Tolerating Latency in Software Distributed Shared Memory Systems Through Non-Binding Prefetching

by

Charles Quoc Cuong Chan

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

© Copyright by Charles Quoc Cuong Chan 1998

Abstract

Tolerating Latency in Software Distributed Shared

Memory Systems Through Non-Binding Prefetching

Charles Quoc Cuong Chan

Master of Science

Graduate Department of Computer Science

University of Toronto

1998

A key obstacle to achieving high performance on software distributed shared memory (DSM) systems is their high memory latencies. *Software-controlled prefetching* tolerates memory latency by overlapping computation with communication.

This thesis proposes and evaluates an implementation of software-controlled non-binding prefetching on a software DSM called TreadMarks. With programmer-inserted prefetching, all of our applications achieve better performance. The overall speedup ranges from 4% to 29%. In addition, we observe that the performance of compiler-inserted prefetching matches that of programmer-inserted prefetching in a few cases. We also investigate prefetching with runtime information. Although using dynamic information to issue prefetches can overcome some of the limitations of statically inserted prefetches, the overheads of this approach often more than offset any gain in memory performance. Finally, we evaluate the combined effects of prefetching and multithreading on application performance. In several cases, the combined approach outperforms either technique alone, but the overall results are mixed.

Acknowledgements

I would like to express my thankfulness to all who have contributed to the research presented in this thesis.

First of all, it is a blessing that I have the opportunity to work with my supervisor, Todd Mowry, who has been extremely patience with me and has guided me throughout the entire process of the research. In addition, I would like to thank my second reader, Michael Stumm, for his time and effort, especially under a tight time constraint. I would also like to thank the entire POW research team, especially Paul Lu and Eric Parsons for managing the network of workstations to allow our experiments to be performed smoothly. I truly appreciate the financial support provided by the University of Toronto Open Fellowship.

I would like to thank Chi-Keung Luk, Adley Lo, and Mabel Wong for their professional advice and for all the tea times that we have spent together. I would also like to thank my parents and my two brothers, Allen Chan and Billy Chan, for their emotional supports. In addition, my little sheltie, Lucky, who has always given me a reason to smile during the hard times.

Last but not least, I would like to express my deepest gratitude towards my beloved Cynthia Lee for always supporting me and praying for me in both good and bad times. You and God have written a wonderful chapter of my life.

Contents

1 Introduction **1**

1.1 Software DSM Performance 2

1.2 Coping with Memory Latency 3

1.2.1 Caching 4

1.2.2 Prefetching 4

1.2.3 Multithreading 6

1.2.4 Overall Approach 6

1.3 Research Goals 7

1.4 Related Work 7

1.5 Organization of Thesis 8

2 Non-Binding Prefetching Implementation **10**

2.1 Memory Consistency Models 10

2.2 Lazy Release Consistency 12

2.2.1 Write Notices and Timestamps 13

2.2.2 Multiple-Writer Protocol 14

2.3 Coherence Protocols 15

2.4 Prefetching Techniques 16

2.4.1 Binding vs. Non-binding Prefetches 18

2.4.2 Implementing Non-binding Prefetching Within Lazy Release Consistency . . . 19
2.5 Implementation within TreadMarks . . . 23
2.5.1 TreadMarks . . . 23
2.5.2 Pseudo-codes . . . 24

3 Non-binding Prefetching Performance **29**
3.1 Experimental Framework . . . 30
3.1.1 Hardware Platform . . . 30
3.1.2 Applications . . . 30
3.2 Programmer-Inserted Prefetching . . . 32
3.2.1 Inserting Prefetches . . . 32
3.2.2 Overall Performance . . . 33
3.2.3 Prefetching Effectiveness . . . 36
3.2.4 Summary . . . 40
3.3 Compiler-Inserted Prefetching . . . 40
3.3.1 Overall Performance . . . 40
3.3.2 Cases where the Compiler Failed . . . 42
3.4 Chapter Summary . . . 44

4 Alternative Prefetching Approaches **46**
4.1 History Prefetching . . . 47
4.1.1 History Prefetching Performance Analysis . . . 48
4.1.2 Summary . . . 50
4.2 Hybrid Prefetching . . . 51
4.2.1 Hybrid Performance Analysis . . . 52
4.2.2 Summary . . . 55

5 Prefetching and Multithreading **56**

5.1 Multithreading 56

5.1.1 Multithreading Implementation and Application Modifications . . 57

5.1.2 Multithreading Performance 58

5.1.3 Summary 60

5.2 Combining Multithreading and Prefetching 61

5.2.1 Prefetch Insertion Strategy 62

5.2.2 Overall Performance 63

5.2.3 Chapter Summary 65

6 Conclusions **67**

Bibliography **69**

List of Tables

3.1 Application descriptions, problem sizes, and parallel execution times . . . 31

3.2 The percentages of unnecessary prefetches and the percentages of remote misses covered by prefetches (i.e. coverage factors) 36

