As in the Berkeley protocol, the state may change as a result of a processor access or an observed memory transaction. The protocol uses a special sharing line that is asserted if at least one other cache has a copy of an accessed line.

On a read miss, the local cache line state is set to either Read-Private if no other shared copies of the line exist or to Shared-Clean if the sharing line is asserted during the miss. If one of the caches has the line in the Private-Dirty state, then that cache supplies the data for the miss and sets the cache line state to Shared-Dirty. If one of the caches is already in the Shared-Dirty state, it simply supplies the data for the miss. If one of the caches has the line in the Read-Private state, it supplies the data for the miss and sets the cache line state to Shared-Clean.

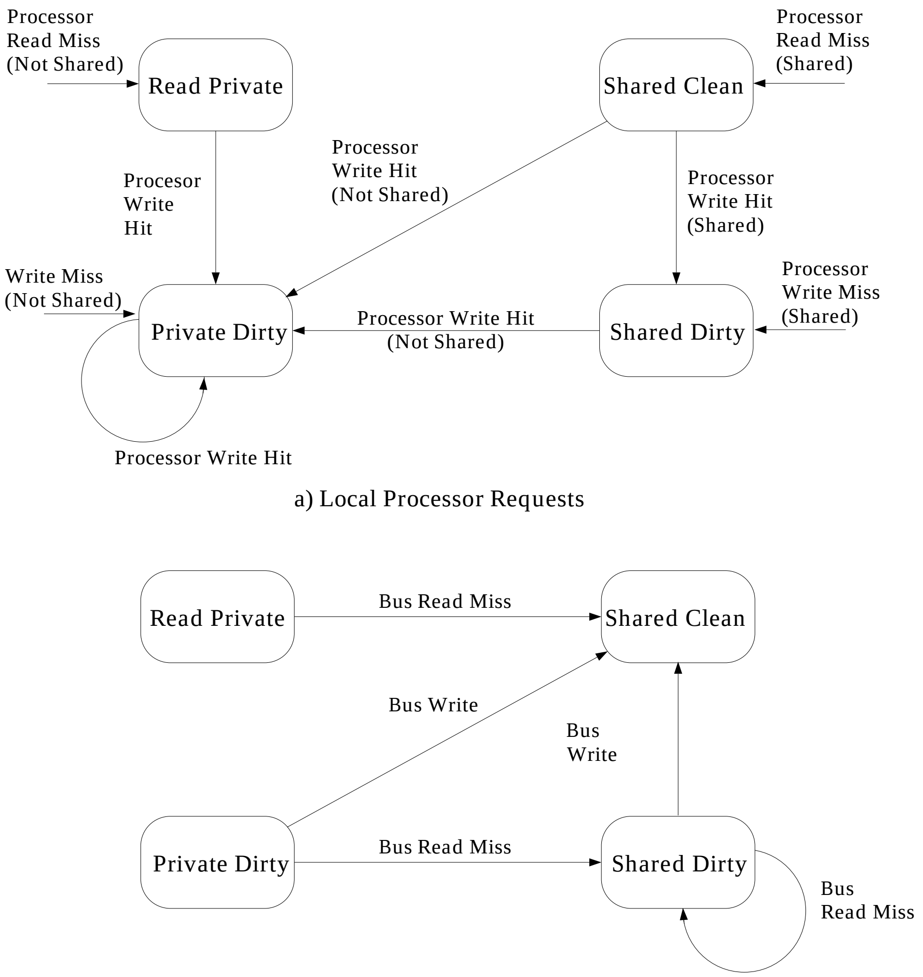
On a write miss, all other copies of the line must be updated. The cache line state is set to Private-Dirty if there are no other shared copies of the line, or it is set to Shared-Dirty if other copies do exist. All caches with a copy of the line update their copy and set the cache line state Shared-Clean. If a cache was in the Private-Dirty state, it supplies the data for the miss. Memory's copy is no longer valid. When the last Shared-Dirty copy is replaced, memory is updated.

For this protocol, exclusive ownership of the line is not required to modify the line, but the protocol does requires the special sharing line. There can be multiple writers of a line unlike the invalidate-based Berkeley protocol. The protocol also uses cache-to-cache transfers.

Several studies have indicated that there is little performance difference between broadcast-based cache coherence protocols which maintain consistency through invalidations or updates [27, 46].

1.3.2 Directory-Based Protocols

The main problem with broadcast-based protocols is that they rely on an interconnect that allows broadcasts of all memory transactions. Interconnects such as buses do not



a) Local Processor Requests

b) Bus Requests (Remote Processor Requests)

Figure 1.6: Dragon Update-Based Cache Coherence Broadcast Protocol

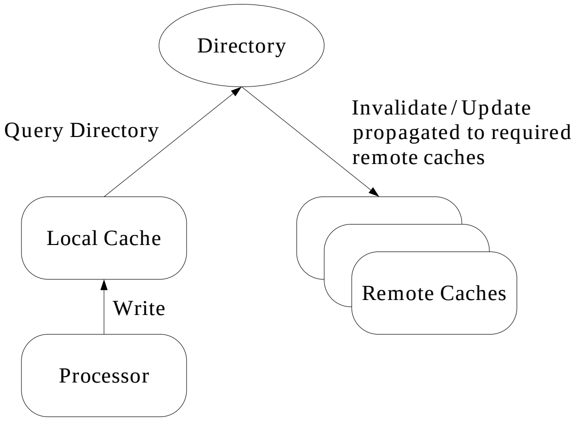


Figure 1.7: Directory-Based Cache Coherence Protocols

scale well as the shared bus saturates after a small number of processors are attached to it [68]. A scalable system is one in which the performance of the system increases linearly with the size of the system. To build such a scalable system, other interconnects that scale better are required. These interconnects, such as two-dimensional meshes, do not allow caches to observe or "snoop" all other memory transactions.

This change in the network structure imposes two new requirements on the cache coherence protocols [66]. First, the protocols must now explicitly notify other caches when a cache line has been modified. To send these notifications, the protocols must be able to send and receive protocol level messages. The second requirement is that a logical list of caches holding a copy of each memory line must be maintained so that each write notification may be sent only to the caches which have a copy of the line. This list of caches holding a copy of each line is stored in what is known as a "directory". The directory may be physically implemented in several ways as will be described in section 3.1, but each directory entry will logically maintain a list of caches holding a copy of the respective memory line.

The directory-based cache coherence protocols use the directory entry to forward write information to the necessary caches. Figure 1.7 shows the basic operations for a directory-based protocol. When a cache receives a write request from the processor, it must query the directory to determine which caches must be invalidated or updated for invalidate-based or update-based protocols respectively. Once all the caches have received and processed the invalidation or update, all caches in the system are consistent. Section 3.3 will describe three different implementations of invalidate-based protocols presented in the literature, and section 3.4 will present two new update-based protocols. The remainder of this dissertation examines the performance difference between these two classes of directory-based cache coherence protocols.

1.4 Organization of Dissertation

This dissertation is organized as follows. First, chapter 2 reviews two techniques for reducing and tolerating memory access latency. These are relaxed memory consistency models and software-controlled data prefetch. Chapter 3 reviews three invalidate-based protocols described in the literature and describes two new update-based protocols. Validation of the update-based protocols is discussed. Chapter 4 describes the methodology used in this dissertation to compare the performance of the cache coherence protocols. This includes a description of the simulation environment, the multiprocessor architecture, and the application space. Next, chapter 5 presents two additional techniques that can be applied to update-based protocols to reduce and tolerate memory access latencies. These include a word synchronization scheme and a write grouping scheme. Next, chapter 6 presents the simulated performance of the protocols studied, and chapter 7 examines the sensitivity of the protocol's performance to variations in architectural parameters and to applications with migratory data. Finally, chapter 8 concludes the dissertation by comparing and contrasting the protocol classes in terms of their correctness, cost and efficiency of maintaining the desired memory consistency model.

1.5 Contributions of Dissertation

The main contribution of this dissertation is the development and analysis of update-based cache coherence protocols for scalable shared-memory multiprocessors. It designs and validates two update-based protocols based on a centralized directory and a distributed directory. The work analyzes the performance of the update protocols and identifies the two fundamental limitations of the update-based protocols: the inefficiency of single word updates and the mismatch between the coarse granularity of the data synchronization and the fine granularity of the data updates. Two techniques are presented to overcome these limitations. The first, a write grouping scheme, is used to improve the efficiency of the updates; the second, a finer grain (word) data synchronization scheme, is used to provide a better match between the granularity of the data synchronization and data updates.

The dissertation compares the performance of the update-based protocols to three high performance invalidate-based protocols presented in the literature. The simulations demonstrate the limitations of the update-based protocols and how the two techniques, write grouping and word synchronization, improve the performance of the update-based cache coherence protocols.

The sensitivity of the protocols to architectural parameters is examined. And, finally, the problem of migratory shared data and update-based protocols is discussed, and a technique to limit the performance loss due to the unnecessary updating of migratory data is presented.

Overall, the dissertation demonstrates that update-based protocols can be designed to significantly improve the performance of fine-grain scientific applications.

Chapter 2

Latency Reducing and Tolerating Techniques

There are several techniques that can be used to either reduce or tolerate memory access latency. These techniques, which were briefly introduced in the last chapter, include processor caches, relaxed memory consistency models and software-controlled data prefetch. The last chapter described the cache coherence problem introduced by the addition of processor caches to shared-memory multiprocessor systems. In this chapter, relaxed memory consistency models and software-controlled data prefetch are described, and their impact on the cache coherence protocols is examined.

2.1 Memory Consistency Models

Memory consistency models describe the ordering of memory access events as seen by the programmer. Censier and Feautrier [12] define memory system coherence as follows

Definition 2.1 A memory scheme is *coherent* if the value returned on a load instruction is always the value given by the latest store instruction with the same address.

The problem with this definition is that the meaning of “latest store” is unclear. To make the definition of memory consistency more precise, several memory consistency models have been defined in the literature. These include sequential consistency [52], processor consistency [38], weak consistency [22] and release consistency [33].

Dubois and Scheurich [21] have defined several terms required to properly define a memory consistency model. The terms define the ordering of memory access events. These events may be a memory load or store if the type is not specified.

Definition 2.2 *Initiated memory request:* A memory access is *initiated* when a processor has sent the request and the completion of the request is out of its control.

Definition 2.3 *Issued memory request:* An initiated request is *issued* when it has left the processor environment and is in transit in the memory system.

Definition 2.4 *Performed load:* A load is considered *performed* at a point in time when the issuing of a store to the same address cannot affect the value returned by the load.

Definition 2.5 *Performed store:* A store to address X by processor i is considered *performed* at a point in time when an issued load to the same address returns the value defined by a store in the sequence $Si(X)+$.

The sequence $Si(X)+$ refers to the stream of stores to address X from processor i. From definition 2.5, the store is complete when a load returns the value of the store or the value of any subsequently performed store.

In systems with atomic memory accesses, a memory access is performed with respect to all processors at the same time. In systems where a store may be performed at different processors at different times a further refinement of the definition of performed is required.

Definition 2.6 *Performed load with respect to processor k:* A load is considered *performed with respect to processor k* at a point in time when the issuing of a store to the same address by processor k cannot affect the value returned by the load.

Definition 2.7 *Performed store with respect to processor k:* A store to address X by processor i is considered *performed with respect to processor k* at a point in time when an issued load to the same address by processor k returns the value defined by a store in the sequence $Si(X)+$.

For non-atomic memory systems, the definition of *performed* is now given by

Definition 2.8 *Performed store:* A store is *performed* when it is performed with respect to all processors.

Definition 2.9 *Globally Performed load:* A load is *globally performed* when it is performed with respect to all processors and if the store which is the source of the returned value is globally performed.

For systems with a write buffer, the definitions must be slightly modified [32].

Definition 2.10 *Performed store with respect to processor k (with write buffering):* A store by processor i eventually performs with respect to processor i. If a load by processor i performs before the last store (in program order) to the same address by processor i performs with respect to processor i, then the load returns the value defined by that store. Otherwise, the load returns the value defined by the last store to the same address (by any processor) that performed with respect to processor i (before the load performs). A store is *performed* when it is performed with respect to all processors.

This new definition is required since a load request may find the desired data in the write buffer and, therefore, be performed without querying the cache or interacting with the coherence mechanism.

With these definitions, several different memory consistency models can now be presented. These include sequential consistency, processor consistency, weak consistency and release consistency. The models vary on how strongly ordered the memory accesses are. Sequential consistency requires the ordering of accesses from all processors. The other consistency models, known collectively as relaxed consistency models, relax this total ordering and only specify the order of accesses by individual processors.

2.1.1 Sequential Consistency Model

Lamport [52] has defined a sequential consistency model as

Definition 2.11 A system is *sequentially consistent* if the results 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 a sequentially consistent multiprocessor, memory accesses appear to execute atomically in some total order [51]. To a programmer, a sequentially consistent multiprocessor would be indistinguishable from a multitasking uniprocessor.

Scheurich and Dubois [21] have described a set of sufficient conditions for sequential consistency. The conditions, slightly modified and combined [69], are

Condition 2.1 *Sufficient conditions for sequential consistency:* Before a load or store access is allowed to perform with respect to any processor, all previous load accesses must be *globally performed* and all previous stores must be *performed.*

Implementation

For invalidate-based protocols, these conditions may be satisfied by delaying the issuing of a memory access from processor i until all previous memory accesses issued by this processor have been performed. With this constraint, load accesses by different processors are not ordered, but the stores to the same address are, and the stores are also ordered with respect to the loads. The invalidation mechanism provides for this store ordering. When a store to address X is issued by processor i, other processors may still globally perform loads from this address. But once the store is performed with respect to processor j, any load from address X issued by processor j cannot be globally performed until the store issued by processor i has been performed.

This ordering of stores is possible in the invalidate-based protocols since the processor issuing the store, processor i, obtains exclusive ownership of the line. Any load request for this line is blocked by processor i until the store has been performed (all invalidations have been acknowledged). For example, figure 2.1 shows a sequence of loads and stores for three processors. First, processor i and k issue a load from address X and processor j issues a store to address X. The loads can be globally performed since the store has not yet been performed with respect to either processor i or k.

Once the invalidation request reaches processor i, the store is considered performed with respect to processor i. The next load from address X issued by processor i is now blocked by processor j since the pending store has not yet been performed (all invalidations acknowledged). Once the invalidation has reached processor k and both processor i and k acknowledge the invalidation, the store is considered performed, and the blocked load of processor i can be globally performed.

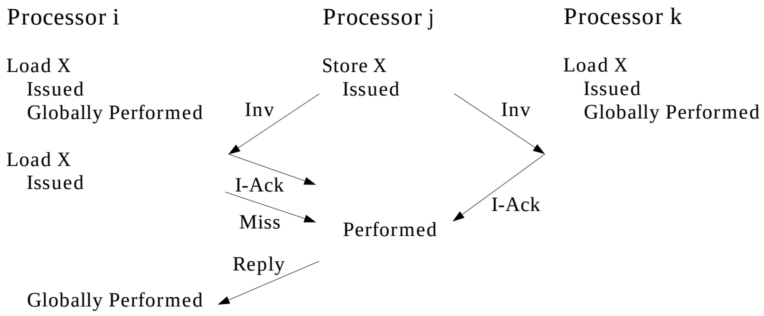


Figure 2.1: Sequential Consistency - Invalidate-Based Protocols

The ordering of accesses to a given line is determined by the relative timing of the invalidation request. If a load is issued by processor i before a store from processor j has been performed with respect to processor i, the load will appear as if it was performed before the store in the total ordering. If the load by processor i is issued after the store is performed with respect to processor i then the load will appear as if it occurred after the store in the total ordering. The invalidation mechanism allows for the ordering of loads with respect to stores.

For update-based protocols, neither the loads nor stores are ordered. This lack of ordering for the stores creates a significant problem for update-based protocols attempting to implement a sequential consistency model. To implement such a model, the update-based protocols must be augmented with a mechanism that will allow for the ordering of stores. Wilson and LaRowe [72] have demonstrated how a two-phase update would allow for this proper ordering of stores.

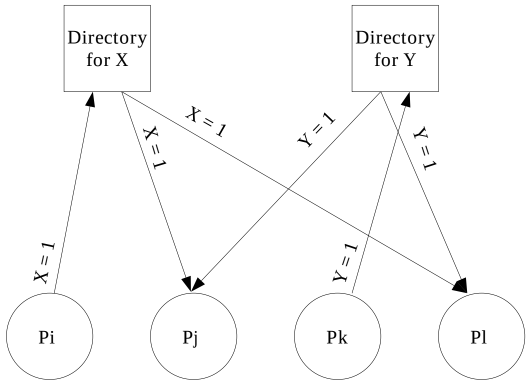
The example shown in figure 2.2a, taken from Wilson and LaRowe [72], demonstrates how an update-based protocol may violate sequential consistency even if a processor delays all memory accesses until all previous memory accesses from the

Processor i	Processor j	Processor k	Processor l

x = 1;	while (x != 1);
	print y;
	print y;
		y = 1;	while (y != 1);
			print x;
			print x;

x and y are initially 0

a) Code example



b) Update flow

Figure 2.2: Violating Sequential Consistency - Update-Based Protocols

same processor have been performed. This requirement was sufficient for invalidate-based protocols to provide sequential consistency.

In the example, processors i and k write to two control variable X and Y respectively. If the system is sequentially consistent, such as with a uniprocessor, then either both processors print two ones, one processor prints a zero followed by a one and the other prints two ones, or one processor prints two zeros and the other prints two ones. If both processors print a zero followed by a one, then sequential consistency has been violated. This ordering may occur in an update-based protocol if the directories for the two variable, X and Y, are located at different nodes. In this case, the path of the updates will differ and the stores to X and Y may be performed with respect to processors j and l in a different order, as shown in figure 2.2b. This case violates the condition that an issued store be performed before any other memory accesses may be issued.

The problem arises because when a store is performed with respect to processor i, the processor obtains the new value of the store. Other processors to which the store has not yet been performed are still able to access the old value. As in the invalidate-based protocols, the load performed at processor j prior to the store being performed at processor j simply appears prior to the store in the total ordering. The problem is that the processors to which the store has been performed with respect to can issue a load which will be performed before the store has been performed. This problem is prevented in invalidate-based protocols since the writing cache controls when the loads can be performed.

A two-phase update prevents this scenario by issuing a store to address X in two phases. The first phase of the store informs each cache with a copy of the line containing X that an update has been issued. These caches would not allow their respective processors to perform accesses to this address. After the first phase of the update has been performed with respect to all processors, the second phase of the update sends the actual data to the appropriate caches. Once cache i receives this update, processor i is able to perform loads from address X. This scheme prevents processor i from performing a load of the new value from address X while processor j may load an old value.

2.1.2 Processor Consistency Model

To improve system performance, the memory consistency model may be relaxed. The first of such relaxed memory consistency models is the processor consistency model defined by Goodman [38]. Goodman gives the conditions for processor consistency as

Condition 2.2 *Conditions for processor consistency:* Before a load is issued by a processor, all previous load accesses by that processor must be performed. Before a store is issued by a processor, all previous memory accesses by that processor must be performed.

Processor consistency requires that stores from a processor be performed with respect to all processors in the same order, but stores from different processors are not ordered. The condition also allows for reads to bypass writes and, therefore, the introduction of a write buffer, but stores may only be retired from the write buffer after they have been performed [31]. This retirement constraint limits the number of outstanding stores per processor to one.

Implementation

For both classes of protocols, implementing processor consistency is not difficult. Since loads are able to bypass stores, the processor may simply issue a load and stall until the load has been performed. For the stores, the processor issues the store to the write buffer (stalls if write buffer is full) and continues execution. As noted above, the stores cannot be retired (removed) from the write buffer until the store has been performed. This retirement constraint guarantees ordering of stores since each processor can have at most one issued, but not yet performed store.

2.1.3 Weak Consistency Model

The next progression in memory consistency models is a weak consistency model [22]. In weak consistency, accesses are divided into either synchronization accesses or normal accesses. These accesses are defined as [69]

Definition 2.12 *Synchronization access:* An access is a *synchronization access* if it is used to order events. For example, a synchronization flag can be used to order the accesses to a block of data by a producer and consumer, or a lock can be used to order process accesses to a critical section of code.

Definition 2.13 *Normal access:* An access is a *normal access* if it is not used to order events. These accesses include simple loads and stores.

If the synchronization events are identified by the programmer or compiler, then the caches and memory must be consistent only at these synchronization points.

Dubois, *et al* [21] proposed a weak consistency model and presented conditions for weak consistency (as slightly modified by Gharachorloo [31])

Condition 2.3 *Conditions for weak consistency:* Before a *normal* access is allowed to perform with respect to any other processor, all previous *synchronization* accesses must be performed, and before a *synchronization* access is allowed to perform with respect to any other processor, all previous *normal* accesses must be performed. *Synchronization* accesses are sequentially consistent with respect to one another.

Implementation

For both classes of protocols, the weak consistency model introduces the need for a fence instruction. A fence instruction stalls the processor until all previous synchronization and normal accesses have been performed. Thus, hardware counters are required to keep track of outstanding accesses. For invalidate-based protocols, these accesses include outstanding write misses and unacknowledged invalidation requests, and similarly for update-based protocols, they include outstanding write misses and unacknowledged write updates. When a processor issues a fence instruction, the processor is stalled until all counters are zero, which indicates that all writes have been performed.

In weak consistency, the burden of synchronization is placed on the compiler and programmer. It is their responsibility to place synchronization accesses at the proper locations in the code to protect critical regions and guarantee correct access order to shared data. Fence instructions are requires around synchronization accesses to satisfy condition 2.3.

2.1.4 Release Consistency Model

The final memory consistency model discussed in this dissertation is the release consistency model [33]. In this model, the conditions of weak consistency are relaxed even further. Release consistency divides memory accesses into two major categories: competing and normal. Two accesses are competing if they are to the same memory location and at least one access is a store. The competing accesses are further divided into synchronization accesses and non-synchronization accesses. The synchronization accesses are used to order events and are divided into acquire or release accesses. These synchronization accesses are defined by Gharachorloo, *et al* [33] as

Definition 2.14 An *acquire* synchronization access (e.g., a lock operation or a process spinning for a flag to be set) is performed to gain access to a set of shared locations.

Definition 2.15 A *release* synchronization access (e.g., an unlock operation or a process setting a flag) grants permission to access a set of shared locations.

Note that an acquire is a load operation and a release is a store operation.

In weak consistency, processors must wait for a synchronization event to be performed before continuing. Release consistency relaxes the consistency model by noting that a process performing a release access does not need to wait for the release's store operation to be performed and that a process performing an acquire access does not need to wait for previous accesses to be performed. The conditions for release consistency are given by Gharachorloo, *et al* [33] as

Definition 2.16 *Conditions for release consistency:* Before a *normal* access is allowed to perform with respect to any other processor, all previous *acquire* accesses must be performed, and before a *release* access is allowed to perform with respect to any other processor, all previous *normal* accesses must be performed. *Acquire* and *release* accesses are processor consistent with respect to one another.

Implementation

As with weak consistency, hardware counters are also required to keep track of outstanding accesses [33]. Again, the burden is placed on the programmer and compiler to insert synchronization accesses and fence instructions at the proper locations in the code, but the release consistency model requires fewer fence operations than the weak consistency memory model.

2.1.5 Summary

In this section four memory consistency models were described. The strongest model, sequential consistency, requires that all memory accesses appear atomic. This requirement presents difficulties for update-based protocols, but a two-phase update-based protocol can be used to implement sequential consistency. Relaxed consistency models were introduced to improve system performance. The relaxed consistency models allow more overlap of memory accesses. This overlap effectively reduces the memory access latency by hiding it behind other useful work. The first relaxed consistency model, processor consistency, required that stores from a given processor be performed with respect to all other processors in the same order as they were issued. Weak consistency and release consistency divide accesses into normal load and store accesses and synchronization accesses and only require memory and caches to be consistent at the synchronization points. Release consistency goes further by dividing synchronization accesses into acquire and releases and relaxes the ordering of these accesses. In the remainder of this dissertation, the applications studied will use a release consistency memory model.

2.2 Software-Controlled Data Prefetch

Another technique to reduce the memory access latency is software-controlled, non-binding data prefetch [55]. The data prefetch moves the requested data to the cache nearest the processor. This data prefetch can be used to hide a portion of the data miss latency in a typical producer and consumer interaction [55], as shown in figure 2.3. In the figure, the producer and consumer share a block of data using a release consistency memory model or what will be referred to as block synchronization throughout the remainder of this dissertation. The producer computes the data, writes the results to a shared buffer, and then uses a fence instruction to stall the processor until all writes have been performed. Once the writes have been performed, the semaphore is set. Consuming nodes wait for the semaphore to be set before attempting to access the shared data.

In the invalidate-based protocols, a write prefetch (or exclusive read) can be used to hide a portion of the producer's fence latency by issuing write requests early and overlapping multiple requests, as shown in figure 2.3. The write prefetch allows the write miss and fence latency to be overlapped with useful work.

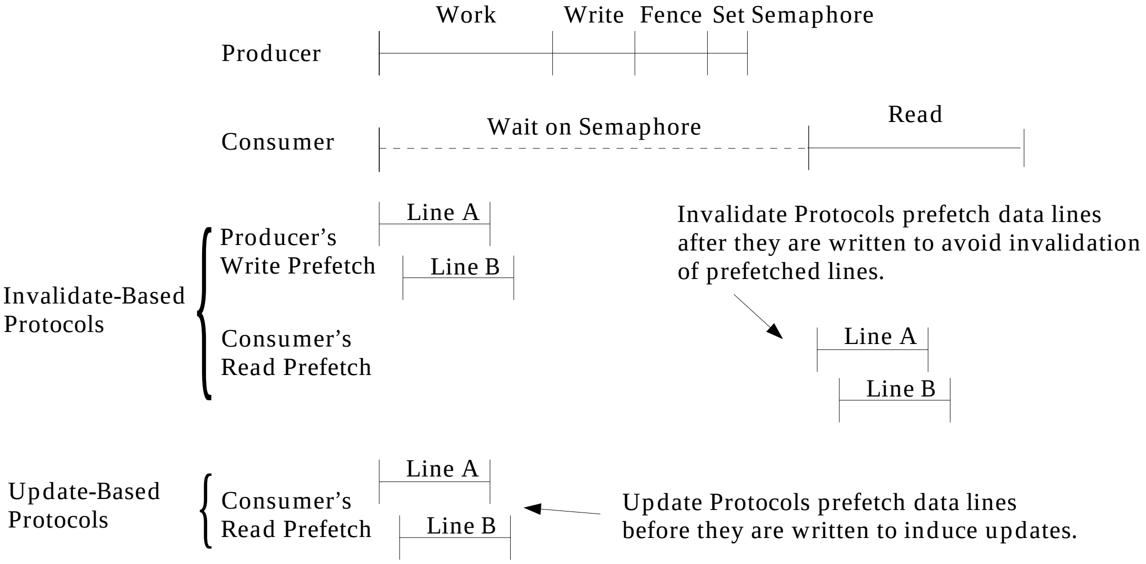

Work
Write
Fence
Set Semaphore
Producer
Consumer
Wait on Semaphore
Read
Line A
Line B
Invalidate-Based Protocols
Producer's Write Prefetch
Consumer's Read Prefetch
Invalidate Protocols prefetch data lines after they are written to avoid invalidation of prefetched lines.
Line A
Line B
Update-Based Protocols
Line A
Line B
Consumer's Read Prefetch
Update Protocols prefetch data lines before they are written to induce updates.

Figure 2.3: Block Synchronization with Prefetch

A read prefetch can also be used to reduce the consumer's read miss latency in an invalidate-based system, but the prefetches should be issued after reaching the synchronization point. If the prefetches are issued earlier, two scenarios are possible. In the first case, the producer has not yet written the data. In this case, the prefetched lines will be later invalidated, and the work done to prefetch and invalidate the lines will be wasted. In the second case, the producer has completed writing the data before the consumer's prefetch is received. In this case, the prefetch will obtain the proper data, but the time saved compared to prefetching after the synchronization point will be minimal since the producer would have also set the semaphore and the consumer's read of the semaphore would have found it set. The latency saved by prefetching early is small compared to the high cost of invalidated prefetches. For the invalidate-based protocols and the applications examined in this dissertation, prefetching before the synchronization point never resulted in a faster execution time than prefetching after the synchronization point for the invalidate protocols.

In the update-based protocols, read prefetches can be used not only to hide read miss latency behind useful work, but they also can be used to induce updates. If the prefetch is issued before the data is written by the producer, the consumer's cache will be updated when the data is written. The prefetch allows the consumers to express an early interest in the data.

2.3 Summary

This chapter reviewed two techniques to reduce and tolerate memory access latency. The first, relaxed memory consistency models, relaxed the ordering of accesses. This technique allowed for more overlap of memory accesses and, therefore, a reduction in memory access latency. The second technique, software-controlled data prefetch, allowed miss latency to be tolerated by overlapping miss requests with useful work or other miss requests. Prefetch also has an important role in update-based cache coherence protocols; it allows the consumers of data to express an early interest in a piece of data. If the consumers prefetch the data, then when it is written by the producer, the consumer's cache will be updated. Chapter 5 presents two other techniques that can be used with update-based protocols to help reduce and tolerate memory access latency.

Chapter 3

Directory-Based Protocols

This chapter describes the directory-based cache coherence protocols studied in this dissertation. First, section 3.1 reviews the directory structures required by the protocols, and section 3.2 describes protocol level deadlock. Next, section 3.3 briefly reviews three invalidate-based protocols presented in the literature, and section 3.4 presents two new update-based protocols. In particular, section 3.4.1 describes a centralized directory update-based protocol and section 3.4.2 describes a singly-linked distributed directory update-based protocol. Finally, section 3.5 discusses validation of the update-based protocols using an exhaustive validation tool called Murφ [19].

3.1 Directory Structures

Directory-based cache coherence protocols must maintain a directory entry for each memory line in the system. This directory entry indicates which caches in the system have a copy of the respective memory line. Each directory entry can be stored in a single, central location (centralized directory protocol) or distributed among the caches holding a copy of the line (distributed directory protocol). In both cases, the directory entries are distributed throughout the system with their respective memory lines.

3.1.1 Centralized Directory

In a centralized directory (CD) protocol, each directory entry contains a pointer to each cache in the system that contains a copy of the respective memory line. In the CD protocols studied in this work, a fully mapped directory is used in which there is a single bit pointer for each cache in the system [66]. For example, figure 3.1 shows a directory entry for a memory line in a four cache system. In the example, caches 1 and 3 have a copy of the given memory line.

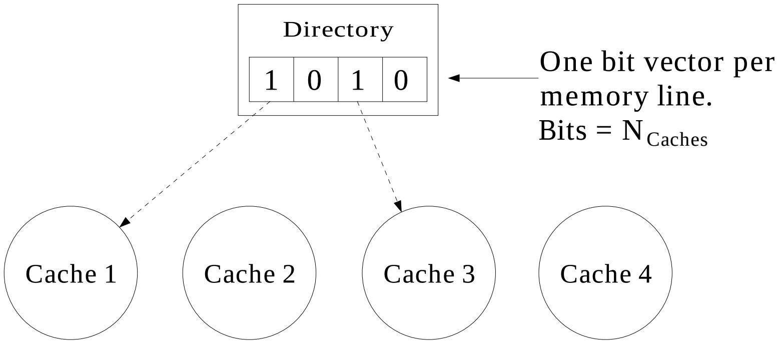


Figure 3.1: Centralized Directory Structure

In this fully mapped scheme, each directory entry contains N_{Caches} bits for a total storage requirement of

$$\begin{aligned} Bits &= N_{Caches} * N_{MemoryLines} \\ &= O(N_{Caches} N_{MemoryLines}) \end{aligned}$$

where N_{Caches} is the number of caches and $N_{MemoryLines}$ is the number of memory lines in the system.

3.1.2 Distributed Directory

In a distributed directory (DD) protocol, a linked list structure is used to maintain the list of caches that have a copy of a given memory line. The directory entry contains a pointer to the head of this list, and each cache line contains the necessary pointers to construct the list. The list may be singly-linked or doubly-linked. It is important to note that the order of the list is not optimized in any way. The order is determined by the order that the requests for the memory line reach the directory.

Singly-Linked Directory Structures

In a singly-linked distributed directory protocol [70], a singly-linked list is used to maintain the list as shown in figure 3.2. In this example, caches 0, 2 and 3 have a copy of the line.

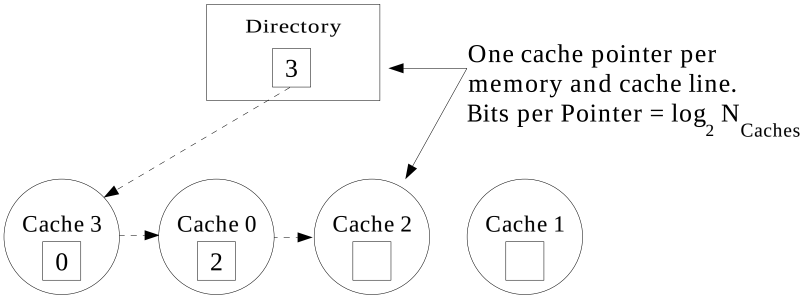


Figure 3.2: Singly-Linked Distributed Directory Structure

In this case, each directory entry contains $log_2(N_{Caches})$ bits, and each cache line must also include a single pointer. The directory now requires a total of

$$\begin{aligned} Bits &= N_{MemoryLines} log_2(N_{Caches}) + N_{CacheLines} log_2(N_{Caches}) \\ &= log_2(N_{Caches})(N_{MemoryLines} + N_{CacheLines}) \\ &= O(log_2(N_{Caches}) N_{MemoryLines}) \end{aligned}$$

which scales better than the fully-mapped CD directory structure as the size of the system (N_{Caches} and $N_{MemoryLines}$) increases.

Doubly-Linked Directory Structures

Alternatively, a doubly-linked directory structure may be used [42], as shown in figure 3.3. In this example, caches 0, 2 and 3 have a copy of the line.

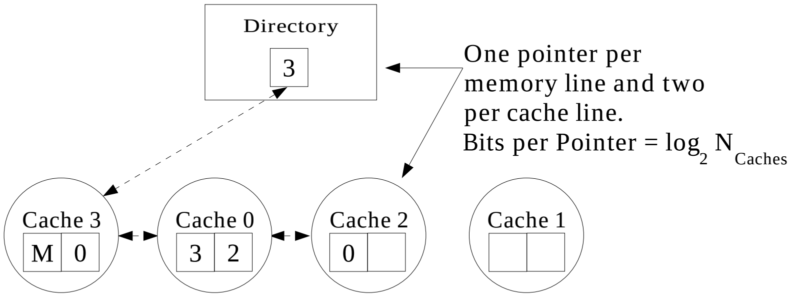

Directory
3
One pointer per memory line and two per cache line.
Bits per Pointer = log2 NCaches
Cache 3
M 0
Cache 0
3 2
Cache 2
0
Cache 1

Figure 3.3: Doubly-Linked Distributed Directory Structure

The amount of storage required is slightly more than that of the singly-linked distributed directory structure since each cache line must now maintain two pointers. The directory now requires

$$\begin{aligned} Bits &= N_{MemoryLines} log_2(N_{Caches}) + 2N_{CacheLines} log_2(N_{Caches}) \\ &= log_2(N_{Caches})(N_{MemoryLines} + 2N_{CacheLines}) \\ &= O(Log_2(N_{Caches})N_{MemoryLines}) \end{aligned}$$

of total storage.

3.1.3 Scalability of Directory Structures

As shown above, the centralized directory structure scales as $O(N_{Caches}N_{MemoryLines})$, while the distributed directories scale better as $O(log_2(N_{Caches})N_{MemoryLines})$. However, several different approaches have been suggested to improve the scalability of the centralized directory schemes. These include limited pointer schemes and cached directories.

The limited pointer schemes limit the number of cached copies of each memory line. When this limit is exceeded, the limited pointer schemes either invalidates one of the copies to make room for the new request [4], assumes all caches now have a copy of the line [4], switches to a coarse grain mode where each bit represents several caches [39] or traps to software to extend the directory list [14]. With a limited pointer scheme, the centralized directory scales as $O(N_{Limited}N_{MemoryLines})$ where $N_{Limited}$ is the number of bits in the limited directory entry.

The other approach notes that the maximum number of cached copies of a memory line is limited by the total size of all caches and not by the size of memory. In this case, a directory cache could be used to cache this smaller set of directory entries [39]. Also, the bits for each directory entry can be dynamically allocated out of a pool of directory bits [62].

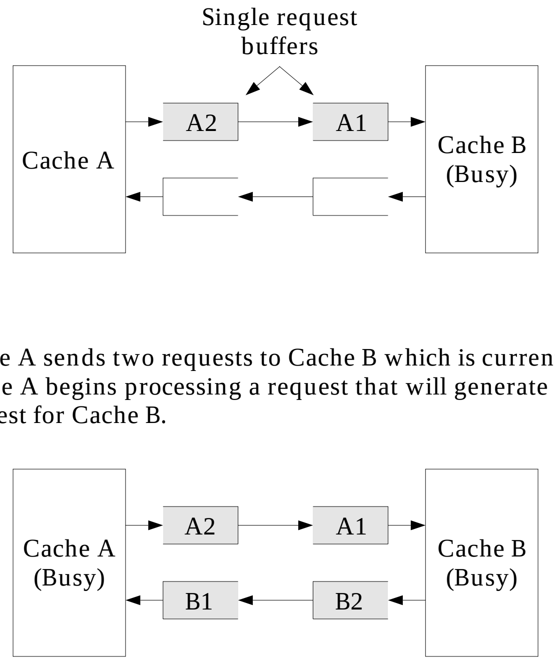
Several studies have suggested that the average number of shared copies of a memory line is small [13, 57, 71, 4, 26]. The results presented by the researchers demonstrate that the limited directory schemes result in a minimal performance loss [4, 14, 39], and similarly, the cached directory has also been shown to have a minimal affect on performance [39]. These results are *very* dependent on the number of shared copies of a line in a given application.

Some of the limited pointer schemes require that the directory be able to invalidate cached copies of a line. The centralized directory update-based protocol presented in this work currently does not support invalidations, but the protocol could be extended to support such directory initiated invalidations.

3.2 Protocol Deadlock

If the system has finite buffering, then protocol level deadlock is possible [69, 53, 35][1]. For example, figure 3.4 shows two caches that are sending requests to each other through a set of finite buffers. Each buffer can hold a single request. First, cache A sends two requests to cache B, and it begins processing a request that will generate another request to cache B. But because the buffers are already full, cache A must wait until a buffer becomes available before it is able to complete the processing of the new request. Meanwhile, cache B generates two requests to cache A, and it attempts to generate a third request. The system is now deadlocked. Neither cache A nor B can complete the processing of their current request because their output buffers are full, and they will never empty. A timeout must be used to detect such deadlocked

[1] The actual network is assumed to be deadlock free.



(a) Cache A sends two requests to Cache B which is currently busy. Cache A begins processing a request that will generate another request for Cache B.

(a) Cache A cannot complete current request since the buffer is full. Meanwhile, Cache B generates two requests to Cache A and attempts to generate a third request. Neither cache can proceed since their output buffers are full. The system is deadlocked.

Figure 3.4: Protocol Level Deadlock

situations. Once detected, there are two basic techniques to handle deadlock.

The first technique attempts to avoid deadlock by using local memory to expand the buffer space as needed [49]. When a buffer fills and a possible deadlock situation is detected, packets are removed from the buffer and placed in a secondary buffer created in the local memory. The cache and directory controllers must then process packets from this secondary buffer until it is empty. This technique essentially creates an almost infinitely sized buffer, but it requires a tight coupling of the cache controller, directory controller and local memory.

The second technique attempts to break the deadlock by removing requests from the deadlocked buffer and sending them back to their source through an exception network [53, 69, 35]. To minimize the probability of deadlock, message types are statically divided by the protocol into request and reply messages. A request message may generate another message and, therefore, lead to deadlock. A reply message never generates any new messages and, therefore, can always be consumed. A packet can be consumed if all resulting packets generated from the request, if any, can be successfully transmitted. The protocols examined in this dissertation use this second technique to avoid protocol level deadlock.

This second technique requires three logical networks: a request, reply and exception network. The reply network is deadlock free since replies can always be consumed. The request network may experience deadlock if the request at the head of the buffer generates another request. If the request generates a reply, then the request will eventually be consumed since the reply network will never deadlock.

The request-request deadlock is broken by sending the request at the head of the deadlocked buffer back to the source of the request to be retried. The act of removing a message from the deadlocked network may break the deadlock by freeing critical network resources. If not, this process would remove another request packet and send it back to the source. This is repeated until the deadlock condition is eliminated.

The separate request and reply networks also require logically separate controllers in both the cache and directory. Otherwise, a cache that is attempting to process a request would not be able to consume pending replies. This would violate the condition that replies always be consumed. Also, since replies are always consumed, the reply network may also be used as the exception network. This requires that the exceptions must always be consumed. If the network resources required to retransmit an exception packet are deadlocked, a trap handler must be invoked. The handler must store the exception packet in memory (using uncacheable accesses) and block the processor from generating any new requests. Eventually, the pending requests will be satisfied.

The frequency of deadlock is dependent on the size of the buffers. If reasonable buffer sizes are used, deadlock is extremely rare [69, 53]. In the system simulated in this work, the cache and memory buffers were 128 words deep, and deadlock never occurred for any of the cases examined [35, 34, 36].

3.3 Invalidate-Based Protocols (INV)

This dissertation compares update-based protocols with three invalidate protocols: a centralized directory protocol (CD-INV) that is similar to DASH [53], a singly-linked distributed directory protocol (DD-INV) [70] and a doubly-linked distributed directory protocol (SCI-INV) [45, 42], which is the IEEE standard protocol. This section gives a brief description of how these three invalidate-based protocols operate.

The invalidate-based protocols require that the writing cache obtain exclusive ownership of the line. With different directory structures, the protocols differ in how the invalidations, used to obtain exclusive ownership of the line, are performed. The protocols also differ in how misses are satisfied.

3.3.1 Centralized Directory (CD-INV)

In the CD-INV protocol, each cache line can be in one of six possible states:

- *Invalid* - the cache's copy of the line is not valid,
- *Reading* - the cache has issued a read miss and is waiting for a reply,
- *Writing* - the cache is waiting for exclusive ownership of the line,
- *Replacing* - the cache line is being replaced,
- *Shared* - the cache has a read-only copy of the line, or
- *Exclusive* - the cache has an exclusive read-write copy of the line (cache is owner of line).

Each memory line can be in one of four states:

- *Absent* - no caches have a copy of the line,
- *Shared* - at least one cache has a read-only copy of the line - memory is consistent,
- *Exclusive* - one cache has an exclusive read-write copy of the line - memory is not consistent, or
- *Update Pending* - the directory is awaiting a write back of the memory line.

The next few sections describe the actions taken for typical processor reads and writes.

Read Miss

Figure 3.5 shows the actions for a processor read miss when the memory line state is *Absent* or *Shared*. The cache first sends a *Read Miss* (RM) to the directory and the cache line state is set to *Reading*. The directory responds with a *Read Miss Reply* (RMR) and the memory line's data. The memory line state is set to *Shared*. When the cache receives the *Read Miss Reply* and a copy of the memory line from the directory, the cache line state is set to *Shared*, and the cache returns the desired word to the processor. The directory now points to requesting cache.

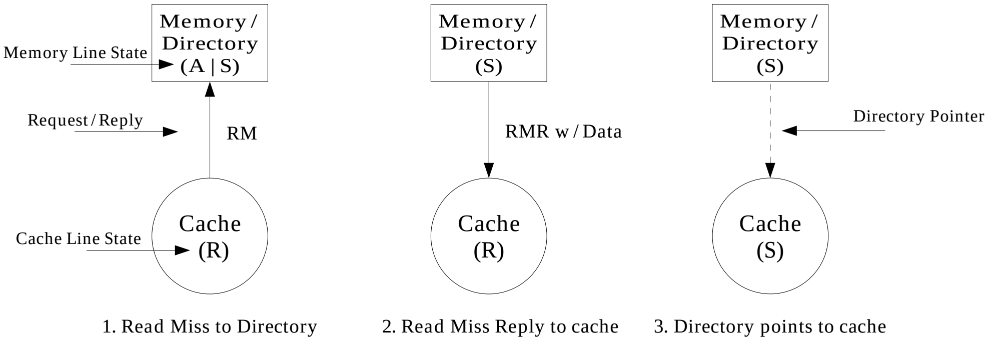


Figure 3.5: CD-INV: Read Miss to *Absent* or *Shared* Memory Line

If the memory line state is *Exclusive* on a read miss, then the memory line's current data must be fetched from the owning cache, as shown in figure 3.6. After receiving the *Read Miss* from the cache, the directory sends a *Write Back Shared* (WBS) to the owning cache, and the memory line state is set to *Update Pending*. This state is needed since all other requests to the line must wait until the line's data is received from the owning cache. Requests to the memory line while its state is *Update Pending* are bounced back to the sender to be retried. After receiving the *Write Back Shared*, the owning cache returns the line's data to memory and to the requesting cache. After receiving the *Update Line* (UL) reply and the line's data from the owning cache, memory is updated and the memory line state is set to *Shared*. The directory now points to both caches which are in the *Shared* state.

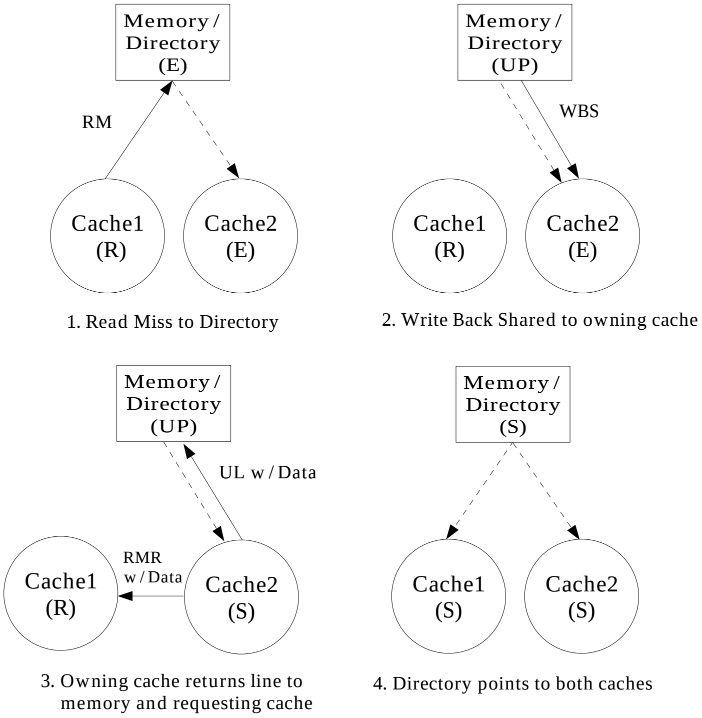


Figure 3.6: CD-INV: Read Miss to *Exclusive* Memory Line

Write Miss

The actions for a write miss to a memory line in the *Absent* state are identical to that of a read miss except that the cache and memory line state is set to *Exclusive* rather than *Shared*, but if the memory line is in the *Shared* state, all other copies of the line must be invalidated, as shown in figure 3.7. After receiving the *Write Miss*, the directory sends a *Write Miss Reply*, the number of pending invalidates and the line's data to the requesting cache. The directory sends an *Invalidate* signal to each cache holding a *Shared* copy of the memory line, and the memory line state is set to *Exclusive*. After receiving the *Invalidate* signal, each remote cache invalidates its copy of the line and sends an *Invalidate Acknowledge (Inv-Ack)* to the writing cache. The writing cache cannot release exclusive ownership of the line until all pending invalidations for the line have been acknowledged.

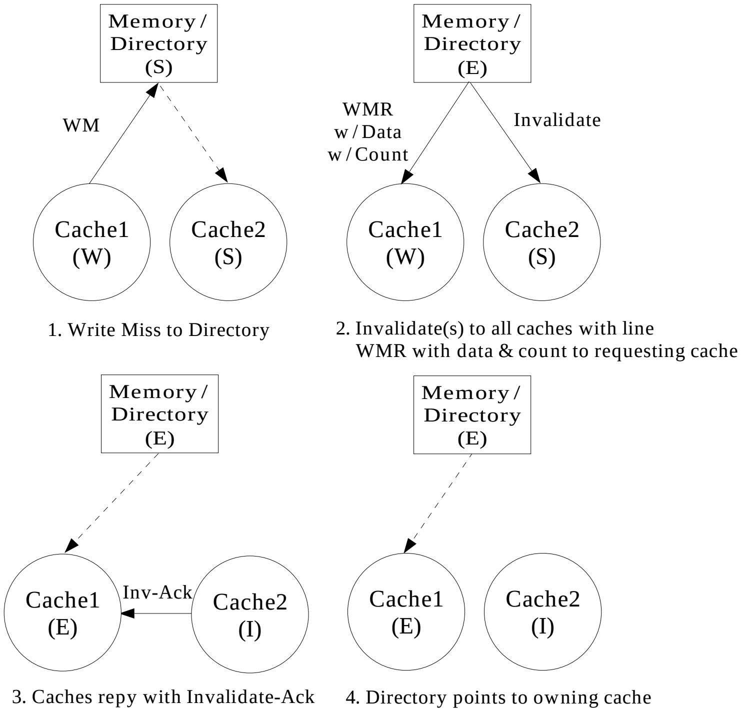


Figure 3.7: CD-INV: Write Miss to *Shared* Memory Line

The actions for a write miss to a memory line in the *Exclusive* state are similar to a read miss to a line in the *Exclusive* state as described above. The writing cache sends a *Write Miss* to the directory, as shown in figure 3.8. The directory must fetch the line from the owning cache and invalidate its copy. The directory does this by sending a *Write Back Invalidate* (WBI) to the owning cache. The owning cache invalidates its copy, forwards the line to the requesting cache, and informs the directory that the *Write Back Invalidate* has been performed. The requesting cache now has exclusive ownership of the line. Note that memory is not updated since the copy of the line held by the old owner is no longer valid since the new owning cache may modify it locally.

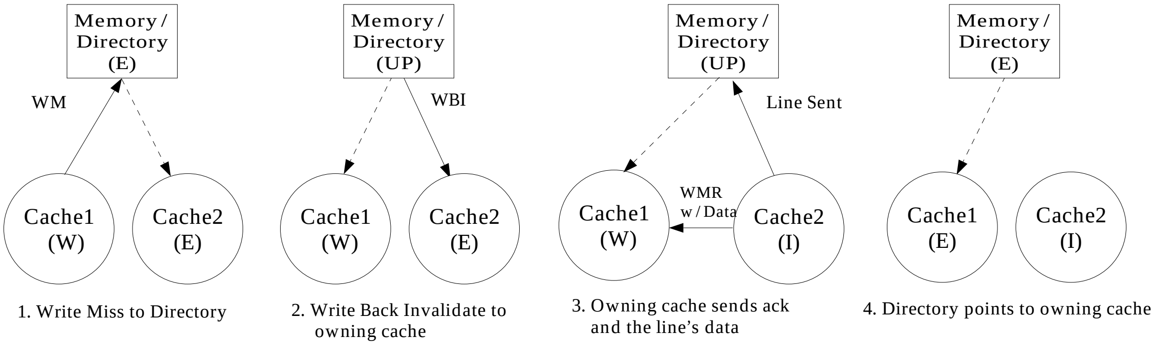


Figure 3.8: CD-INV: Write Miss to *Exclusive* Memory Line

Write Request

A write to a cache line that is in the *Shared* state requires that the cache obtains exclusive ownership of the line. The required actions are similar to those of a *Write Miss* when other shared copies of the line exist, as shown in figure 3.9. The only difference is that the cache already has a copy of the line so the directory responds with a *Write Granted* (WG) message and a count of pending invalidations. The *Write Granted* message gives the cache exclusive ownership of the line.

A write to a line in a pending state, states that are waiting on responses, must also be handled correctly. A write to a cache line in the *Writing* state can be issued since the cache's previous request for ownership of the line will eventually be granted. But a write to a line in the *Reading* state must be blocked at the write buffer until

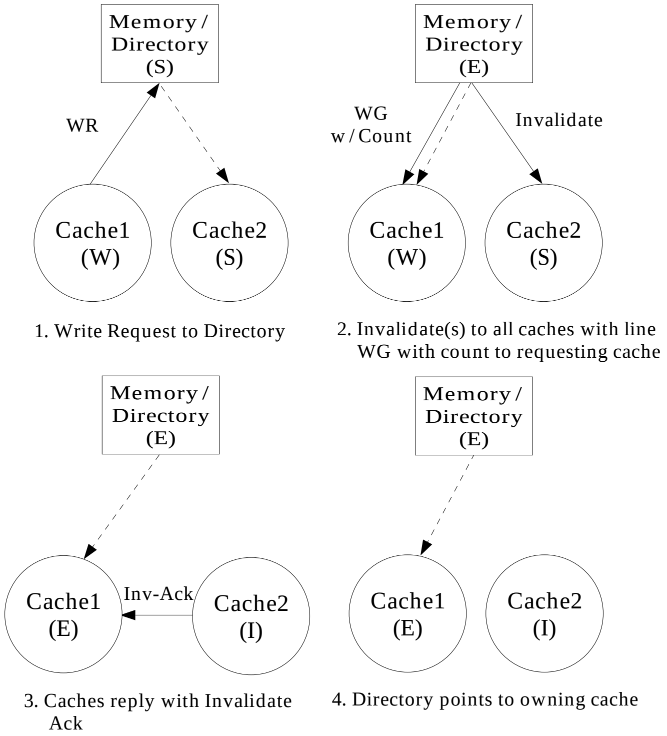


Figure 3.9: CD-INV: Write Request

the outstanding *Read Miss Reply* is received[2].

In summary, a writing cache must obtain exclusive ownership of the line which requires the invalidation of all other cached copies of the line. The CD structure allows the invalidations to be done in parallel, and the updating of memory on a write back allows subsequent misses to be serviced by the directory.

3.3.2 Distributed Directory

The state diagrams for the distributed directory protocols, DD-INV and SCI-INV, contain more than twice the number of states than in the CD-INV protocol. These extra states are needed to control and maintain the linked lists of caches. Therefore, because of this complexity, the state diagrams for these protocols will not be presented here, only the actions required for reads and writes will be examined. The terminology used to describe the actions will be similar to those used for the CD-INV protocol rather than the actual terminology used by the SCI-INV and DD-INV protocols. This similar terminology will make the comparison easier to follow. A detailed description of the DD-INV protocol is given by Thapar in his Ph.D. dissertation [69], and the details of the Typical SCI-INV protocol are given in the IEEE SCI P1596 specifications [42].

3.3.3 Singly-Linked Distributed Directory Protocol

This section briefly describes the singly-linked distributed directory invalidate-based (DD-INV) protocol.

Read Miss

For a read miss to a memory line that is absent from all other caches, the actions are identical to the previous protocol. If the memory line is present in other caches, the *Read Miss* signal is forwarded to the cache at the head of the list, as shown in figure 3.10. This cache sends a *Read Miss Reply* along with the data to the requesting

[2] A write to a line in the *Reading* state can occur if a read bypasses a write stored in the write buffer, or a write follows a prefetch.

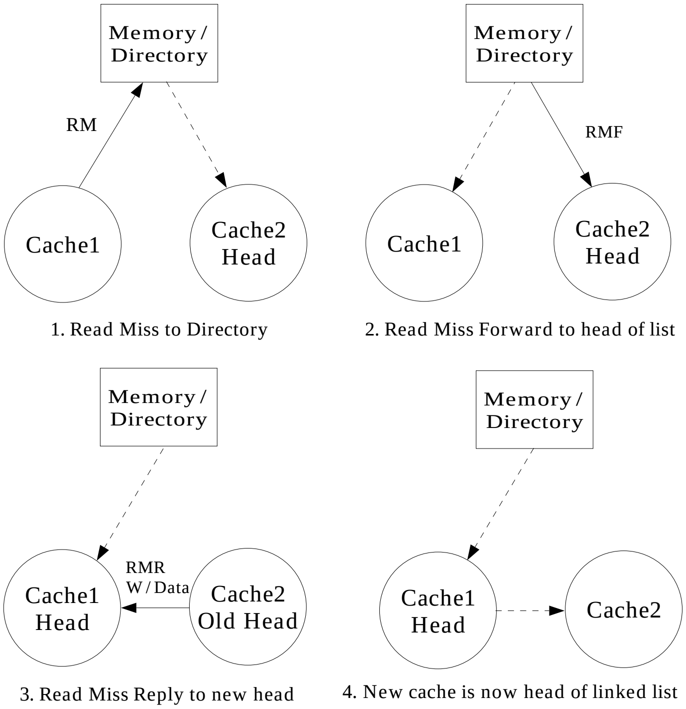
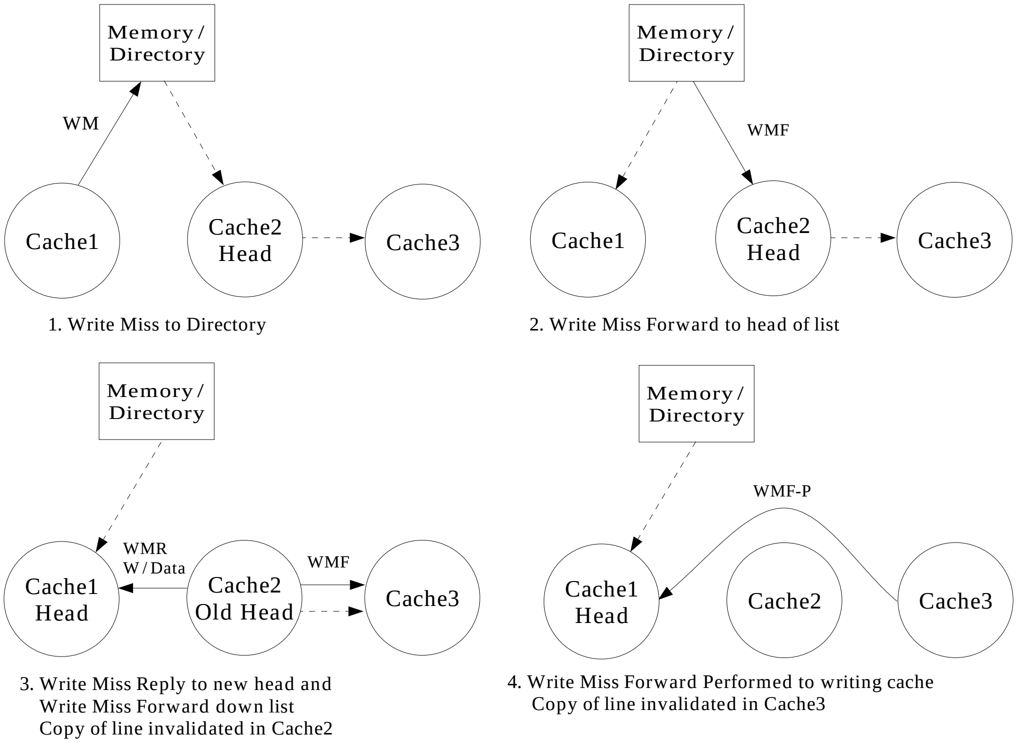


Figure 3.10: DD-INV: Read Miss

cache. The reading cache now becomes the head of the list. The next read miss will be satisfied by this cache.

Write Miss

For a write miss, all other cached copies of the line must be invalidated. The writing cache sends a *Write Miss* to the directory, as shown in figure 3.11. After receiving the miss, the directory sends a *Write Miss Forward* (WMF) signal to the cache at the head of the list of caches. This cache invalidates its copy, sends the *Write Miss Forward* down the list of caches, and replies to the writing cache with a *Write Miss Reply* and the line's data. The caches down the list invalidate their copy of the line as they receive the *Write Miss Forward* signal. The final cache in the list replies to the writing cache with a *Write Miss Forward Performed* (WMF-P). The writing cache now has exclusive ownership of the line.



1. Write Miss to Directory

2. Write Miss Forward to head of list

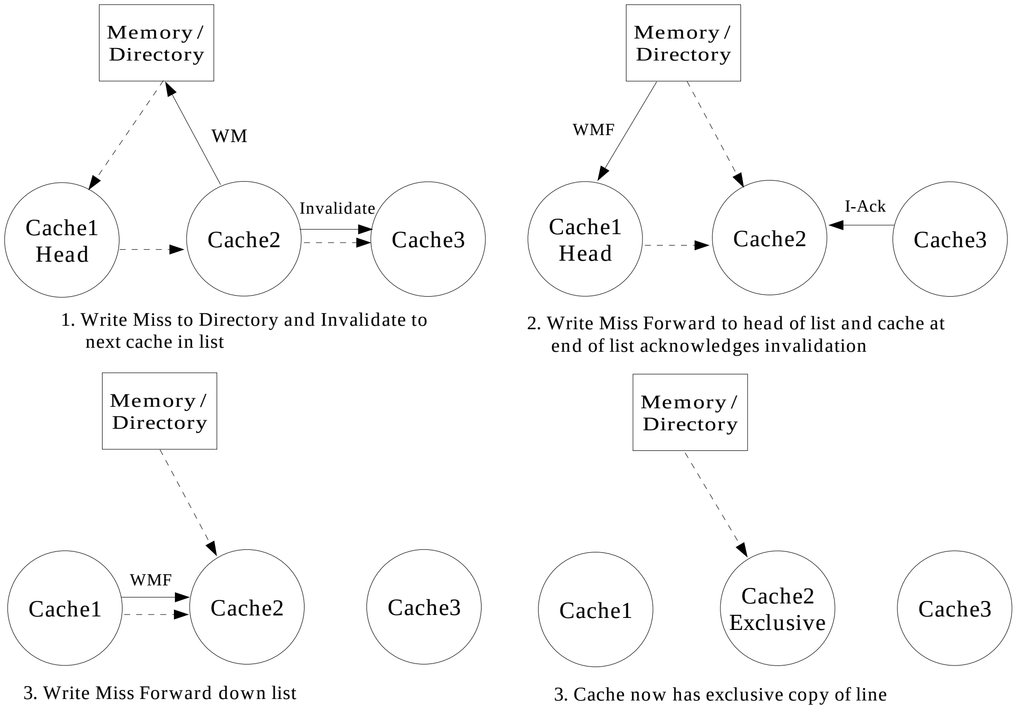
3. Write Miss Reply to new head and
Write Miss Forward down list
Copy of line invalidated in Cache2

4. Write Miss Forward Performed to writing cache
Copy of line invalidated in Cache3

Figure 3.11: DD-INV: Write Miss

Write Request

For a write to a cache line in the *Shared* state, the writing cache sends an *Invalidate* signal to the next cache in the list, as shown in figure 3.12. The caches down the list invalidate their copy as they receive the *Invalidate* signal. The final cache in the list sends an acknowledgment back to the writing cache. This operation has invalidated all caches down the list from the writing cache. If the writing cache is not at the head of the list, it also sends a *Write Miss* to the directory. When the directory receives the *Write Miss*, it sends a *Write Miss Forward* to the head of the list. This cache invalidates its copy of the line and forwards the signal to the next cache. When the writing cache receives both the acknowledgment from the invalidations and the *Write Miss Forward*, if it was not the head of the list, the cache has obtained exclusive ownership of the line.



1. Write Miss to Directory and Invalidate to next cache in list

2. Write Miss Forward to head of list and cache at end of list acknowledges invalidation

3. Write Miss Forward down list

3. Cache now has exclusive copy of line

Figure 3.12: DD-INV: Write Request

As in the CD-INV protocol, writes to a line in the *Writing* state are allowed to be performed. But writes to a line in the *Reading* state must be blocked at the write buffer.

In summary, the DD-INV protocol requires exclusive ownership of the line if the line is to be modified. The DD-INV protocol uses a pipelined technique to invalidate all other copies of a line. Memory is not updated when exclusive ownership of the line is released, but rather, the cache at the head of the list services all miss requests.

3.3.4 Doubly-Linked Directory Directory Protocol

This section briefly describes the Typical IEEE SCI invalidate-based (SCI-INV) protocol.

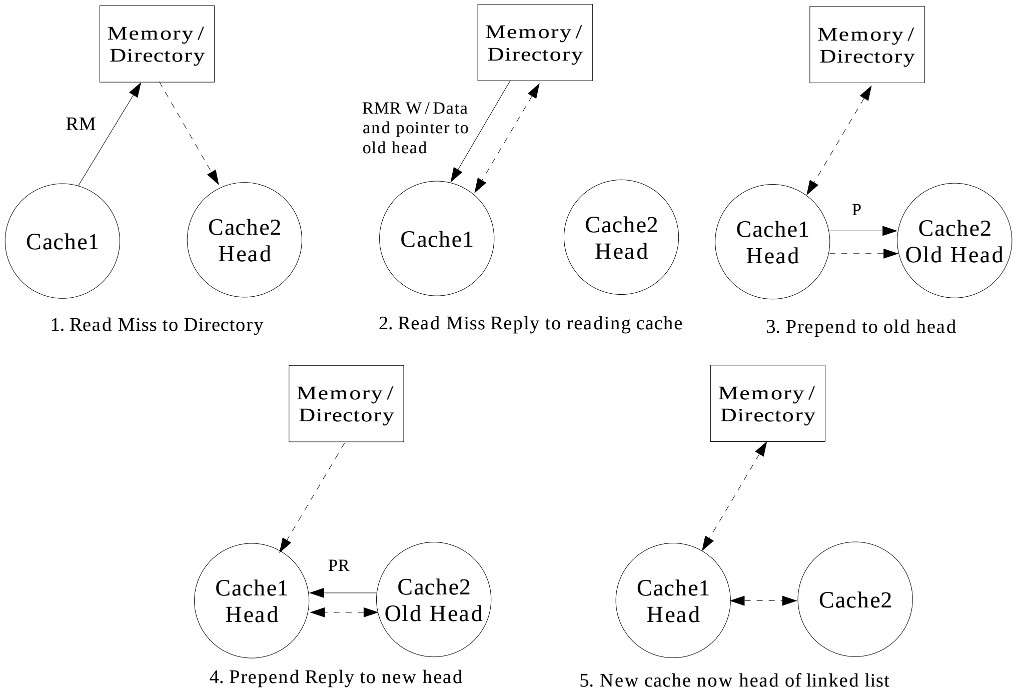
Read Miss

For a read miss to a memory line that is absent from all other caches, the actions are identical to that of the previous protocols. The cache sends a *Read Miss* to the directory, and the directory responds with a *Read Miss Reply* and a copy of the memory line. The directory now points to the cache.

On a read miss to a line that is present in other caches, the new cache must be added to the head of the linked list, as shown in figure 3.13. The directory responds to the *Read Miss* with the memory line's data and a pointer to the old head of the linked list. The new cache sets its backward pointer to point to the directory and its forward pointer to point to the old head of the list. It then sends a *Prepend* (P) signal to the old head of the list to add itself to the head of the list. The old head changes its backward pointer to point to the new cache and sends a *Prepend Reply* (PR) to the new cache. The new cache has now been added to the head of the linked list. If memory's copy was not valid, then the old head of the list would include the data in the *Prepend Reply* message.

Write Miss

For a write miss, the writing cache first becomes the head of the linked list of caches, as was shown in figure 3.13. Once the new cache is at the head of the list, all caches with a copy of the line must be invalidated. The writing cache sends an *Invalidate* to the next cache in the list and waits for an *Invalidate Reply* from the cache, as shown in figure 3.14. The reply carries a pointer to the next cache in the list. This process continues until all caches have been invalidated.



1. Read Miss to Directory

2. Read Miss Reply to reading cache

3. Prepend to old head

4. Prepend Reply to new head

5. New cache now head of linked list

Figure 3.13: SCI-INV: Read Miss to *Valid* Memory Line

Write Request

For a write to a cache line that is in a read-only state, the cache must obtain exclusive ownership of the line. The cache first removes itself from the linked list by sending a request to the caches pointed to by its forward and backward pointers. Once these caches acknowledge the deletion of the writing cache from the doubly-linked list, the writing cache sends a *Write Miss* to the directory, and the actions proceeds as described above. Writes to a line in the *Pending* State must be blocked at the write buffer because the protocol does not distinguish between lines in a *Reading* or *Writing* state.

In summary, the invalidations are done one at a time; each invalidate requires an

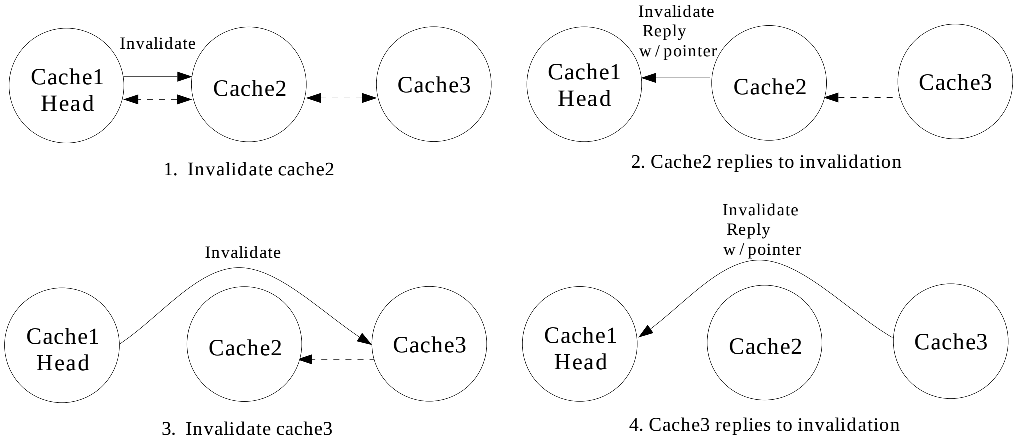


Figure 3.14: SCI-INV: Write Miss to Clean Line

acknowledgment before the next invalidation is performed. The protocol also requires more operations to add a cache to the doubly-linked list. The doubly-linked list is used to allow caches to remove themselves from the list without a request having to traverse the entire length of the list. See the SCI-INV specifications [42] and Thapar's Ph.D. thesis [69] for a more detailed discussion and comparison of the distributed directory invalidate-based protocols.

3.3.5 Summary

The invalidate-based protocols differ in how invalidation is performed. In the CD-INV protocol, the invalidates can be sent out in parallel since the directory entry contains all the necessary information. For the distributed directory protocols, the invalidates flow down the linked list of caches; the length of the list determines the latency of the invalidations. The DD-INV uses a pipelined technique; the invalidations flow down the list of caches. The SCI-INV protocol invalidates the caches in the list one at a time. The writing cache invalidates the next cache in the list and waits for an acknowledgment before invalidating the subsequent cache in the list.

The source of data for a miss reply also differs among the invalidate-based protocols. The CD-INV protocol supplies the data from main memory if the memory line is clean. If the line is dirty in another cache, the protocol uses cache-to-cache transfers to forward the data to the requesting cache, and if the request was a *Read Miss*, the protocol also writes the data back to memory. For the DD-INV protocol, cache-to-cache transfer is also used, but the line's data is not written back to memory. The cache at the head of the linked list always supplies the data on a miss. For the SCI-INV protocol, the directory can supply the data until it is modified by a cache after which cache-to-cache transfers are used. For a detailed comparison of the invalidate-based protocols see the work by Thapar [70, 69].

3.4 Update-Based Protocol (UP)

This section presents the details of the update-based protocols. Since the scalability of the directory structures is still an open research topic, update-based protocols are presented for both a fully-mapped centralized directory and a singly-linked distributed directory. Both update-based protocols presented require an order preserving network. An order preserving network guarantees that messages between nodes are received in the same order that they are sent.

3.4.1 Centralized Directory (CD-UP)

In this section, the details of the centralized directory update-based (CD-UP) protocol are presented.

Cache and Memory Line States

In the CD-UP protocol, a cache line can be in one of five states:

- *Invalid* - the cache's copy of the line is not valid,
- *Pending* - the cache has issued a miss and is waiting for a reply,
- *Shared* - the line is shared by multiple caches - writes must update other cached copies of the line and memory,
- *Replacing* - the cache's copy is being replaced, or

- *Exclusive* - the cache's copy is the only copy of the memory line (cache is owner of line).

A memory line can be in one of four states:

- *Absent* - no cached copies of the line exist,
- *Shared* - at least one cache has a copy of the line - memory is consistent,
- *Exclusive* - one cache has a copy of the line - memory is not consistent, or
- *Pending* - the directory is awaiting a write back for the memory line.

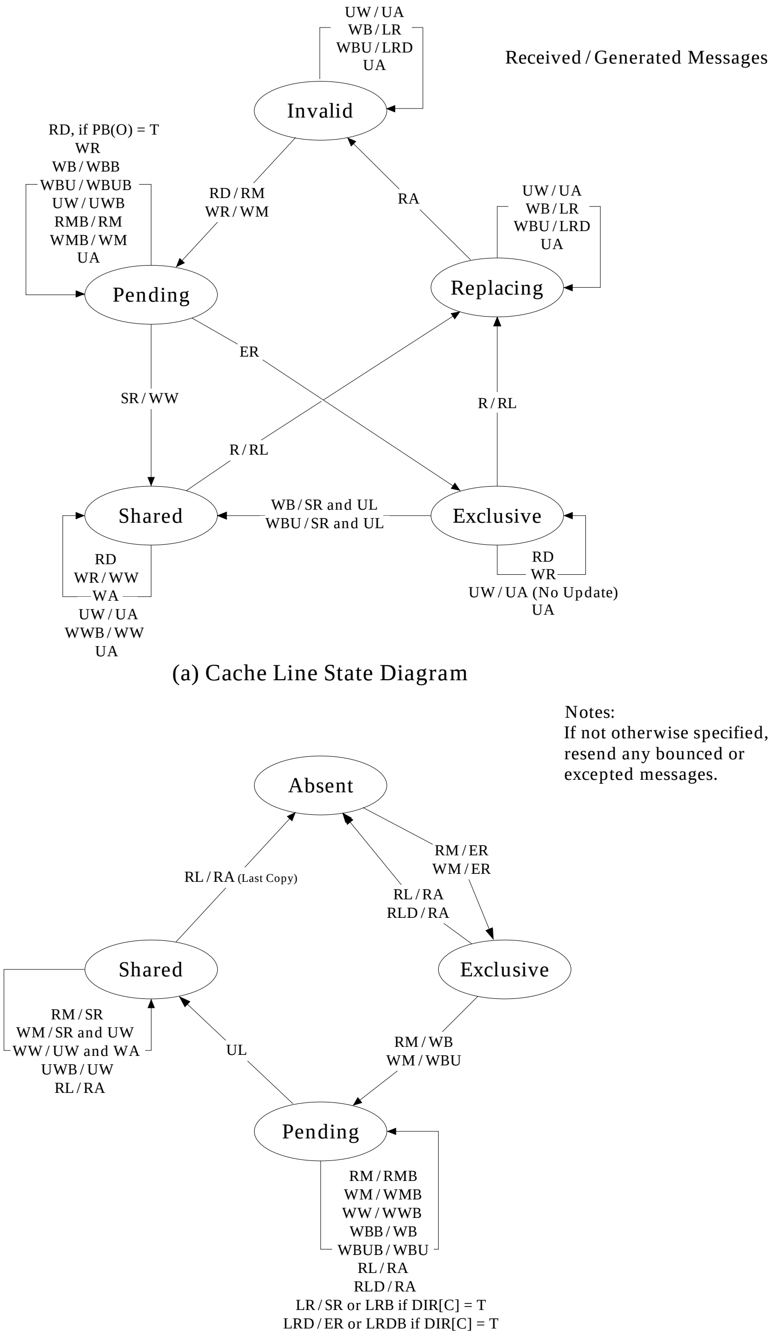
Table 3.1 describes the protocol messages. The first column of the table gives the message name, and the next two columns indicate the source and destination for each message. The source (Src) and destination (Dst) may be a processor (P), a cache (C) or a directory (D). The next column specifies if the message is a request (Req) or reply (Rep). The message type determines which network channel the message will traverse. The next column gives a brief description of the message, and the data column specifies what data type, a word or line, is sent with the message. Finally, the last column specifies what actions are taken by the destination node. These actions may include incrementing and decrementing the pending write counter (PWC) and the pending update counter (PUC). These counters are used to determine when all issued writes have been performed. Other actions include writing the data value (V) from the message into the cache line data (CD) or memory line data (MD) at the specified cache line offsets (O) and setting or clearing the directory (DIR) pointers. A Stall indicates that the processor is stalled until the proper reply is received by the cache, and Block indicates that the write is not processed, and it is not retired from the write buffer. A pending bit (PB) is used to control the writing of data into the cache line for certain conditions. The pending bits are identical to the valid bits used in the invalidate-based protocols [69].

Figure 3.15 shows the state transition diagrams for cache and memory lines. The diagrams show the state changes for each received message and the resulting messages generated, if any.

Message	Src	Dst	Type	Description	Data	Destination Actions
RD	P	C		Processor read		If *Pending* & PB(O) = F Then Stall
WR	P	C		Processor write	Word	If *Shared* & PB(O) = T Then Stall Else CD(O) = V If *Invalid* Then PWC += 1 If *Pending* Then PB(O) = T If *Shared* Then PWC += 1, PB(O) = T
R	P	C		Line replacement		If ForAny O: PB(O) = T Then Stall
FENCE	P			Stall until writes performed		Stall until PWC = PUC = O
RM	C	D	Req	Read miss		DIR(Src) = T
WM	C	D	Req	Write miss	Word	MD(O) = V, DIR(Src) = T
WW	C	D	Req	Write word to directory	Word	MD(O) = V
RL	C	D	Req	Replace line		DIR(Src) = F
RLD	C	D	Req	Replace line with data	Line	MD = V, DIR(Src) = F
SR	D or C	C	Req	Shared miss reply with update count (U)	Line	PWC -= 1, PUC += U Forall O: If PB(O) = F Then CD(O) = V Else PB(O) = F, send WW with CD(O)
ER	D	C	Rep	Exclusive miss reply	Line	PWC -= 1 Forall O: If PB(O) = F Then D(O) = V Else PB(O) = F
WB	D	C	Req	Write back line to directory		
WBU	D	C	Req	Write back line to directory with data update		CD(O) = V
UW	D	C	Req	Update word to cache(s)	Word	If PB(O) = F Then CD(O) = V
RA	D	C	Req	Replacement ack		
WA	D	C	Req	Write ack with update count (U)		PWC -= 1, PUC += U, PB(O) = F
UA	C	C	Rep	Update ack		PUC -= 1
LR	C	D	Rep	Line already replaced		
LRD	C	D	Rep	Line already replaced/Data	Word	
UL	C	D	Rep	Write back of line	Line	MD = V
RMB	D	C	Req	Bounce RM to cache		Resend RM
WMB	D	C	Req	Bounce WM to cache		Resend WM, V = CD(O)
WWB	D	C	Req	Bounce WW to cache		Resend WW, V = CD(O)
LRB	D	C	Req	Bounce LR to cache		Resend LR
LRDB	D	C	Req	Bounce LRD to cache	Word	Resend LRD
UWB	C	D	Req	Bounce UW to directory		Resend UW, V = MD(O)
WBB	C	D	Req	Bounce WB to directory		Resend WB
WBUB	C	D	Req	Bounce WBU to directory		Resend WBU, V = MD(O)

Legend	
P	Processor
C	Cache
D	Directory
Req	Request packet
Rep	Reply packet
PB(O)	Pending bit for offset O
CD(O)	Cache data for offset O
MD(O)	Memory data for offset 0
PWC	Pending write counter
PUC	Pending update counter
DIR(i)	Directory pointer for node i
PB(O)	Pending bit for offset O
U	Pending update ack count
V	Data value in packet

Table 3.1: CD-UP: Description of Messages



(a) Cache Line State Diagram

(b) Memory Line State Diagram

Figure 3.15: CD-UP: Cache and Memory Line State Diagrams

The next few sections describe the actions taken by the CD-UP protocol for typical read misses, write misses, write hits and line replacements. Protocol races and exception handling are also discussed.

Read Miss

For a read miss, the requesting cache sends a *Read Miss* to the directory, and the state of the cache line is set to *Pending*, as shown in figure 3.16. When the directory receives the miss request, the directory can reply with the line's data if the memory line state is *Absent* or *Shared*. If the memory line state is *Absent*, then the directory responds with an *Exclusive Reply* (ER) and the memory line state is set to *Exclusive*. If the memory line state is *Shared*, then the reply is a *Shared Reply* (SR). After the cache receives the reply, the cache line state is set to *Exclusive* for an *Exclusive Reply* or *Shared* for a *Shared Reply*.

If the memory line state is *Exclusive* on a *Read Miss*, the line's data must be fetched from the owning cache, as shown in figure 3.17. Once the requesting cache receives the line's data, the cache line state is set to *Shared*, and when the memory receives the *Update Line* with the line's data, the memory line state is also set to *Shared* and memory is updated.

Write Miss

The actions for a write miss to a memory line in the *Absent* state are identical to a read miss except that the cache sends a *Write Miss* to the directory. If the memory line is in the *Exclusive* state, the actions are similar to a *Read Miss* except that the directory sends a *Write Back Update* (WBU) to the owning cache. The *Write Back Update* contains the write value. This allows the owning cache to update its copy of the line before sending a *Shared Reply* to the requesting cache and an *Update Line* to the directory. After the transaction is complete, the states of both cache lines and the memory line are *Shared*.

If the memory line is in the *Shared* state on a write miss, all other copies of the line must be updated, as shown in figure 3.18. The writing cache sends a *Write*

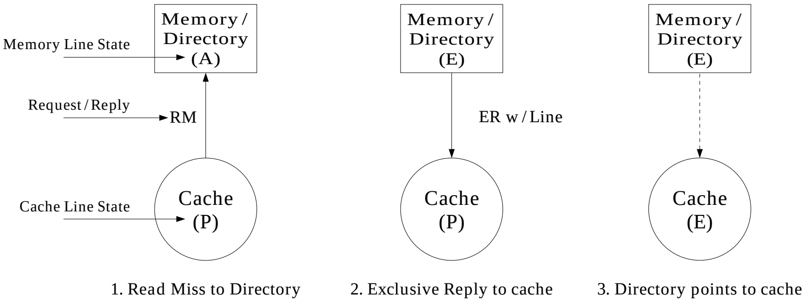


Figure 3.16: CD-UP: Read Miss to *Absent* Memory Line

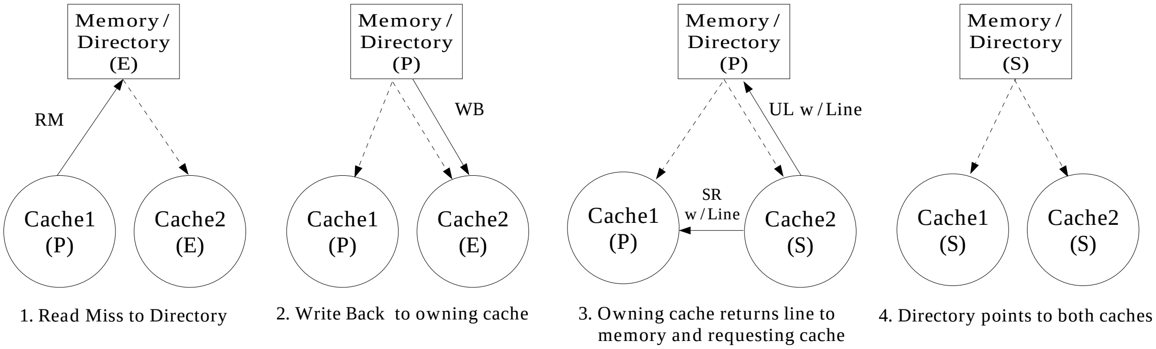


Figure 3.17: CD-UP: Read Miss to *Exclusive* Memory Line

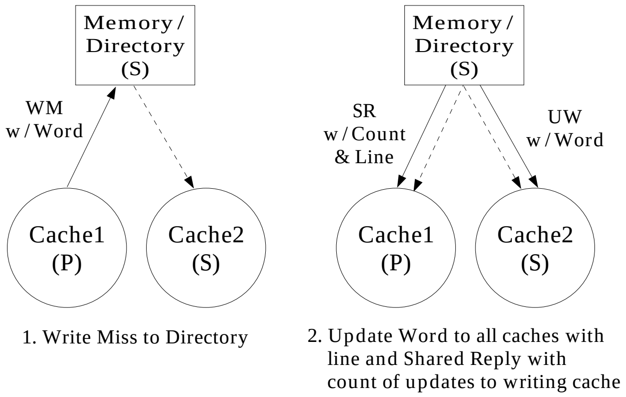


Figure 3.18: CD-UP: Write Miss to *Shared* Memory Line

Miss along with the new value to the directory, increments the pending write counter (PWC) and sets the cache line state to *Pending*. Once the directory receives the miss request, it sends an *Update Word* with the new value to all caches that have a copy of the line, and the directory also sends a *Shared Reply* with a count of the updates sent and the line's data to the writing cache. When the writing cache receives the reply, it decrements the pending write counter and adds the number of updates sent to the pending update counter (PUC). The updated caches receive the *Update Word*, update their copy of the line and send an *Update Ack* to the writing cache. Upon receipt of an *Update Ack*, the writing cache decrements the pending update counter. When both the pending write and pending update counters are zero, all previously issued writes have been performed.

Write Hit

Write hits to a cache line in the *Shared* state must also update all other cached copies of the line, as shown in figure 3.19. The resulting actions are similar to that of a *Write Miss*. The only difference is that the directory responds to the writing cache with a *Write Ack* rather than a *Shared Reply* since the writing cache already has a copy of the line.

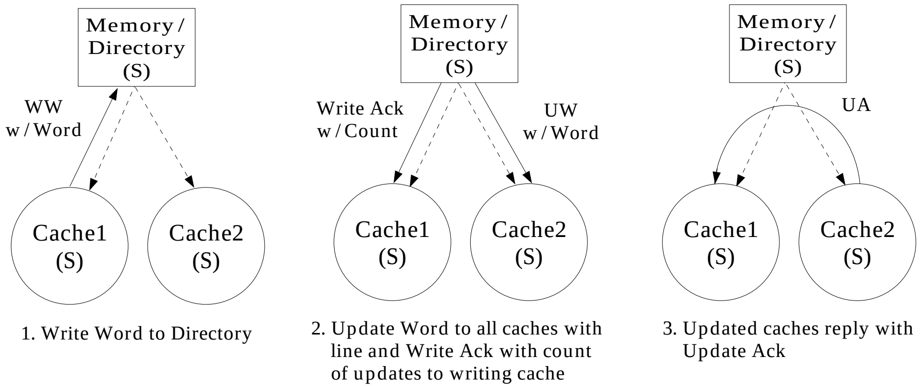


Figure 3.19: CD-UP: Write Hit to *Shared* Memory Line

The CD-UP protocol limits the number of outstanding updates per offset to one per cache. When the processor issues a write to a cache line in the *Shared* state, the update-pending bit for the offset (PB(O)) is set. When the write is acknowledged by a *Write Ack*, the bit is cleared. Any write to a cache line in the *Shared* state with the corresponding update-pending bit set is blocked. If the line is in the *Pending* state, the value is written into the cache line and the corresponding update-pending bit is set, but unlike writes to a line in the *Shared* state, writes are not blocked if the update-pending bit is already set. They overwrite any previous value. If the pending miss reply is a *Shared Reply*, then all offsets with the update-pending bit set are sent to the directory in a *Write Word* message. If the miss reply is an *Exclusive Reply*, then no updates are required. In both cases, the update-pending bits are cleared after the reply is received.

This restriction on outstanding updates should have little, if any impact of the performance of the protocols. If the writes are closely spaced, then the compiler may optimize these updates since they are actually a write after write hazard [41]. Any synchronization point between the writes will be blocked until the previous writes complete. If multiple writes to the same offset occurs, then the write grouping schemes that will be described in chapter 5 should be able to group the writes.

Line Replacement

Line replacement in the CD-UP protocol is straightforward, as shown in figure 3.20. The cache sends a *Replace Line* (RL) to the directory. The directory clears the cache's bit pointer in the directory entry for the memory line and acknowledges the replacement with a *Replacement Ack* (RA). If the cache line was in the *Exclusive* state and the line had been modified, then the cache must send the line's data along with the replacement request to the directory.