3.3 Communication characteristics of programmer-inserted prefetching 39

3.4 The percentage of unnecessary prefetches with programmer-inserted prefetching and with compiler-inserted prefetching 42

4.1 The number of invalidated prefetch misses due to lazy propagation of write notices and issuing prefetches too early 51

4.2 The impact of hybrid prefetching on the number of invalidated prefetch misses 54

List of Figures

1.1 Application performance using TreadMarks on 8 workstations connected by an ATM LAN 2

1.2 How prefetching improves performance 5

2.1 Illustration of a limitation of prefetching in lazy release consistency. . . . 16

2.2 Example of when a binding prefetching is illegal. 18

2.3 Simple non-binding prefetching within LRC (only parts of the modifications are retrieved). 20

2.4 To request the most up-to-date modification within LRC, a new interval is created upon servicing a prefetch request. 22

2.5 Non-binding prefetching pseudo-code. (Continued on next page.) 25

2.5 Non-binding prefetching pseudo-code. (Continued from previous page.) . 26

3.1 Performance improvement from programmer-inserted prefetch 34

3.2 Breakdown of impact of programmer-inserted prefetching on the original remote misses 37

3.3 Performance improvement from compiler-inserted prefetching 41

4.1 Performance improvement from history prefetching 49

4.2 Performance improvement from hybrid prefetching 53

5.1 Performance comparison between prefetching and multithreading using 2, 4, and 8 threads 59
5.2 Eliminating unnecessary prefetches in multithreading by inserting a conditional test 62
5.3 Performance improvement from combining prefetching and multithreading using 2, 4, and 8 threads 64

Chapter 1

Introduction

There has been considerable interest recently in exploiting collections of workstations connected by commodity networks as the less expensive alternatives to tightly-coupled multiprocessors. Software distributed shared-memory (DSM) is an all-software approach that enables different processes on different machines to share a single address space [25]. Compared with tightly-coupled multiprocessor machines like the SGI Origin [22], software DSMs are relatively inexpensive and are widely applicable across a broad range of hardware platforms. For some classes of applications, software DSMs are able to deliver performance comparable to that of hardware cache-coherent shared memory machines at the same scale [8, 15].

A key obstacle to achieving higher performance on software DSMs is their relatively high communication latencies. For example, a remote memory access on a tightly-coupled multiprocessor machine takes on the order of half a microsecond [22] while that on a software DSM system running on commodity hardware requires close to half a *millisecond* [9]—i.e. roughly three orders of magnitude slower. Since memory miss latency is known to be a performance bottleneck for tightly-coupled multiprocessors, it is reasonable to expect that the even higher memory miss latency on software DSMs will also be a significant problem.

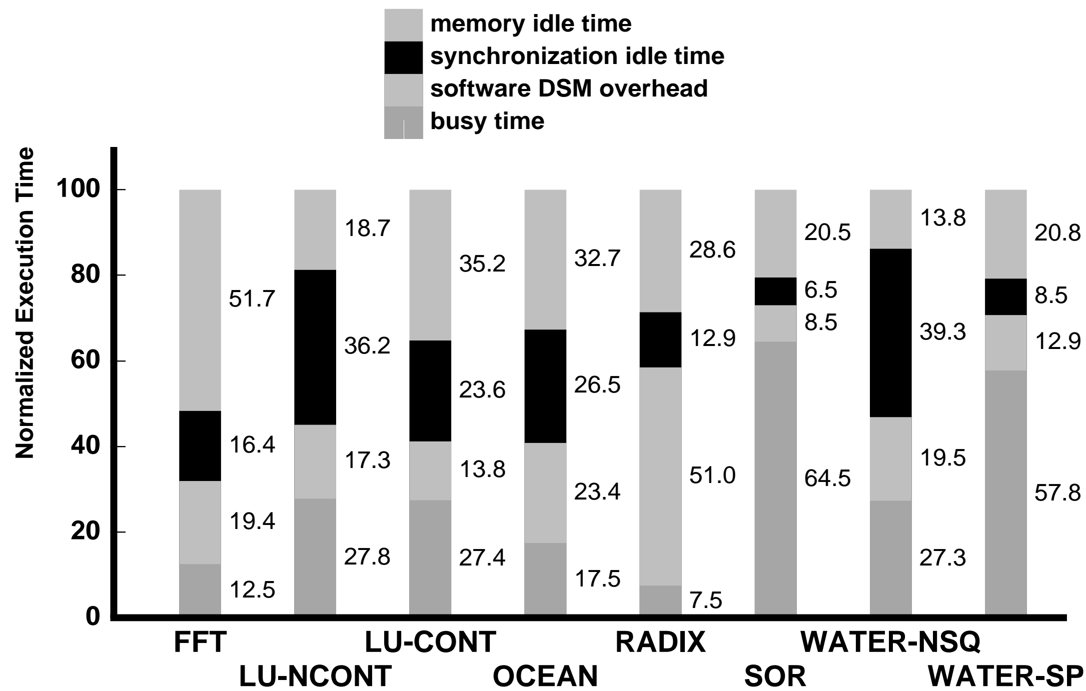

Figure 1.1: Application performance using TreadMarks on 8 workstations connected by an ATM LAN