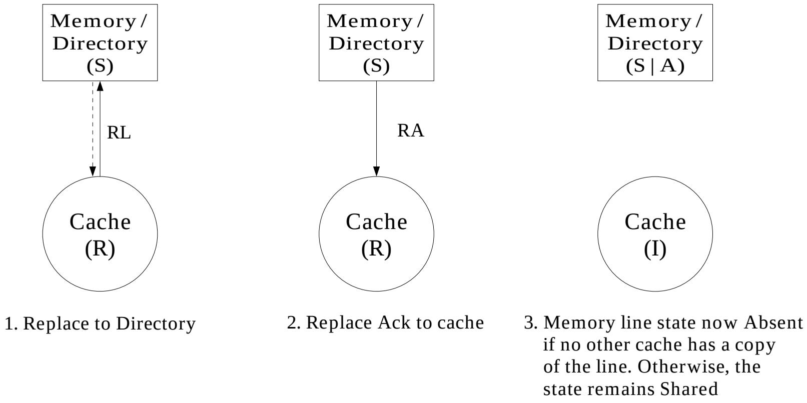
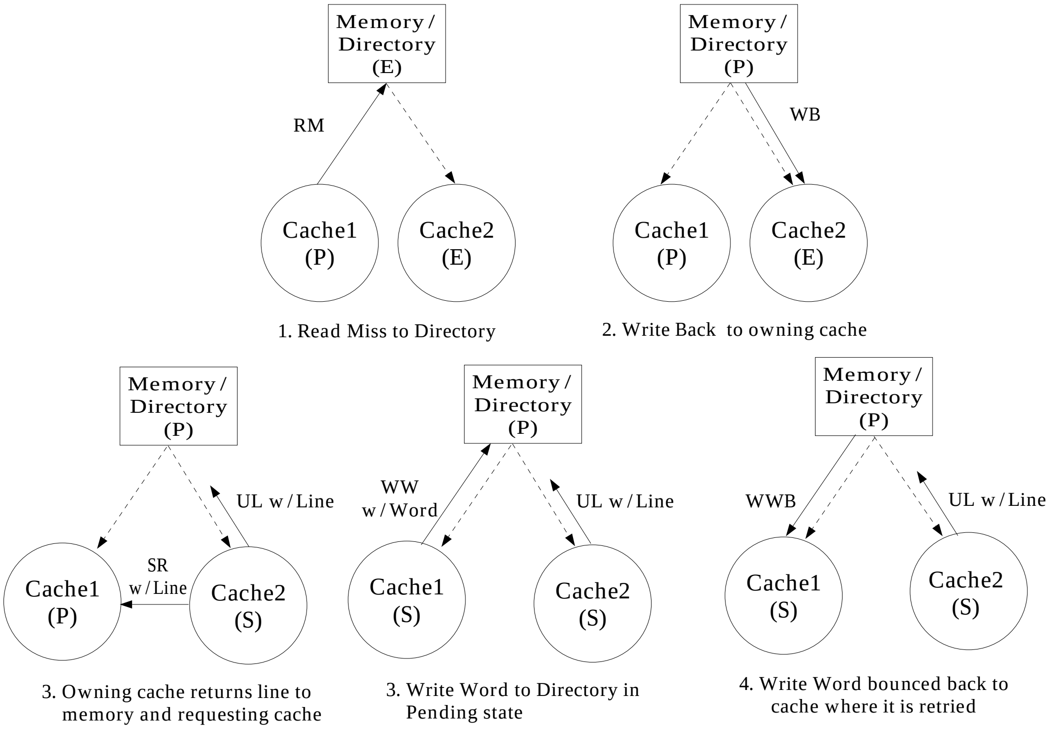

Memory / Directory (S)
RL
Cache (R)
1. Replace to Directory
Memory / Directory (S)
RA
Cache (R)
2. Replace Ack to cache
Memory / Directory (S | A)
Cache (I)
3. Memory line state now Absent if no other cache has a copy of the line. Otherwise, the state remains Shared

Figure 3.20: CD-UP: Replacing Cache Line in the *Shared* State

Protocol Races

There are several types of protocol races in the CD-UP protocol. The first type occurs when the directory receives a request to a memory line in the *Pending* state. These requests include *Read Miss*, *Write Miss* and *Write Word*. These races may occur when a cache has sent a miss request for a line that is currently owned by another cache and either the requesting cache initiates a *Write Word* or a third cache sends a miss request for the line before the *Update Line* from the owning cache is received by the directory. Figure 3.21 shows how a *Write Word* race might occur. For all three request types, the directory can bounce the request back to the sender to be retried.

The second type of protocol race is a request to a cache line in the *Pending* state. These requests include *Update Word* and *Write Back*. The *Update Word* race may



1. Read Miss to Directory

2. Write Back to owning cache

3. Owning cache returns line to memory and requesting cache

3. Write Word to Directory in Pending state

4. Write Word bounced back to cache where it is retried

Figure 3.21: CD-UP: Race: Write Word to *Pending* Memory Line

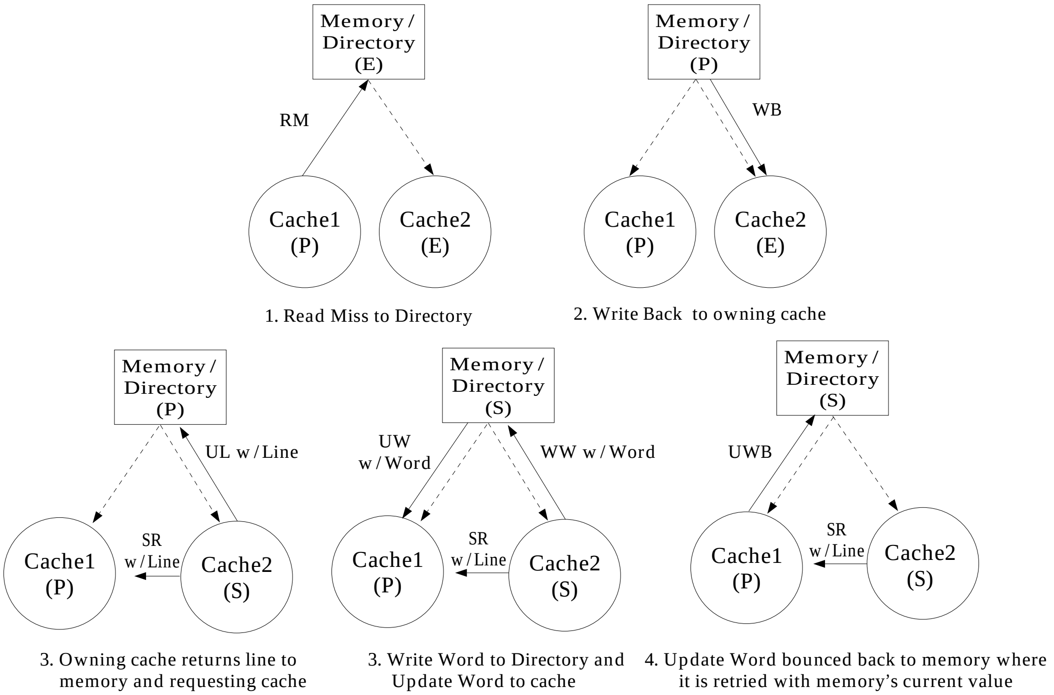
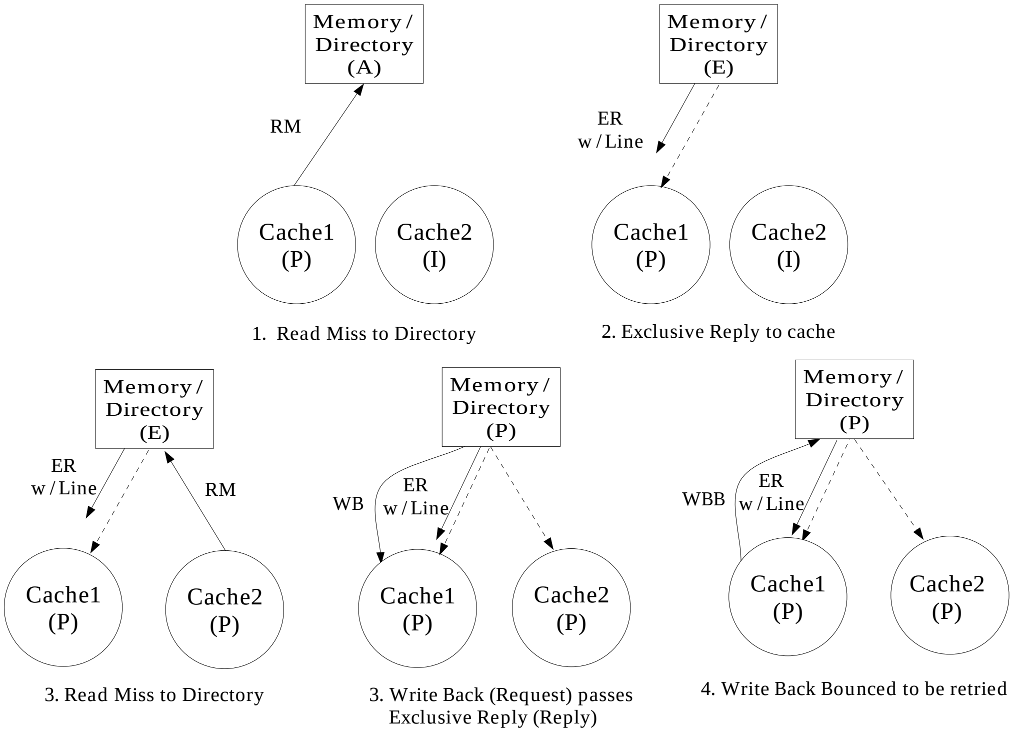


Figure 3.22: CD-UP: Race: Update Word to *Pending* Cache Line

occur if a cache sends a miss request to the directory, and if before the cache receives the miss reply, the cache receives an *Update Word* from a write by another cache, as shown in figure 3.22. The *Update Word* may reach the pending cache before the *Shared Reply* since the messages follow different paths: the *Update Word* from the directory and the miss reply from another cache. In this case, the cache must bounce the *Update Word* back to the directory. When the directory receives the bounced update, it resends the *Update Word* with the latest value of the memory word. Thus, a cache may not see all updated values if there are multiple writes to the same address without any synchronization events controlling the writes, but it will see the latest global value. The directory acts as the write serialization point.

The *Write Back* race may occur since an *Exclusive Reply* to a cache is a reply and the subsequent *Write Back* is a request; and therefore, they travel down different networks and may reach the cache out of order, as shown in figure 3.23. In this case, the *Write Back* is bounced back to the directory to be retried.



1. Read Miss to Directory

2. Exclusive Reply to cache

3. Read Miss to Directory

3. Write Back (Request) passes Exclusive Reply (Reply)

4. Write Back Bounced to be retried

Figure 3.23: CD-UP: Race: Write Back to *Pending* Cache Line

Message	Src	Dst	Description	Type
RL	C	D	Replace line	OD
UWB	C	D	Bounce UW to directory	OD
RM	C	D	Read miss	OI
WM	C	D	Write miss	OI
WW	C	D	Write word	OI
WB	D	C	Write back line	OI
WBU	D	C	Write back line w/update	OI
SR	D or C	C	Shared miss reply	OI
LR	C	D	Line replaced	OI
RMB	D	C	Bounce RM to cache	OI
WMB	D	C	Bounce WM to cache	OI
WWB	D	C	Bounce WW to cache	OI
WBB	C	D	Bounce WB to directory	OI
WBUB	C	D	Bounce WB to directory	OI
LRB	D	C	Bounce LR to cache	OI

Table 3.2: CD-UP: Messages That May Deadlock

The final type of race is a request to a cache line in the *Replacing* state. These requests include *Update Word* and *Write Backs*. For the *Update Word* request, the cache can simply acknowledge the update. For the *Write Back* requests, the cache line must have been in the *Exclusive* state previously and just sent a *Replace Line* request to the directory. In this case, the cache sends a *Line Replaced* (a *Line Replaced Data* with the update data if the request was a *Write Back Update*) back to the directory. When the directory receives the *Line Replaced*, it will have already received the *Replace Line* request, and memory's data will be valid. The directory may now respond to the initial miss request with memory's current data.

Exceptions

As described in section 3.2, the protocol must be able to break request-request deadlocks. Table 3.2 shows the request messages that may generate another request. These messages can be divided into two classes. The first class of messages are essentially order-independent (OI). They do not rely on the order preserving nature of the network and, therefore, do not introduce any additional complexity to the protocols if they are bounced back to the sender as an exception. For example, the *Read Miss* message is OI. Once a cache sends a *Read Miss* to the directory, the cache will not generate any other messages relating to this line, and until the directory receives the

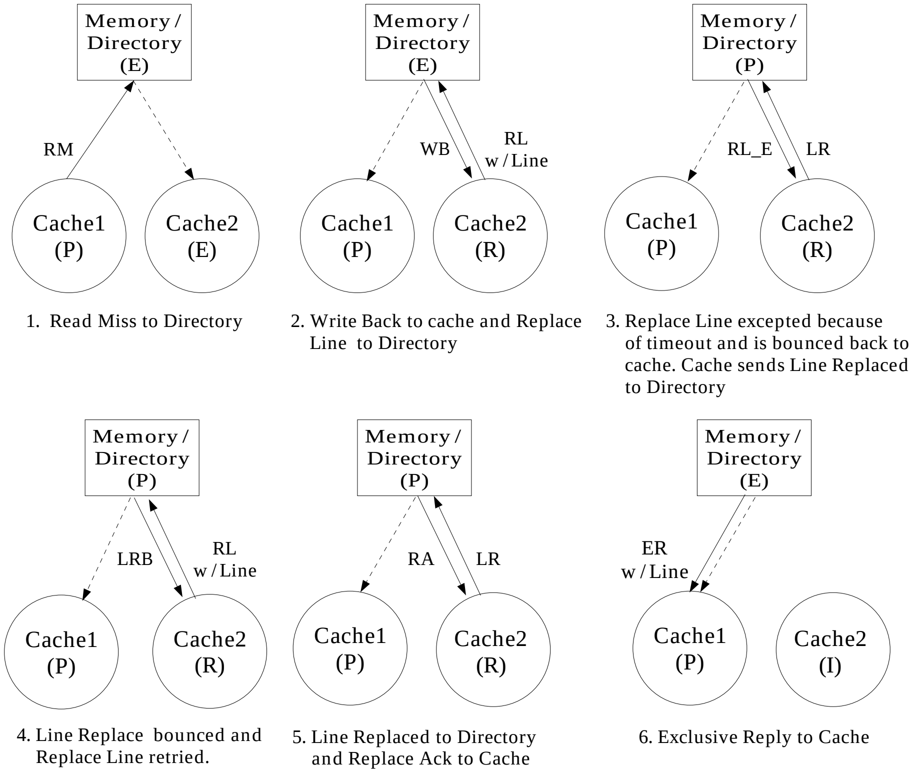

Memory / Directory (E)
RM
Cache1 (P)
Cache2 (E)
1. Read Miss to Directory
Memory / Directory (E)
WB
RL w / Line
Cache1 (P)
Cache2 (R)
2. Write Back to cache and Replace Line to Directory
Memory / Directory (P)
RL_E
LR
Cache1 (P)
Cache2 (R)
3. Replace Line excepted because of timeout and is bounced back to cache. Cache sends Line Replaced to Directory
Memory / Directory (P)
LRB
RL w / Line
Cache1 (P)
Cache2 (R)
4. Line Replace bounced and Replace Line retried.
Memory / Directory (P)
RA
LR
Cache1 (P)
Cache2 (R)
5. Line Replaced to Directory and Replace Ack to Cache
Memory / Directory (E)
ER w / Line
Cache1 (P)
Cache2 (I)
6. Exclusive Reply to Cache

Figure 3.24: CD-UP: Race: Line Replaced to *Pending* Memory Line

miss request, it will not send the cache any message pertaining to the line. The *Read Miss* message can take any path from the cache to the directory including being sent back to the cache as an exception without introducing any new complexities to the protocol.

Of the order-dependent (OD) messages, the exception of a *Replace Line* message may cause a race condition, as shown in figure 3.24. In this case, a *Line Replaced* may reach the directory before the *Replace Line*. The directory must bounce the *Line Replaced* back to the replacing cache. This action is repeated until the directory receives the *Replace Line* message and the line's current data. Now when the *Line Replaced* is received, the directory may respond to the pending miss request.

An *Update Word Bounce* may also result in an exception, but since an *Update Word Bounce* does not carry data (it only carries a promise of an update), it can take an arbitrary path between the cache and directory without introducing any additional complexities to the protocol. Once the directory is able to process the *Update Word Bounce*, it sends an *Update Word* with the memory's current data. If the destination of the update no longer has a copy of the line, the directory acknowledges the writing cache directly.

3.4.2 Distributed Directory (DD-UP)

In this section, the details of the singly-linked distributed directory update-based protocol (DD-UP) are presented. The DD-UP is based on the directory structure and singly-linked lists of the DD-INV protocol [69, 70].

Cache and Memory Line States

In the DD-UP protocol, a cache line may be in one of 12 states:

- *Invalid* - the cache's copy of the line is not valid,
- *Pending* - the cache has issued a miss and is waiting for a reply,
- *Exclusive* - the cache's copy is the only copy of the line (cache is owner of line),
- *Shared* - the line is shared by multiple caches and the cache is neither the head nor the tail of the list - writes must update other cached copies of the line,
- *Shared-Head* - the line is shared by multiple caches and the cache is the head of the list - writes must update other cached copies of the line,
- *Shared-Tail* - the line is shared by multiple caches and the cache is the tail of the list - writes must update other cached copies of the line,
- *Replacing-Head* - the cache's copy, which is the head of the list, is being replaced,
- *Replacing* - the cache's copy, which is neither the head nor the tail of the list, is being replaced,
- *Replacing-Tail* - the cache's copy, which is the tail of the list, is being replaced,

- *Replacing-Exclusive* - the cache's copy, which is the only copy of the line, is being replaced,
- *Exclusive-Replacing* - the cache's copy was the head of the list and the next cache in the list, which was the tail of the list, is replacing itself. The head of the list will contain the exclusive copy of the line once the replacement is complete, or
- *Shared-Head-Replacing* - the cache's copy is the head of the list and a cache in the list is replacing itself.

A memory line may be in one of 3 states:

- *Absent* - no cached copies of the line exist,
- *Present* - at least one cache has a copy of the memory line - memory is not consistent, or
- *Replacing* - a cache in the list is removing itself from the list - memory is not consistent.

Table 3.3 gives a brief description of the protocol messages. The columns have the same meaning as in the CD-UP protocol. The DD-UP protocol can store (Store) requests to cache lines in the *Pending* state in the pointer field of the cache line. When the appropriate reply is received and the cache line state is changed, the pending signal is processed (Process-Stored). The state *Shared-States* implies any of the three shared states: *Shared-Head*, *Shared* or *Shared-Tail*. The Aux field is an additional pointer field used by some of the message types. Figure 3.25 shows the state transition diagrams for cache and memory lines.

The next few sections describe the actions taken by the DD-UP protocol for typical read misses, write misses, write hits and line replacements. Protocol races and exception handling are also discussed.

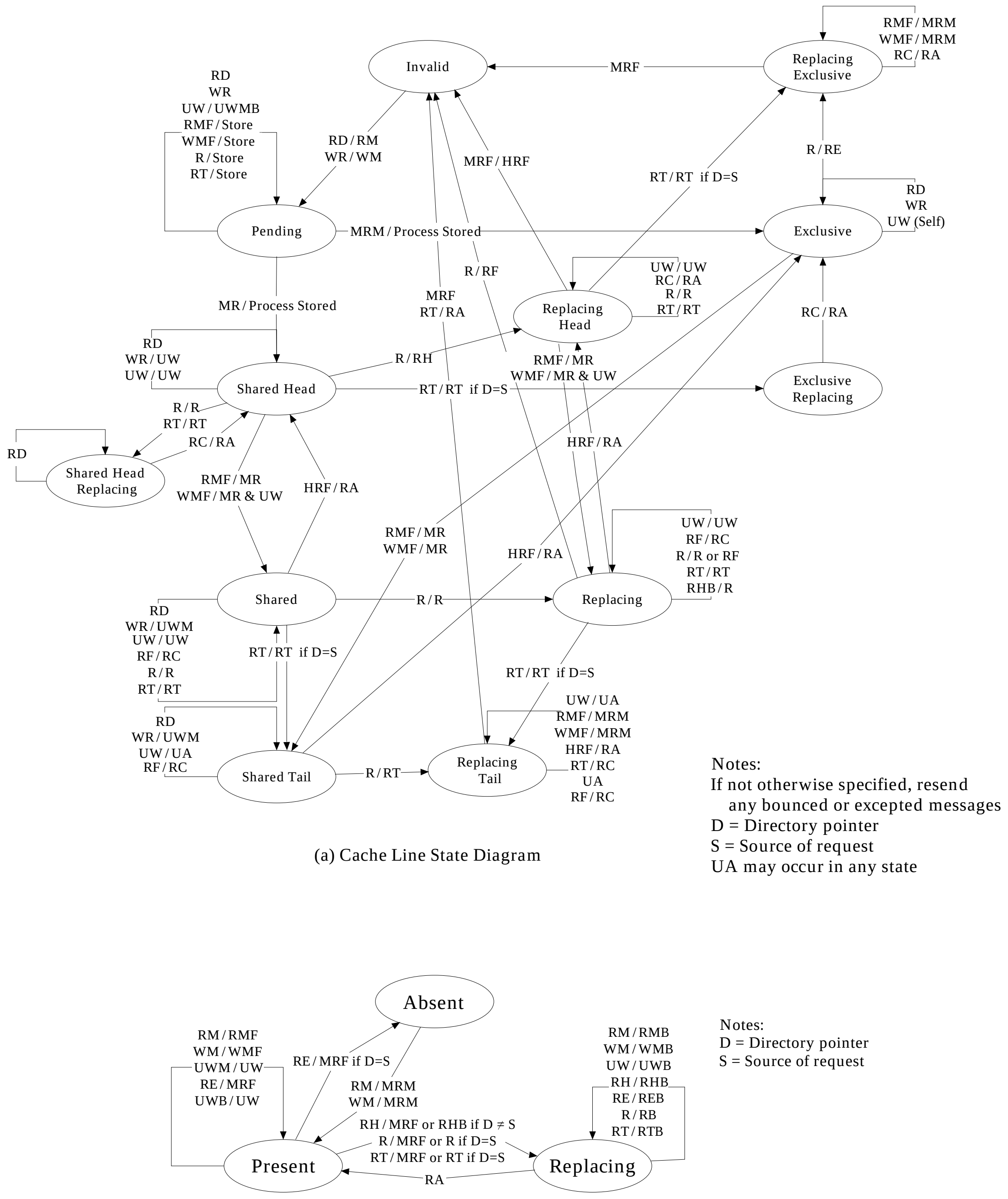
Read Miss

For a read miss, the cache sends a *Read Miss* request to the directory. If the state of the memory line is *Absent*, then the directory replies to the miss with a *Miss Reply*

Message	Src	Dst	Type	Description	Data	Destination Actions
RD	P	C		Processor read		If *Pending* & PB(O) = F Then Stall
WR	P	C		Processor write	Word	If *SharedStates* & PB(O) = T Then Stall Else CD(O) = V If *Invalid* Then PWC += 1 If *Pending* Then PB(O) = T If *Shared-States* Then PWC += 1, PB(O) = T
R	P	C	Req	Line replacement		If ForAny O: PB(O) = T Then Stall Else Aux = Dir
FENCE	P	C		Stall until writes performed		Stall until PWC = PUC = O
RM	C	D	Req	Read miss		Dir = Src
WM	C	D	Req	Write miss	Word	Dir = Src, Aux = Src
UWM	C	D	Req	Update word memory	Word	Aux = Src
UW	D or C	C	Req	Update word	Word	If PB(O) = F Then CD(O) = V If Src = Aux Then PB(O) = F
RE	C	D	Req	Replace exclusive line		
RH	C	D	Req	Replace shared-head line		Dir = Aux
R	C D or C	D C	Req Req	Replace shared line Replace shared line		If Dir = Src Then Dir = Aux If Dir = Aux Then Dst is replacing cache
RT	C D or C	D C	Req Req	Replace shared-tail line Replace shared-tail line		
HRF	C	C	Req	Head replace flush		
RC	C	C	Req	Replace complete		
MRF	D	C	Req	Memory replace flush		
RF	C	C	Req	Replace flush		
MR	C	C	Req	Miss reply with update count (U) (U = 0,1)	Line	PWC -= 1, PUC += U, DIR = Src Forall O: If PB(O) = F Then CD(O) = V Else send UW with CD(O), PB(O) = F
MRM	D	C	Rep	Memory miss reply	Line	PWC -= 1 Forall O: If PB(O) = F Then CD(O) = V Else PB(O) = F
RMF	D	C	Req	Read miss forward		
WMF	D	C	Req	Write miss forward with data update		CD(O) = V
RA	D	C	Req	Replace ack		
UA	C	C	Rep	Update ack		PUC -= 1, PB(O) = F
RMB	D	C	Req	Bounce RM to cache		Resend RM
WMB	D	C	Req	Bounce WM to cache		Resend WM, V = CD(O)
UWB	D C	C D	Req Req	Bounce UW to cache Bounce UW to directory		Resend UW, V = CD(O) Resend UW
RHB	D	C	Req	Bounce RH to cache		Resend RH
REB	D	C	Req	Bounce RE to cache		Resend RE
RTB	D	C	Req	Bounce RT to cache		Resend RT
RB	D	C	Req	Bounce R to cache		Resend R

Legend	
P	Processor
C	Cache
D	Directory
Req	Request packet
Rep	Reply packet
PB(O)	Pending bit for offset O
CD(O)	Cache data for offset O
MD(O)	Memory data for offset 0
PWC	Pending write counter
PUC	Pending update counter
DIR	Directory pointer
PB(O)	Pending bit for offset O
U	Pending update ack count
V	Data value in packet

Table 3.3: DD-UP: Description of Messages



Notes:
If not otherwise specified, resend
any bounced or excepted messages
D = Directory pointer
S = Source of request
UA may occur in any state

(a) Cache Line State Diagram

Notes:
D = Directory pointer
S = Source of request

(b) Memory Line State Diagram

Figure 3.25: DD-UP: Cache and Memory Line State Diagrams

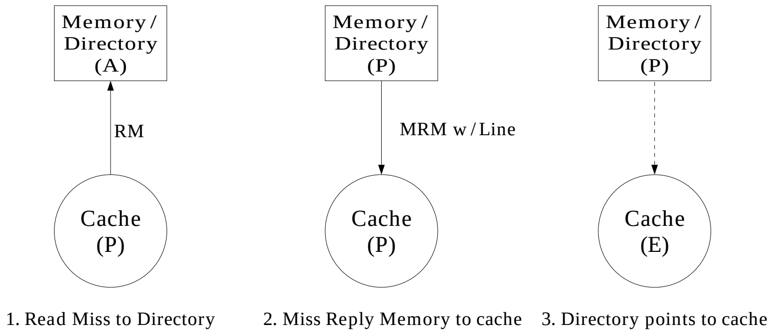


Figure 3.26: DD-UP: Read Miss to *Absent* Memory Line

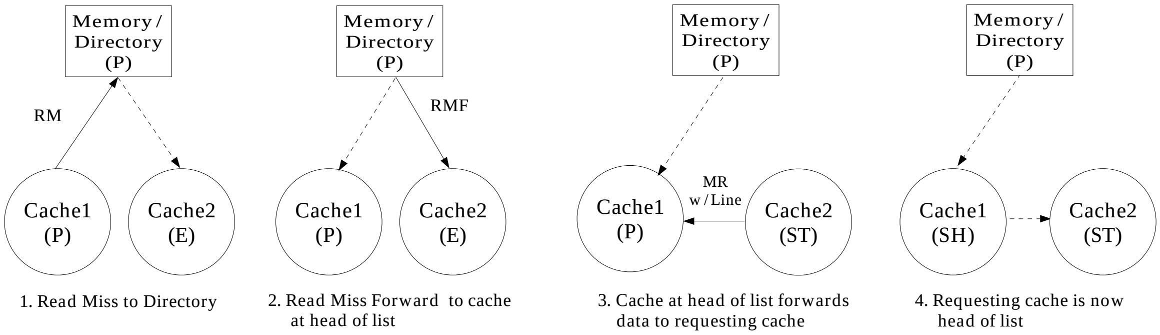


Figure 3.27: DD-UP: Read Miss to *Present* Memory Line

Memory (MRM). The state of the memory line is set to *Present*, and the cache line state is set to *Exclusive*, as shown in figure 3.26.

If the memory line state is *Present*, then the data must be fetched from the cache at the head of the list, and the requesting cache must be added to the head of the list, as shown in figure 3.27. The directory responds to the miss request by sending a *Read Miss Forward* (RMF) to the cache at the head of the list, and the directory pointer is changed to point to the requesting cache, the new head of the list. The old head of the list responds to the *Read Miss Forward* by sending a *Miss Reply* (MR) with the line's data to the requesting cache. After receiving the reply, the requesting cache sets its directory pointer to point to the cache that sent the reply.

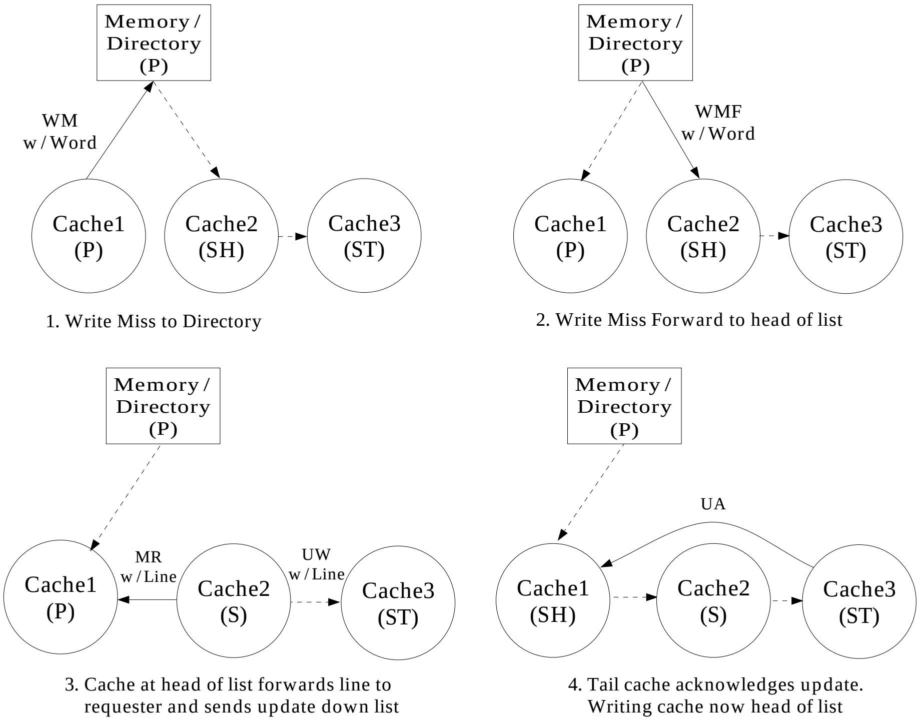


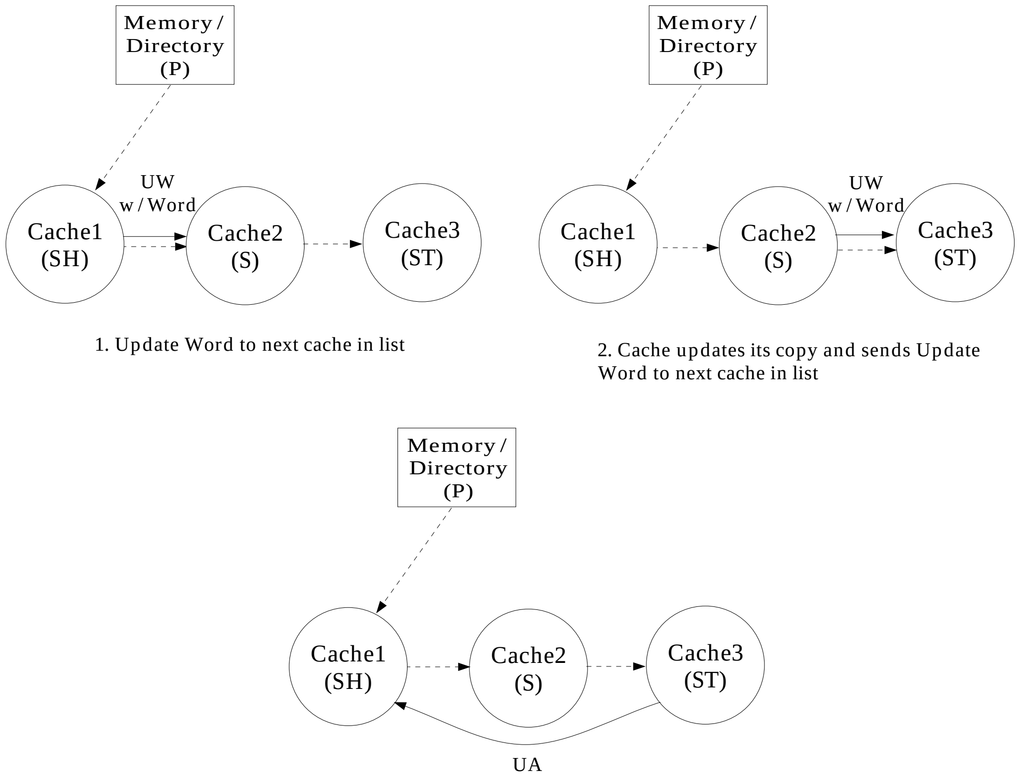
Figure 3.28: DD-UP: Write Miss to *Present* Memory Line

Write Miss

The actions for a write miss are identical to a read miss if the memory line state is *Absent*. If the memory line state is *Present*, then the directory sends a *Write Miss Forward* (WMF) and the new data value to the old head of the list, as shown in figure 3.28. The old head of the list updates its copy of the line, sends a *Miss Reply* along with the line's data to the requesting cache and sends an *Update Word* down the list of caches. The cache at the end of the list acknowledges the update, and the writing cache becomes the head of the list.

Write Hit

On a write hit to a cache line in any of the three shared states, *Shared-Head*, *Shared* or *Shared-Tail*, the other caches in the list must be updated. If the writing cache is at the head of the list, it can simply send an *Update Word* to the next cache in the



1. Update Word to next cache in list

2. Cache updates its copy and sends Update Word to next cache in list

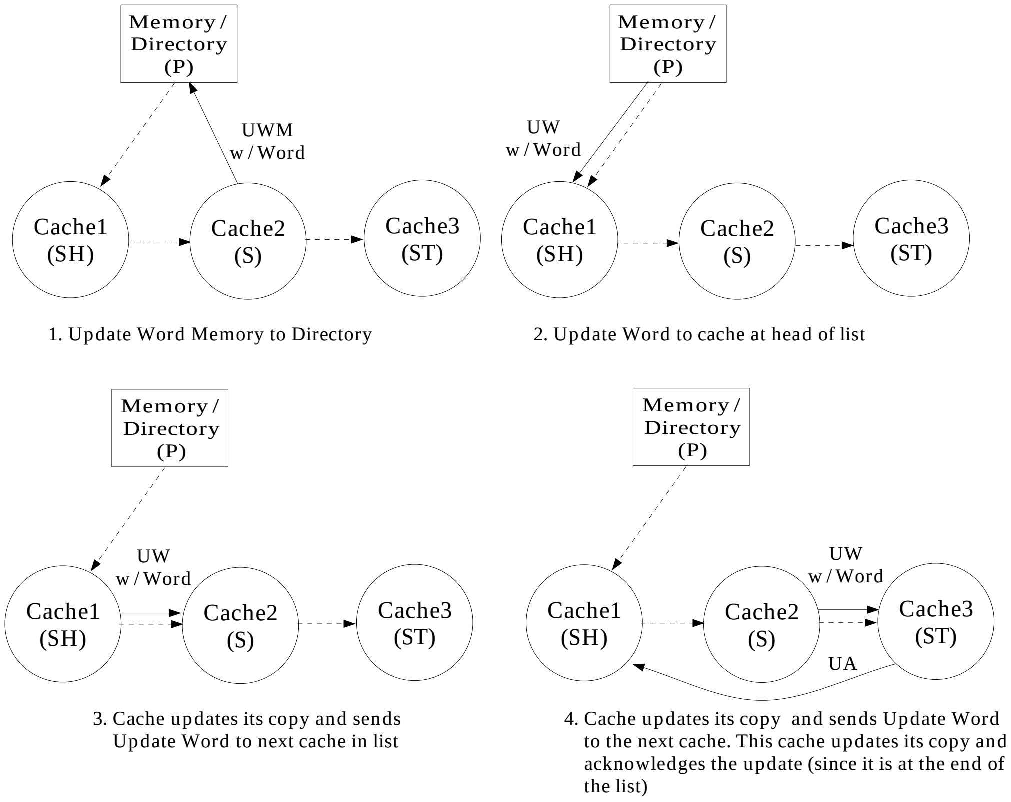
3. Tail cache acknowledges update

Figure 3.29: DD-UP: Write Hit to Cache Line in *Shared Head* State

list, as shown in figure 3.29. This cache updates its copy and forwards the *Update Word* down the list. The cache at the end of the list acknowledges the update.

If the cache is not the head of the list, the write value must be forwarded to the head of the list. To do this, the writing cache sends an *Update Word Memory* (UWM) to the directory, as shown in figure 3.30. The directory forwards it to the head of the list, and an *Update Word* is propagated down the list. The cache at the end of the list acknowledges the update. Note that the writing cache will receive the update request, but must forward the update on to the next cache in the list.

As in the CD-UP protocol, the DD-UP protocol limits the number of outstanding updates per offset to one per cache. When a cache modifies a cache line in the *Shared* or *Shared Tail* states, the pending bit for the offset(PB(O)) is set. When the cache



1. Update Word Memory to Directory

2. Update Word to cache at head of list

3. Cache updates its copy and sends Update Word to next cache in list

4. Cache updates its copy and sends Update Word to the next cache. This cache updates its copy and acknowledges the update (since it is at the end of the list)

Figure 3.30: DD-UP: Write Hit from Cache Line in *Shared* State

receives its own *Update Word*, the bit is cleared. Writes to words in shared lines with the corresponding pending bit set are blocked and updates to these words do not update the cache line, but the update is still forwarded to the next cache in the list. If there are multiple writes to the same word by different processors, each processor may not see all values, but the final value will be consistent. The order that the updates reach the cache at the head of the list determines the total ordering of the writes; the head of the list acts as a write serialization point.

Line Replacement

Line replacement is much more difficult in the DD-UP protocol than in the CD-UP protocol. In the CD-UP protocol, the message path from the directory to each cache is fixed. But in the DD-UP protocol, the path from the directory to a given cache is dependent on the current structure of the directory list for the line. As caches are added and deleted from this list, the structure of the list is altered. These alterations change the path of messages sent between the directory and caches and between caches.

To make line replacement possible without a significantly more complex protocol, a new state, *Replacing*, was added to the possible states of a memory line. If a memory line is in this state, all requests to the line are bounced back to the requester. This new state prevents new messages from attempting to traverse the list while it is being altered. Messages currently traversing the list are flushed out with special flushing messages, which will be described in the next few paragraphs.

For a cache line in the *Exclusive* state, line replacement is identical to the CD-UP protocol. The cache sends a *Replace Line* message to the directory and the directory responds with a *Replace Ack*.

If the cache line is in the *Shared Head* state, the cache sends a *Replace Head* (RH) message to the directory, as shown in figure 3.31. The directory changes the memory line state to *Replacing* (R) and replies to the replacing cache with a *Memory Replace Flush* (MRF). The replacing cache then sends a *Head Replace Flush* (HRF) to the next cache in the list. This cache now becomes the head of the list, or if it is the tail of the list, the state of the cache line state is set to *Exclusive*. This cache sends a *Replace*

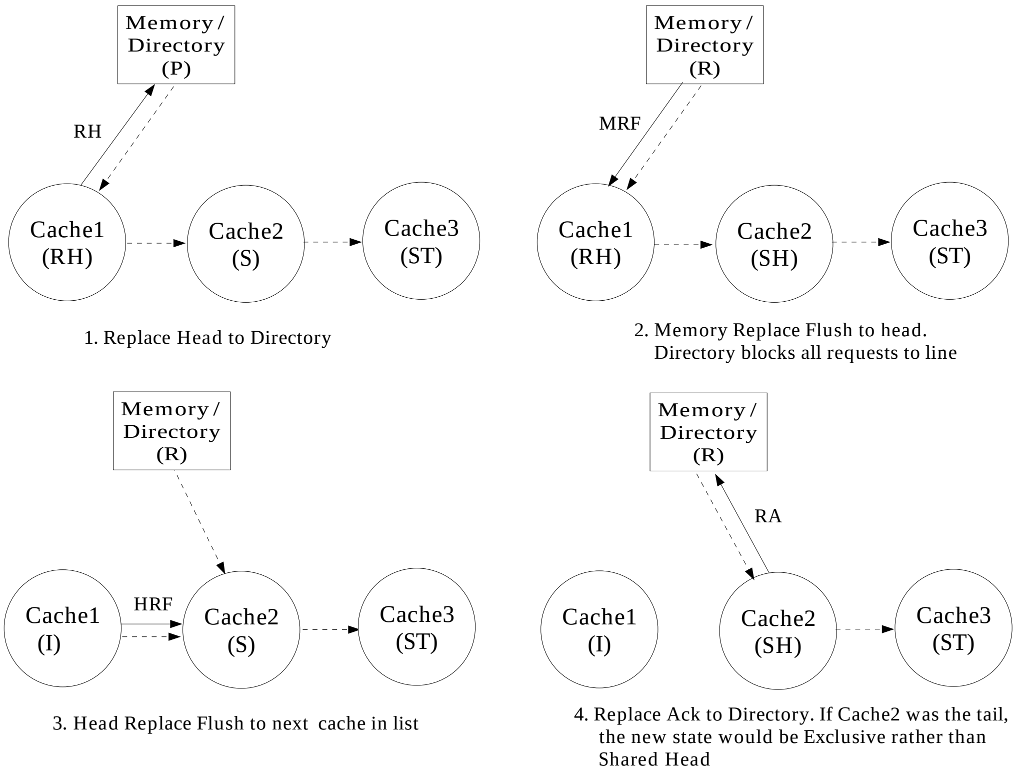

Memory / Directory (P)
RH
Cache1 (RH)
Cache2 (S)
Cache3 (ST)
1. Replace Head to Directory
Memory / Directory (R)
MRF
Cache1 (RH)
Cache2 (SH)
Cache3 (ST)
2. Memory Replace Flush to head. Directory blocks all requests to line
Memory / Directory (R)
HRF
Cache1 (I)
Cache2 (S)
Cache3 (ST)
3. Head Replace Flush to next cache in list
Memory / Directory (R)
RA
Cache1 (I)
Cache2 (SH)
Cache3 (ST)
4. Replace Ack to Directory. If Cache2 was the tail, the new state would be Exclusive rather than Shared Head

Figure 3.31: DD-UP: Replacing Cache Line in *Shared Head* State

Ack (RA) back to the directory. This acknowledgment indicates that the replacement is complete, and the memory line state is set back to *Present*. The *Memory Replace Flush* and the *Head Replace Flush* are used to flush any pending messages that might be traversing the list.

To replace a cache line in the *Shared* state, the cache sends a *Replace* message to the directory, as shown in figure 3.32. The directory sets the memory line state to *Replacing* and forwards the *Replace* to the head of the list. The cache at the head of the list changes its cache line state to *Shared Head Replacing* (SHR). This new state prevents this cache from generating any new updates while the list is being altered. Each cache in turn forwards the *Replace* request down the list. The request specifies the replacing cache and the next cache in the list after the replacing cache. When a cache receives the request, it checks if its directory pointer points to the replacing

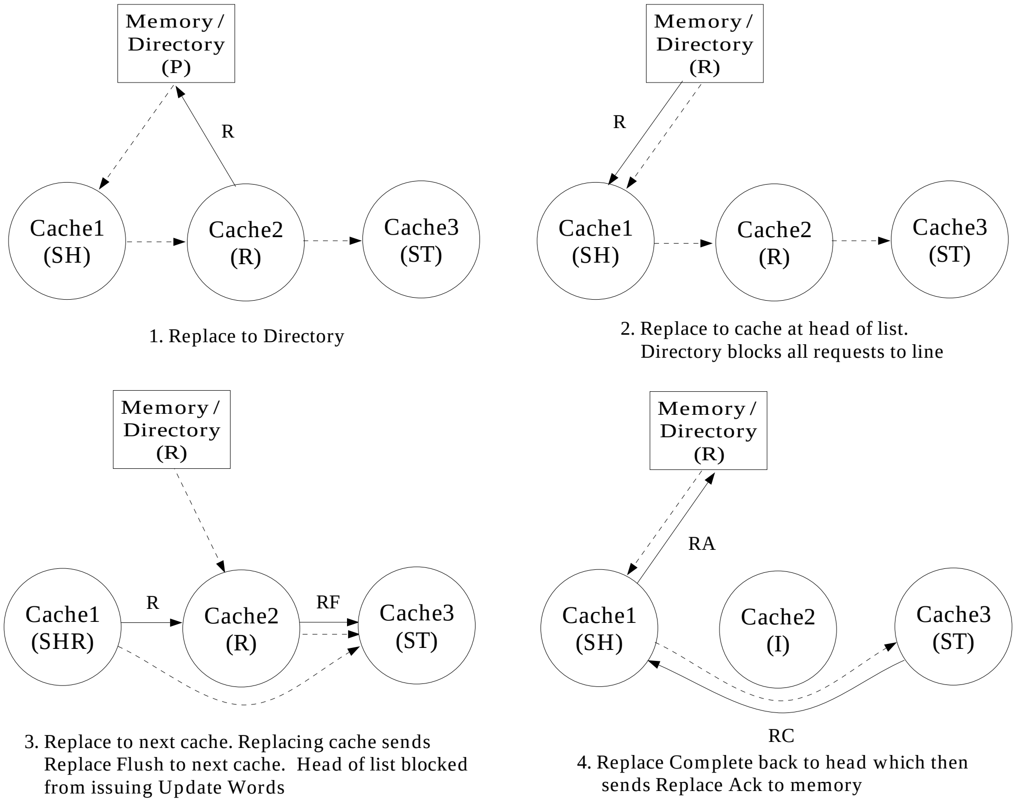
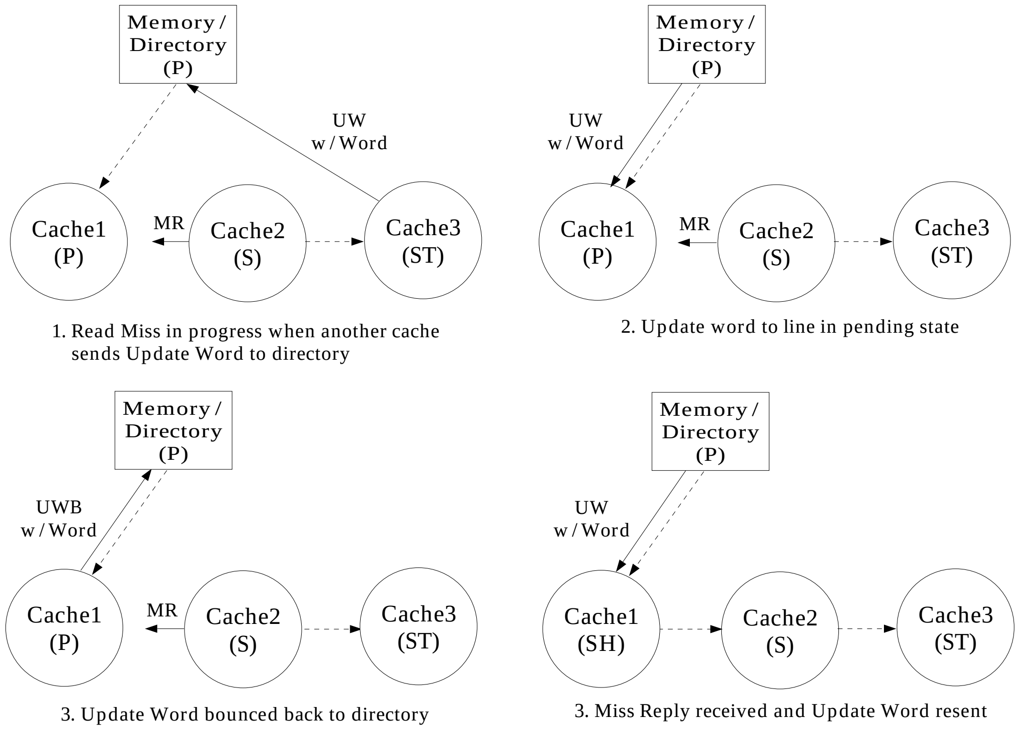

Memory / Directory (P)
R
Cache1 (SH)
Cache2 (R)
Cache3 (ST)
1. Replace to Directory
Memory / Directory (R)
R
Cache1 (SH)
Cache2 (R)
Cache3 (ST)
2. Replace to cache at head of list. Directory blocks all requests to line
Memory / Directory (R)
Cache1 (SHR)
R
Cache2 (R)
RF
Cache3 (ST)
3. Replace to next cache. Replacing cache sends Replace Flush to next cache. Head of list blocked from issuing Update Words
Memory / Directory (R)
RA
Cache1 (SH)
Cache2 (I)
Cache3 (ST)
RC
4. Replace Complete back to head which then sends Replace Ack to memory

Figure 3.32: DD-UP: Replacing Cache Line in *Shared* State

cache. If so, it sets its directory pointer to point to the cache following the replacing cache. When the replacing cache receives its own request, it sends a *Replace Flush* (RF) to the next cache in the list, and sets its cache line state to *Invalid*. The *Replace* and *Replace Flush* messages have flushed out any requests that were flowing down the list. Once the next cache in the list receives the *Replace Flush*, it sends a *Replace Complete* to the head of the list. The cache at the head of the list changes its cache line state back to *Shared Head* and sends a *Replace Ack* to the directory. The memory line state is then changed back to *Present*.