This thesis investigates a technique called *software-controlled prefetching* which helps alleviate the impact of the high memory latency on software DSMs. In Section 1.1, we provide further motivation for improving the memory performance of software DSMs. In Section 1.2, we describe various techniques that cope with memory latency. Section 1.3 and Section 1.4 present our research goals and summarize related work respectively. Finally, Section 1.5 provides an overview of the remaining chapters.

1.1 Software DSM Performance

To better understand the significance of the memory latency problem in software DSMs, we evaluate the performance of eight parallel applications. The experiments were performed on a state-of-the-art software DSM called TreadMarks [17], using eight 133 MHz

IBM RS/6000 workstations running AIX 4.1 connected by a 155 Mbps FORE System ATM LAN (further details of the hardware platform are given in Section 3.1.1). Seven of the eight applications we used are taken from the SPLASH-2 suite [37], except SOR, which is taken from the TreadMarks distribution. All timing measurements were done using the high-resolution timers provided by AIX 4.1.

Figure 1.1 shows the normalized parallel execution times for the applications. The execution times are broken down as follows. "*Memory idle time*" represents the time the memory subsystem stalls for remote memory requests to be fulfilled. "*Synchronization idle time*" is the time the system stalls for synchronization. "*Software DSM overhead*" is the overhead of executing the software DSM layer, which includes handling remote messages, executing the memory consistency protocol, and performing garbage collection. Finally, "*busy time*" is the amount of time spent doing useful computation (including interrupt times associated with software DSM, which we cannot isolate).

As we see in Figure 1.1, five of the eight applications spend 25% or more of their execution times stalled waiting for remote memory references. Therefore, techniques for coping with memory latency are clearly important.

1.2 Coping with Memory Latency

To cope with the large memory latency on a software DSM system, we can (i) reduce the number of remote memory accesses by using caches and (ii) overlap computation with communication using latency tolerance techniques such as prefetching and multithreading. In our study, we focus on purely software-based approaches to cope with memory latency because they are widely applicable across a broad range of hardware platforms.

1.2.1 Caching

Caches reduce memory access latency by turning a remote memory access into a local cache access. In software DSMs, local memories are used as caches for remote data. To provide the abstraction of a single shared memory, a software DSM must control accesses to these cached data through a well-defined *consistency model*. Since memory consistency is at the heart of a software DSM, it is very important to choose an appropriate consistency model for a specific environment. We will discuss consistency models further in Section 2.1. One of the factors which affects the number of remote memory accesses for a consistency model is its coherence protocol. For example, an update protocol propagates updates of shared data to every processor that has a copy. By trying to keep the locally cached variables valid, an update protocol can reduce the number of remote memory accesses. The tradeoffs of these protocols will be discussed in Section 2.3. Since caching remote data is now an integral part of most software DSMs, the techniques we are about to discuss build upon caching as a foundation.

1.2.2 Prefetching

The two main techniques for tolerating read latency as well as write latency are prefetching [3, 6, 32] and multithreading [2, 14, 19, 21]. The central idea of tolerating read latency is to split the *request* for the data from the *use* of that data, and find enough *parallelism* to keep the processor busy in between. The difference between prefetching and multithreading is that the former finds parallelism within *a single thread* of execution while the latter exploits parallelism with *multiple threads*. To find parallelism within a single thread, the request of a data item must be issued far enough in advance of its use in the execution stream. Prefetching therefore requires knowledge about the future access patterns of the application. In contrast, multithreading splits the data request