The actions to replace a cache line in the *Shared Tail* state are almost identical to the replacement of a *Shared* line, but since the replacing cache is at the end of the list, it does not need to flush the next section of the list. The replacing cache can send the *Replace Complete* to the cache at the head of the list once it receives its own



1. Read Miss in progress when another cache sends Update Word to directory

2. Update word to line in pending state

3. Update Word bounced back to directory

3. Miss Reply received and Update Word resent

Figure 3.33: DD-UP: Race: Update Word to *Pending* Cache Line

replace request.

Protocol Races

There are two types of protocol races in the DD-UP protocol. The first type is a request to a memory line in the *Replacing* state. In this case, the directory can bounce the request back to the sender to be retried. The second type is an *Update Word* to a cache line in the *Pending* state, as shown in figure 3.33. In this case, the *Update Word* can be bounced back to the directory to be retried. The *Update Word Bounce* still carries the data value, but this does not create an update ordering problem since the update order is determined by the order that the *Update Words* reach the cache at the head of the list. The bounced update has yet to reach the head of the list and, therefore, is not yet an ordered update.

Message	Src	Dst	Description	Type
UW	C	C	Update Word	OD
RM	C	D	Read Miss	OI
WM	C	D	Write Miss	OI
UWM	C	D	Update word to directory	OI
RE	C	D	Replace Exclusive	OI
RH	C	D	Replace Shared-Head	OI
R	C	D	Replace Shared	OI
RT	C	D	Replace Shared-Tail	OI
UW	D	C	Update Word	OI
R	D	C	Replace Shared	OI
RT	D	C	Replace Shared-Tail	OI
MRF	D	C	Memory Replace Flush	OI
BOUNCE	D	C	All Bounced messages	OI
RF	C	C	Replace Flush	OI

Table 3.4: DD-UP: Messages That May Deadlock

Exceptions

As described in section 3.2, the protocol must be able to avoid protocol level deadlock. Table 3.4 shows the request messages that may generate another request in the DD-UP protocol. As with the CD-UP protocol, the order independent (OI) requests can be sent back to the source as an exception without adding any complexity to the protocol.

The *Update Word* message, the only order-dependent message, may change the order of updates flowing down the list. For example, figure 3.34 shows two updates from different processors flowing between two caches. The first update results in an exception and is sent back to the sender. The second update reaches the destination cache and updates the cache line. Once the sending cache receives the excepted update, it must resend it with the *latest* value, which is the value from the second update. The destination cache receives this update and updates its cache line. Now both caches are consistent with the value from the second update, although the second cache never saw the update of the first value.

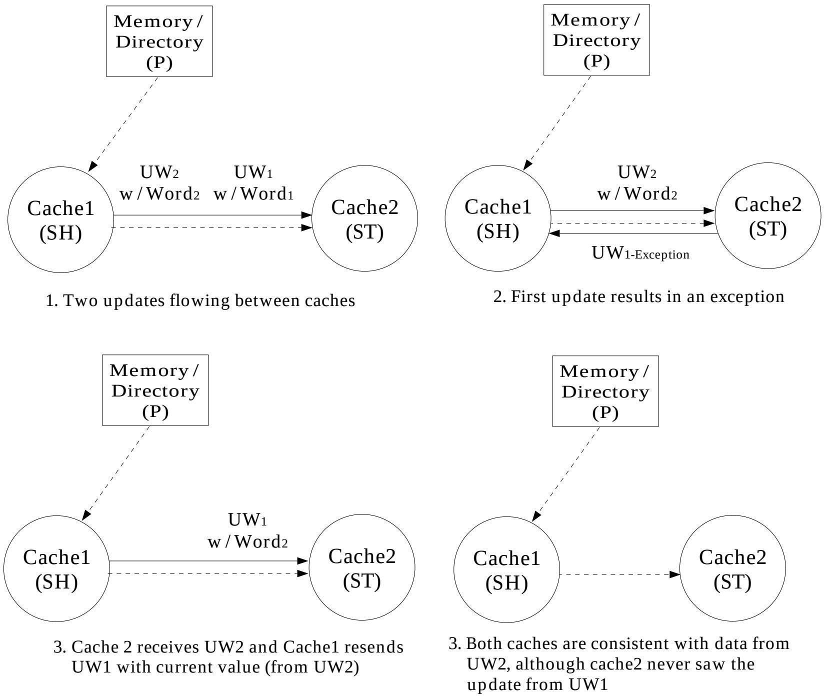

Memory / Directory (P)
Cache1 (SH)
Cache2 (ST)
UW2 w / Word2
UW1 w / Word1
1. Two updates flowing between caches
Memory / Directory (P)
Cache1 (SH)
Cache2 (ST)
UW2 w / Word2
UW1-Exception
2. First update results in an exception
Memory / Directory (P)
Cache1 (SH)
Cache2 (ST)
UW1 w / Word2
3. Cache 2 receives UW2 and Cache1 resends UW1 with current value (from UW2)
Memory / Directory (P)
Cache1 (SH)
Cache2 (ST)
3. Both caches are consistent with data from UW2, although cache2 never saw the update from UW1

Figure 3.34: DD-UP: *Update Word* Exception

3.5 Protocol Verification

An exhaustive verification tool called Murφ [19, 20] was used to verify the update-based protocols. To verify a protocol using Murφ, a description of the system and a behavioral description of the protocol is required. From this, Murφ builds a system state and attempts to traverse it by applying rules from the behavioral description of the protocol. Error statements and invariants are used to detect errors.

For example, figure 3.35 shows the architectural model on which the update-based protocols were verified. The system consists of three caches and one directory/memory. The caches each have one request and one reply buffer, and the directory has four request buffers and one reply buffer. Each network, request and reply, consists of ordered paths between each cache and the directory, and it also includes ordered paths between each pair of caches. The memory consists of a single line with a one-bit data word. The single line is sufficient since protocol actions do not interact between lines. The single, one-bit data word is also sufficient since if there is a case in which a "wrong" value overwrites a "correct" value, then the exhaustive nature of the tool will find the same case for the two values used for the data word.

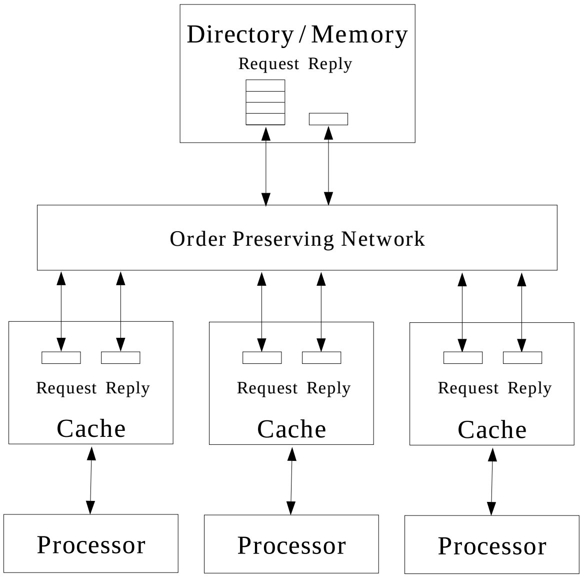


Figure 3.35: Verification Model

The system state created by Murφ consists of a concatenation of all the state bits in the system. This state includes the bits in the cache and memory line data and state information and the message data in the network buffers. In this case, the maximum number of state would be $2^{state-bits}$ states, a significant number of states.

The actual number of states traversed is dependent on the behavioral rules of the protocol. For the update-based protocols examined, this number quickly consumes all the memory available for the verification since Murφ must remember which states have been visited. There are two techniques to reduce the number of states traversed and, therefore, Murφ's memory requirements.

The first technique uses symmetry to eliminate redundant states [43, 44]. Symmetry in a system allows Murφ to find states that are equivalent in their current and future behavior with respect to error checking. During verification, only one member of each equivalence class needs to be examined. This technique is able to significantly reduce the total number of states examined. Using symmetry does not affect the correctness or coverage of the protocol verification.

The second technique limits the number of concurrent actions. Since Murφ is an exhaustive verification tool, every possible combination of events must be verified. Therefore, the more active events, the larger number of traversed states. In the verification of the update-based protocols, the number of outstanding updates was limited to one per cache word since the actual protocol limits the number of outstanding updates per word to one.

Overall, the verification tool was useful in verifying the correctness of the protocols. Errors were detected very quickly, but the state explosion problem limited the size and scope of the verification. As discussed above, the only limitation of the verification that might affect correctness was the limited number of outstanding updates, but the combination of Murφ verification, running simulation with the update-based protocols and hand verification have produced correct update-based protocols with a high confidence level.

3.6 Summary

In this chapter, the actions required for both invalidate-based and update-based cache coherence protocols were discussed. Table 3.5 summarizes the important characteristics of the protocols. The invalidate-based protocols require a writing cache to obtain exclusive ownership of a line before it may be modified, but the update-based protocols allow multiple readers and writers of any given memory line. The centralized directory protocols can use the structure of the directory to send invalidations or updates in parallel. The distributed directory protocols were forced to send the invalidate and updates down the list of cache; the longer the list, the longer the latency of the operation. The centralized directory protocols updated memory[3]. The update of memory allows subsequent misses to be serviced by the directory. The distributed directory protocols do not update memory and, therefore, the data must always be fetched from the cache at the head of the list.

Protocol	Invalidates/Updates	Memory Update	Source of Miss Data
CD-UP	Parallel Updates	Always	Memory or Cache
DD-UP	Pipelined Updates	Never	Cache
CD-INV	Parallel Invalidates	For read miss	Memory or Cache
SCI-INV	Sequential Invalidates	Never	Cache
DD-INV	Pipelined Invalidates	Never	Cache

Table 3.5: Protocol Summary

The remainder of this dissertation will focus on the performance differences between these five protocols.

[3] All protocols update memory if a cache replaces the last cached copy of a memory line and memory's copy is not consistent.

Chapter 4

Simulation Methodology

In order to compare the performance of the various cache coherence protocols, a simulation environment must be built. This environment must include a multiprocessor architecture model, a set of applications and the underlying event simulator. This chapter describes these components in detail. Section 4.1 describes the architectural models used, and section 4.2 describes the simulation environment. Finally, section 4.3 describes the applications simulated.

4.1 System Architecture

The simulated architecture consists of a 64 node shared-memory multiprocessor. The nodes arranged in an 8 by 8 mesh as shown in figure 4.1. Each node consists of a processor and memory element (PME) connected to its four nearest neighbors through a set of network queues. Although the figure shows only one network, the nodes are actually connected together by both a reply and a request network to avoid protocol level deadlock.

The PME consists of a processor, cache, memory, cache coherence directory and an interface to the network, as shown in figure 4.2. The elements of a PME are connected together through a reply and request bus. A separate reply and request bus and network are required to avoid protocol deadlock, as described in section 3.2. The reply network can also be used as the exception network.

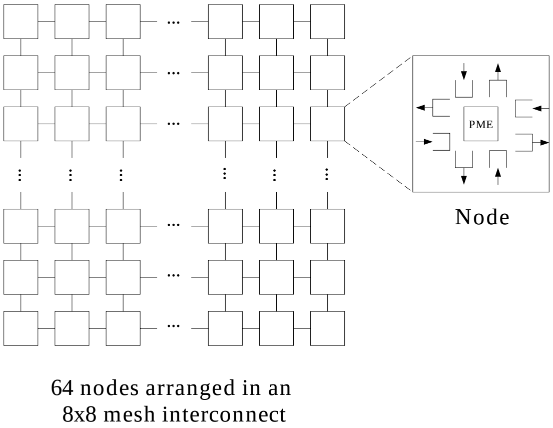


Figure 4.1: Network Topology

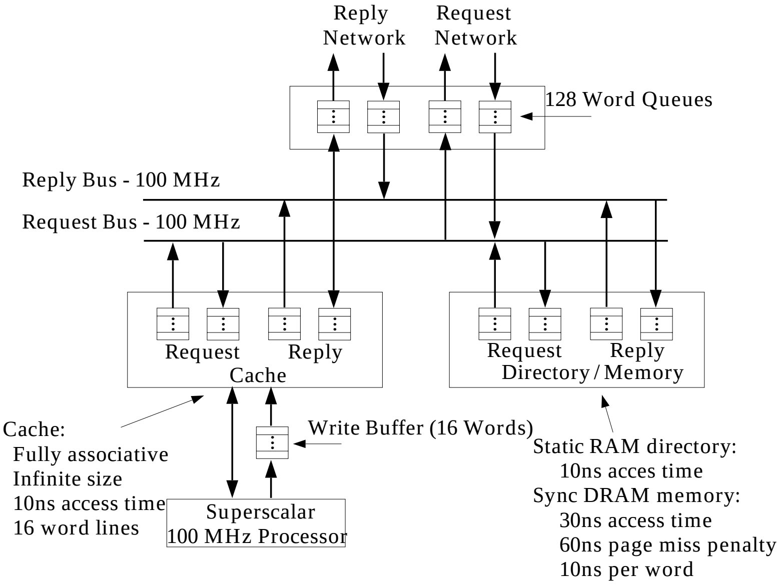


Figure 4.2: Processor/Memory Element

4.1.1 Processor Model

The processor is a 100 MHz superscalar processor that is assumed to be load/store limited. The load/store limited model assumes that non-load/store instructions can be executed in parallel with the load and store instructions. This assumption was validated by compiling the inner loops of the applications studied on an Alpha superscalar processor. The resulting cycle count did indicate that the processor tended to be load/store limited.

4.1.2 Write Buffer

A write buffer is used to buffer writes from the processor to the cache. The write buffer is 16 words deep and can accept a new write every cycle. The latency through the write buffer is one cycle.

4.1.3 Cache Model

The cache is lockup-free [61], and it is fully associative with infinite size. An infinitely sized cache is used to separate the effects of a limited cache size and the actions required by the cache coherence protocols. The cache has a single cycle access time and a line size of 16 words. The cache controller uses a fixed priority scheme with processor read requests having the highest priority followed by network requests and replies, and then write buffer requests.

4.1.4 Directory/Memory Model

Each directory/memory module consists of a single bank of 100 MHz synchronous DRAMs supporting page mode operation. The SDRAMs have 30 ns access time for a page access with a page miss penalty of an additional 60 ns. The directory consists of a 10 ns access time SRAM for all protocols.

4.1.5 Bus Model

The buses are a word wide and operate at 100 MHz. The bus arbitration uses a fixed priority scheme for the three sources of requests for the bus: the cache, the directory/memory and the network interface. The cache has the highest priority followed by the directory/memory and then the network interface. The bus arbitration requires one cycle.

4.1.6 Network Model

Each network link has a bandwidth of 400MB/s and the network latency through a node is 8 cycles. The network queues are 8 and 20 words deep for the input and output queues respectively. The output queue must be able to hold the largest possible network packet (20 words) so that the multicast scheme presented at the end of this section may be implemented.

The network is order preserving with static, wormhole routing [16] and multicast. The network routing function requires one cycle. Multicast is only used by the centralized directory protocols when there are multiple caches to update or invalidate. Multicast is used to send identical packets to multiple targets. It is not a full broadcast.

Topology

As noted in the beginning of this section, a 2 dimensional mesh topology was used. There has been significant work in the study of cache coherence protocols on other types of network topologies. These include rings [10], hierarchy of buses [73], multistage interconnect networks [56, 5, 11]. A 2 dimensional mesh was chosen because it offers the least message latency when interconnect locality and bisection bandwidth are taken into account [15, 2]. Both the work by Thapar [69] and the DASH project [53] also assume a 2 dimensional mesh.

Multicast

The multicast scheme is very simple. Each multicast packet includes an extra two words of header (64 bits) which are used as a bit vector to specify the targets of the multicast packet. Each bit represents a target cache in the system. The routers convert the bit vector into a list of destination ports of the router. When a message has multiple output destination ports, it is only allowed to proceed if the entire packet can be sent to each destination queue. Thus, the destination queues must be large enough to hold the largest possible packet (20 words).

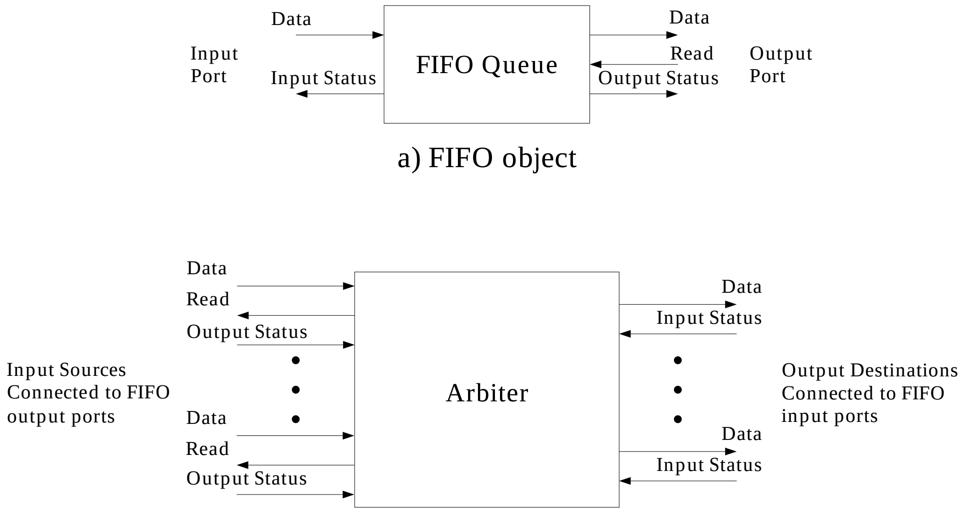
With this restriction, multicast aborts are not required since a packet is never sent down a network channel unless the entire packet can be successfully sent down all channels; the status signals from the queues can be used to determine if there is enough room in the queues to hold the packet. Deadlock can be avoided since the multicast packet does not acquire any output channels until all channels are available. These restrictions may result in increased latency for multicast packets, but they significantly reduce the complexity of the multicast mechanism.

4.2 Simulation Environment

The Simple simulation environment [18, 60] was used to simulate the multiprocessor architecture running the applications that will be described in section 4.3. Simple is a lisp-based, object-oriented simulation environment.

4.2.1 Basic Objects

In this environment, each simulated object consists of a behavior description and a set of ports. The objects communicate by sending messages across the port connections. For example, figure 4.3 shows the two basic building block used to build the multiprocessor models: a first-in, first-out (FIFO) queue object and an arbiter object.



a) FIFO object

b) Arbiter object

Figure 4.3: Basic Simulation Objects

FIFO

The FIFO queues are used to connect arbiters that perform actions on the data flowing through the system. The FIFO is a passive object because it only responds to requests from arbiters. An arbiter may send data to a FIFO though its input port or it may request data from a FIFO through its output port. The status ports indicate the number of words in the FIFO. These status signals can be used to determine if the FIFO is full or empty.

The behavior of a FIFO queue object is very simple. If the FIFO object receives a data signal on the data input port, the data is inserted into the FIFO queue and the status ports are updated. If the FIFO receives a request signal on the read port, the next word in the FIFO is placed on the output data port and the status ports are updated. The basic FIFO can send and receive data at one word per cycle, and the minimum latency through the FIFO is two cycles.

Arbiter

The behavior of an arbiter is more complicated and consists of the following steps:

1. Select input source FIFO
2. Read packet from output port of selected input FIFO
3. Process packet - Specialize this step to perform desired work
4. For each output packet generated
    (a) Determine output destination(s)
    (b) If input port of FIFO(s) is available, send data if not wait

The arbiter object is then specialized to perform the desired actions. For example, to specialize an arbiter object to act as a cache, step 3, the process packet step, is specialized to perform the following steps:

1. Read cache line state for specified memory line
2. Process request for cache line in current state
    (a) Access cache line data and update cache line state in parallel
    (b) Generate any resulting protocol messages

The second step is then specialized for each cache coherence protocol.

The protocol specifications consist of a set of actions for each cache line state for all possible protocols messages. For example, figure 4.4 shows the protocol specification for an update message to a cache line in the *Shared* state for the CD-UP protocol described in section 3.4.1

4.2.2 Node Object

Figure 4.5 shows how a node is constructed from the two basic objects. The cache and directory/memory objects are arbiter objects specialized to perform the desired protocols. The network router and bus are also arbiter objects specialized to perform the desired routing function, and the network object is an arbiter specialized to connect two nodes. The figure only shows one network, but the actual model includes two network. The two networks, a reply and request network, are required to avoid protocol level deadlock, as described in sections 3.2.

```
;;
;: SHARED cache line State
;;
(defmethod (CD-UP-CACHE :SHARED) (Cache-Line Operation Packet)

     (case Operation
         ;;
         ;; Request was a :UPDATE-WORD
         ;;
         (:UPDATE-WORD

            ;; Update cache line data with new word
            (WRITE-HIT Cache-Line offset data)

            ;; Increment current simulation time (in cycles)
            (WRITE-WORD-EVENTS 1)

            ;; Acknowledge Update
            (SET-OPERATION Packet :UPDATE-ACK)
            ;; Destination node was Source of original Write
            (SET-DESTINATION Packet (PACKET-SOURCE packet))
            ;; Set Packet Size
            (SET-WORD-SIZE Packet ***Header-Size***)
            ;; Queue Packet to be sent no earlier than current time
            (SEND-COHERENCE-SIGNAL packet))
```

Figure 4.4: Protocol Source Code Example

The latencies through a network object, network router, bus router and FIFOs are all two cycles. Therefore, the latency from network input to network output is eight cycles, and the latency from the local bus to the network output is also eight cycles.

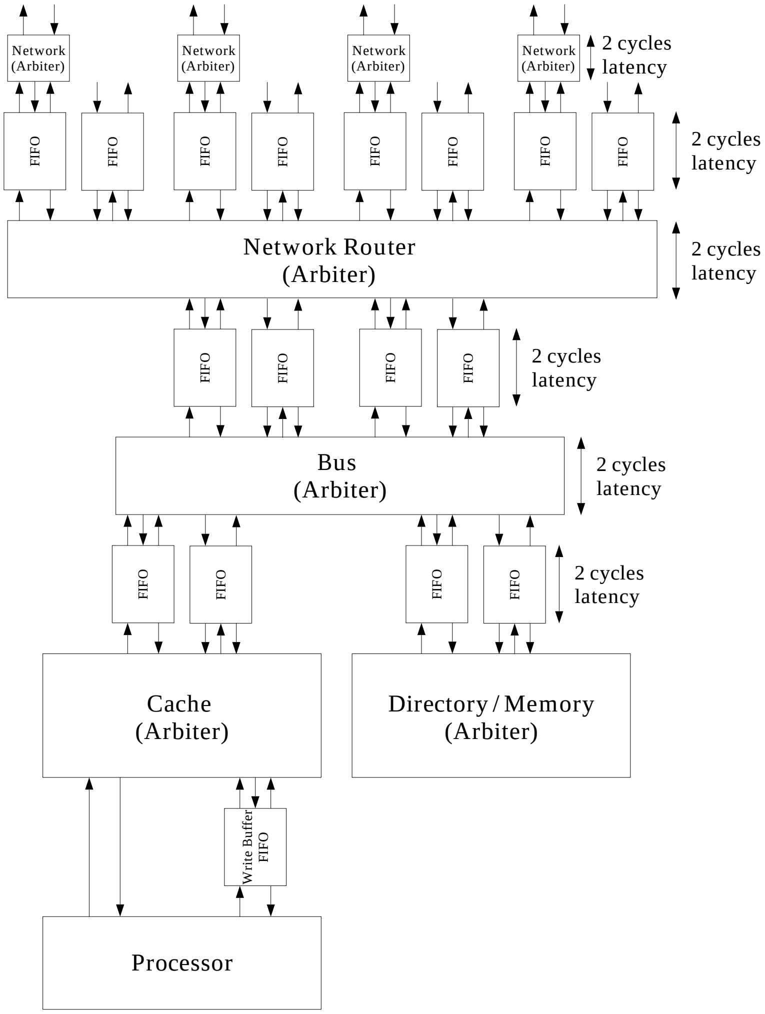


Figure 4.5: Simulated PME

4.3 Scientific Application Domain

To compare the performance of the cache coherence protocols, a set of applications must be specified which represents an important domain for large scale shared-memory multiprocessors. One such domain is the scientific and engineering domain. The applications studied here include a simple, iterative partial differential equation solver (PDE), a 3-D iterative partial differential equation solver using FFTs (3DFFT), and three different methods of factorizing a matrix into triangular matrices: a multifrontal solver (MF), sparse Cholesky factorization (SPCF), and LU decomposition. Appendix A presents the details of the scientific applications. Two migratory applications (MP3D and TASK) are also examined, and appendix B presents the details of these applications.

The applications can be classified along the two dimensions as shown in figure 4.6. The y-axis is the number of consumers for each data block. This gives a measure of general contention for each object and the maximum number of invalidates or updates that might be needed when the data is modified. The x-axis is the line utilization. The line utilization is the percentage of each memory line that is modified by the producer. For example, if the data is a dense vector, the producer is likely to modify all the words in the given line. This would result in a line utilization of 100%. If the data is a structure, the producer might only modify a few words, which would result in a low line utilization.

These measures give a good prediction of the performance of the cache coherence protocols. The line utilization determines the efficiency of the single word updates, and the average number of consumers will determine the latency of updates and invalidates for the different protocols. The distributed directory protocols will be more sensitive to this parameter since the latency of updates and invalidates is dependent on the number of shared copies of the line.

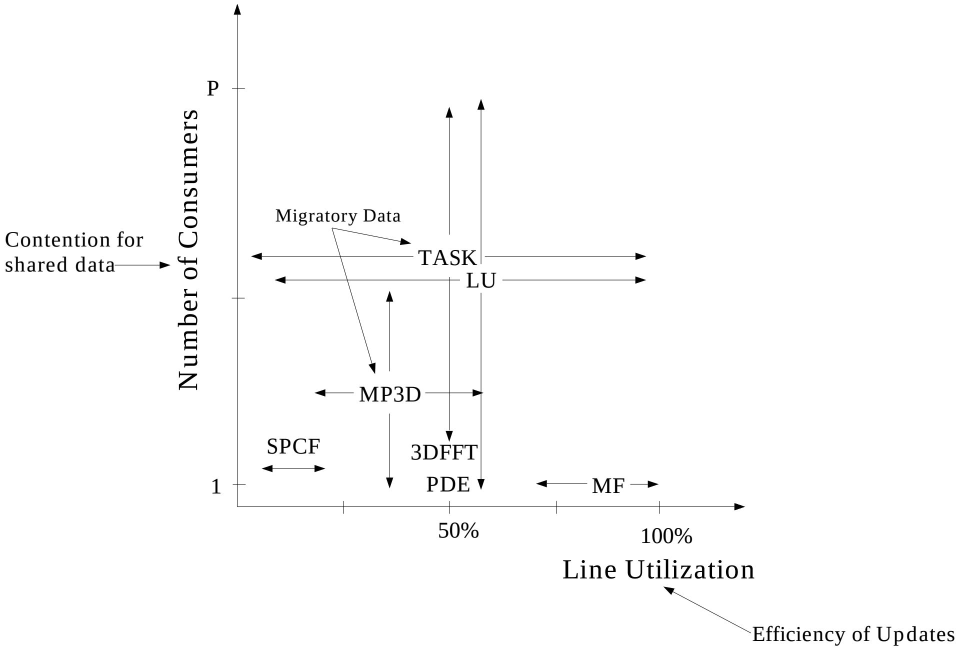


Figure 4.6: Application Domain

Chapters 5 and 6 discuss the performance of the protocols running the 5 scientific applications, and section 7.2 describes the performance of the protocols for applications with migratory data.

Chapter 5

Enhancements to Update-Based Protocols

This chapter motivates the need for two techniques to enhance the performance of the update-based protocols. The first enhancement, a word synchronization scheme, attempts to match the granularity of the data synchronization with the fine-grain data updates. The second enhancement, a write grouping scheme, attempts to groups write updates to improve the efficiency of the updates. Section 5.1 describes word synchronization, and section 5.2 describes write grouping.

5.1 Word Synchronization

This section examines the interaction between data synchronization and cache coherence protocols. First, block and word synchronization schemes are described. Next, the actions required by each class of cache coherence protocols for both synchronization schemes are examined. Finally, the simulation results are presented which demonstrate that word synchronization always improves the performance of the five scientific applications compared to the block synchronization case when update-based protocols are used, but that the results vary for invalidate-based protocols.

5.1.1 Block Synchronization

In shared-memory multiprocessors, data is often shared between a producer and one or more consumers. To prevent the consumers from using stale or incorrect data, the consumers must not access the data until the producer notifies them that it is available. Typically, such systems use a block synchronization scheme to synchronize this production and consumption of data.

In such schemes, simple flags can be used to indicate that a given block of data has been produced. For example, figure 5.1 shows a timing line for an exchange of data using a block synchronization scheme. First, the producer creates the data and writes it to a shared buffer. Next, the producer issues a fence instruction that stalls the processor until all writes have been performed. Finally, the producer sets the synchronization flag. The consumers, who have been waiting for the synchronization flag to be set, see the flag set and begin consuming the data. Block synchronization schemes may also use other synchronization methods such as barriers. The block synchronization scheme examined in this dissertation assumes a release consistency memory model [33], as was described in section 2.1.4.

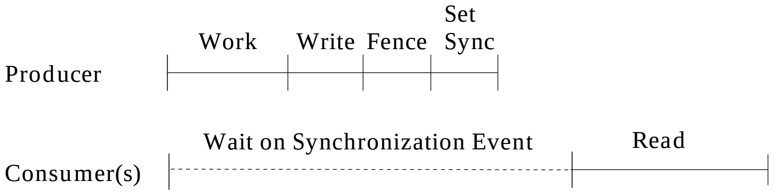


Figure 5.1: Block Synchronization

A block synchronization scheme has two basic disadvantages. First, the scheme requires an expensive synchronization operation such as a flag or barrier to synchronize the production and consumption of data. Second, the consumers are forced to wait until the entire data block has been produced before they are able to begin consuming the data. Both disadvantages are related to the block size of the data being produced. If the block size is increased, the relative overhead, or cost, of the synchronization event is reduced. But as the block size increases, the delay before consumers can begin consuming the data also increases.

5.1.2 Word Synchronization

An alternative to block synchronization is a word-based synchronization scheme. In such a scheme, the synchronization information for each data word is combined with the word. For example, figure 5.2 shows how the timing line for the producer and consumer interaction would be simplified. In this case, the producer creates the data and writes it to a shared buffer. The consumers wait for the desired word to become available and then consume it.

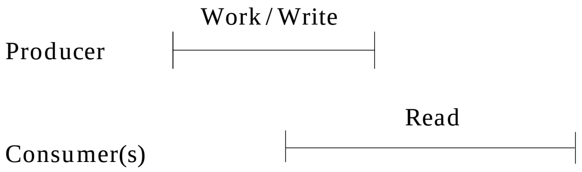


Figure 5.2: Word Synchronization

Word synchronization has several advantages. First, the consumers are able to consume each word as soon as it is available. This early consumption allows the consumption time of the available data to overlap the production time of subsequent words. Moreover, no expensive synchronization operation is required; thus, the producer never needs to wait for the writes to be performed.

There have been several systems designed with word synchronization. These include but are not limited to Alewife [48], HEP [65], Tera [6] and MDP [17]. This dissertation is the first to examine the performance of word synchronization on both update-based and invalidate-based cache coherent multiprocessors. The work on the Alewife system examines the implementation details of a word synchronization scheme, and their work gives an excellent description of the software requirements for such a scheme. In their work, they demonstrate that a word synchronization scheme may improve the performance of the given applications running on an invalidate-based cache coherent system. This dissertation does not contradict their findings, but rather expands them to demonstrate the instability of the combination of invalidate-based cache coherence protocols and word synchronization.

5.1.3 Performance of Block Synchronization

This section reviews the performance of block synchronization schemes for both invalidate-based and update-based cache coherent systems. As briefly mentioned in the last section, block synchronization uses a synchronization event, such as a flag, to synchronize the production and consumption of data.

Figure 5.3 shows the type of diagram that will be used to demonstrate the work required for the purpose of sharing a single data line for the two synchronization schemes and the two different classes of cache coherence protocols: invalidate-based and update-based. The horizontal lines indicate required network transactions. These transactions may go to the cache coherence directory, the memory or other caches while traveling between the producer and consumer. The diagram shows only one consumer, but there may be many. The vertical axis is time.

Invalidate-Based Protocols

Figure 5.3 shows the actions required for invalidate-based protocols using a block synchronization scheme. After producing the data, the producer writes it to a shared buffer and waits for the writes to be performed. The writes are considered performed when the producer's cache has obtained exclusive ownership of the line (all necessary invalidations have been performed) and the data has been written into the cache. The figure assumes that the producer has already obtained exclusive ownership of the data lines through write prefetching. After the writes have been performed, the producer sets the synchronization flag. If the consumers have already read the flag, then this write must invalidate the consumers' copies of the flag. The consumers, who are still waiting for the flag to be set, will immediately reread the flag after it is invalidated, but the producer cannot release the flag's line until all the invalidations have been performed. Once the consumers see the flag set, they can begin reading and consuming the data.

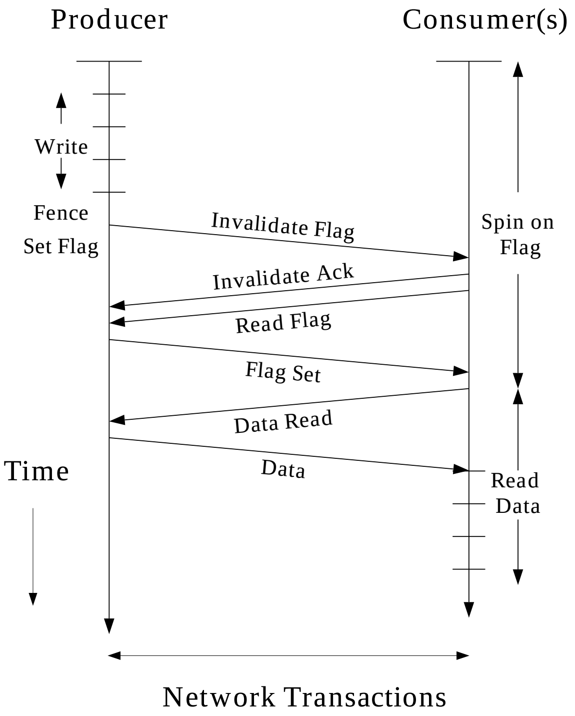


Figure 5.3: Block Synchronization and Invalidate-Based Protocols

Figure 5.3 illustrates the cost of a block synchronization scheme using a simple flag. The invalidation and reread of the flag by each consumer requires four network transactions and the transfer of a line of data. The work to transfer the synchronization information is more than the work to transfer one line of data. The size of the data block could be increased to reduce this synchronization overhead, but the larger block size also increases the waiting time of the consumers.

Update-Based Protocols

Figure 5.4a shows the actions required for update-based protocols. In this case, all of the consumers have prefetched the synchronization flag and data block before the data is written. When the producer writes the data, the consumers' caches are updated. The producer must wait for the writes to be performed (all updates acknowledged) before setting the synchronization flag, and this write of the flag will result in the consumers' caches being updated. When consumers see the flag updated, they can

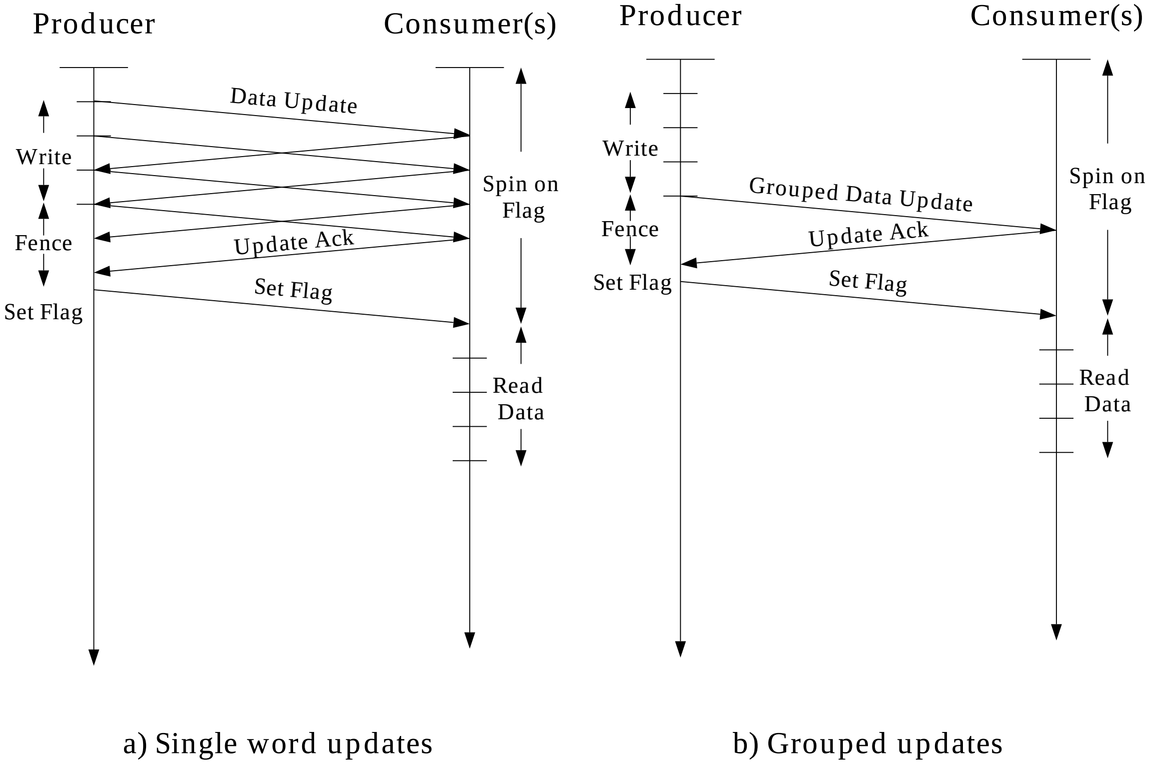


Figure 5.4: Block Synchronization and Update-Based Protocols

begin using the data, which is already in their caches. Figure 5.4b shows the resulting network transactions if a write-grouping scheme, as examined in the next section, is used. In this case, the data updates are grouped into a larger, more efficient update packet.

As in the invalidate-based case, the block synchronization scheme has disadvantages. First, the cost of the block synchronization operation is high. But the cost is not in transferring the synchronization information itself, it is in waiting for the updates to be performed (acknowledged). This synchronization scheme also forces the consumers to wait until the synchronization point is reached before accessing the data, but, unlike the invalidate-based case, the desired data is already in the consumers' caches as a result of the earlier updates. The block synchronization scheme does not allow the system to take advantage of these single word data updates.

5.1.4 Performance of Word Synchronization

A word synchronization scheme can overcome many of the disadvantages of block synchronization by combining the synchronization information with the data. Such a scheme may be implemented in either hardware or software. In a hardware-based scheme, a full/empty bit is associated with each memory word [65]. The full/empty bit is set to full when the data word is written, and the word may be read only after the bit is set to full. Alternatively, a software-based scheme may be used in which an invalid code, such as NaN in a floating point application, is used to indicate an empty word.

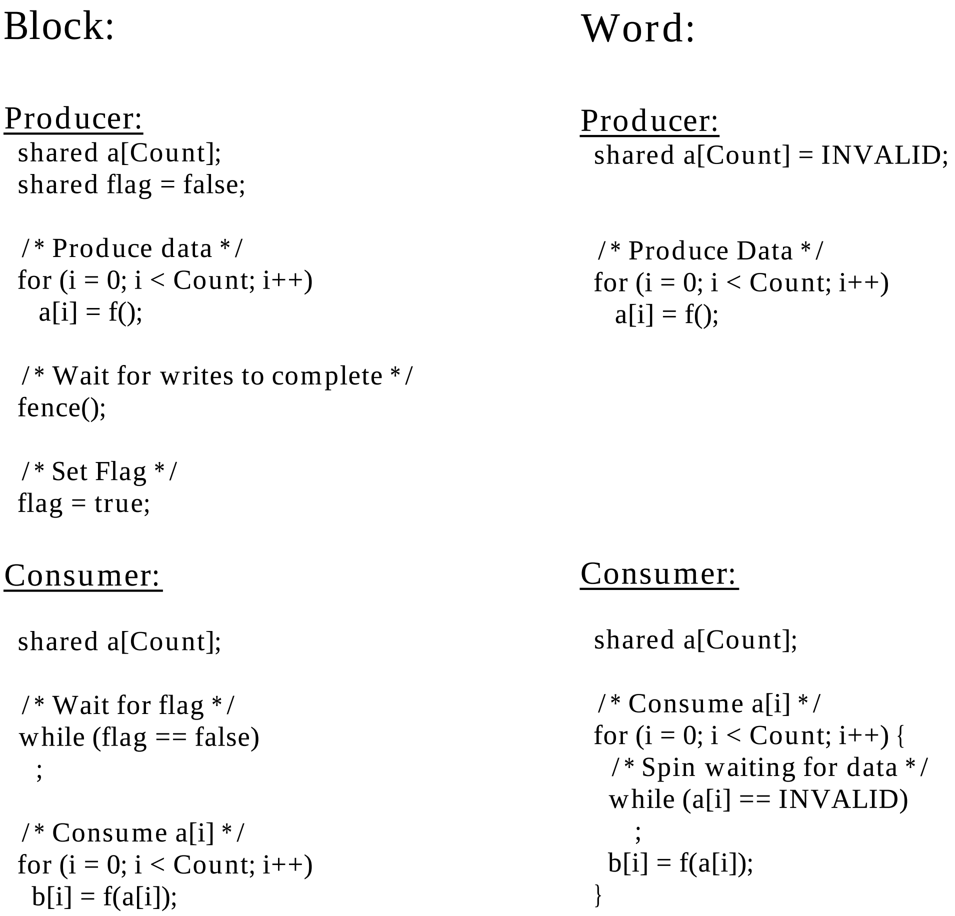
Figure 5.5 demonstrates how a producer and consumer interaction might be coded for both block and word (using a software-based scheme) data synchronization. For the block case, a simple flag, initialized to false, is used to synchronize the production and consumption of data. For the word synchronization case, the data is initialized to an invalid code. The consumer waits for each word to become valid and then consume it. For iterative applications, the data must be set to an invalid code between iterations.

There is little performance difference between hardware-based and software-based word synchronization schemes. The main difference is in the clearing (or emptying) of blocks of data. A hardware-based scheme may have facilities that allow a block to be cleared in a single operation. This facility would allow the hardware-based scheme to clear data in the initialization phase more quickly or, what is more important, clear data between the iterations of iterative applications. Currently, all the applications under study use a software-based scheme.

In the remainder of this section, the actions required by invalidate-based and update-based protocols to implement a word synchronization scheme are described.

Invalidate-Based Protocols

By their very nature, invalidate-based protocols are not well matched to word synchronization schemes. The protocols do not allow consumers to maintain copies of a data line while a producer writes to the line. This results in a very unstable solution.



Block:

Producer:

```
shared a[Count];
shared flag = false;

/* Produce data */
for (i = 0; i < Count; i++)
  a[i] = f();

/* Wait for writes to complete */
fence();

/* Set Flag */
flag = true;
```

Consumer:

```
shared a[Count];

/* Wait for flag */
while (flag == false)
  ;

/* Consume a[i] */
for (i = 0; i < Count; i++)
 b[i] = f(a[i]);
```

Word:

Producer:

```
shared a[Count] = INVALID;


/* Produce Data */
for (i = 0; i < Count; i++)
  a[i] = f();
```

Consumer:

```
shared a[Count];

/* Consume a[i] */
for (i = 0; i < Count; i++) {
  /* Spin waiting for data */
  while (a[i] == INVALID)
     ;
  b[i] = f(a[i]);
}
```

Figure 5.5: Code Examples for Block and Word Synchronization

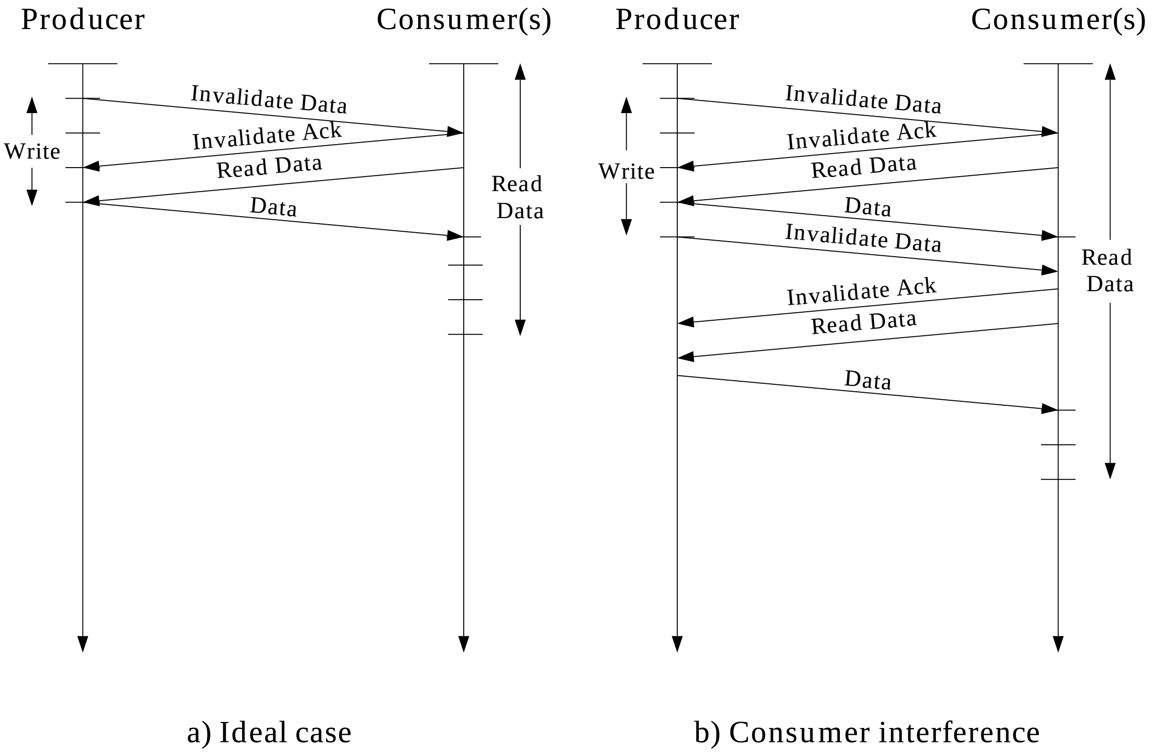


Figure 5.6: Word Synchronization and Invalidate-Based Protocols

Figure 5.6a shows the ideal timing diagram for invalidate-based protocols when a word synchronization scheme is used. First, all the consumers read the initial word of the data block. When the producer writes this word, all the consumers' copies must be invalidated. But since the consumers are eagerly waiting for the data, they will immediately reread the data line once it is invalidated. The invalidate-based protocols studied in this dissertation allow the producer to continue writing into the cache line while invalidations are pending. These non-blocking writes prevents the producer from observing any write delay as the consumers read and reread the line. In the ideal case, the invalidation latency is greater than the producer's write time for the line. In this case, when consumers reread the line, they will find the line completely written. The producer will not invalidate the line again; this gives consumers all the time they need to consume the line's data for this ideal case.

However, figure 5.6b demonstrates the problem with invalidate-based protocols and word synchronization schemes. In this case, consumers have reread the line after the initial invalidation but before the producer has completed writing the data. Now the producer is required to invalidate the consumers' copies of the line again, and the consumers are forced to reread the line. The producer is able to release the line again only after all the pending invalidations have been performed.

The relative timing of the writes and reads will have an enormous impact on the performance of the system. If the writes occur in bursts, as they often do, the producer will usually be able to produce many words of data between each reread by the consumers. But the consumers, who will reread the line immediately after it is invalidated, will only receive the data after all the consumers' copies of the line have been invalidated. The consumers will then be able to consume data only until the producer invalidates the line again, which may occur soon after the consumer receives the data if the write rate is high.

The frequency with which these invalidations and rereads occur depends on two characteristics of the applications. First, the probability that the producer finishes writing the line before the consumers attempt to reread it depends on the number of words written to the line. This measure, known as the line utilization, was used in section 4.3 to classify the application space. The other characteristic, also used in section 4.3, is the number of consumers. The more consumers, the higher the probability that consumers will interfere with the producer's writing of the data. The performance results discussed in section 5.1.6 will demonstrate the impact of these two application characteristics on the performance of the invalidate-based protocols when word synchronization is used.

As described in chapter 2, data prefetch can be used in block synchronization schemes to hide the latency of data accesses [55]. But in a word synchronization scheme, prefetching may result in many more invalidations. The prefetched line itself may be invalidated before it can be used, or the prefetched line may contain valid data that may be consumed. The relative timing of the prefetch will determine which of these two cases actually occurs. The performance gains of prefetching will be examined in the section 5.1.5.

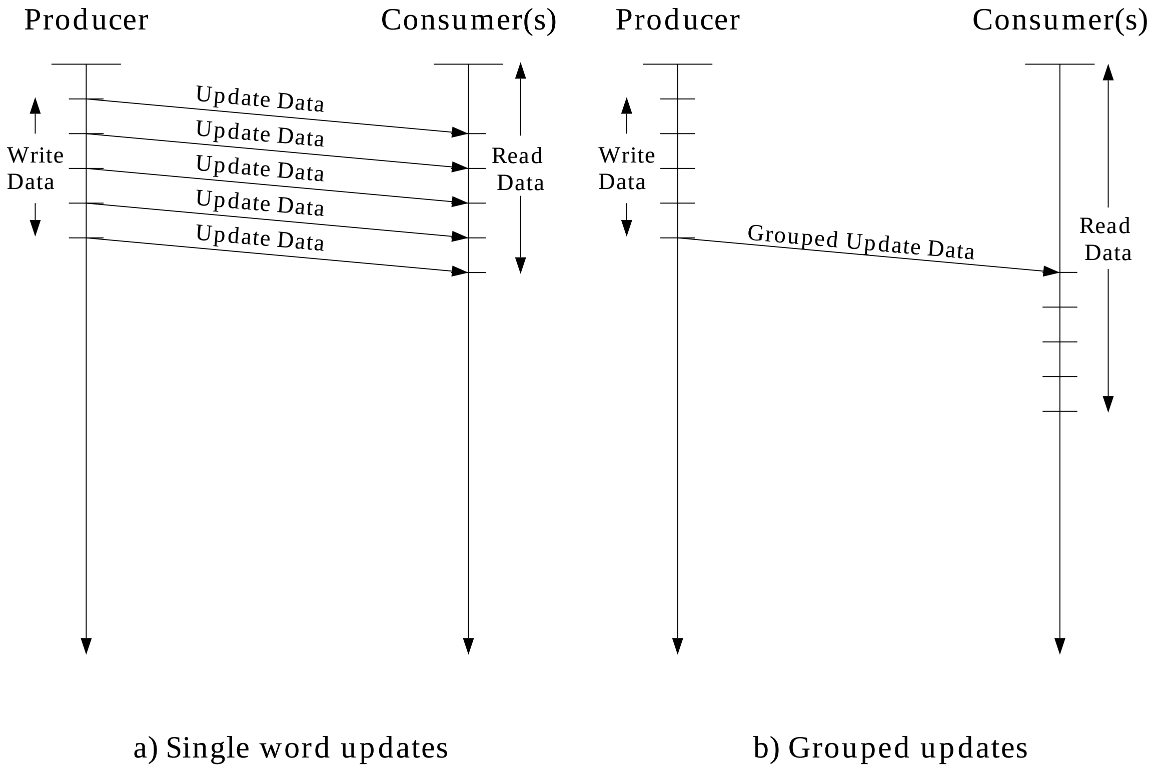


Figure 5.7: Word Synchronization and Update-Based Protocols

Update-Based Protocols

The definition of update-based protocols offer a better match to word synchronization schemes. Unlike the invalidate-based protocols, update-based protocols allow consumers to maintain a copy of the data line while a producer writes to the line. Also, the producer's write of the data results in an update of all the consumers' caches: a proactive distribution of data rather than a reactive approach as in the invalidate-based protocols in which each consumer is responsible for refetching an invalidated line.

For example, figure 5.7a shows the actions required for update-based protocols using a word synchronization scheme. First, all the consumers prefetch the desired data lines. The prefetch allows the consumers to express an early interest in the data so that when the data is written by the producer, the consumers' caches will be updated. As the producer writes the data, the consumers can consume the data as soon as it arrives. Figure 5.7b shows the required network transactions when a write grouping scheme is used.

Update-based protocols can also take advantage of word synchronization in another way. Update-based protocols using a block synchronization scheme require that updates be acknowledged before the synchronization flag is set, but in a word synchronization scheme, no update acknowledgment is needed because the producer is never required to wait for the updates to be performed. The elimination of the acknowledgments has the largest impact on the distributed-directory update-based (DD-UP) protocol described in chapter 3. In this protocol, the update acknowledgment latency is dependent on the length of the list of caches with a copy of a given line. Eliminating the requirement for update acknowledgments reduces the impact of the length of this list on the performance of the DD-UP protocol.

The update-based protocols using write grouping offer a much more robust solution when word synchronization is used compared to the combination of word synchronization and invalidate-based protocols. Data prefetch cannot degrade the performance of the word synchronization scheme as with the invalidate-based protocols. The prefetch can be issued as early as desired by the consumers. When the producer writes the data, the consumers' caches are updated. The inefficiency of these updates is addressed by the write grouping scheme described in section 5.2. Also, the relative timing of the producer's writes and consumers' reads cannot affect performance as it may when invalidate-based protocols are used. *The amount of work is fixed regardless of any variations in the timing of reads or writes.*

5.1.5 Performance Impact of Data Prefetching

As described in chapter 2, software-controlled data prefetch has been shown to be effective in hiding memory access latencies in applications using block synchronization [55]. In this section, the performance impact of prefetching on the applications using word synchronization is examined. Figure 5.8 shows the relative execution times of the applications using data prefetch compared to the non-prefetching case for the applications using word synchronization.

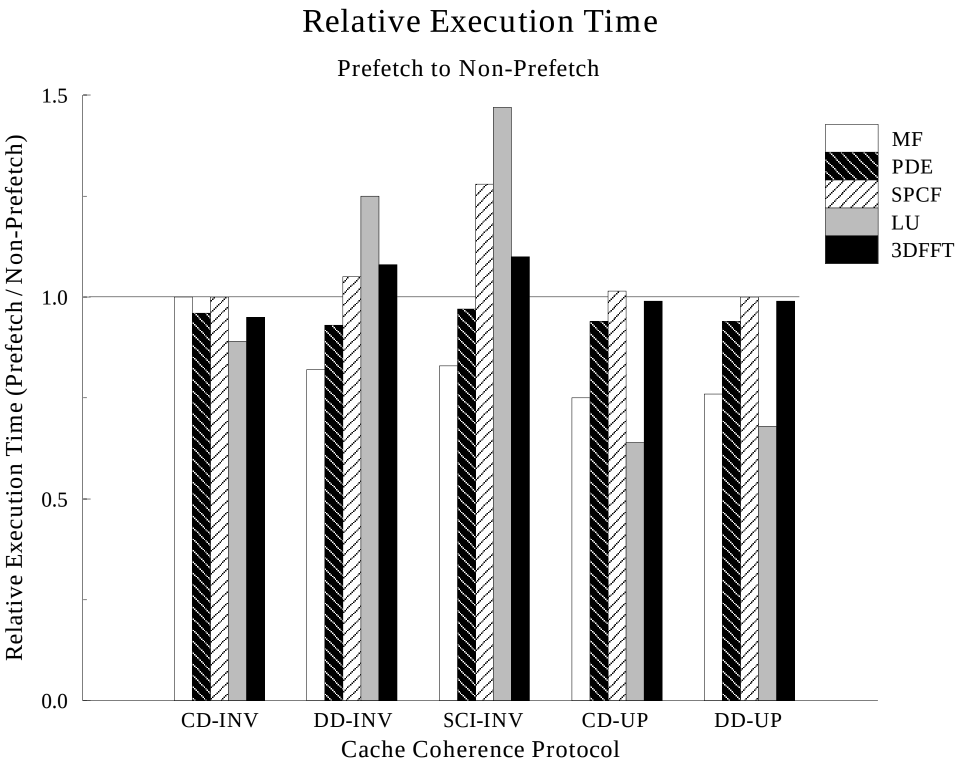


Figure 5.8: Relative Execution Time of Applications Using Word Synchronization and Prefetch Compared to the Non-Prefetch Case

Invalidate-Based Protocols

As discussed above, prefetch may improve the performance of an application by hiding memory access latencies, or it may degrade performance by increasing consumer interference and, therefore, create many more invalidations. Figure 5.8 indicates that prefetch slightly improved the performance of the CD-INV protocol, but for the DD-INV and SCI-INV protocols, prefetch may increase the number of invalidations and, therefore, the total execution time. The performance degradation of prefetching increases as the number of consumers in the applications increase. The CD-INV protocol is less sensitive to this effect since it is able to issue invalidates in parallel, but in DD-INV and SCI-INV protocols, the invalidation latency is dependent on the length of the sharing list, which is determined by the number of shared copies of each line.

Overall, prefetch improved the performance of the CD-INV protocol, and it is beneficial to the distributed directory protocols DD-INV and SCI-INV when the number of consumers is small. Therefore, the applications studied will include prefetch when word data synchronization is used because it improves the performance of the better invalidate-based protocols, CD-INV and DD-INV, compared to the non-prefetching case.

Update-Based Protocols

Unlike the invalidate-based protocols, prefetching with update-based protocols does not adversely affect the performance of the word synchronization scheme. The data prefetch allows the consumers to express an early interest in the data. The producer's writes of this data will result in the consumers' caches being updated. Prefetch may induce more updates, which might tend to congest the system, but, as described in the next section, a write grouping scheme can be used to address this problem.

5.1.6 Performance of Word Synchronization Compared to Block Synchronization

In this section, the performance of the word synchronization scheme is compared to the block synchronization scheme. Figure 5.9 shows the relative performance of each application for each cache coherence protocol. Table 5.1.6 gives the ratio of invalidates (updates) required for the word synchronization case compared to the block synchronization case for the invalidate-based (update-based) protocols, and table 5.2 gives the relative change in total network traffic for the word synchronization case compared to the block synchronization case. This section will discuss the relevance of these measures for both the invalidate-based and update-based protocols.

Invalidate-Based Protocols

For the invalidate-based protocols, the performance of the word synchronization scheme varies. For the applications with small blocks and few consumers (PDE and

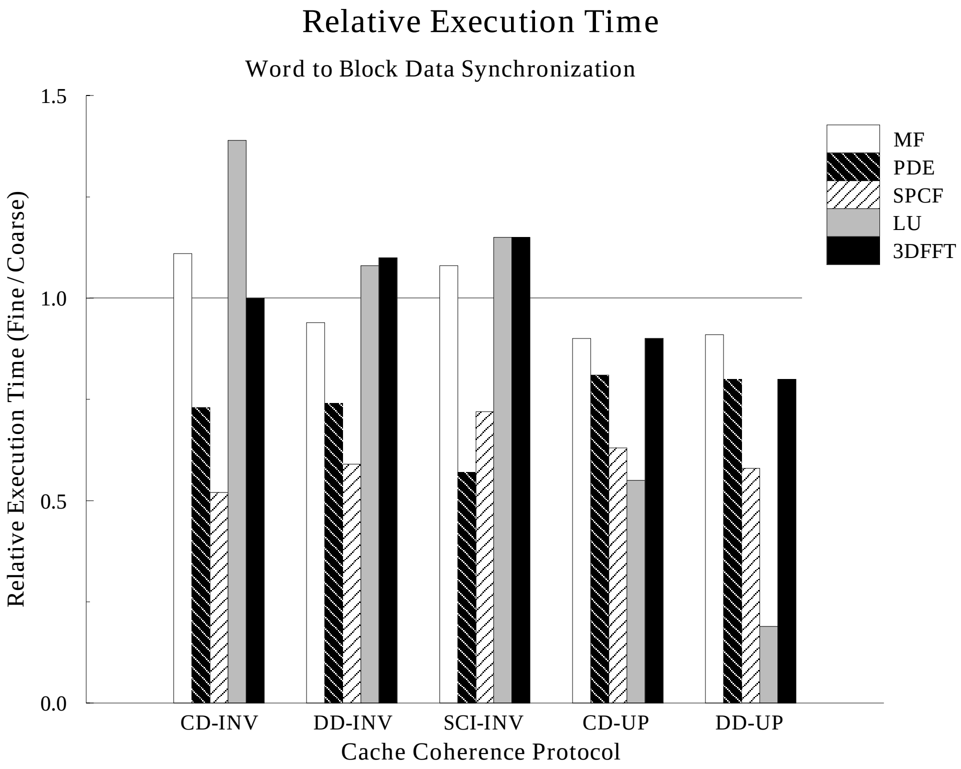


Figure 5.9: Relative Execution Time of Word Synchronization Compared to Block Synchronization

SPCF), the word synchronization scheme improved the performance of the applications compared to the block synchronization case. In these cases, the block synchronization operation was costly, as each synchronization point protects only a small block of data: 8 words for the PDE application and 4.9 words for the SPCF application, as summarized in tables A.2 and A.3. Therefore, the elimination of this block synchronization operation outweighed any extra invalidations or traffic generated by the word synchronization scheme. As illustrated in table 5.1.6, the word synchronization has actually reduced the total number of invalidations compared to the block case for these two applications. With the small block sizes of these applications, the producer was able to write the full block before the consumers could reread the line after the initial invalidation. The single invalidation per block essentially acted as a synchronization or triggering event. The elimination of the explicit synchronization also significantly reduced the network traffic for these applications, as shown in table 5.2. This reduction resulted in an improvement in execution times for the PDE and SPCF applications, as shown in figure 5.9.

Protocol	CD-INV	DD-INV	SCI-INV	CD-UP	DD-UP
MF	10.7	7.08	4.07	0.98	0.84
PDE	0.93	0.93	0.93	1.85	1.91
SPCF	0.94	0.97	0.95	0.73	0.73
LU	4.42	3.67	3.42	0.40	0.41
3DFFT	1.00	1.08	1.14	1.38	1.38

Table 5.1: Ratio of Invalidation/Update Count for Word Compared to Block Synchronization

Protocol	CD-INV	DD-INV	SCI-INV	CD-UP	DD-UP
MF	0.81	0.81	0.80	0.84	0.84
PDE	0.67	0.55	0.51	0.87	1.09
SPCF	0.41	0.51	0.50	0.40	0.44
LU	1.24	1.32	1.59	0.34	0.71
3DFFT	0.98	1.07	0.99	0.84	1.13

Table 5.2: Ratio of Network Traffic for Word Compared to Block Synchronization

As the line utilization increased, the consumer and producer interference also increased. For the MF application, the number of invalidations was small for the block synchronization case which indicates that the consumers were not always eagerly consuming the data. Word synchronization increased the number of invalidations significantly, but the overall traffic was reduced since the extra invalidation traffic was less than the traffic eliminated by the elimination of the explicit block synchronization events. The actual execution time increased for the CD-INV and SCI-INV protocols, but decreased for the DD-INV protocol. The difference in execution time between the invalidate-based protocols arose from the particular producer-consumer interaction that was interfered with. For the DD-INV protocol, the interference was off the critical timing path of the application, and it was in the critical path for the other two invalidate-based protocols.

For the 3DFFT application, the iterative nature of the application required approximately twice the number of shared writes for the word synchronization case as compared to the block synchronization case. These extra writes created at least as many invalidations as were eliminated by the word synchronization, as shown in table 5.1.6. The resulting network traffic, as shown in table 5.2, remained about the same for the same reason. The small number of consumers increased the execution time of the distributed directory invalidate-based protocols (DD-INV and SCI-INV) slightly more than the centralized-directory protocol (CD-INV). Overall, the word synchronization scheme did not improve the execution time for the 3DFFT application when invalidate-based cache coherence protocols were used, as shown in figure 5.9.

As the number of consumers and line utilization increased, the consumer and producer interference also increased. For the LU application, word synchronization increased the number of invalidations and network traffic as illustrated in tables 5.1.6 and 5.2, respectively. With relatively inexpensive block synchronization in this application, the increase in invalidations and network traffic outweighed any performance gains from the elimination of the block synchronization operations. Again, word synchronization offered no improvement in execution time for systems with invalidate-based protocols, as shown in figure 5.9.

Update-Based Protocols

As figure 5.9 demonstrates, a word synchronization scheme always improved the performance of the applications when update-based cache coherence protocols were used. The performance gains came from the elimination of both the block synchronization operation and the update acknowledgments. The word synchronization decreased both the number of updates and the network traffic for non-iterative applications as shown in tables 5.1.6 and 5.2 respectively. For the iterative applications, the extra writes to clear the data between iterations increased the number of updates for both update-based protocols. The network traffic was reduced for the CD-UP protocol, but it increased slightly for the DD-UP protocol.

The word synchronization scheme had the largest impact on the DD-UP protocol when the number of consumers was greater than one. In these cases, the block synchronization scheme required the producer to wait for the updates to be propagated down the list of caches and then acknowledged. This limited the performance of the DD-UP protocol. The word synchronization scheme removed the need for these update acknowledgments.

5.1.7 Relative Performance of Word Synchronization Compared to a Common Base

In this section, the execution times of the applications using a word synchronization scheme are compared to a common base (CD-INV) for each application. This allows the absolute performance of the cache coherence protocols to be compared.

Figure 5.10 shows the execution time of the applications using word synchronization and prefetch compared to the base invalidate-based cache coherence protocol CD-INV. As shown in the figure, the update-based protocols always perform better than any of the invalidate-based protocols when both use a word synchronization scheme. The update-based protocols use the hardware-based write grouping scheme presented in section 5.2.2.

Invalidate-Based Protocols

The DD-INV protocol performed better for the applications with a single consumer. In these cases, the memory write backs of the CD-INV protocol were unnecessary since the data was not read from memory again because cache-to-cache transfers were used to transfer the data to the single consumer. But as the number of consumers increased, the invalidation latency of the distributed directory protocols resulted in longer execution times compared to the base CD-INV protocol.

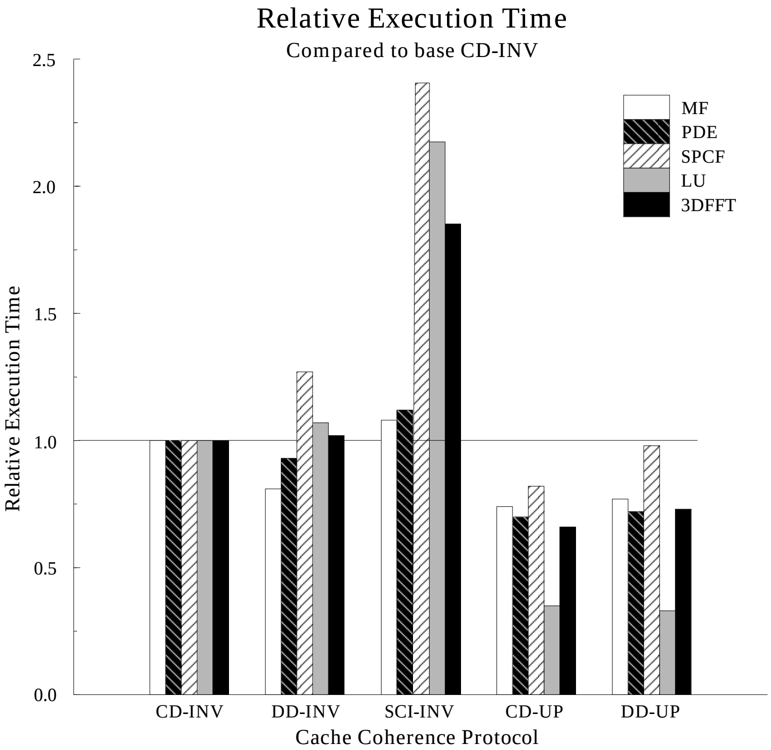


Figure 5.10: Relative Execution Time of Word Synchronization Compared to a Common Base

Update-Based Protocols

For the update-based protocols, the performance of the two protocols was almost identical except for the SPCF application. The improvement in performance compared to the base CD-INV protocol was best for applications with high line utilization and a large number of consumers (LU). As the number of consumers decreased, the improvement from the update-based protocols also decreased compared to the CD-INV protocol. Compared to the DD-INV protocol, the improvement of the update-based protocols was less for applications with a single consumer, but it was more for applications with multiple consumers.

5.1.8 Summary

In summary, the word synchronization scheme, when used with invalidate-based protocols, did not offer a robust solution. It only improved the performance for applications with a small number of consumers and a low line utilization. These application characteristics tended to avoid consumer interference. Conversely, systems with update-based protocols could take advantage of the word synchronization scheme. The resulting execution times were always less than the block synchronization case, and they were always less than the corresponding execution times for the invalidate-based systems using word synchronization.

The performance of word synchronization was not stable when invalidate-based cache coherence protocols were used because the definition of the invalidate-based protocols prevented the producer and consumers from actively sharing a memory line. The producer was required to obtain exclusive ownership of the line before writes could be performed. Consumers who might be consuming data from the line were forced to give up their copy of the line. The simulated results of the scientific applications demonstrated the performance loss that can occur as the producer and consumers interfere with each other.

The definition of update-based protocols offered a much better match to word synchronization. The protocols allowed multiple consumers and producers to maintain copies of a given memory line at the same time, and the producer and consumers of data could not interfere with each other.

The system architecture studied tended to favor the ideal case for the invalidate-based protocols. The load/store limited processor model created bursty read and write streams which reduced the probability of consumer and producer interference. The simulations did not simulate other events that might introduce bubbles into the write and read streams. These events include private data misses, operating system actions, page faults and context switches. Any of these could increase the number of invalidated lines when invalidate-based protocols were used. These events would not increase the work required by the update-based protocols.

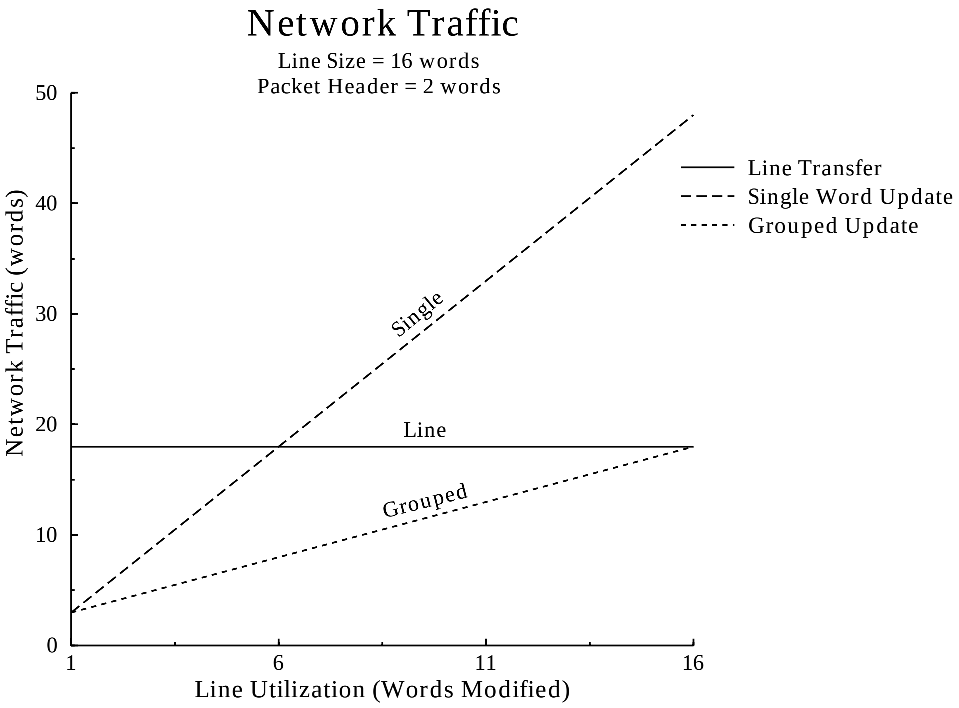


Figure 5.11: Network Traffic

5.2 Write Grouping

As mentioned in the last section, write grouping can be used to improve the efficiency of write updates. The goal of write grouping is to group single writes destined for the same memory line into larger, more efficient write groups. Grouping is able to improve performance in two ways. First, grouping writes allows the cost of the network header to be amortized across all words in the group. The second advantage of grouping is that the cost of memory updates can also be amortized across more words in a grouped update than in a single word update.

Efficient grouping can significantly reduce the network traffic. For example, figure 5.11 shows the number of network words required to transfer a portion of a memory line using a line transfer, single word updates or a grouped update. The y-axis is the resulting network traffic, and the x-axis is the line utilization. The line utilization is the number of modified words in the memory line that must be transferred. For line transfers, as used by invalidate-based protocols, the network traffic is constant regardless of the line utilization. For single word updates, every write update requires three network words: a two word header plus the data word. But by grouping the writes, the resulting network traffic is always less than or equal to the traffic required by the line transfer; the grouped updates result in the minimal network traffic necessary to transfer the line's modified data.

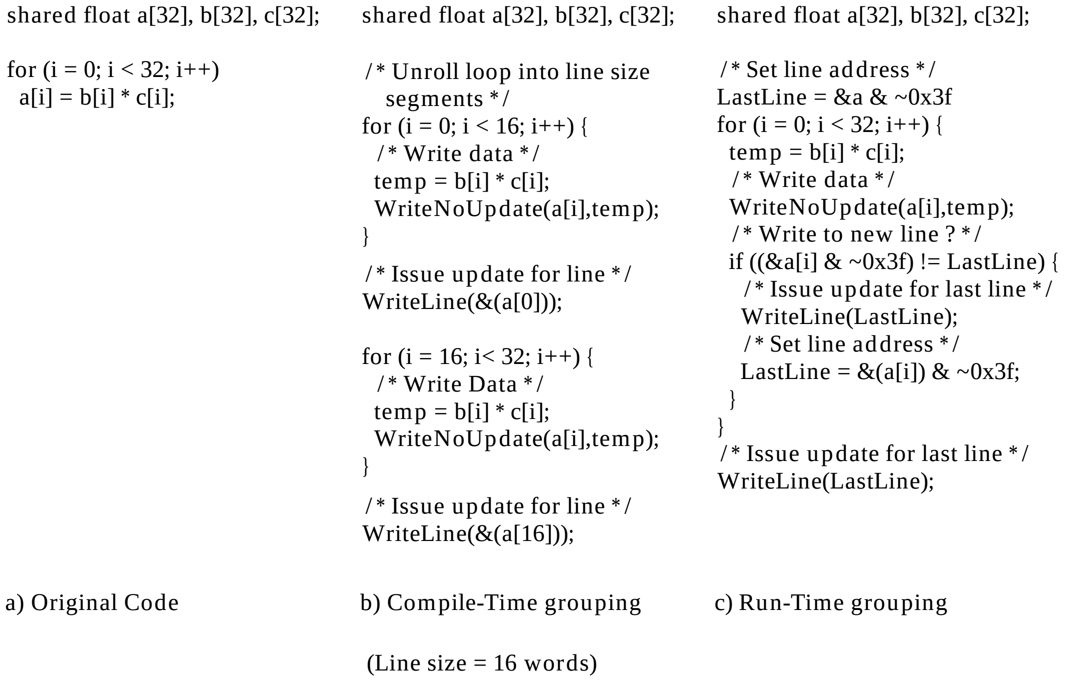
Write grouping can be accomplished using a software-based or hardware-based scheme. In a software-based scheme, compile time analysis is used to identify writes to shared data. The compiler issues these writes with a special non-updating write and issues special write-line instructions to initiate the data updates. Section 5.2.1 explains the software-based scheme in more detail. In a hardware-based scheme, writes to the same memory line can be grouped into larger write groups at the write buffer, as described in section 5.2.2.

5.2.1 Software-Based Write Grouping

With perfect knowledge of the application's write pattern, software-based write grouping is able to achieve optimal efficiency by grouping all shared writes to the same memory line into a single write group. Compile time analysis is used to identify writes to shared data, and the compiler would issue these writes using a special write-no-update instruction. This instruction would write the word into the cache and set a special update-pending bit for the word[1]. After all the words written to a given memory line have been issued, a special write-line instruction is issued that causes the words in the cache line with the pending-update bit set to be grouped into an update packet and forwarded to the directory. These special instructions are ignored if the line is in an exclusive state where there are no other shared copies of the line, and therefore, no updates are required.

The scheme places a large burden on the compiler. First the compiler must identify writes to shared data and issue them with a special write-no-update instruction. Next, the compiler must issue the special write-line instruction after all the writes to a given

[1] Each cache line already has a set of valid bits used to allow the protocols to write words into the line while the line is in a pending state awaiting a miss reply. These bits can also be used as the update-pending bits when the cache line is in a shared state.



```
shared float a[32], b[32], c[32];

for (i = 0; i < 32; i++)
  a[i] = b[i] * c[i];
```

a) Original Code

```
shared float a[32], b[32], c[32];

/* Unroll loop into line size
   segments */
for (i = 0; i < 16; i++) {
  /* Write data */
  temp = b[i] * c[i];
  WriteNoUpdate(a[i],temp);
}
/* Issue update for line */
WriteLine(&(a[0]));

for (i = 16; i< 32; i++) {
  /* Write Data */
  temp = b[i] * c[i];
  WriteNoUpdate(a[i],temp);
}
/* Issue update for line */
WriteLine(&(a[16]));
```

b) Compile-Time grouping

(Line size = 16 words)

```
shared float a[32], b[32], c[32];

/* Set line address */
LastLine = &a & ~0x3f
for (i = 0; i < 32; i++) {
  temp = b[i] * c[i];
  /* Write data */
  WriteNoUpdate(a[i],temp);
  /* Write to new line ? */
  if ((&a[i] & ~0x3f) != LastLine) {
    /* Issue update for last line */
    WriteLine(LastLine);
    /* Set line address */
    LastLine = &(a[i]) & ~0x3f;
  }
}
/* Issue update for last line */
WriteLine(LastLine);
```

c) Run-Time grouping

Figure 5.12: Software-Based Write Grouping

line have been issued. Note that a write line instruction is required for each cache line with pending updates. If the application is data independent, the compiler may simply insert these instructions at the proper location in the code. Alternatively, the compiler may insert code around each write to generate the necessary write-line instructions at run time. In this case, the code must maintain the line address of the last shared write. When the line address of the writes change, a write-line instruction is issued for the last line address. This run-time implementation would add several instructions per write and would require the use of a register to maintain the last line address.

Figure 5.12 shows an example loop and illustrates how the software-based scheme may be implemented. The original code is a simple loop that computes a set of values for the shared vector a. For the compile-time grouping, the loop is unrolled into segments that are of length equal to the memory line size. The writes are issued using the special $WriteNoUpdate$ instruction that writes the values into the cache but does not initiate an update. Then, after the full line of data has been produced, the

WriteLine instruction is issued for the line to initiate the update. This is repeated for each memory line. For the run-time grouping, each write is again issued with the *WriteNoUpdate* instruction, but the line address of the write must be compared with the line address of the last write. If they are not equal, a *WriteLine* instruction is issued for the last line, and the last line pointer is set to the new line address. A final *WriteLine* instruction must be issued to initiate the update for the last line's data.

As illustrated from the above discussion, the software-based scheme may be difficult to implement efficiently, but the scheme is able to achieve optimal grouping efficiency by grouping all writes to a given cache line. The compile-time software-based scheme will be used to compute the optimal grouping efficiency for each application studied and to demonstrate that the hardware-based scheme presented in the next section can approach this efficiency.

5.2.2 Hardware-Based Write Grouping

In a hardware-based write grouping scheme, writes are grouped as they are written into the write buffer, as shown in figure 5.13. The scheme requires an additional grouping bit for each word in the write buffer. As the processor issues the writes, the line address of the write is compared with the line address of the last write inserted into the write buffer. If the write is to the same line, the additional grouping bit is set to 1, indicating that the write should be grouped with the last write inserted into the write buffer. When the cache processes the writes from the write buffer, it consumes all writes in a write group, and it is able to issue the write update, if needed, in a larger, more efficient packet.

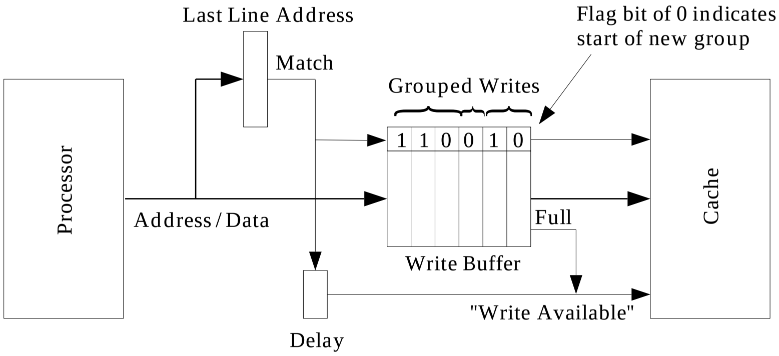


Figure 5.13: Hardware-Based Write Grouping at the Write Buffer

A write group is delayed in the write buffer until either

1. the write buffer fills,

2. a write to a new line is received, or

3. the delay timer expires.

The first two conditions are obvious. Once the write buffer fills or the write buffer receives a write to a new line, no more writes can be grouped with the current write group. The third condition is used to increase the grouping window for each write group. This delay prevents the cache from consuming writes as fast as the processor is able to issue them; the delay sets the minimal grouping window for each write group. The base case sets the delay count at five cycles.

Figures 5.14 shows an example of hardware-based write grouping. In the figure, the processor issues 7 writes labeled Wi where i is the address of the write. In this example, the line size is 2 words and the write grouping delay is 3 cycles. Writes W0 and W1 are grouped into a two word write group because they are writes to the same line. Write W5 cannot be grouped with the W0,1 group since it is a write to another line. This write triggers the sending of the first write group, W0,1, to the cache. No writes follow W5 and the delay timeout expires resulting in W5 being sent to the cache. Writes W2 and W3 are grouped, but write W9 is a write to another line so it triggers the sending of write group W2,3 to the cache. Finally, write W0 is issued, which causes write group W9 to be sent to the cache. No writes follow W0 so the delay timeout expires and the write is sent to the cache. Notice that the delay timeout only affects the last write in a write burst.

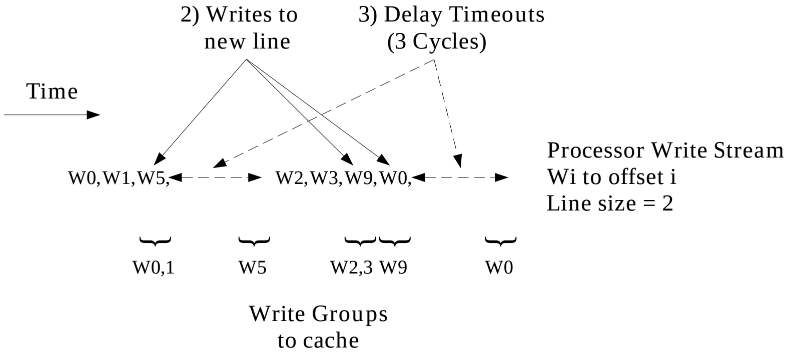


Figure 5.14: Hardware-Based Write Grouping Example

The grouping hardware adds an extra pipeline stage to the write buffer. Writes now take a minimum of two cycles to reach the cache, but the write buffer may still accept writes at a rate of one per cycle.

5.2.3 Grouped Update Network Packet

To send the grouped update efficiently, a bit vector is used to indicate the line offsets of the grouped writes. In this vector, bit b_i would be 1 if the data words in the packet included an update for offset i. The data following the packet header must be

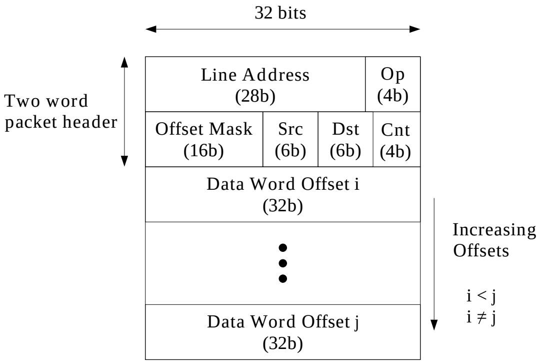


Figure 5.15: Grouped Update Packet

arranged in ascending offset order, as shown in figure 5.15. To achieve this ordering for the software-based scheme, the cache simply reads the update-pending words in the proper offset order. For the hardware-based scheme, the cache must first read the write group from the write buffer. These writes may be in any offset order and may contain multiple writes to the same offset. Next, the words are written into the cache line using the valid bits, as in the software-based scheme, to indicate which words of the cache line have been modified. Finally, the words are read from the cache line in the proper offset order, and the update packet is generated. The valid bits are then cleared.

5.2.4 Write Grouping Performance

Grouping Efficiency

Table 5.3 shows the average write group size for the two update-based cache coherence protocols using both software-based (SW) and hardware-based (HW) write grouping schemes. The average write group size is given for both block and word data synchronization schemes for the five applications under study. Block synchronization requires explicit synchronization events such as flags or barriers, and these variables are allocated on separate memory lines to avoid false sharing. Therefore, synchronization writes cannot be grouped with any other writes since no other data is allocated on the same line as the synchronization variable. This allocation policy results in a write group of one word which will reduce the average write group size. In contrast, word synchronization combines the synchronization information with the data word. In this case, the average write group sizes are slightly larger than in the block synchronization case; the difference indicates the relative frequency of synchronization writes to data writes.

With an ideal knowledge of the applications' write patterns, the software-based scheme is able to group all writes destined for the same line into optimal write groups. The actual size of the optimal write group is determined by each application's average line utilization, which is given in table A.5. The higher the line utilization, the larger the optimal write group. The maximum write group size is 16 words, a full memory

Applications	Sync	CD-UP-SW	CD-UP-HW	DD-UP-SW	DD-UP-HW
MF	Block	13.2	13.2	13.2	13.2
	Word	14.9	14.9	14.9	14.9
PDE	Block	4.5	4.4	4.5	4.4
	Word	8.0	7.9	8.0	7.9
SPCF	Block	1.4	1.4	1.4	1.4
	Word	1.9	1.9	1.9	1.9
3DFFT	Block	3.0	1.8	3.0	1.9
	Word	3.5	2.2	3.5	2.3
LU	Block	9.5	9.5	9.5	9.5
	Word	9.5	9.5	9.5	9.5

Table 5.3: Grouping Efficiency - Words Per Update Packet

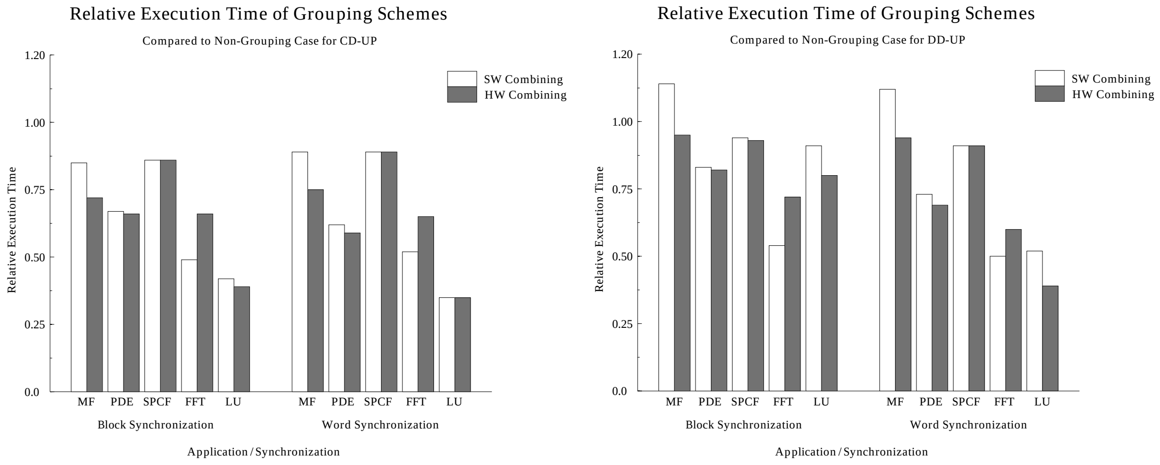
line.

As indicated in table 5.3, the hardware-based grouping scheme is almost able to achieve the same write grouping efficiency as the software-based grouping scheme for both update-based protocols. The one exception is the 3DFFT application. In this application, the average shared data write rate is too low to be captured by the hardware grouping scheme using a five cycle grouping window. In section 5.2.5, the effect of varying the length of the write grouping window will be examined.

Execution Time

Figure 5.16 shows the relative execution times for the applications using the two grouping schemes compared to the non-grouping case for each update-based protocol. For all applications, except 3DFFT, the hardware-based scheme resulted in about the same or faster execution time compared to the software-based grouping scheme for both update-based protocols. The hardware-based scheme did not perform as well for the 3DFFT since the write grouping was sub-optimal compared to the software-based scheme, as was described in section 5.2.4,

The software-based grouping scheme resulted in a significantly longer execution time than the hardware-based scheme for the MF application for both protocols and the LU application for the DD-UP protocol. In the MF application, the data was


Relative Execution Time of Grouping Schemes

Compared to Non-Grouping Case for CD-UP

Relative Execution Time of Grouping Schemes

Compared to Non-Grouping Case for DD-UP

A) CD-UP Protocol

B) DD-UP Protocol

Figure 5.16: Relative Execution Time of Grouping Schemes Compared to Non-Grouping Case

not as eagerly shared as in the other applications. Most of the writes were to cache lines in the exclusive state, which did not require updates. In this case, the extra cycles introduced by the software-based scheme outweighed any performance gain from the relatively few write groupings. For the LU application, the observed write miss latency was increased because the delay to issue the writes, introduced by the software-based scheme, reduced the amount of the miss latency which could be hidden behind useful work. This resulted in a slightly longer execution time for the DD-UP protocol.

The performance improvement from write grouping was larger for the CD-UP protocol than for the DD-UP protocol. The write grouping improved the performance of the CD-UP protocol by reducing both the network traffic and the memory update overhead. The DD-UP protocol did not update memory on each write and, therefore, only benefited from the reduction in network traffic.

5.2.5 Variations in Hardware-Based Grouping

This section examines the performance of the hardware-based write grouping scheme when the grouping delay was varied and the location of the write grouping buffer was

		CD-UP-HW					DD-UP-HW				
	Sync	0	5	10	15	20	0	5	10	15	20
MF	Block	4.6	13.2	13.2	13.2	13.2	4.8	13.2	13.2	13.2	13.2
	Word	4.8	14.9	14.9	14.9	14.9	5.0	14.9	14.9	14.9	14.9
PDE	Block	2.9	4.4	4.5	4.5	4.5	2.8	4.4	4.5	4.5	4.5
	Word	6.8	7.9	8.0	8.0	8.0	6.4	7.9	8.0	8.0	8.0
SPCF	Block	1.2	1.4	1.4	1.4	1.4	1.2	1.4	1.4	1.4	1.4
	Word	1.2	1.9	1.9	1.9	1.9	1.2	1.9	1.9	1.9	1.9
3DFFT	Block	1.2	1.8	2.7	2.9	3.0	1.2	1.9	2.7	2.9	3.0
	Word	1.5	2.2	3.2	3.5	3.5	1.6	2.4	3.2	3.4	3.5
LU	Block	3.5	9.5	9.5	9.5	9.5	3.1	9.5	9.5	9.5	9.5
	Word	3.3	9.5	9.5	9.5	9.5	4.1	9.5	9.5	9.5	9.5

Table 5.4: HW Grouping Delay (Cycle) - Words Per Write Group

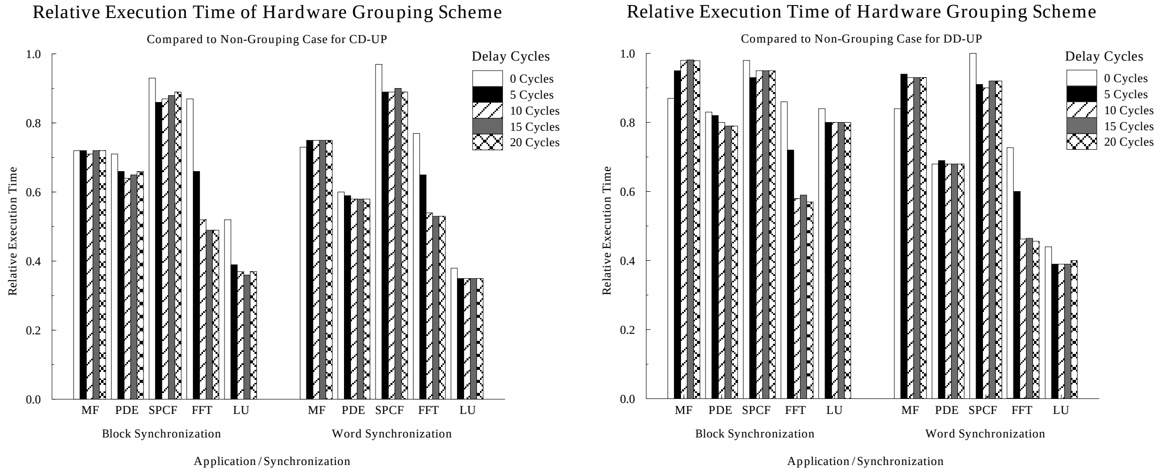
moved to the output of the cache or the input of the memory/directory.

Write Grouping Delay

Table 5.4 shows the words per write group as the write grouping delay was varied from no delay to 20 cycles of delay. With no delay, the write grouping scheme was able to group only a small fraction of the writes. This grouping only occurred when the writes were delayed in the write buffer because the cache was busy responding to a read or network request.

As the write grouping delay was increased, the grouping scheme captured many more writes. As noted in section 5.2.4, a five cycle delay was sufficient to group almost all writes that can be grouped. The one exception was the 3DFFT application, but as the write grouping delay was increased for this application, more writes were grouped. A ten cycle delay resulted in an almost optimal write grouping efficiency for all applications.

However, increasing the write grouping delay was not without its cost since the increase in delay also affected the total execution of the applications, as shown in figure 5.17. The figure shows the relative execution times as the write grouping delay was varied from 0 cycles to 20 cycles. In the figure, the execution times of the applications improved with increasing delay until the delay was large enough to group a significant portion of the writes. Increasing the delay beyond this point only affected the last write of each data block. Intermediate writes were grouped as new writes were inserted into the write buffer. If these writes were efficiently grouped, then the write rate was faster than the grouping delay, and only the last write in the group was delayed by the full delay timeout period, as was described in section 5.2.2.



A) CD-UP Protocol

B) DD-UP Protocol

Figure 5.17: Relative Execution Time for Alternative Grouping Delays for Hardware-Based Grouping

The relative impact of the delay is largest for the block synchronization case. In this case, the processor must stall until all writes have been performed (all updates acknowledged). The performance loss depends on the block size and the write completion latency. If the data block is large, the grouping delay will be small compared to the total write time. But in applications such as SPCF where the block size is small, the delay becomes a noticeable fraction of the total write time. The effect is clearly illustrated in figure 5.17 for this application using block synchronization, especially for the CD-UP protocol. For the DD-UP protocol, the write completion latency is longer than in the CD-UP protocol, and it increases with the number of caches that are updated on each write. The write grouping delay is small compared to this write completion latency, and therefore, the impact of the increased write grouping delay

		CD-UP-HW			DD-UP-HW		
Applications	Sync	WB	CO	MI	WB	CO	MI
MF	Block	13.2	7.0	6.6	13.2	7.1	7.0
	Word	14.9	6.8	6.2	14.9	7.3	7.3
PDE	Block	4.4	3.5	1.1	4.4	3.3	1.3
	Word	7.9	5.3	1.1	7.9	1.2	1.2
SPCF	Block	1.4	1.2	1.2	1.4	1.2	1.2
	Word	1.9	1.4	1.2	1.9	1.3	1.3
3DFFT	Block	1.8	1.8	1.1	1.9	1.8	1.1
	Word	2.2	2.2	1.2	2.4	2.3	1.2
LU	Block	9.5	8.2	1.0	9.5	7.8	1.1
	Word	9.5	7.5	1.0	9.5	2.3	2.3

Table 5.5: Grouping Location - Words Per Write Group

was negligible for the DD-UP protocol. The one exception the MF application. As noted in the last section, data was not eagerly consumed in this application. Thus, the delay only increased the execution time.

For word synchronization, the added delay for the last word written had little impact on the total execution time of the application, as shown in figures 5.17. When an application used word synchronization, the consumption time of earlier data words overlapped the latency of subsequent data updates.

Location of the Grouping Buffer

Write grouping could be done at any request buffer in the system. Two other candidates for write grouping are the request output buffer of the cache (CO) and the request input buffer of the memory (MI). Table 5.5 shows the write group sizes when write grouping is introduced at these buffers. Grouping writes at either of these buffers is not as efficient as grouping at the write buffer (WB), and increasing the grouping delay beyond 5 cycles had little impact on the grouping efficiency at these other buffers. The write buffer grouping efficiency was always the best.

Grouping at these other locations does not work well since other network packets tend to disrupt the flow of writes from the caches to the directory/memory. The grouping scheme works by attempting to group a new write packet with the last packet inserted into the buffer. The complexity of the buffers could be increased to allow writes to be grouped with any appropriate write packet currently in the buffer. This might improve grouping, but, as discussed in section 5.2.4, the write grouping efficiency at the write buffer is almost optimal. The additional cost of increasing the buffer complexity would result in no performance gain compared to the inexpensive write buffer grouping.

5.2.6 Summary

In summary, this section has explored the performance gains from write grouping in update-based cache coherent systems. Two types of grouping schemes were discussed: a software-based and a hardware-based scheme. The software-based scheme required compiler support, but the scheme was able to optimally group writes. The hardware-based scheme was shown to almost achieve this optimal write grouping efficiency if a grouping delay window was introduced, and the scheme required only a small amount of hardware support and little, if any, software support.

The write-buffer write grouping improved the performance of the CD-UP protocol the most. The improvements in execution time ranged from about 10% to over 50%. The gains came from two sources. First, the write grouping significantly decreased network traffic resulting from the write updates. Second, the larger write groups decreased the average memory update latency per word as the memory access time could be amortized across more data words. The DD-UP protocol also benefited from write grouping. The improvement in execution time ranged from only a few percent to over 50%. Since the DD-UP protocol did not update memory, the gain came only from a reduction in write update network traffic.

Increasing the write grouping delay beyond the base 5 cycles had little impact on the efficiency of the write grouping. The one exception was the 3DFFT application in which a 10 cycle delay was required to achieve near optimal write grouping efficiency. Also, increasing the delay beyond 5 cycles had a minor affect on the total execution time of the application. Moving the write grouping to the cache output buffer or the memory input buffer resulted in poor write grouping and, therefore, little improvement in total execution time compared to the non-grouping case.

Overall, write grouping is essential for improving the performance of update-based cache coherent system with a general interconnect, and a write grouping delay window of 5 cycles was sufficient to achieve efficient write grouping in most applications without adversely affecting non-grouped writes.

Chapter 6

Performance Results

The last chapter discussed the performance improvements from word synchronization and write grouping schemes for update-based cache coherence protocols. In this chapter, the performance of the update-based protocols are compared directly to the three invalidate-based protocols for each application.

As described previously, the actions of a producer and consumer interaction using block synchronization (release consistency memory model) can be divided into the basic operations shown in figure 6.1.

The performance of each cache coherence protocol is dependent on how efficiently it can minimize each of the latency components shown in the figure: the write, fence, synchronization and read latencies. The write latency is simply the time to "issue" or send the writes to the write buffer. The fence latency is the latency until all writes have been performed. In the invalidate-based protocols, a write is considered

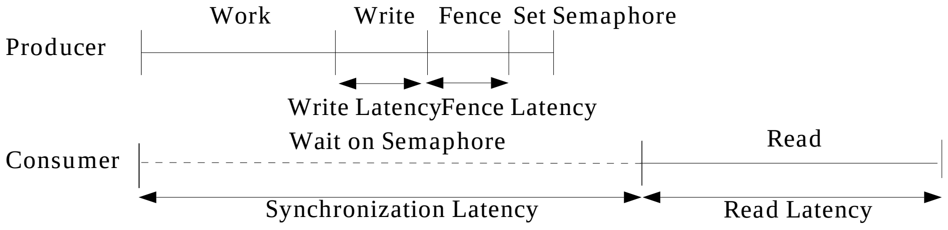


Figure 6.1: Block Synchronization

to have been performed when the cache receives exclusive ownership of the line. For the update-based protocols, a write is considered to be performed when all necessary updates have been acknowledged. The synchronization latency is the time a consumer waits for the desired semaphore to be set; and finally, the read latency is the time to read the data once the semaphore has been set. For word synchronization, the latency components are reduced to the write latency and the read/synchronization latency, as shown in figure 6.2. The discussion of the results will rely heavily on these latency components.

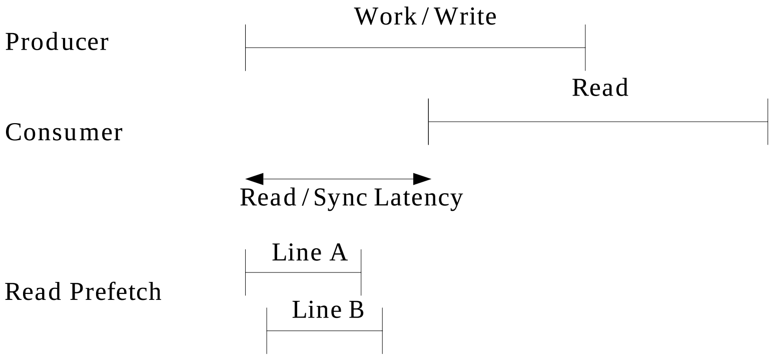


Figure 6.2: Word Synchronization

The next five sections describe the simulation results for the five fine-grain, scientific applications examined. This discussion begins by examining applications with a single consumer and moderate to high line utilization: MF and PDE. Next, the impact of a small number of consumers will be examined by studying the SPCF and 3DFFT applications, and finally, the impact of both high line utilization and multiple consumers is examined in the LU application.

For each application, a graph of the relative execution time of each protocol compared to the base CD-INV invalidate-based protocol is presented. On each graph, four differences are labeled (one set for each update-based protocol - solid lines for CD-UP and dashed lines for DD-UP). Difference BS represents the difference in the performance of the update-based protocols without write buffer grouping and the CD-INV protocol using block synchronization. Difference BS-G represents the improvement in the block synchronized update-based protocols when write buffer grouping is added.

Difference WS represents the improvement in the update-based protocols when word synchronization is used rather than block synchronization, and difference WS-G represents the improvement in the word synchronized update-based protocol when write buffer grouping is added.

Table C.1 in appendix C summarizes the relative execution times of each protocol for the five applications examined.

6.1 MF Application

For the block synchronization case without write grouping, the CD-UP protocol increased the execution time by 26% compared to the CD-INV protocol, and the DD-UP protocol has similar performance to the DD-INV protocol, as shown by difference BS in figure 6.3. The update-based protocols were able to reduce the read latency by updating the consumers' caches, but because of the high line utilization of the application, these single word updates were very inefficient compared to the line transfers of the invalidate-based protocols. These inefficient updates increased the congestion in the system which resulted in longer write and synchronization latencies.

The addition of write grouping improved the efficiency of the updates which decreased the network congestion. The write grouping improved the execution time of the CD-UP and DD-UP protocols by 28% and 4% respectively, as shown by difference BS-G in figure 6.3. Write grouping was more effective in the CD-UP protocol because the larger update packets reduced the average overhead of each memory update. The update-based protocols were now able to outperform the invalidate protocols by a slight margin.

With single word updates significantly congesting the system, word synchronization was able to hide only a portion of the update latency, which resulted in a minimal improvement in the execution time of the update-based protocols, as indicated by difference WS in figure 6.3. Word synchronization improved the performance of the CD-UP protocol by 13% and the performance of the DD-UP protocol by 7%.

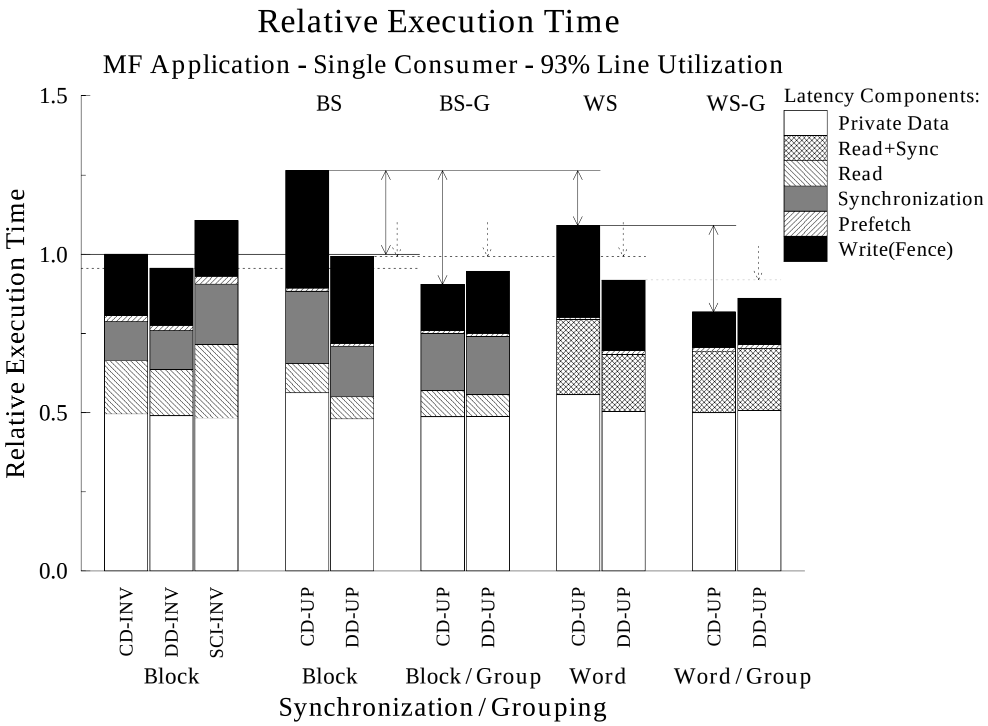


Figure 6.3: Relative Execution Time for MF

The use of both word synchronization and write grouping improved the performance of the CD-UP and DD-UP protocols by 25% and 7% respectively compared to the non-grouping case, as shown by difference WS-G in figure 6.3. As in the block synchronization case, write grouping reduced the congestion, which decreased the write latency compared to the word synchronization case without write grouping.

Overall, the update-based protocols with the two enhancements were able to outperform the invalidate-based protocols. For the CD-UP protocol, the block synchronization case with write grouping performed better than the word synchronization case without write grouping. It improved the performance of the application by 9% compared to the CD-INV protocol. For the DD-UP protocol, the performance of the word synchronization case without write grouping was slightly better than the block synchronization case with write grouping. For this case, the improvement in the execution time was 4% compared to the DD-INV protocol. The use of both word synchronization and write grouping allowed the update-based protocols to improve performance of the application by 18% for the CD-UP protocol compared to the CD-INV protocol and 10% for the DD-UP protocol compared to the DD-INV protocol.

6.2 PDE Application

The update-based protocols using block synchronization were able to decrease the total execution time of the PDE application by 3% for the CD-UP protocol compared to the CD-INV protocol and by 12% for the DD-UP protocol compared to the DD-INV protocol, as shown by difference BS in figure 6.4. The main source of the improvement was a reduction in the synchronization latency resulting from the update of the semaphores. The update of the shared data also reduced the read latency, but the overall impact was small since the invalidate-based protocols were able to effectively prefetch the necessary data.

Write grouping was able to group the 8 data writes destined for each neighboring node into single update packets. These more efficient packets reduced the overhead and congestion in the network and at the caches. This reduction in congestion resulted in a decrease in the fence latency which in turn reduced the synchronization latency. The sum of these latency reductions resulted in an improvement in the execution time of 35% and 18% for the CD-UP and DD-UP protocols respectively compared to the non-grouping case, as shown by difference BS-G in figure 6.4.

The improvement in the execution time of the update-based protocols when word synchronization and no write grouping was used was minor compared to the block synchronization case without write grouping, as shown by difference WS in figure 6.4. In this iterative application, the producer was required to clear the data between iterations. This extra traffic limited the possible improvements in performance from word synchronization.

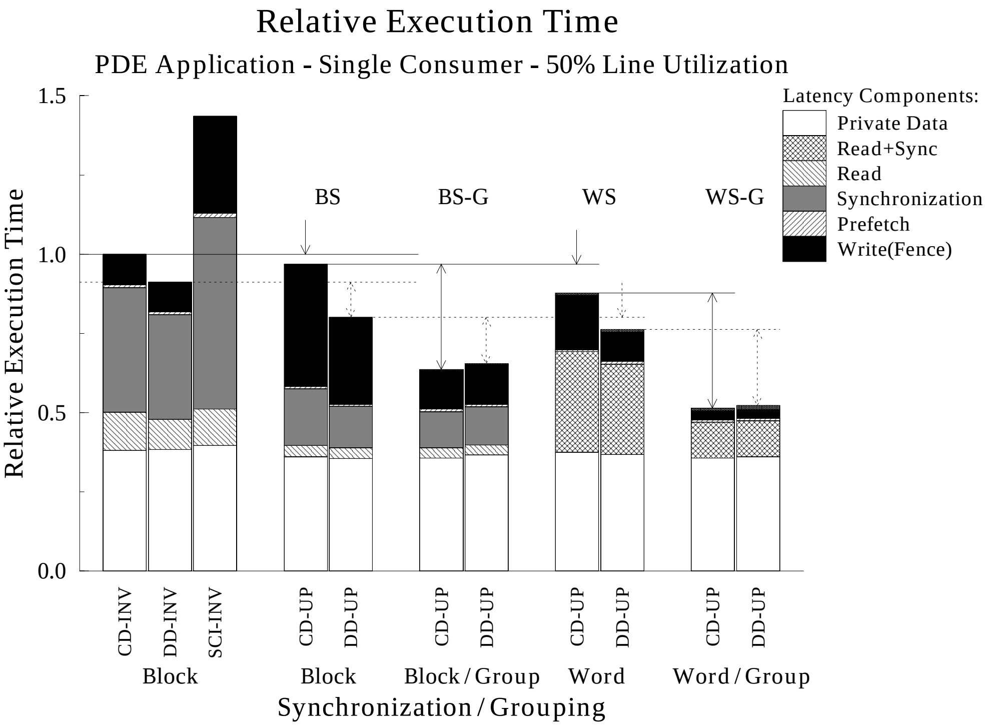


Figure 6.4: Relative Execution Time for PDE

As in the block synchronization case, write grouping with word synchronization was able to group all writes destined for each consumer into a single update packet. The resulting synchronization and read latency was reduced to half that of the update-based protocol using word synchronization without write grouping. This reduction in latency along with the elimination of the fence latency reduced the execution times of the update-based protocols by 42% and 31% for the CD-UP and DD-UP protocols respectively, as shown by difference WS-G in figure 6.4.

Overall, the update-based protocols were able to significantly improve the execution time of this iterative application. The update-based protocols performed well in the block synchronization case, and the addition of write grouping to the block synchronization case improved the execution time by about 35% compared to the CD-INV protocol. The use of word synchronization and write grouping allowed the update-based protocols to improve the execution time by 49% for the CD-UP protocol compared to the CD-INV protocol and 43% for the DD-UP protocol compared to the DD-INV protocol.

The difference between the CD-UP and DD-UP protocols arises from the difference in the path of the data updates. In the CD-UP protocol, updates are sent to the directory where they are forwarded to the consumer. In the DD-UP protocol, the path of the update depends on the producer's position in the cache list. If the producer is at the head of the list, the update is sent directly to the consumer, but if the producer is not at the head of the list, the update must be sent to the directory first and then forwarded to the consumer at the head of the list. In this particular application, each update required an average of 1.5 update hops indicating that the producer's cache was at the head of the list half of the time. This reduction in update hops allowed the DD-UP protocol to perform slightly better than CD-UP for the non-grouping cases. When write grouping was introduced, the reduction in congestion reduced the cost of the extra half hop which minimized the difference in the performance between the update-based protocols.

6.3 SPCF Application

When using block synchronization, the CD-UP and DD-UP protocols were able to reduce the total execution time of the SPCF application. The CD-UP protocol reduced the execution time by 21% compared to the CD-INV protocol, and the DD-UP protocol reduced the execution time by 16% compared to the DD-INV protocol, as shown by difference BS in figure 6.5. The main source of this reduction was a decrease in the synchronization latency which accounted for a significant portion of the execution time. The reduction in read latency was significant even though the invalidate-based protocols were able to use prefetch to hide a large portion of the read latency, but overall, the read latency was a small portion of the total execution time.

Since the application has a low line utilization, the write grouping scheme had little opportunity for grouping writes. In both update-based protocols, the grouped update packet contained an average of only two words, which resulted in a minimal improvement in the execution time, as shown by difference BS-G in figure 6.5.

In this application, word synchronization allowed for the elimination of the costly explicit synchronization. With each consumer only consuming a fraction of the data produced by each producer, the cost of an explicit synchronization semaphore was high. As a result of the elimination of this explicit synchronization and the fence operation, the execution time of the CD-UP protocol decreased by 39%, and the execution time of the DD-UP protocol decreased by 40% compared to the block synchronization case, as shown by difference WS in figure 6.5.

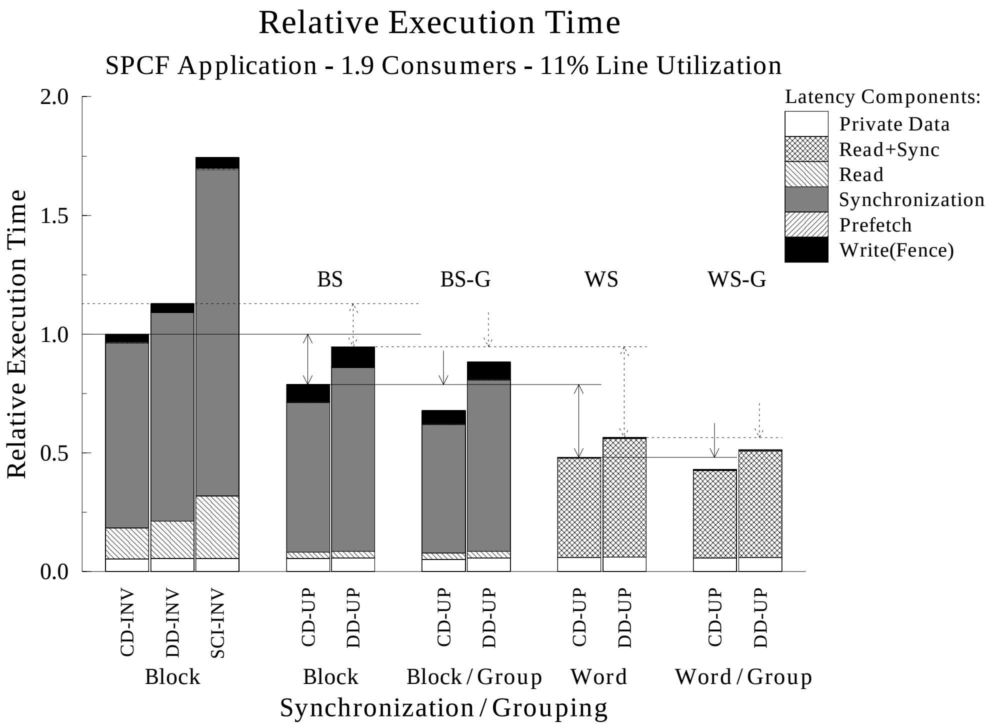


Figure 6.5: Relative Execution Time for SPCF

The use of write grouping with word synchronization also had a minimal impact on execution time for both update-based protocols, as shown by difference WS-G in figure 6.5. For both protocols, the early consumption of data permitted by the word synchronization scheme was able to hide a majority of the latency of subsequent updates.

The difference between the CD-UP and DD-UP protocol was due to the increased latency introduced by the longer sharing lists of caches in the DD-UP protocol. On the average, the list of caches was only 2.3 caches long, but the maximum reached 10 caches. The longer list increased the fence latency as updates were required to traverse the entire list before being acknowledged. The synchronization latency also increased as caches at the end of the list had to wait longer before receiving the data updates.

6.4 3DFFT Application

For the block synchronization case, the congestion created by the inefficient updates prevented the update-based protocols from improving the performance of the application compared to the CD-INV and DD-INV invalidate protocols, as shown by difference BS in figure 6.6. The update-based protocols did reduce the read and synchronization latencies, but the increase in the write and fence latency outweighed any gain from the reduced latencies. The CD-UP protocol increased the execution time of the application by 11% compared to the CD-INV protocol, and the DD-UP protocol increased the execution time of the application by 37% compared to the DD-INV protocol.

Since the application had a relatively high line utilization, the write grouping scheme was able to group a majority of the single word updates into more efficient packets. This grouping resulting in a significant reduction in the write latency and a small reduction in the read and synchronization latencies. These reductions allowed the write grouping to improve the performance of the CD-UP protocol by 34% and the DD-UP protocol by 28%, as shown by difference BS-G in figure 6.6.

Word synchronization resulted in a minimal improvement in the execution time of the application, as shown by difference WS in figure 6.6. The iterative nature of the application resulted in almost twice the number of shared writes as in the block synchronization case since the shared data had to be cleared between each iteration.

These extra writes outweighed any performance gains from the word synchronization.

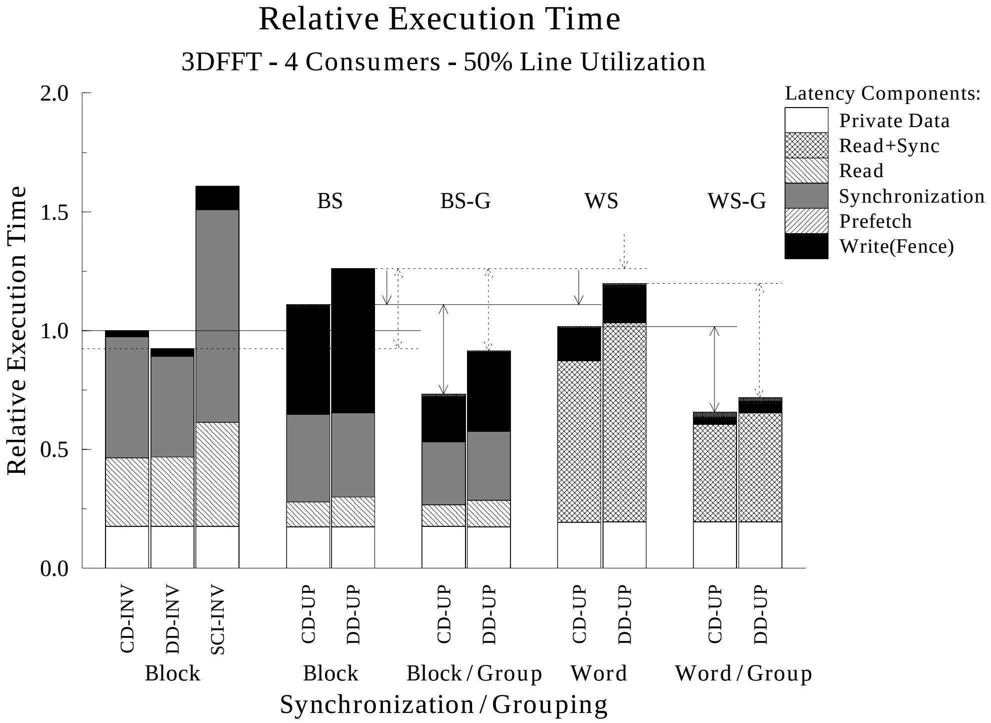


Figure 6.6: Relative Execution Time for 3DFFT

As with the block synchronization case, write grouping was able to improve the performance of the application when word synchronization was used, as shown by difference WS-G in figure 6.6. The grouped writes reduced the network congestion created by the updates. This allowed for a significant reduction in the write and read/synchronization latencies.

Overall, the CD-UP protocol was able to improve the performance of the application by 34% compared to the CD-INV protocol, and the DD-UP protocol was able to improve the performance of the application by 22% compared to the DD-INV protocol when both enhancements are used. The difference between the CD-UP and DD-UP protocols was due to the multiple consumers for each data block. The multiple consumers increased the update latency in the DD-UP protocol since each update had to traverse the list of caches sharing the updated line.

6.5 LU Application

For the LU application with a high line utilization and a large number of consumers, the single word updates of the update-based protocols were extremely inefficient. The updates created significant congestion in the system, and the large number of consumers increased the fence latency since each update had to be acknowledged. The resulting execution times of the update-based protocols using block synchronization were over twice that of the CD-INV protocol, as shown by difference BS in figure 6.7.

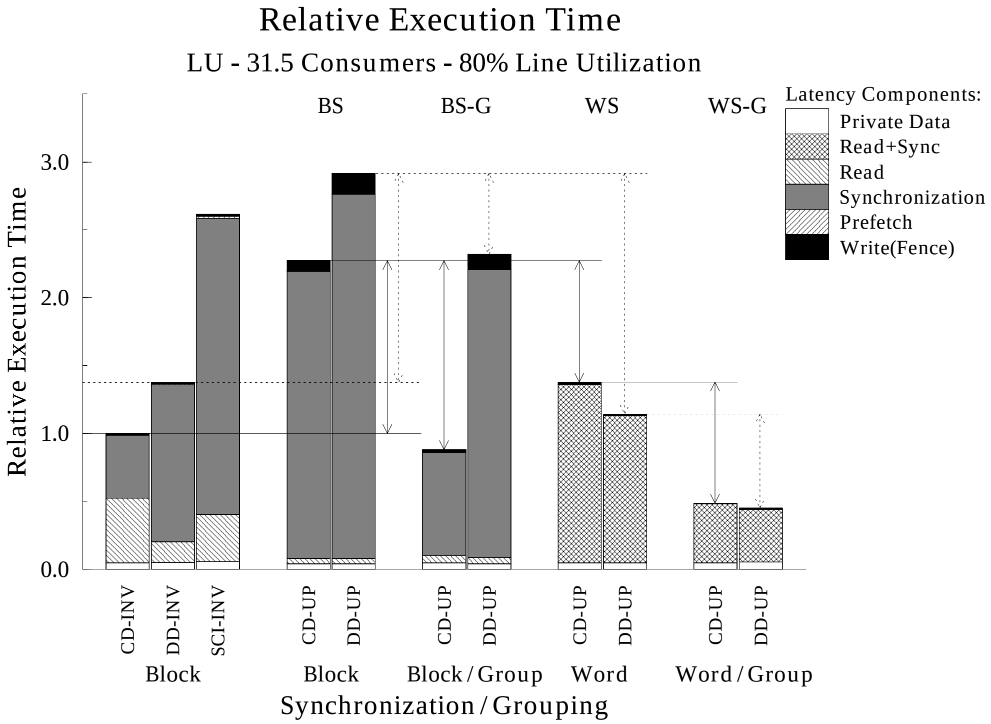


Figure 6.7: Relative Execution Time for LU

The high line utilization allowed for ample write grouping. The fewer grouped update packets decreased the fence and synchronization latencies. The resulting improvement in performance for the update protocols is shown by difference BS-G in figure 6.7. For the CD-UP protocol, the resulting reduction in execution time was significant indicating that congestion dominated the execution time in this case. But for the DD-UP protocol, these reductions did little to improve the execution time indicating that the update latency which resulted from the longer lists of caches dominated the execution time.

The addition of word synchronization eliminated the need for the write fence and allowed the consumption of early data words to hide a portion of the update latency of subsequent words. In the CD-UP protocol, the early consumption helped to hide a large portion of the congested update latency and improve the performance of the protocol by 39%, as shown by difference WS in figure 6.7. For the DD-UP protocol, the word synchronization improved the performance by 60%. The elimination of the fence latency reduced the performance impact of the longer lists of caches. Caches began consuming data as soon as it arrived. Caches at the end of the list only experienced a long wait for the first word of the data to arrive; the subsequent words followed closely behind the first.

The use of write grouping with word synchronization improved performance even more. Write grouping improved the performance of the update-based protocols by 65% and 61% for the CD-UP and DD-UP protocols respectively, as shown by difference WS-G in figure 6.7. In both protocols, the grouping decreased congestion in the network and caches. In the CD-UP protocol, the larger update packets reduced the average memory update latency because the memory access overhead per word decreased with increasing update packet size.

6.6 Summary

The relative performance of the update-based protocols was dependent on the line utilization and the number of consumers. Figure 6.8 summarizes these characteristics for the applications that were studied.

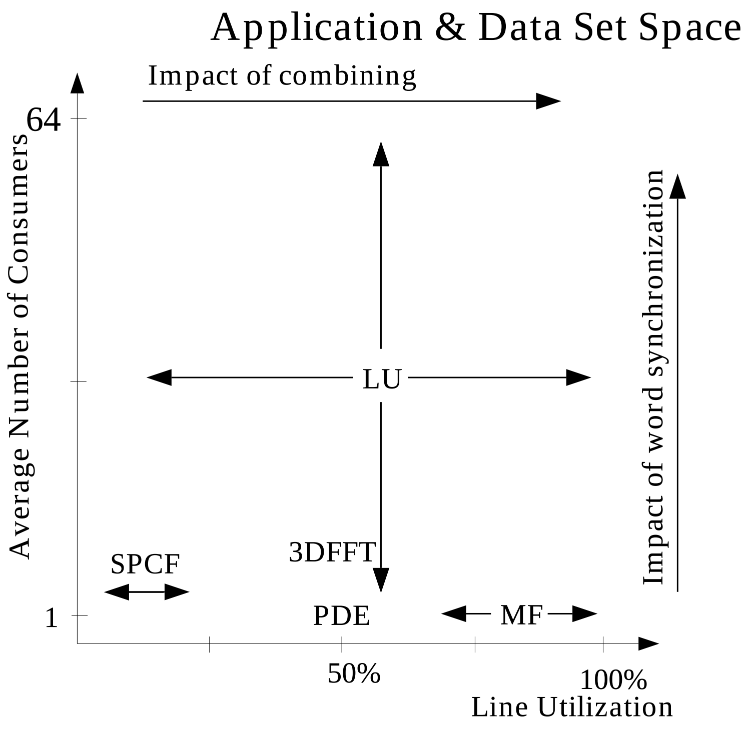

Application & Data Set Space
Impact of combining
64
Average Number of Consumers
Impact of word synchronization
LU
SPCF
3DFFT
1
PDE
MF
50%
100%
Line Utilization

Figure 6.8: Application Space

The line utilization determined the efficiency of the updates. In the applications with a low line utilization, the single word updates were more efficient than the line transfers of the invalidate protocols. But as the line utilization increased, the efficiency of these updates decreased. These inefficient updates tended to congest the network, caches and memories. Write grouping was introduced to address this problem.

Write grouping was able to group single updates into larger update packets, which improved the efficiency of the updates and decreased congestion. This improvement in efficiency was dependent on the line utilization of the application, as indicated in figure 6.8. When the line utilization was low, the possibility of write grouping was small. But as the line utilization increased, the write grouping scheme was able to group a significant number of updates. These new update packets were as efficient as the line transfers of the invalidate protocols, even for the applications with line utilizations approaching 100%. The improvement in the execution time of the update-based protocols when write grouping was added ranged from 4% to 61% for the block synchronization case; the improvement was largest for the applications with the high line utilization.

Word synchronization eliminated the explicit synchronization semaphores from the applications and allowed the consumers to begin consuming the data as soon as possible. The elimination of the semaphores removed the need for update acknowledgments and for the fence operation. In applications with few consumers, this had little impact on performance. But as the number of consumers increased, the impact of the elimination of the update acknowledgments increased, especially in the DD-UP protocol. In this protocol, the latency of the update acknowledgments and the fence operation was dependent on the number of the consumers, which determined the length of the list of caches that each update had to traverse before being acknowledged. The improvement in execution time of the update protocols ranged from about 10% to almost 40% for the CD-UP protocol and from less than 5% to over 50% for the DD-UP protocol when word synchronization was used compared to the block synchronization case.

The use of both enhancements allowed the update-based protocols to significantly improve the performance of the applications when compared to the invalidate-based protocols; the improvements in execution times ranged from about 15% to over 50% compared to the CD-INV protocol. The applications with both high line utilization and a larger number of consumers benefited the most from the enhancements. Even in applications with high line utilization and a single consumer, which tend to favor the invalidate-based protocols, the update protocols were able to improve the execution time.

With both enhancements, the difference between the CD-UP and DD-UP protocols was small. Write grouping improved the performance of the CD-UP protocol more than the DD-UP protocol because the overhead of updating each memory word decreased with the larger packet size. Word synchronization had the largest impact on the DD-UP protocol. It eliminated the need for update acknowledgments which

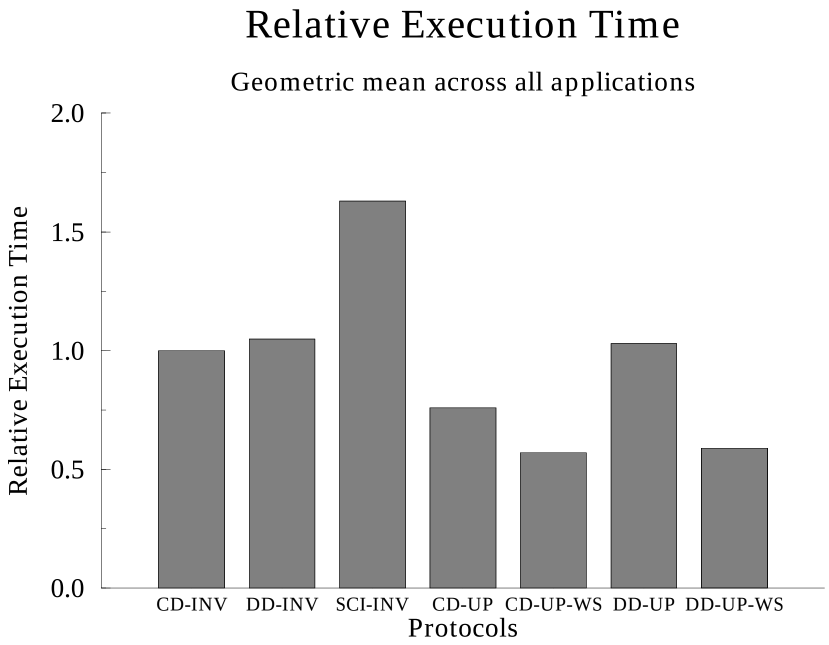


Figure 6.9: Relative Execution Time Averaged Across All Applications

reduced the impact of the length of the sharing list on the performance of the protocol. The choice of which update protocol to use will be dependent on other issues such as the scalability of the directory structure.

Figure 6.9 plots the relative execution time of each protocol, averaged (geometric mean) across all 5 scientific applications, compared to the base CD-INV protocol. For the invalidate-based protocols, the CD-INV protocol slightly outperformed the DD-INV protocol, and both of these performed significantly better than the SCI-INV protocol. For the update-based protocols, the graph plots the performance of the protocols with the write grouping scheme for both the block (CD-UP and DD-UP) and word (CD-UP-WS and DD-UP-WS) synchronization cases. For the block synchronization case, the performance of the CD-UP protocol averaged about 24% better than the base CD-INV invalidate-based protocol, but the update latency of the DD-UP protocols limited the overall performance of the protocol; the performance of the DD-UP protocol was similar to the DD-INV protocol. The use of the word synchronization scheme and write grouping allowed the update-based protocols to improve the performance of the applications by about 40% averaged across all applications.

Chapter 7

Sensitivity of Protocols

This chapter examines the sensitivity of the protocols to both variations in architectural parameters and to shared-memory applications with migratory data. In each case, only the specified parameter is changed. All others remain constant.

7.1 Architecture Parameter Sensitivity

The actions required by the cache coherence protocols to maintain consistency directly affects the latency of shared data accesses. These actions can be divided into three basic operations: accessing cache line state and data, accessing directory state and memory data, and sending protocol messages between nodes to change the state of cache and memory lines at other nodes in the system.

There are several architectural parameters which directly affect the performance of the protocols. The first parameter is the cache access time which affects the access time of the cache line state and data. The second closely related parameter is the directory access time, and the last set of parameters is the network bandwidth and latency. These affect the latency of network transactions performed by the protocols.

In this chapter, the sensitivity of the cache coherence protocols to these architectural parameters is examined. The study will include the two update-based protocols: the centralized directory protocol (CD-UP) and the distributed directory protocol

(DD-UP). Both update-based protocols use the hardware-based write grouping described in chapter 5. The study also examines the three invalidate based protocols: the centralized directory protocol (CD-INV), the singly-linked distributed directory protocol (DD-INV) and the doubly-linked distributed directory protocol (SCI-INV). The performance of the update-based protocols is examined for both the block (CD-UP and DD-UP) and word synchronization (CD-UP-WS and DD-UP-WS) schemes. The invalidate-based protocols assume a block synchronization scheme.

The graph in each section plots the geometric mean of the relative execution time for each protocol averaged across the five scientific applications (MF, PDE, SPCF, 3DFFT and LU). The relative execution times for each application are given in the tables in appendix C.2.

7.1.1 Cache Access Time

This section examines the sensitivity of the protocols to changes in the cache access time for both state and data accesses. Increasing the cache access time will increase the latency of processor load and stores that access the cache, and it will also increase the time for the cache controllers to process coherence operations from remote nodes that may access cache line state or data.

In the base model, each node has a single level cache with a one cycle access time for both state and data accesses. If the cache is busy and the processor has a request, the processor is blocked. The graph in figure 7.1 shows the relative execution time of the protocols compared to the base CD-INV protocol when the cache access time is increased from one to five cycles.

For the invalidate-based protocols, the increase in cache access time affects the distributed directory protocols more than the centralized-directory protocol. The DD-INV and SCI-INV protocols satisfy most miss requests by fetching data from caches rather than from memory as the CD-INV protocol does. The distributed directory protocols also require more coherence operations at the cache since the directory is distributed among the caches, and the SCI-INV protocol is even more sensitive than the DD-INV protocol because it requires more coherence operations at the cache to maintain the doubly-linked lists. These two characteristics result in the distributed directory protocols being more sensitive to the cache access time than the CD-INV protocol.

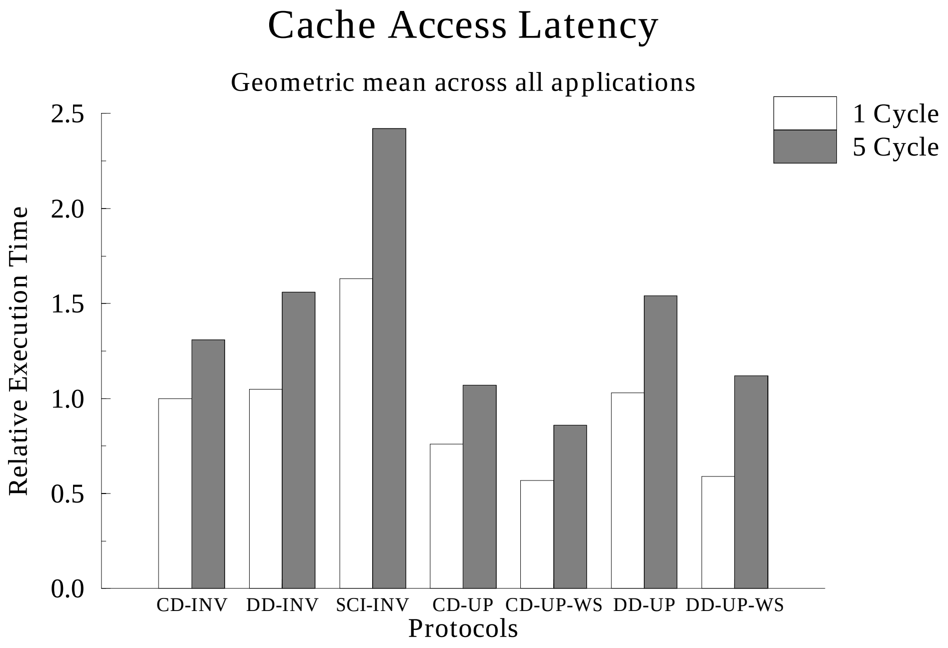


Figure 7.1: Sensitivity to Cache Access Time

For the update-based protocols, the distributed directory protocol (DD-UP) is more sensitive to the cache access time than the CD-UP protocol. The reasons are the same as with the invalidate-based protocols, but the DD-UP protocol is also more sensitive because the data updates must traverse the linked list of caches before being acknowledged. In the distributed directory, invalidate-based protocols, the increased latency of the invalidation operations, which must also traverse the linked list, was often hidden by write prefetching.

Increasing the cache access time actually reduced the overall system congestion in many cases. By increasing the cache processing time, the rate that the cache injected new requests into the system also decreased. This decrease resulted in a lower offered bandwidth which actually reduced the latency of other operations in the system. These reductions tended to offset a portion of the increased cache access time.

Overall, the distributed directory protocols were more sensitive to the cache access time. These protocols required more coherence operations at the cache, and they tend to fetch data from the cache more often than the centralized directory protocols. The increase in cache access time directly affected both of these operations.

7.1.2 Directory Access Time

This section examines the sensitivity of the protocols to changes in the directory access time. Increasing the directory access time has the greatest impact on the protocols which require more directory accesses.

In the base model, the directory is modeled as a single cycle access static RAM, and the data accesses were from synchronous DRAMs. Figure 7.2 shows the relative execution time of the protocols compared to the base CD-INV protocol averaged across all applications when the directory access time is increased from one to five cycles. The five cycle access time might approximate a system in which the directory entries were cached in a fast SRAM and the main memory was used to hold the uncached directory entries. An average of five cycles would result in such a system if the cached directory hit rate was 50% and the memory access time was 9 cycles.

For the invalidate-based protocols, the impact of the increased directory access time was minimal, but the impact was largest for the CD-INV protocol. This protocol requires more coherence operations at the directory since the entire directory is stored there. The SCI-INV protocol was also impacted by the longer directory access time since it requires many operations at the directory even though the directory is distributed among the caches. The DD-INV protocol was not affected by the increase in directory access time.

For the update-based protocols, the increase in directory access time had a negligible affect on the execution times. Data prefetch allowed a significant portion of miss request latency to be hidden behind other useful work. When the directory was accessed for updates, the increase in the directory access time was a very small portion of the total update latency.

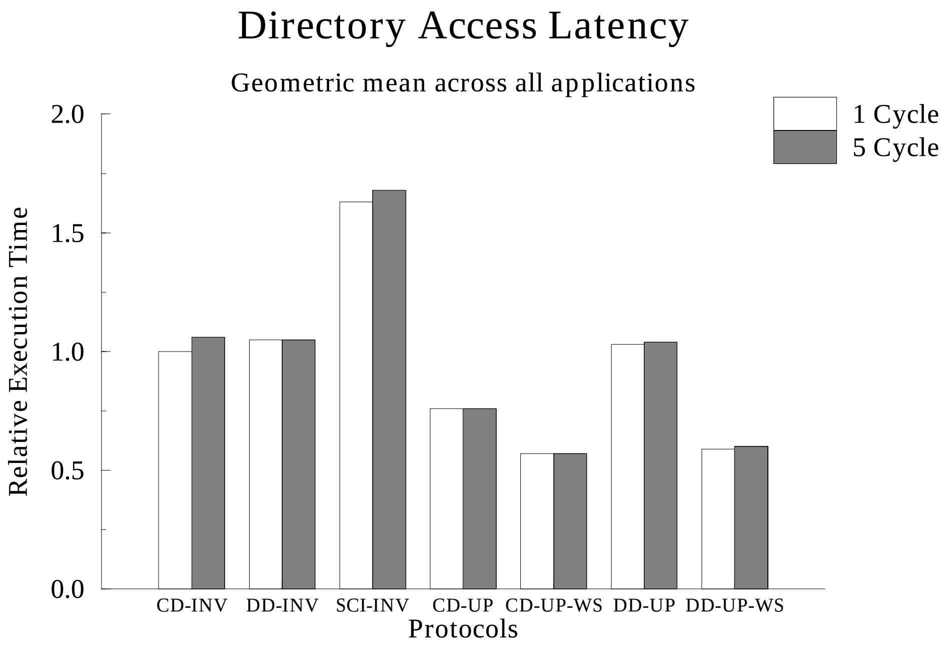


Figure 7.2: Sensitivity to Directory Access Time

Overall, increases in the directory access time had a minimal impact on the performance of the protocols. The only protocols to show a noticeable increase in execution time were the CD-INV and SCI-INV protocols. These protocols tended to access the directory more during the critical path of the applications. The increase in directory access time also decreased the rate of coherence operations, such as invalidates and updates, from the directory in the centralized directory protocols. This decrease actually decreased the offered bandwidth slightly which allowed for better utilization of the network and local bus.

7.1.3 Network Bandwidth

In this section, the sensitivity of the protocols to the network bandwidth is examined. The sensitivity of the protocols was dependent on the amount of traffic generated by each protocol. When the offered bandwidth of a protocol was much less than the network bandwidth, the time for a packet to traverse the network tended to be limited by the network latency. When the offered bandwidth was close to the network bandwidth then the congestion created by the traffic dominated the time required for a packet to traverse the network. The increase in this latency tends to be exponential with respect to the offered bandwidth.

In the base model, the network and local bus are a word (32 bit) wide and run at 100Mhz for a bandwidth of 400MB/s per link. Figure 7.3 shows the performance of the protocols when the bandwidth was reduced by a factor of 2 or increased by a factor of 2.

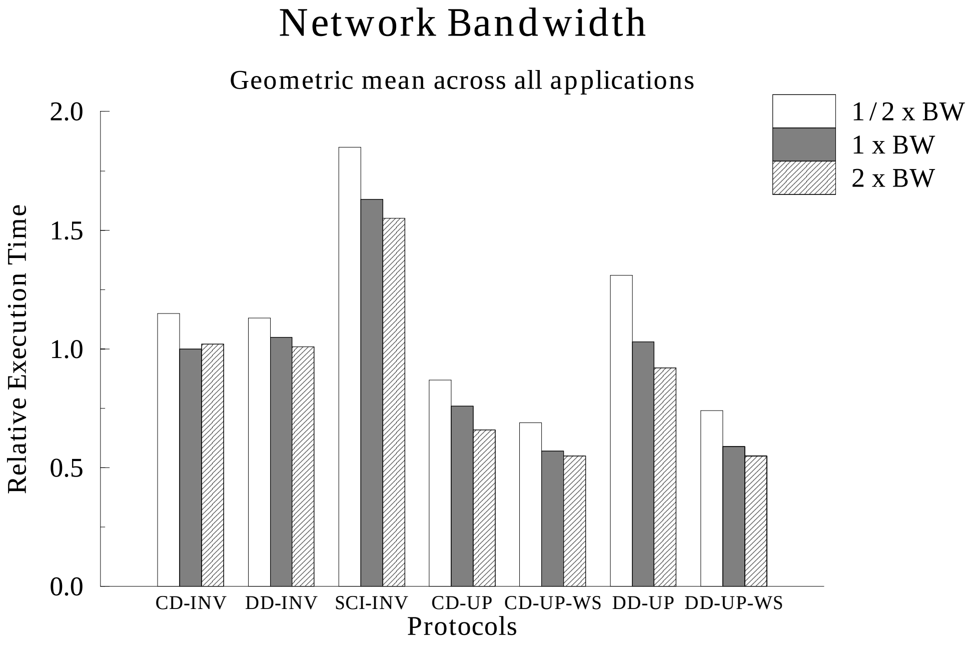


Figure 7.3: Sensitivity to Network Bandwidth

For the invalidate-based protocols, the performance of the protocols tracks the amount of traffic generated by each protocol. The SCI-INV protocol was the most sensitive to the changes in the network bandwidth as it generated the most traffic. The CD-INV protocol generated slightly less traffic and was, therefore, less sensitive to changes in the bandwidth. When the bandwidth was doubled for the CD-INV protocol, the execution time of the protocol (averaged across all applications) actually increased. The increase came from the LU application (See table C.4). In this application, the increased bandwidth actually allowed more consumers to request the data before it was produced. This required more invalidations and created more congestion as the consumers competed for the data. Finally, the DD-INV protocol, which generated the least amount of traffic, was the least sensitive to changes in the network bandwidth. Doubling the bandwidth had a small impact on the performance of this protocol.

For the update-based protocols, the DD-UP protocol was more sensitive to changes in the network bandwidth than the CD-UP protocol. The DD-UP protocol generated more update traffic since it did not use multicasts to send updates. For the block synchronization case, the update-based protocols were still more sensitive to the network bandwidth than the invalidate based protocols. This indicated that the traffic produced by the protocols still affected performance of the protocol even after write grouping was introduced. In the word synchronization case, the sensitivity was less since the traffic was reduced and portions of the update latency were often hidden behind other useful work.

Overall, the update-based protocols using a block synchronization scheme were more sensitive to changes in the network bandwidth when compared to the better invalidate based protocols, CD-INV and DD-INV. Moving to a word synchronization scheme reduced the sensitivity of the update-based protocols to the network bandwidth.

7.1.4 Network Latency

In this section, the sensitivity of the protocols to changes in the network latency is examined. Changes in the network latency affect the time for coherence operations to traverse the network. Protocols which require many coherence operations will tend to be more sensitive to changes in the network latency.

The base model assumes a network latency of 8 cycles between two adjacent network nodes or between the local bus and the network interface [1]. Figure 7.4 shows the resulting relative execution time of the protocols when the network latency was

[1] The simulator accurately simulated the network traffic.

halved or doubled.

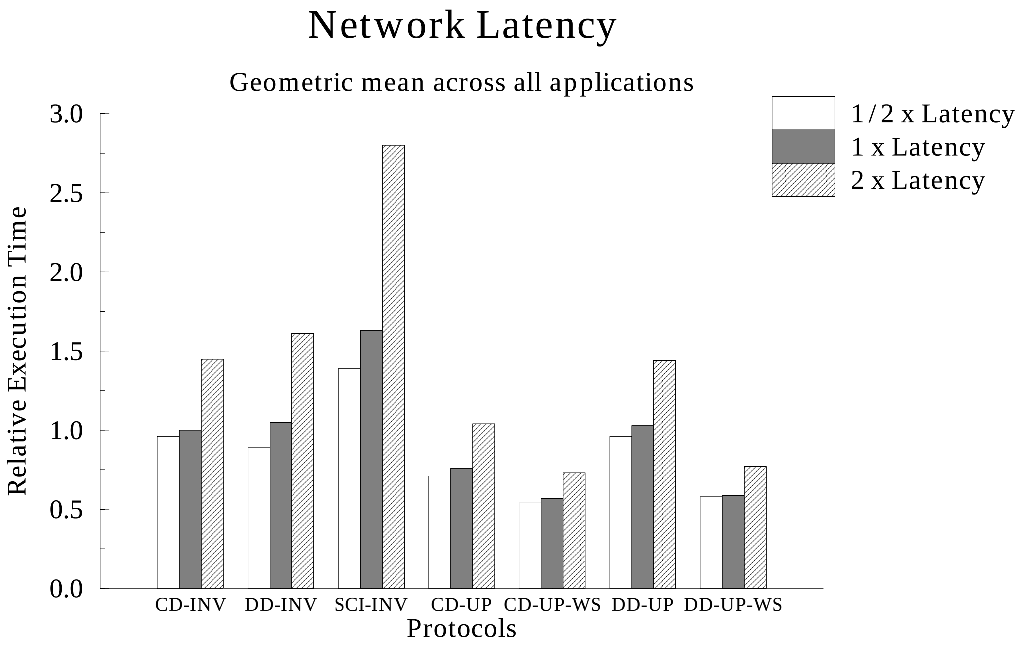


Figure 7.4: Sensitivity to Network Latency

For the invalidate-based protocols, the distributed directory protocols were more sensitive to the network latency than the centralized directory protocols. The SCI-INV protocol was extremely sensitive to the network latency because the protocol required significantly more network hops than the other invalidate-based protocols. The DD-INV protocol was also sensitive to the network latency because of its distributed directory structure, but the impact was less than the SCI-INV protocol since invalidations were pipelined in the DD-INV protocol. The CD-INV protocol was the least sensitive to the network latency. Reducing the network latency had almost no affect on the performance of the protocol which implies that network congestion was dominating the time it took for packets to traverse the network.

The update-based protocols were less sensitive to changes in the network latency than the invalidate-based protocols. The update-based protocols required fewer network hops during the critical path of the applications. The consumers prefetched the data early, and when the data was produced, the producer forwarded the data to the consumers. The DD-UP protocol was slightly more sensitive to the network latency than the CD-UP protocol because the DD-UP protocol required more network hops to send an update packet down the linked list of caches. Finally, the word synchronization scheme was less sensitive to the network latency compared to the block synchronization case as it was able to hide a significant portion of the update latency.

Overall, the proactive nature of the update-based protocols allowed it to be less sensitive to changes in the network latency. The invalidate-based protocols required more hops to obtain data during the critical path of the application. Word synchronization allowed the update-based protocols to hide a portion of the update latency and, therefore, be even less sensitive to the network latency.

7.1.5 Network Multicast

In the base architecture, the network supports multicast packets. The CD-INV protocol uses the multicast to send invalidates to multiple nodes, and the CD-UP protocol uses multicast to send data updates to multiple nodes. This section examines the sensitivity of the protocols to availability of multicast.

Figure 7.5 shows the performance of the centralized directory protocols with (base case) and without multicast averaged across the applications which have multiple consumers (SPCF, 3DFFT and LU). The CD-INV gains little from multicast. For the CD-UP protocol, the multicast is essential for the 3DFFT and LU application. In these applications, network traffic is a problem, and multicast decreased the total traffic and reduced the generation time of the update packets. For the block synchronization case, multicast resulted in an average improvement of about 30% for the CD-UP protocol, and for the word synchronization case, the average improvement was slightly less, 25%.

The sensitivity of the centralized directory protocols to the multicast would be less if the performance was averaged across all 5 scientific applications because the execution time of the protocols with respect to the MF and PDE applications is identical with and without multicast.

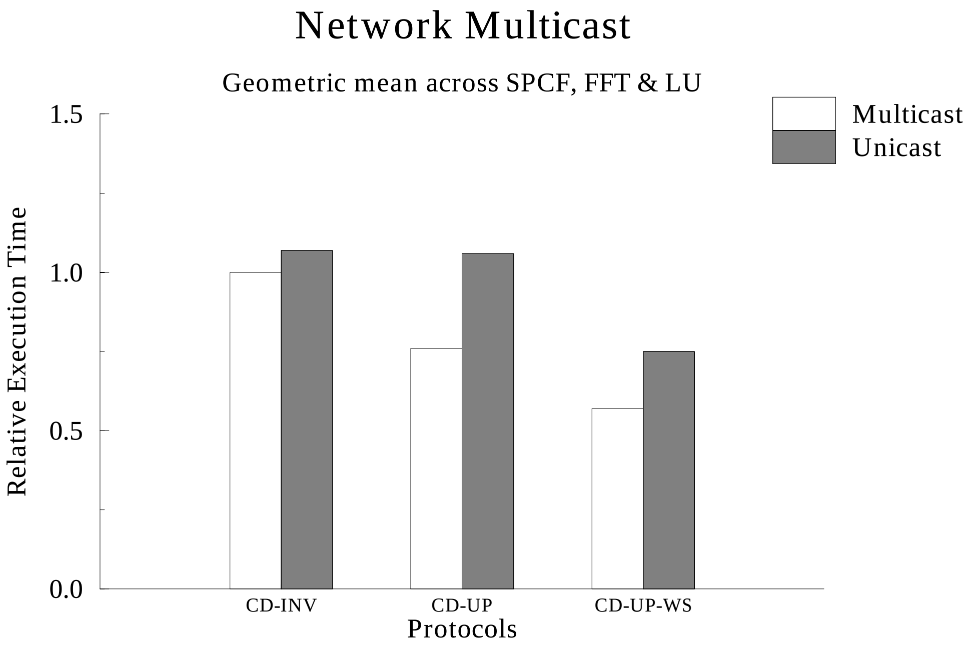


Figure 7.5: Sensitivity to Network Multicast

7.1.6 Load/Store Cycles

In the base architecture, the processor is assumed to be load/store limited. All non-load/store operations are assumed to execute in parallel with load and store operations. This section examines the sensitivity of the protocols to the relaxation of this assumption.

Figure 7.6 shows the performance of the cache coherence protocols for various load/store cycle counts. The load/store cycle count is the number of cycles charged for each processor load or store. The base case assumes one cycle per processor load/store. The additional cycles represent the time to execute other non-load/store instructions.

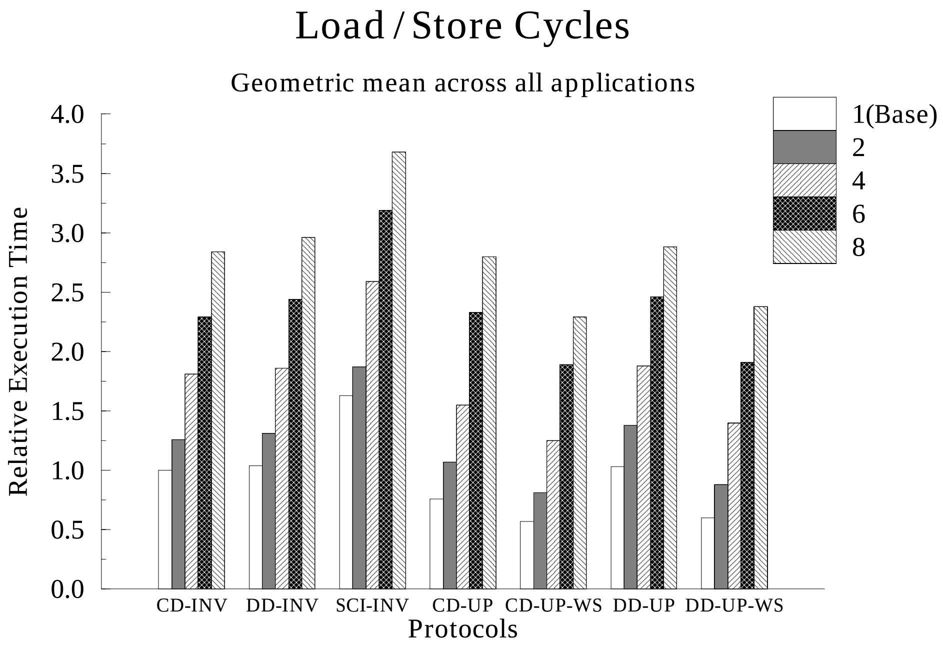


Figure 7.6: Sensitivity to LDST Cycles

As the load/store cycle count increases, the absolute difference between the protocols remains approximately the same, but this difference becomes a smaller portion of the total execution time.

For the invalidate-based protocols, the CD-INV protocol always performs slightly better than the DD-INV protocol and significantly better than the SCI-INV protocol, but the relative difference in the total execution time drops with increasing load/store cycle count.

For the CD-UP protocol, the absolute improvement in performance compared to the CD-INV protocol does not remain constant. The main reason for this is the increasing execution time of the LU application as the load/store cycle count increases (See table C.7 in appendix C). As noted in section 7.1.3, the LU application is not stable. As the production of data slows, more consumers are able to express their interest in the data before it is produced in the LU application. For the CD-UP protocol this results in more update traffic and longer update latencies as the load/store cycle count increases. Once the load/store cycle count reaches 6, the extra traffic is enough to significantly increase the execution time of the application. The increase is large enough to increase the average execution time of the CD-UP protocol across all applications compared to that of the CD-INV protocol. But once word synchronization is introduced, the extra traffic and update latency can be tolerated. In this case, the absolute performance gain of the CD-UP-WS protocol is almost constant across all load/store cycle counts.

The DD-UP protocol with block synchronization performs about the same as the DD-INV protocol across all load/store cycle counts. For the DD-UP protocol with word synchronization, the absolute improvement also remains almost constant with increasing load/store cycle count.

Overall, the percentage decrease in execution time from the update-based protocols is dependent on the total percentage of the execution time spent accessing shared data. If the majority of an application's execution time is spent computing or accessing private data, then the choice of cache coherence protocol will have little impact on the total execution time. But, as processors become more superscalar and load/store limited and systems increase in size, the latency of shared data accesses will be a significant portion of the total execution time. Thus, the selection of a cache coherence protocol will be an extremely important choice.

7.1.7 Summary

Studying the sensitivity of the protocols to architectural parameters is difficult. When a parameter is changed, the resulting actions taken by a protocol may be different. For example, when the network bandwidth was increased in the CD-INV protocol, the execution time actually increased because the change allowed more consumers in the LU application to request the data before it was produced. Changes also affected the performance of other parts of the system. The best example is how increases in the cache and directory access times actually reduced the latency of other packets through the network. The increase in the cache and directory latencies reduced their offered bandwidth back into the system, and in the cases where the backup at the cache and directory did not spill over into the rest of the system, the reduction in the traffic rate resulted in faster packet transfer in the remainder of the system.

But overall, the results follow what was expected. The distributed directory protocols were more sensitive to increases in the cache access time. The centralized directory protocols were more sensitive to the directory access time, although the sensitivity was minimal. The update-based protocols were more sensitive to the network bandwidth compared to the CD-INV and DD-INV protocols. The extra coherence operations generated by the SCI-INV protocols made it extremely sensitive to both the network bandwidth and latency. The update-based protocols were less sensitive to the network latency because they required fewer hops to transfer data between the producer and consumer. Word synchronization reduced the sensitivity of the update-based protocols to the network parameters. The scheme reduced the amount of traffic generated and allowed early consumption of data to hide portions of the update latency of subsequent words.

7.2 Migratory Data

Migratory data is data that migrates from processor to processor during program execution. A good example is a database. In a database, a particular piece of data may be required by many different processors. Assuming that the total data working set fits into the processor caches, then each piece of migratory data may eventually reside in almost all caches. In an invalidate-based system, each time the data is modified all other cached copies of the migratory data will be invalidated (or purged) from the remote caches. If the modification rate is high, the number of migratory copies will remain small. But in update-based protocols, there is no mechanism to purge the migratory copies. When a processor modifies a piece of migratory data, all other copies must be updated. After a long enough period of time, a piece of data may reside in all caches. Now each write must update all copies. The resulting traffic and congestion would result in extremely poor system performance.

In this section, the performance of update-based protocols is examined for applications with migratory data. First, the performance of the MP3D migratory application is examined. The low rate of data migration in MP3D motivates the need for the

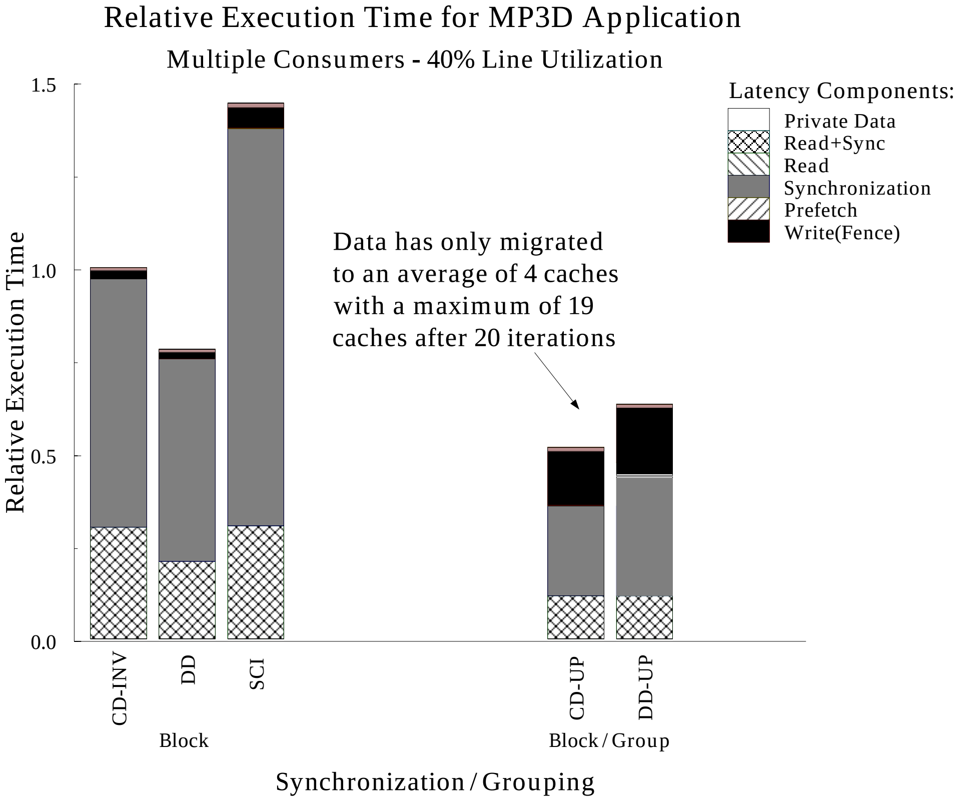


Figure 7.7: Relative Execution Time for MP3D

synthetic migratory data application (TASK). The details of both applications are presented in appendix B. The performance analysis of the protocols demonstrates the difficulties that update-based protocols have with migratory data, but a simple replacement scheme is presented which allows systems with update-based protocols to purge migratory data.

7.2.1 MP3D Application

Figure 7.7 shows the relative execution time of the protocols for the MP3D application. The graph shows that the update-based protocols using block synchronization perform significantly better than the invalidate-based protocols, but from the discussion in the previous section, the extra traffic created by the updates of the migratory data should significantly degrade the performance of the update-based protocols. The problem is that for this application, the data migration rate is too low to be simulated within a reasonable amount of time. The average rate of migration after 20 iterations was only 4 caches with a maximum of 19. This amount of migratory data was too small to have much impact on the performance of the update-based protocols.

7.2.2 Synthetic Task Application

To examine the impact of migratory data on update-based protocols the rate of data migration must be extreme with respect to the cache line replacement rate and the total amount of migratory data must also be a large fraction of the of shared data. Therefore, a synthetic application that could be simulated significantly faster than MP3D, but still had the same basic characteristics of MP3D, was required. The details of the application are presented in Appendix B.

Figure 7.8 shows the performance of the centralized directory invalidate-based protocol (CD-INV) and the two update-based protocols CD-UP and DD-UP with the hardware-based write grouping. The x-axis is the step number of the iteration of the application, and the y-axis is the relative execution time compared to the execution time of the CD-INV protocol for the first time step. The graph demonstrates the performance problems of the update-based protocols for applications with migratory data.

For the DD-UP protocol, the execution time increased almost exponentially with the step number. During each step of the application, the shared data migrated throughout the system. In this distributed directory protocol, the length of the shared list of cache increased with each new cache that the data migrates to. After only a few steps, the update latency dominated the execution time. At about the 200th step, the average update had to traverse a list of over 50 caches. The resulting execution time was more than 6 times that of the CD-INV protocol. For the CD-UP protocol, the performance loss was not as severe as the DD-UP protocol. In the CD-UP protocol, the updates are sent out in parallel. In this case, the extra updates did not directly increase the latency of the updates, rather they increased the traffic and network congestion. After 500 steps, the CD-UP protocol took almost twice as long to execute as the CD-INV protocol.

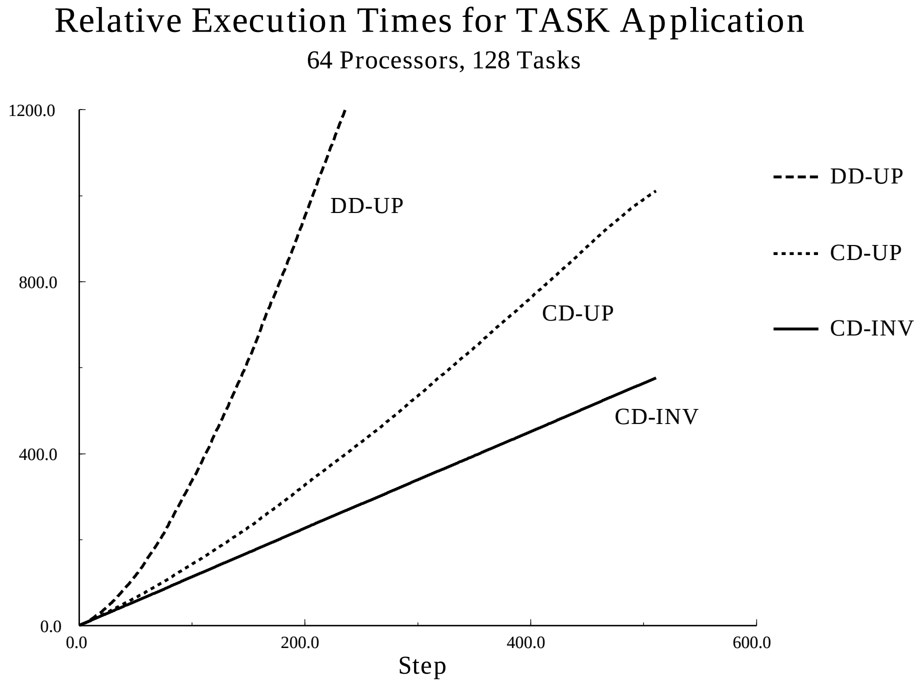


Figure 7.8: Relative Execution Time for TASK Application

Replacement Scheme

To improve the performance of the update-based protocols for applications with migratory data, a scheme to purge migratory data is required. The scheme should mimic the invalidate-based protocol's inherent ability to purge the migratory data from remote caches whenever a cache modifies the data.

A scheme similar to the one used in adaptive protocols on bus based systems can be used [46]. First, a small counter is added to each cache line. Every time the cache receives an update packet, the counter is incremented. Every time the local processor accesses the cache line, the counter is reset. Once the counter reaches it maximum count, the line is replaced out of the cache as if it were being replaced due to a conflict miss in the cache. Currently, the maximum counter is set to four. If the cache line

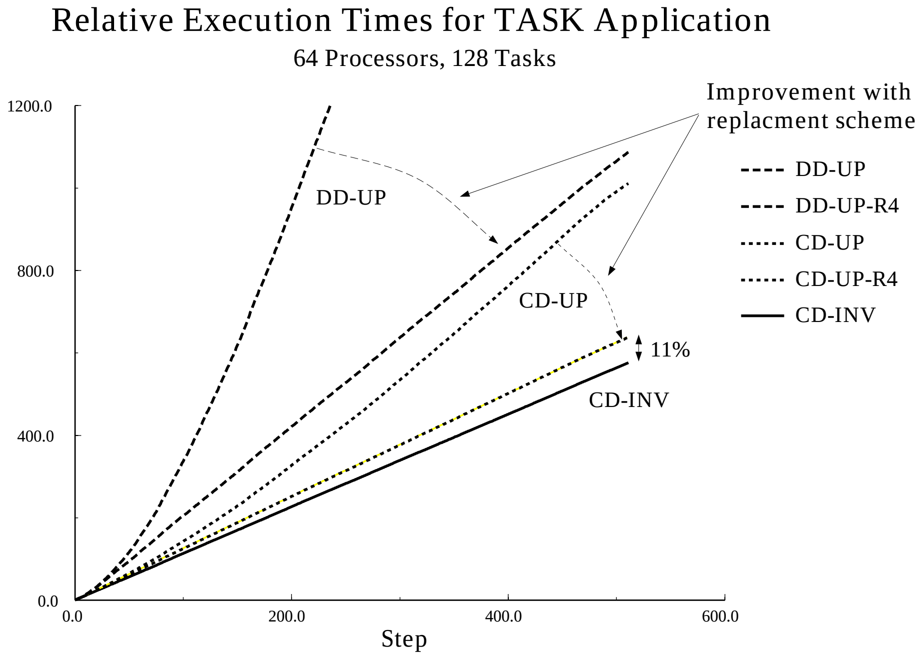


Figure 7.9: Relative Execution Time for TASK application

receives four updates without any local processor accesses, the line will be replaced. The update-based protocols use the write grouping scheme presented in chapter 5. Therefore, each update is a grouped update of many words. The scheme assumes no knowledge of the application access pattern. A line could be replaced a cycle before the local processor decided to use it.

Figure 7.9 shows the performance improvements when the replacement scheme is added to the update-based protocols. For the DD-UP protocol the improvement is significant. Purging the migratory data significantly reduced the length of the list of cache that each update must traverse. As noted above, this latency dominated the execution time of the application when this protocol was used. For the CD-UP protocol, the improvement in performance was not as significant. The extra traffic introduced by the migratory data increased the network congestion. Once the traffic level was decreased to the point that congestion was not a problem, then the performance of the CD-UP protocol could not be improved significantly by purging

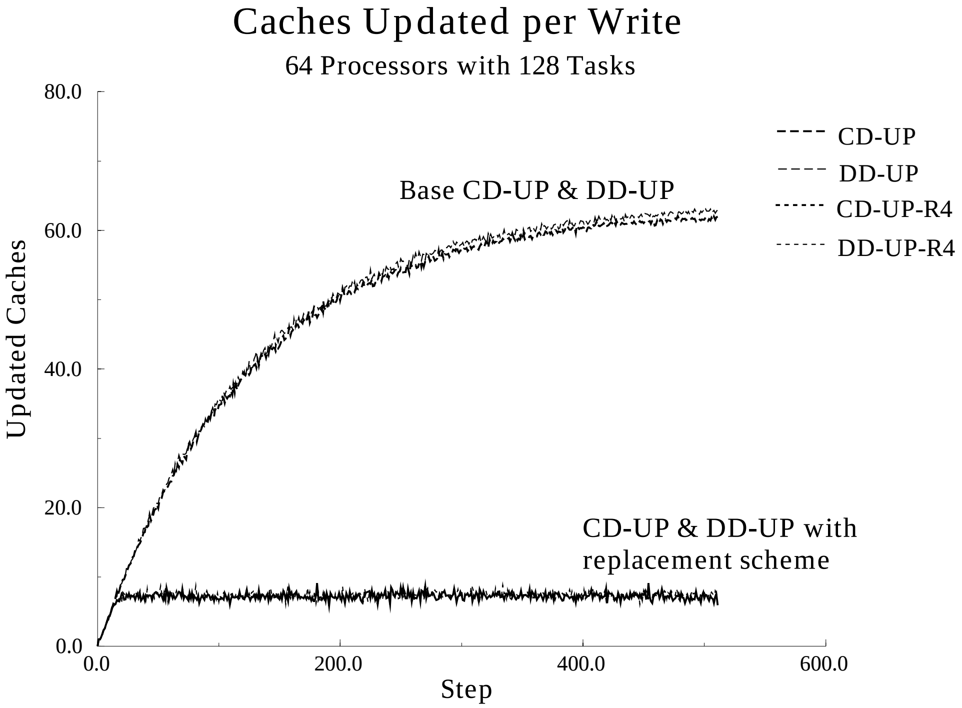


Figure 7.10: Cache Updates per Iteration for TASK Application

more copies of the migratory data.

Figure 7.10 shows the average number of caches updated on each write for each iteration. For the update-based protocols without the replacement scheme, the number of updates quickly approaches the maximum number of 63 caches. Thus, this application is indeed a worst case migratory application because every cache is being updated on each write during the last 100 or so iterations.

The figure also demonstrates the ability of the replacement scheme to limit the number of updates. With a replacement after every four updates and a 50% probability of each consumer modifying the line, the number of caches updated tended to remain around eight caches. Increasing the probability of a write to 100% would reduce the average to four, but the total update traffic would stay about the same. Every iteration a write would update four caches where in the original case a write would update eight caches every other iteration. As the write rate drops even further, the total update traffic would tend to stay about the same as more cache were updated less frequently.

Even with the replacement scheme, the CD-INV protocol still outperforms the update-based protocols. The CD-UP protocol came within 11% of the CD-INV protocol. In this synthetic application, the data migration and write rate were high. In the invalidate-based protocols, each data write would purge all other copies. The next processor to access the data would fetch the data from the writer's cache. With a 50% chance of modifying the data, the average number of readers between writers was one. Therefore, the CD-INV protocol operated very efficiently for this type of application.

For the update-based protocols the replacement count had to be greater than one. If it were one, then the line would be replaced on the first update. The update would be acting as an invalidation of the line, but the extra traffic would result in the update acting as a rather inefficient invalidation. Based on simulation results not presented here, a replacement count of four was deemed to be the smallest count that could be used with a variety of migratory applications without adversely affecting performance of those applications with a lower migratory data write rate.

Overall, an artificial replacement scheme may allow update-based protocols to perform much better for applications with migratory data. But for applications with a significant amount of migratory data, efficient invalidate-based protocols such as the CD-INV protocol will tend to perform better. As the amount of migration decreases and the number of unnecessary updates decrease, update-based protocols will improve the performance of the applications when compared to the invalidate-based protocols. The MP3D application examined in the last section is a typical example. In this case the gains from update-based protocols outweighed the performance loss from the less frequent migratory data updates.

Chapter 8

Conclusions and Future Work

A cache coherence protocol must provide a correct, low-cost and efficient implementation of a shared-memory abstraction for a multiprocessor system. For scalable systems, the protocols require a directory structure and the ability to send protocol level messages between nodes. This work has demonstrated that update-based protocols can be designed to be correct, low-cost and more efficient than invalidate-based protocols for a set of fine-grain scientific applications.

8.1 Correctness

The correctness of a protocol with respect to a memory consistency model is an absolute. Essentially, a correct protocol will guarantee that the processors have a consistent view of memory, where the exact meaning of consistent is determined by the memory consistency model used. The sequential consistency model requires that all memory accesses appear atomic. This model forces a total ordering on a set of distributed writes [52]. The weaker memory consistency models require that each node only maintains the ordering of accesses that they issue [38, 22, 33]. These weaker models simplify the protocols.

The protocols must also be deadlock free. Protocol level deadlock results because of finite buffering in the system. Two possible solutions were described in chapter 3. The first scheme required a timeout mechanism and packet exceptions to avoid deadlock. As demonstrated in the description of the update-based protocols, the exceptions may add complexity to the protocols. The other solution, extending the queues into local memory, does not increase the complexity of the protocols, but may affect performance as the number of outstanding requests must be limited to avoid overflowing the additional buffer space preallocated in memory.

The protocols must also be free of livelock. Livelock results when a protocol level request is continually circulated in the system and forward progress is not maintained. In the protocols described in this work, livelock may occur when a request is bounced back to the sender to be retried. The request may be bounced back every time the destination receives it because some other intervening request changed the state of the line. Although very unlikely, the occurrence is still a possible source of problems.

The directory structure may also add to the complexity of the protocols. In the centralized directory scheme, a request sent between a directory and cache will always traverse the same path, assuming an in-order network. In the distributed directory scheme, the path between the directory and a cache is only stable when the list is stable. Every time the list changes, the path between the directory and each cache may also change. The design of the DD-UP protocol in chapter 3 demonstrated the additional complexities that a distributed directory may add to the protocol.

Overall, the simpler a protocol is, the more likely it is correct. As was demonstrated in section 3.5, validating the correctness is an extremely difficult problem. Both the deadlock avoidance scheme and the directory structure directly impact the complexity of a protocol.

8.2 Cost of Protocol Implementation

Another important characteristic of a cache coherence protocol is its implementation cost. The cost of implementation is difficult to measure. The cost of the directory structure is one component of the implementation cost that has been examined by many researchers [4, 39, 14, 62]. The proponents of the distributed directory protocols argue that it is a more efficient implementation of a directory structure as the size of the system increases. Those in favor of centralized directory protocols have presented several techniques, described in section 3.1, to reduce the cost of a fully-mapped centralized directory scheme. The scalability and cost of the directory structures is still an open research question.

The complexity of the protocol can also influence the cost of implementation. The more complex a protocol is, the more states and message types it may have. This could easily increase the complexity of the state machines and associated bit fields.

Unfortunately, implementation cost is not a measure that can easily be included in the performance analysis of the protocols. As with the correctness of the protocol, the costs of the implementations can be discussed and even quantified in some cases, but the resulting cost versus performance curve is still difficult to describe.

8.3 Efficiency of Protocols

The final requirement of a protocol is that it efficiently maintains a consistent view of memory. The efficiency can be measured by the amount of work, or number of transactions, required to maintain consistency among the caches and memory.

In the applications examined, a producer produces a block of data that was shared by one or more consumers. In the block synchronization case, the producers used a simple synchronization event, such as a flag, to indicate that a given block of data was available. Once the consumers see the flag change, they begin consuming the data. In the word synchronization case, the synchronization information was combined with the data word.

For such applications, invalidate-based protocols are reactive protocols. Assuming that the consumers are eagerly awaiting the data, they will react to the producer's invalidation of the synchronization flag. Each consumer will acknowledge the invalidation and each consumer will attempt to read the data block. In the centralized directory protocol (CD-INV), the directory/memory may become a hot spot if multiple consumers request the same memory lines. In this protocol, the producer of the data block cannot release ownership of a line to a requester unless all invalidations have been acknowledged. If they have not been, then the consumer's request is bounced back to the consumer to be retried. The distributed directory protocols address this problem by distributing the responsibility of satisfying the miss requests to the previous cache that requested the line. If the line is not available, the request is queued. It is not bounced back to the requester.

Conversely, update-based protocols are proactive protocols. A consumer is able to express an interest in a block of data by prefetching the data (actually each line in the data block). Then when the block is written, the producer forwards the data to the consumers; a more controlled distribution of data. These prefetches may result in more network traffic, but the data prefetch can be issued far in advance of the data production. The combination of early data prefetch and updates may significantly reduce the protocol actions required during the critical path of the application compared to the invalidate-based protocols.

The efficiency or performance of a protocol is also dependent on how well other latency reducing and tolerating techniques can be used in conjunction with the protocol. Data prefetch was effective for both classes of protocols. The invalidate-based protocols could use write-prefetching to hide a significant portion of any data invalidation latency, but the protocols could only use read prefetch to overlap read requests. Each consumer was forced to wait until the synchronization point was reached before issuing the data prefetches. As noted above, data prefetches allowed consumers in the update-based systems to express an interest in data before it was produced. In these cases, the prefetch acted as a notification to the producer. Both protocols classes could take advantage of the relaxed consistency memory models. The models allowed the protocols to overlap requests to hide a portion of the data access latency.

Efficiency of the protocols can also be measured in their sensitivity to system parameters, as demonstrated in chapter 7. The most important observation was the dependency of the invalidate-based protocols on the network latency. When the latency was doubled, the performance of the protocols suffered because the cost of the extra network traversals began to dominate the execution time. For the update-based protocols, the write grouping scheme reduced the network traffic, but the performance of the protocols still improved with increasing network bandwidth. This improvement indicates that traffic was still a problem, but the addition of write grouping and word synchronization resulted in update-based protocols that were able to reduce and tolerate a significant portion of the update latency.

The sensitivity to the characteristics of the applications is also important. The applications were characterized by the line utilization and the number of consumers. The efficiency of the updates were originally determined by the line utilization. For low line utilizations, single word updates were more efficient than line transfers. For high line utilization, the line transfers were more efficient. But once write grouping was introduced, the grouped writes were always more efficient than line transfers. The number of consumers had the largest impact on the distributed directory protocols. In the distributed directory, invalidate-based protocols, a portion of the write invalidation latency, which was dependent on the length of the shared cache list, could be hidden. But in the distributed-directory, update-based protocol (DD-UP), the update latency dominated the execution time of the application when the list of caches was long. The introduction of the word synchronization scheme reduced this dependency because consumers could begin consuming data as soon as it arrived rather than waiting for all updates to propagate through the list before the synchronization event could be reached.

The TASK application with migratory data demonstrated how unnecessary updates can reduce the efficiency of the update-based protocols. In the CD-UP protocol, the extra updates resulted in more network traffic, but the extra updates in the DD-UP protocol resulted in a longer execution time since each update had to traverse a continually growing list of caches. The suggested replacement scheme was able to limit the number of sharers of each data word and, therefore, the number of updates required. In the CD-UP protocol, this reduced the traffic enough to allow the performance of the protocol to approach that of the CD-INV protocol, but even a modest list of caches still prevented the DD-UP protocol from performing well.

8.4 Summary

Overall, this dissertation has demonstrated that a correct, low-cost and efficient update-based protocol could be designed such that it could significantly outperform the best high-performance invalidate-based protocols. The CD-UP protocol offered the simplest and most efficient protocol, but the actual cost of the directory is still open to debate. If schemes to limit the size of the directory, such as a sparse directory, perform well across a wide range of application, then the CD-UP protocol is the best performing protocol of the five protocols examined.

An update-based protocol is essentially another latency tolerating and reducing technique. The proactive nature of the protocol allows the programmer to reduce the latency to obtain shared data. But as with other latency tolerating techniques, it must be used correctly to obtain the possible gains in performance. Simply changing the underlying protocol in a system will not result in the performance gains observed in this work. The consumers must prefetch the data or no data updates will occur. A write grouping technique must also be employed to reduce the network traffic generated by the protocols, and a word synchronization scheme is essential if the system is to take full advantage of the updates. The application must also minimize unnecessary data updates which result from false sharing or intermediate results.

8.5 Future Work

Future work should look at expanding the application set and examining other node architectures. This dissertation examined a set of five fine-grain scientific application. The analysis should be expanded to include other classes of applications including commercial applications. The system node used to simulate the protocols was a loosely coupled cache, memory and network interface connected by two logical networks (reply and request). Future work might examine the performance of the protocols on a more tightly integrated system such as the architecture of the Alewife [3] or Flash [50] systems. These systems combine the cache and directory controller into a single unit with a buffered interface to the memory. The system should also explore how the different protocol deadlock avoidance schemes impact the performance of the protocols.

Appendix A

Scientific Applications

A.1 Multifrontal Solver (MF)

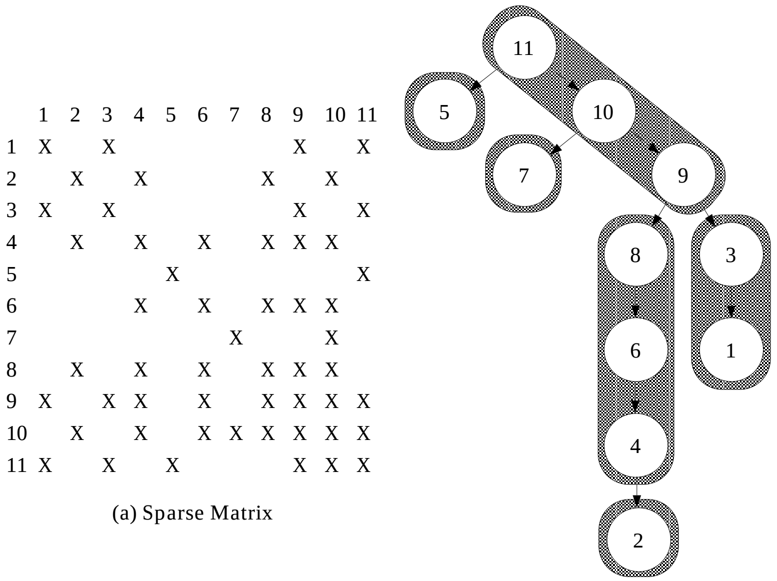
A.1.1 Algorithm

The Multifrontal solver applies Gaussian elimination to a sparse N by N matrix A such as the matrix shown in figure A.1. Since the matrix is sparse, application of the Gaussian elimination update for pivot row k

$$a_{ij} = a_{ij} - a_{kj} * a_{ik}/a_{kk}$$

for column j, in which $a_{kj} = 0$, will not affect elements in column j. Therefore, pivoting by row j will be independent of the pivot by row k. Applying this test, a dependency tree can be created for the pivot operations [23]. At any given time, all leaf nodes of the dependency tree can be performed in parallel. For the example in figure A.1a, the dependency tree in figure A.1b can be constructed. Since element $a_{1,2}$ is zero, the pivot operation using pivot row 1 will not modify $a_{2,2}$. Therefore, the pivot using row 2 can be done in parallel with the pivot using row 1. These independent pivot operations can be compacted into dense matrices by removing columns where $a_{ik} = 0$ and rows where $a_{kj} = 0$. The final value for each element is given by

$$a_{ij} = a_{ij} - \sum_{a_{kj} \neq 0, k=i+1}^{k<N} a_{kj} \frac{a_{ik}}{a_{kk}}$$



	1	2	3	4	5	6	7	8	9	10	11
1	X		X						X		X
2		X		X				X		X	
3	X		X						X		X
4		X		X		X		X	X	X	
5					X						X
6				X		X		X	X	X	
7							X			X	
8		X		X		X		X	X	X	
9	X		X	X		X		X	X	X	X
10		X		X		X	X	X	X	X	X
11	X		X		X				X	X	X

(a) Sparse Matrix

(b) Dependency Tree

Figure A.1: Sparse Matrix and Dependency Tree for Multifrontal Solver

which is the original value minus each update term.

Figure A.2 shows how the nodes in the example can be compacted. For example, the pivot by row 5 can be compacted by eliminating all columns and rows that are zero in row or column 5. This results in the 2x2 matrix shown at the top of the figure. The pivot by row 5 will create one of the Gaussian update terms needed to compute $a_{11,11}$. This update term will be needed by the parent node.

A.1.2 Implementation

In the shared memory implementation, the dependency tree is precomputed and the nodes are statically allocated to processors. The shaded grouping of nodes in the figure A.1b represents the sets of nodes that would be allocated to the same processor in this simple example. If a node had multiple children, the child with the largest shared data update was grouped with the parent to reduce network traffic.

Each processor starts with its leaf nodes. The shared update terms are computed and written to a known shared buffer, which is allocated in the parent node's local memory. The processors then start to move up the tree, reading and combining the

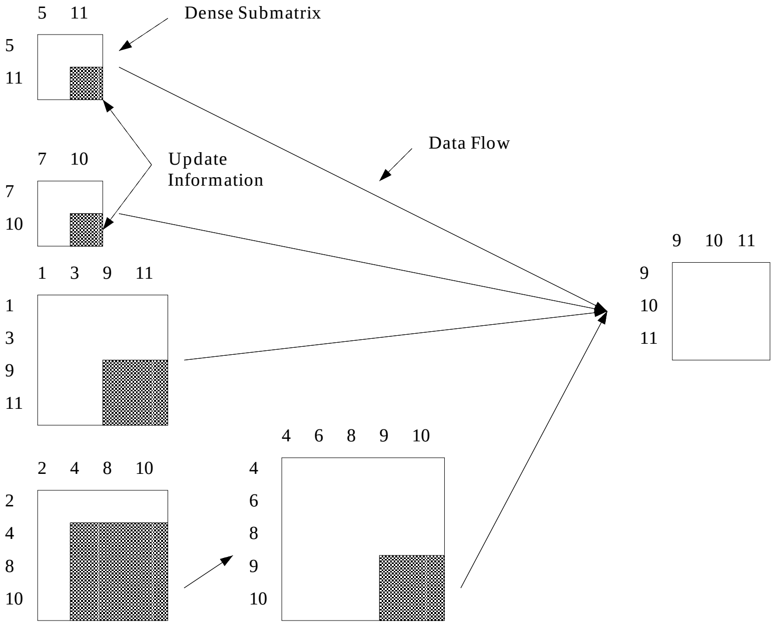


Figure A.2: Dense Submatrices and Update Data Flow

necessary update information from the children nodes. Once all the updates have been combined, dense Gaussian elimination is performed on the dense submatrix and the results are written to the proper buffer where they are then consumed by the parent node.

Table A.1 summarizes the characteristics of the multifrontal solver application. Each data block was consumed by one consumer, and the dense nature of the shared data resulted in very high line utilization.

The MF application has two distinct phases of operation [59]. Initially, the application exhibits large amounts of parallelism in the computation of independent submatrices. As the computation continues, the number of independent submatrices decreases, at which time each submatrix can be computed using a parallel LU technique. Therefore, to maintain a reasonable load balance, the computation was limited to the first 512 submatrices of the total 589 submatrices obtained from decomposing the 1000 x 1000 data matrix.

MF Application	
Sparse Data Set	1000 x 1000 (95% Sparse)
Avg. Consumers per Block	1.0 Consumer
Line Utilization	93.0 %
Synchronization Type	Flag
Iterative	No
Private Loads/Stores	297,608
Shared Loads	16,519
Shared Stores	55,248
Synchronization Events	512
Total Cycles	598,235

Table A.1: Characteristics of Multifrontal Solver Application

A.2 Simple Iterative PDE Solver (PDE)

A.2.1 Algorithm

In the PDE application, the voltage and current through a set of cross-coupled, lossy lines on a resistive ground in an integrated circuit are simulated. The relationship between the current and voltage is

$$I = C\frac{dV}{dt}.$$

The area under simulation is divided into a mesh as shown in figure A.3. At each time step k, the current through each point (i, j) is dependent on the voltage at the point and its four nearest neighbors, and the voltage at each point is dependent on the voltage at the point during the last time step, the capacitance at the point and the current through the point.

$$\begin{aligned} I_{i,j,k} &= f(V_{i-1,j-1,k}, V_{i-i,j+1,k}, V_{i+1,j-1,k}, V_{i+1,j+1,k}, R) \\ V_{i,j,k+1} &= V_{i,j,k} + \frac{dt}{C} I_{i,j,k} \end{aligned}$$

This function is repeated for many iterations until the desired data accuracy is obtained. This application has a SOR-like data sharing pattern.

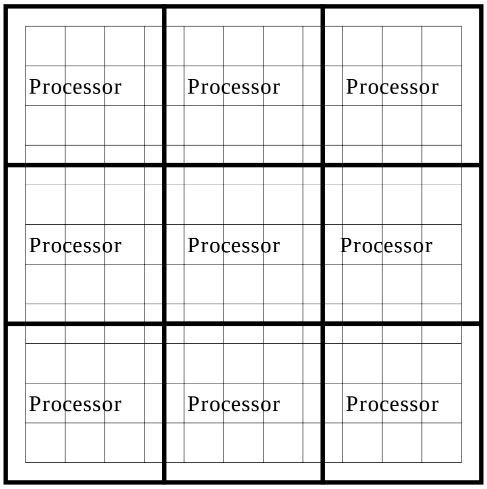


Figure A.3: PDE Solver Mesh

A.2.2 Implementation

In the shared memory implementation, the mesh is divided into 32 by 32 submeshes. The mesh was divided into equal sections and randomly allocated to the processors[1]. For a 64 processor system, each processor was responsible for a 4x4 mesh. At each time step, the processors exchanged values with its four neighboring processors. Two shared buffers were used for each pair of processors. On one iteration, a processor would write into one buffer and read from the other. On the next iteration, the roles of the buffers would be swapped. A total of five iterations were performed.

Table A.2 summarizes the characteristics of the PDE application. The five iterations resulted in 1120 blocks of 4 complex words each that were exchanged between a single producer and consumer pair. The small block size resulted in a low line utilization.

[1] Although the mesh shown is a regular mesh, a more likely scenario would be a irregular mesh which could not be allocated in such a uniform manner. Random allocation is an attempt to mimic such an irregular layout.

PDE Application	
Dense Data Set	32 x 32
Ave. Consumers per Block	1.0 Consumers
Line Utilization	50.0 %
Synchronization Type	Flag
Iterative	5 Iterations
Private Loads/Stores	238,060
Shared Loads	8,960
Shared Stores	10,080
Synchronization Events	1,120
Total Cycles	626,961

Table A.2: Characteristics of PDE Application

A.3 Sparse Cholesky Decomposition (SPCF)

A.3.1 Algorithm

If a matrix A is symmetric and positive definite, Cholesky factorization can be used to factor the matrix into

$$A = LL^T,$$

where L is a lower triangular matrix. Once the factorization is complete, forward and backward substitutions are required to solve the system. The original matrix A can be factored into LL^T by applying the following set of equations.

$$\begin{aligned} l_{jj} &= \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2} \\ l_{ij} &= (a_{ij} - \sum_{k=1}^{k<j} l_{jk} l_{ik})/l_{jj} \quad i = j+1, \ldots, n \end{aligned}$$

These equations can be implemented using a left-looking (column-Cholesky) approach or with a right-looking (submatrix-Cholesky) approach.

If the data matrix is sparse, many of the entries will be zero. If $l_{jk} = 0$, then the update of column j by column k will not contribute to the update and can be skipped.

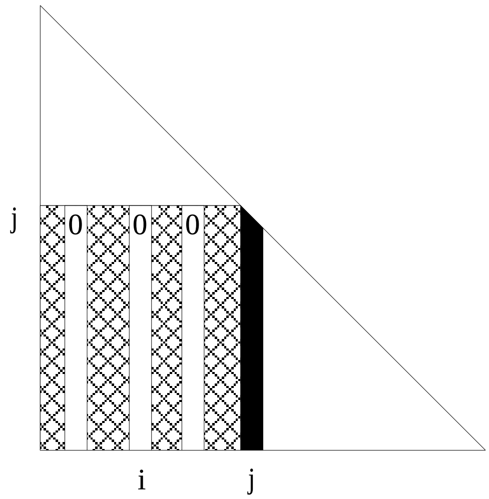


Figure A.4: Data Flow For Cholesky Factorization

The higher the sparsity of the data, the fewer update columns will be needed for each column updated.

A.3.2 Implementation

A left-looking approach was chosen to compute L. In the left-looking algorithm, each column j of L is computed by combining columns from the left. The columns were distributed evenly among the processors and stored locally.

Figure A.4 shows the data flow to produce column j of L. Applying the above algorithm requires data from columns to the left of column j for rows below row j. Since A is a sparse matrix, some elements of L will remain zero. If l_{ji} is zero, then column i is not required to calculate column j of L since column i of L is multiplied by l_{ji} before being summed into column j. The amount of data needed to compute each column is data set dependent.

Table A.3 summarizes the characteristics of the Cholesky factorization application. The sparsity of the data set, from the Harwell-Boeing data set [24], resulted in very small data blocks that were read by very few consumers and had a very low line utilization.

SPCF Application	
Sparse Data Set	1138 x 1138 (98% Sparse)
Ave. Consumers per Block	1.9 Consumers
Line Utilization	11.2 %
Synchronization Type	Flag
Iterative	No
Private Loads/Stores	125,603
Shared Loads	10,357
Shared Stores	3,256
Synchronization Events	2,118
Total Cycles	2,236,441

Table A.3: Characteristics of Cholesky Factorization Application

A.4 3D PDE Solver using FFTs (3DFFT)

The 3DFFT application, from the NAS Parallel Benchmark set [9], solves a set of partial differential equations (PDE) using forward and inverse fast Fourier transforms (FFT).

A.4.1 Algorithm

The algorithm, as described in [9], is as follows. Consider the PDE

$$\frac{t\partial u(x,t)}{\partial t} = \alpha \nabla^2 u(x,t)$$

where x is a position in 3-dimensional space. When a Fourier transform is applied to each side, this equation becomes

$$\frac{\partial v(z,t)}{\partial t} = -4\alpha\phi^2|z|^2 v(z,t)$$

where v(z,t) is the Fourier transform of $u(x,t)$. This has the solution

$$v(z,t) = e^{-4\alpha\phi^2|z|^2 t} v(z,0)$$

Now consider the discrete version of the original PDE. Following the above, it can be solved by computing the forward 3-D discrete Fourier transform (DFT) of the

3DFFT Application	
Dense Data Set	8 x 8 x 16
Ave. Consumers per Block	4.0 Consumers
Line Utilization	50.0 %
Synchronization Type	Barrier
Iterative	3 Iterations
Private Loads/Stores	480,640
Shared Loads	14,336
Shared Stores	18,944
Synchronization Events	12
Total Cycles	2,716,459

Table A.4: Characteristics of 3DFFT Application

original state array $u(x, 0)$, multiplying the results by certain exponentials, and then performing an inverse 3-D DFT.

A.4.2 Implementation

The actual implementation is also taken from the description in [9], and it is repeated here. Assume that the data in the input n_1 x n_2 x n_3 complex array A is organized so that for each j and k, all elements of A $(A_{i,j,k})$ are allocated at a single memory node. First perform an n-point 1-D FFT on each of the $n_2 n_3$ complex vectors by copying the shared data in A into a private vector, computing the FFT and then copying the data back to A. Transpose the matrix and repeat this process two more times as shown in figure A.5 to form a single iteration of the computation. This computation is repeated for several iterations with barriers used to synchronize the data production and consumption.

Table A.4 summarizes the characteristics of the 3DFFT application. Each data block was consumed by a single consumer, but the rotation of the shared data block resulted in false sharing among three other nodes.

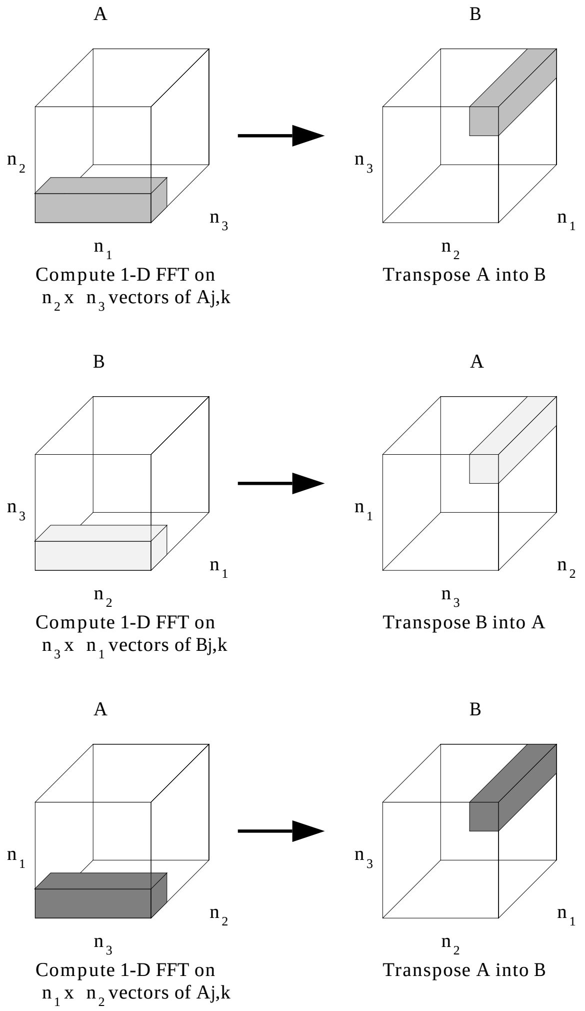


Figure A.5: Data Flow for Single Iteration of 3DFFT

A.5 LU Decomposition (LU)

A.5.1 Algorithm

In many applications, the solution to the linear system of equations $Ax = b$ must be found for many different values of b. The LU decomposition algorithm allows the matrix A to be decomposed into two triangular matrices L and U such that

$$PAx = LUx = Pb.$$

where P is the permutation matrix, L is a lower triangular matrix with ones on the diagonal, and U is an upper triangular matrix. This new system can then be solved by first solving $Ly = Pb$ for y using forward substitution and then solving $Ux = y$ for x using backward substitution [28].

A.5.2 Implementation

In the shared memory implementation, each processor is given the responsibility for computing a set of columns of L and U. Each column of A is stored at the memory local to the processor responsible for that column. The memory locations used for A can also be used to store the values of U and L. A second shared matrix is used to store the pivot row indices for each step.

All processors step through the columns of A. If the current column k is in the set of columns that a processor is responsible for, the processor determines the best pivot row, stores the pivot row index, swaps the necessary rows and computes the multiplier terms. All other processors read the pivot row index, swaps the necessary rows and apply the multipliers to each column that they are responsible for. Each column requires the multiplier terms from all columns to the left. Figure A.6 shows the shared data flow needed to compute column j of L and U. Table A.5 summarizes the characteristics of the LU decomposition application.

LU Application	
Dense Data Set	64 x 64
Ave. Consumers per Block	31.5 Consumers
Line Utilization	59.0 %
Synchronization Type	Flag
Iterative	No
Private Loads/Stores	189,341
Shared Loads	89,440
Shared Stores	2,080
Synchronization Events	64
Total Cycles	4,360,869

Table A.5: Characteristics of LU Decomposition

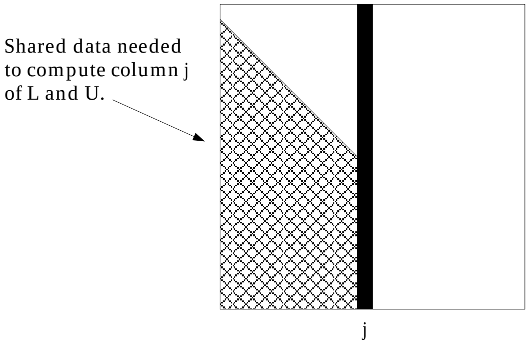


Figure A.6: Data Flow For LU Decomposition

The application required the movement of a relatively large amount of data. Multipliers computed for column j were read by all columns to the right. This algorithm resulted in large shared data blocks and a large number of consumers for each block. The density of the data allowed for the high line utilization.

A.6 Summary of Scientific Applications

Table A.6 summarizes the two characteristics, the average number of consumers and the average line utilization, which were used in section 4.3 to define the application space.

Application	Consumers	Line Utilization %
MF	1	93.0
PDE	1	50.0
SPCF	1.9	11.2
3DFFT	4	50.0
LU	31.5	59.0

Table A.6: Application characteristics

Appendix B

Migratory Data Applications

B.1 MP3D Particle Simulator

B.1.1 Algorithm

The MP3D application is taken from the Splash benchmarks set [63]. The description of the application from Splash documentations is summarized here. MP3D solves a problem in rarefied fluid flow simulations. MP3D uses a Monte Carlo method to simulate the trajectories of a collection of molecules that are subject to collisions with boundaries of the physical domain or other molecules. MP3D employs five degree-of-freedom simulation of idealized diatomic molecules in a three-dimensional active space. The active space is a rectangular tunnel with openings at each end and reflecting walls. Molecules that exit the tunnel are kept in a reservoir and are later

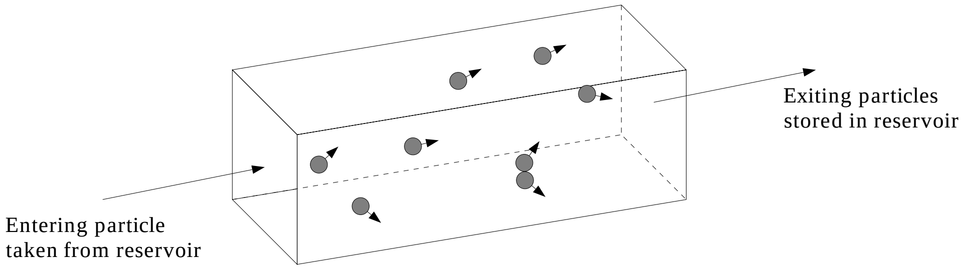


Figure B.1: Data Flow For MP3D Decomposition

MP3D Application	
Dense Data Set	1024 Molecules
Avg. Consumers per Block	Migratory
Line Utilization	40.0 %
Synchronization Type	Barrier
Iterative	20 Iterations
Private Loads/Stores	34,218
Shared Loads	245,450
Shared Stores	232,509
Synchronization Events	120
Total Cycles	14,653,999

Table B.1: Characteristics of MP3D Application

reinserted into the tunnel. Figures B.1 shows the active space.

B.1.2 Implementation

Each processor is responsible for computing the movement of a set of molecules. The active space is divided into unit-size cells, and molecules may only collide with other molecules in the same cell. The results of each collision are statistically determined.

Each time step is divided into 5 basic steps: initialize, move, add reservoir-move and reservoir-collide. The initialization step resets the appropriate variables, and in the move step, molecules are moved and collisions are resolved. This step accounts for over 90% of the execution time. The add, reservoir-move and reservoir-collide steps deal with the molecules entering and exiting the tunnel and the molecules in the reservoir.

Table B.1 summarizes the characteristics of the MP3D application.

B.2 Synthetic Task Application

In the migratory application MP3D, the data tends to migrate slowly especially in terms of simulation time. Therefore, a synthetic migratory application was needed to

TASK Application	
Dense Data Set	Single Task
Avg. Consumers per Block	Migratory
Line Utilization	50.0 %
Synchronization Type	Ideal
Iterative	512 Iterations
Private Loads/Stores	3,256,924
Shared Loads	280,440
Shared Stores	140,472
Synchronization Events	0
Total Cycles	20,895,173

Table B.2: Characteristics of the TASK Application

examine performance of applications with significant data migration. In this synthetic task application (TASK), each processor obtains a task from a global task queue, operates on the task and returns the task to the task queue. A task consists of

1. Reading a line of data
2. Performing work on the data
3. Possibly modifying the data

The work time is randomly distributed with an uniform distribution between 1 and 16 cycles, and the probability of modifying the line is set at 50%. With the load/store limited superscalar processor model, the work time can be interpreted as the number of private loads and stores required to process each task.

The task queue has 128 tasks and the application runs until each processor has operated on 512 tasks. The operations to add and remove tasks from the queue are not simulated.

Table B.2 summarizes the characteristics of the synthetic TASK application.

Appendix C

Tables of Relative Execution Times

C.1 Base Simulation Times

Table C.1 gives the relative execution times plotted in figures 6.3 through 6.7. All times are relative to the CD-INV protocol.

C.2 Sensitivity Simulation Times

Tables C.2 through C.7 present the relative execution times of each protocol for the sensitivity studies of chapter 7. All execution times are relative to the execution time of the base CD-INV protocol presented in chapter 6 for the respective application. The table also gives the geometric mean of the relative execution time across all applications for each protocol that were plotted in chapter 7.

Applications	MF	PDE	SPCF	3DFFT	LU	Mean
CD-INV	1.00	1.00	1.00	1.00	1.00	1.00
DD-INV	0.96	0.91	1.13	0.92	1.37	1.04
SCI-INV	1.11	1.44	1.74	1.61	2.61	1.63
CD-UP	1.26	0.97	0.79	1.11	2.27	1.19
CD-UP-HW	0.91	0.64	0.68	0.73	0.88	0.76
DD-UP	0.99	0.80	0.95	1.26	2.91	1.23
DD-UP-HW	0.95	0.66	0.88	0.91	2.32	1.03
CD-UP-WS	1.09	0.88	0.48	1.02	1.38	0.92
CD-UP-HW-WS	0.82	0.51	0.43	0.66	0.49	0.57
DD-UP-WS	0.92	0.76	0.57	1.20	1.15	0.89
DD-UP-HW-WS	0.86	0.52	0.51	0.72	0.45	0.60

Table C.1: Relative Execution Times

Applications	MF	PDE	SPCF	3DFFT	LU	Mean
CD-INV	1.88	1.24	1.20	1.14	1.20	1.31
DD-INV	1.88	1.45	1.41	1.32	1.83	1.56
SCI-INV	1.93	2.08	2.44	2.45	3.46	2.42
CD-UP-HW	1.57	0.92	1.02	0.82	1.16	1.07
CD-UP-HW-WS	1.46	0.80	0.63	0.85	0.77	0.87
DD-UP-HW	2.09	0.99	1.35	1.08	2.90	1.55
DD-UP-HW-WS	1.93	0.96	0.78	1.02	1.21	1.12

Table C.2: Cache Access (5 Cycle) Relative Execution Times

Applications	MF	PDE	SPCF	3DFFT	LU	Mean
CD-INV	1.00	1.02	1.02	1.02	1.28	1.06
DD-INV	0.98	0.91	1.15	0.93	1.31	1.05
SCI-INV	1.11	1.43	1.81	1.64	2.83	1.68
CD-UP-HW	0.91	0.64	0.70	0.73	0.84	0.76
CD-UP-HW-WS	0.82	0.52	0.44	0.67	0.48	0.57
DD-UP-HW	0.97	0.64	0.90	0.91	2.35	1.04
DD-UP-HW-WS	0.85	0.52	0.52	0.75	0.44	0.60

Table C.3: Directory Access (5 Cycle) Relative Execution Times

Applications	BW	MF	PDE	SPCF	3DFFT	LU	Mean
CD-INV	1/2x	1.15	1.43	1.06	1.26	0.92	1.15
	2x	0.96	0.90	0.99	0.92	1.42	1.02
DD-INV	1/2x	1.00	1.16	1.21	1.15	1.17	1.13
	2x	0.95	0.85	1.15	0.85	1.32	1.01
SCI-INV	1/2x	1.19	1.75	1.90	1.99	2.76	1.85
	2x	1.07	1.34	1.79	1.51	2.32	1.55
CD-UP-HW	1/2x	1.11	0.74	0.75	0.97	0.84	0.87
	2x	0.82	0.60	0.66	0.64	0.62	0.66
CD-UP-HW-WS	1/2x	0.90	0.61	0.47	0.88	0.60	0.67
	2x	0.77	0.49	0.42	0.58	0.55	0.55
DD-UP-HW	1/2x	1.19	0.74	1.00	1.24	3.58	1.31
	2x	0.83	0.61	0.84	0.76	2.04	0.92
DD-UP-HW-WS	1/2x	1.00	0.61	0.58	1.10	0.59	0.74
	2x	0.78	0.49	0.49	0.64	0.41	0.55

Table C.4: Network Bandwidth Relative Execution Times

Applications	BW	MF	PDE	SPCF	3DFFT	LU	Mean
CD-INV	1/2x	0.98	0.96	0.85	0.90	1.14	0.96
	2x	1.10	1.50	1.84	1.65	1.28	1.45
DD-INV	1/2x	0.94	0.85	0.99	0.80	0.89	0.89
	2x	1.06	1.43	2.13	1.60	2.11	1.62
SCI-INV	1/2x	1.06	1.27	1.51	1.42	1.81	1.39
	2x	1.37	2.44	3.41	2.87	5.23	2.80
CD-UP-HW	1/2x	0.91	0.62	0.61	0.67	0.79	0.71
	2x	0.95	0.87	1.16	1.03	1.22	1.04
CD-UP-HW-WS	1/2x	0.80	0.52	0.39	0.62	0.47	0.54
	2x	0.86	0.54	0.72	0.87	0.69	0.73
DD-UP-HW	1/2x	0.93	0.61	0.78	0.85	2.15	0.96
	2x	1.07	0.80	1.45	1.31	3.74	1.44
DD-UP-HW-WS	1/2x	0.84	0.52	0.46	0.70	0.46	0.58
	2x	0.92	0.55	0.88	0.99	0.61	0.77

Table C.5: Network Latency Relative Execution Times

Applications	SPCF	3DFFT	LU	Mean
CD-INV	1.00	1.00	1.22	1.07
CD-UP-HW	0.70	1.45	1.17	1.06
CD-UP-HW-WS	0.44	1.49	0.65	0.75

Table C.6: Relative Execution Times Without Multicast

Applications	Cycles	MF	PDE	SPCF	3DFFT	LU	Mean
CD-INV	1	1.00	1.00	1.00	1.00	1.00	1.00
	2	1.50	1.36	1.17	1.13	1.20	1.26
	4	2.68	2.04	1.53	1.46	1.58	1.81
	6	3.92	2.81	1.88	1.83	1.66	2.29
	8	5.15	3.54	2.25	2.20	2.04	2.84
DD-INV	1	0.96	0.91	1.13	0.92	1.37	1.04
	2	1.47	1.24	1.31	1.05	1.52	1.31
	4	2.66	1.95	1.69	1.38	1.84	1.86
	6	3.90	2.71	2.02	1.75	2.32	2.44
	8	5.13	3.44	2.38	2.13	2.52	2.96
SCI-INV	1	1.11	1.44	1.74	1.61	2.61	1.63
	2	1.62	1.76	1.97	1.75	2.31	1.87
	4	2.80	2.44	2.34	2.06	3.53	2.59
	6	4.03	3.12	2.66	2.41	4.08	3.19
	8	5.27	3.83	3.01	2.74	4.04	3.68
CD-UP-HW	1	0.91	0.64	0.68	0.73	0.88	0.76
	2	1.48	1.00	0.82	1.11	1.05	1.07
	4	2.69	1.75	1.12	1.33	1.29	1.55
	6	3.93	2.60	1.47	1.58	2.90	2.33
	8	5.15	3.32	1.79	1.85	3.04	2.80
CD-UP-HW-WS	1	0.82	0.51	0.43	0.66	0.49	0.57
	2	1.38	0.89	0.55	0.95	0.54	0.81
	4	2.58	1.66	0.84	1.25	0.68	1.25
	6	3.80	2.45	1.14	1.59	1.44	1.89
	8	5.00	3.21	1.43	1.99	1.38	2.29
DD-UP-HW	1	0.95	0.66	0.88	0.91	2.32	1.03
	2	1.44	1.02	1.01	1.29	2.60	1.38
	4	2.65	1.75	1.27	1.46	2.76	1.88
	6	3.87	2.57	1.63	1.71	3.23	2.46
	8	5.09	3.29	1.92	1.98	3.13	2.88
DD-UP-HW-WS	1	0.86	0.52	0.51	0.72	0.45	0.60
	2	1.36	0.90	0.64	1.05	0.66	0.88
	4	2.56	1.66	0.91	1.33	1.03	1.40
	6	3.75	2.44	1.22	1.67	1.36	1.91
	8	4.96	3.22	1.51	2.03	1.56	2.38

Table C.7: Processor Load/Store Cycle Count

Bibliography

[1] Sarita Adve and Mark Hill. Weak Ordering - A New Definition. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 2–14, 1990.

[2] Anant Agarwal. Limits on Network Performance or Moderate Dimensions are Better. *Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[3] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Mutltiprocessing. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 104–114, 1990.

[4] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 280–289, 1988.

[5] M. S. Algudady, C. R. Das, and M. J. Thazhuthaveetil. A Write Update Cache Coherence Protocol for MIN-Based Multiprocessors with Accessibility-Based Split Caches. In *Supercomputing '90*, pages 544–553, 1990.

[6] Gail Alverson, Robert Alverson, and David Callahan. Exploiting Heterogeneous Parallelism on a Multithreaded Multiprocessor. In *Workshop on Multithreaded Computers, Proceedings of Supercomputing '91.*, November 1991.

[7] James K. Archibald. The Cache Coherence Problems in Shared-Memory Multiprocessors. Available as Technical Report 87-02-06, University of Washington, February 1987.

[8] James K. Archibald and Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):274–298, November 1986.

[9] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS Parallel Benchmarks. Report RNR-91-002, NAS Systems Division, Applied Research Branch, NASA Ames Research Center, January 1991.

[10] Luis A. Barroso and Michel Dubois. The Performance of Cache-Coherent Ring-Based Multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 268–277, May 1993.

[11] Sandra Johnson Baylor, Kevin P. McAuliffe, and Bharat Deep Rathi. An Evaluation of Cache Coherence Protocols for MIN-Based Multiprocessors. Research Report RC 15220, IBM Research Division, T. J. Watson Reasearch Center, December 1989.

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

[13] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache-Coherence in Large-Scale Multiprocessors. *IEEE Computer*, pages 49–58, June 1990.

[14] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234, 1991.

[15] William J. Dally. Wire-Efficient VLSI Multiprocessors Communication Networks. In *Advanced Reseach in VLSI–Proceedings of the 1987 Stanford Conference*, pages 391–415, 1987.

[16] William J. Dally and Charles L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[17] William J. Dally, et al. Architecture of a message-driven processor. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 189–196, June 1987.

[18] Bruce A. Delagi, Nakul P. Saraiya, Sayuri Nishimura, and Gregory T. Byrd. Instrumented Architectural Simulation. In *Proceedings of the Third International Conference on Supercomputing*, pages 8–11, March 1988.

[19] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1992.

[20] Andreas J. Drexler and C. Norris Ip. *Murφ Annotated Reference Manual.* Stanford University, 1992.

[21] Michel Dubois and Christoph Scheurich. Memory Access Dependencies in Shared-Memory Multiprocessors. *IEEE Transaction on Software Engineering*, 16(6):660–673, June 1990.

[22] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 434–442, June 1986.

[23] Iain S. Duff. Parallel Implementation of Multifrontal Schemes. *Parallel Computing*, 3:193–204, 1986.

[24] Iain S. Duff, Roger G. Grimes, and John G. Lewis. Users' Guide for the Harwell-Boeing Sparse Matrix Collection. Computer Science and Systems Division, Harwell Laboratory, Didcot, Oxon OX11 ORA, England, February 1988.

[25] Thomas H. Dunigan. Kendall Square Multiprocessor: Early Experience and Performance. Technical Report ORNL/TM-12065, Oak Ridge National Laboratory, March 1992.

[26] Susan J. Eggers and Randy H. Katz. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 373–382, May 1988.

[27] Susan J. Eggers and Randy H. Katz. Evaluating the Performance of Four Snooping Cache Coherence Protocols. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 2–15, May 1989.

[28] Lars Elden and Linde Wittmery-Koch. *Numerical Analysis, An Introduction.* Academic Press, Inc., 1990.

[29] Mark Hill, et al. Design Decisions in SPUR. *IEEE Computer*, 19(11):8–22, November 1986.

[30] S. J. Frank. Tightly Coupled Multiprocessor Systems Speeds Memory-Access Times. *Electronics*, 57(1):164–169, January 1984.

[31] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 245–257, 1991.

[32] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Revision to Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. Technical Report CSL-TR-93-568, Computer Systems Laboratory, Stanford University, April 1993.

[33] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip. Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, 1990.

[34] David B. Glasco, Bruce A. Delagi, and Michael J. Flynn. The Impact of Cache Coherence Protocols on Systems Using Fine-Grain Data Synchronization. In *Parallel Architectures and Compilation Techniques*, pages 79–88, August 1994.

[35] David B. Glasco, Bruce A. Delagi, and Michael J. Flynn. Update-Based Cache Coherence Protocols for Scalable Shared-Memory Multiprocessors. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 534–545, January 1994.

[36] David B. Glasco, Bruce A. Delagi, and Michael J. Flynn. Write Grouping for Update-Based Cache Coherence Protocols. In *6th IEEE Symposium on Parallel and Distributed Processing*, pages 334–341, October 1994.

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

[38] James R. Goodman. Cache Consistency and Sequential Consistency. Technical Report Computer Sciences #1006, University of Wisconsin, Madison, February 1991.

[39] Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. Technical Report No. CSL-TR-90-417, Computer Systems Laboratory, Stanford University, 1990.

[40] E. Hagersten, A. Landin, and S. Haridi. DDM - A Cache-Only Memory Architecture. *IEEE Computer*, 25(9):45–54, September 1992.

[41] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[42] IEEE Standards Department, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331. *Scalable Coherent Interface: Logical, Physical and Cache Coherence Specifications*, p1596/d2.00 edition, November 1991.

[43] C. Norris Ip and David L. Dill. Better Verification Through Symmetry. In *Proceedings of the 11th International Symposium on Computer Hardware Description Languages and Their Applications*, pages 87–100, April 1993.

[44] C. Norris Ip and David L. Dill. Efficient Verification of Symmetric Concurrent Systems. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, pages 230–234, 1993.

[45] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurinder S. Sohi. Scalable Coherent Interface. *IEEE Computer*, 23(6):74–77, June 1990.

[46] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive Snoopy Caching. In *Proceedings of the 27th Annual Symposium of Fundations of Computer Science*, pages 244–254, 1984.

[47] Kendall Square Research, Waltham, MA. *KSR1 Principles of Operation*, revision 5.5 edition, October 1991.

[48] David Kranz, Beng-Hong Lim, David Yeung, and Anant Agarwal. Low-Cost Support for Fine-Grain Synchronization in Multiprocessors. In *International Symposium on Computer Architecture*, 1992.

[49] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *7th ACM International Conference of Supercomputing*, 1993.

[50] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 302–313, May 1994.

[51] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[52] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, c-28(9):690–691, September 1979.

[53] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

[54] E. McCreight. The Dragon Computer System: An Early Overview. In *NATO Advanced Study Institute on Microarchitecture of VLSI Computers*, July 1984.

[55] Todd Mowry and Anoop Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, pages 87–106, 1991.

[56] Ashwini K. Nanda and Hong Jiang. Analysis of Directory Based Cache Coherence Schemes with Multistage Networks. In *1992 ACM Computer Science Conference*, pages 485–492, March 1992.

[57] Brian W. O'Krafka and A. Richard Newton. An Empirical Evaluation of Two Memory-Efficient Directory Methods. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 138–147, June 1990.

[58] M. Papamarcos and J. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th Annual Symposium on Computer Architecture*, pages 348–354, June 1984.

[59] Edward Rothberg and Anoop Gupta. A Comparative Evaluation of Nodal and Supernodal Parallel Sparse Matrix Factorization: Detailed Simulation Results. Technical Report CSL-TR-90-416, Computer Systems Laboratory, Stanford University, February 1990.

[60] Nakul P. Saraiya, Bruce A. Delagi, and Sayuri Nishimura. Simple/Care an Instrumented Simulator for Multiprocessor Architectures. Technical Report KSL-90-66, Knowledge Systems Laboratory, Stanford University, 1990.

[61] C. Scheurich and M. Dubois. Lockup-Free Caches in High-Performance Multiprocessors. In *Journal of Parallel and Distributed Computing*, pages 25–36, 1991.

[62] Richard Simoni and Mark Horowitz. Dynamic Pointer Allocation for Scalable Cache Coherence Directories. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, April 1991.

[63] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

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

[65] Burton J. Smith. Architecture and Application of the HEP Multiprocessor Computer System. *Real Time Signal Processing IV*, 298:241–248, August 1981.

[66] Per Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.

[67] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite Jr. Firefly: A Multiprocessor Workstation. In *Proceedings of the Second International Conference on Architectural Support of Programming Languages and Operatings Systems*, pages 164–172, October 1987.

[68] Shreekant Thakkar, Michel Dubois, Anthony T. Laundrie, and Gurindar S. Sohi. Scalable Shared-Memory Multiprocessor Architectures. *IEEE Computer*, 23(6):71–83, June 1990.

[69] Manu Thapar. Cache Coherence for Scalable Shared Memory Multiprocessors. Technical Report CSL-TR-92-522, Computer Systems Laboratory, Stanford University, May 1992.

[70] Manu Thapar, Bruce A. Delagi, and Michael J. Flynn. Linked List Cache Coherence for Scalable Shared Memory Multiprocessors. In *7th International Parallel Processing Symposium*, pages 34–43, April 1993.

[71] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Systems (ASPLOS III)*, pages 243–256, April 1989.

[72] Andrew W. Wilson Jr. and Richard P. LaRowe Jr. Hiding Shared Memory Reference Latency on the Galactica Net Distributed Shared Memory Architecture. In *Journal of Parallel and Distributed Computing*, pages 351–367, 1992.

[73] Qing Yang, George Thangadurai, and Laxmi N. Bhuyan. Design of an Adaptive Cache Coherence Protocol for Large Scale Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):281–293, May 1992.