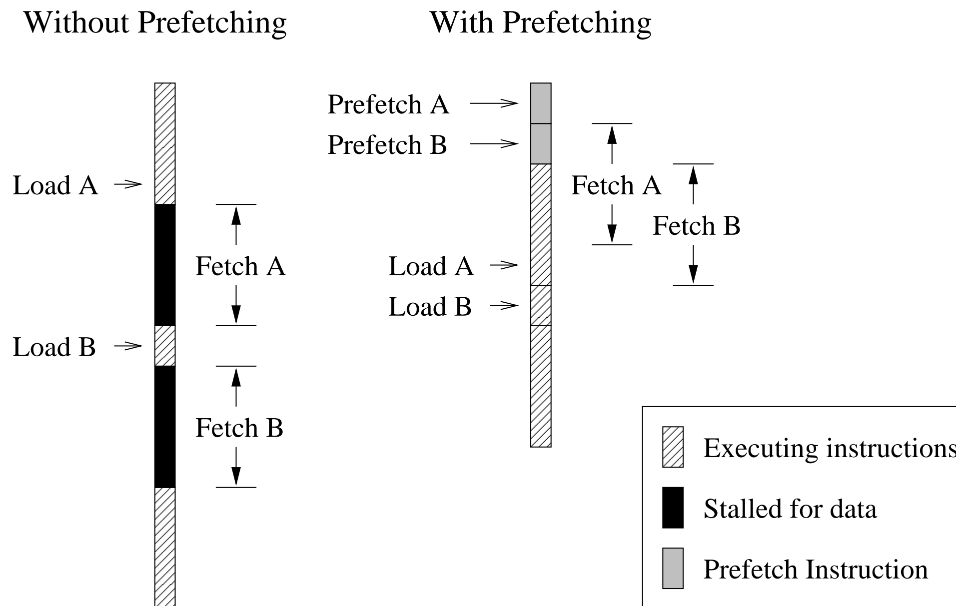

Figure 1.2: How prefetching improves performance

from its use by swapping out the currently active thread when it suffers an access miss and executing another thread to keep the processor busy. When the data comes back, the original active thread will be swapped in and its execution continued. In this subsection, we focus on prefetching. Multithreading will be investigated later in Section 1.2.3.

Figure 1.2 shows how prefetching can improve application performance. Suppose A and B are remote memory locations. In the case without prefetching, memory accesses of A and B force the processor to stall for the data to return before it can continue its execution. In the case with prefetching, however, prefetches are issued for the two data elements far enough in advance of their uses so that these two memory accesses are fulfilled locally and the processor does not stall. We see that prefetching not only overlaps communication with computation but also allows pipelined read accesses. One of the challenges of software-controlled prefetching in a software DSM is to maintain memory consistency.

1.2.3 Multithreading

For multithreading, when a long latency event occurs, instead of stalling for the data, the processor switches the current active thread to another thread within the same program so that the computation of the other thread can overlap with the communication of the current thread. The advantages of multithreading over prefetching are that it can handle arbitrarily complicated data access patterns and tolerate both memory latency and synchronization latency. However, to hide the long latency, multithreading might demand more concurrency than the program can offer. In addition, as different threads have different working sets, the interactions among threads may have constructive and destructive effects on performance. These problems are also discussed by Thitikamol *et al.* [35] (in their multithreaded software DSM called CVM [16]) and by Lo [26].

1.2.4 Overall Approach

Having discussed different techniques that cope with memory latency on software DSMs, we now consider how prefetching fits into the whole picture. First, we want to reduce as much memory latency as we can. Reducing latency is preferable over tolerating latency because it reduces the memory bandwidth requirement, which can be crucial to application performance. To reduce memory latency, caches can be used to reduce the number of remote memory accesses.

After reducing memory latency, we would like to tolerate any remaining remote memory access latency. To tolerate read latency, both prefetching and multithreading can be used. Prefetching is applicable when the data access patterns can be determined ahead of time. Multithreading, on the other hand, tolerates both memory and synchronization latency without requiring the ability to know the data access patterns far enough in advance.

1.3 Research Goals

The key goal of our research is to understand the performance of software-controlled prefetching in software DSMs. Our approach is to start by evaluating the performance of statically inserted prefetching operations, and then to introduce runtime information to try to enhance the prefetching performance. In addition to evaluating prefetching on its own, we also evaluate its performance when combined with multithreading.

1.4 Related Work

Prefetching has been applied to tightly-coupled multiprocessor machines [28, 31] where data that reside farther away in the memory hierarchy (such as the remote cache or the physical memory) are prefetched to the primary cache of the local processor. Mowry *et al.* [28, 31] have shown that software-controlled prefetching is effective at tolerating memory latency on both uniprocessor and large-scale multiprocessor architectures. The compiler can also insert prefetches into array-based code automatically, thus relieving programmers from the burden of inserting prefetches manually [28].

Recently, Mowry, Demke, and Krieger [30] have also applied prefetching to hide the latency of out-of-core applications by prefetching page faults. Similar to out-of-core applications, software DSM applications also have extremely high memory latency. Therefore, applying prefetching to software DSMs can potentially improve performance significantly.

In the context of software DSMs, Bianchini *et al.* [5] studied history prefetching. At synchronization points, the runtime system issues prefetches based on a prefetching heuristic. The authors discuss briefly the limitations of their prefetching scheme—e.g., it increases synchronization latency and does not work with small critical sections. Our scheme is different from theirs in that our prefetching operation is non-binding and software-controlled, which provides us with more flexibility in inserting prefetches.

Dwarkadas *et al.* [9] implemented a `Validate` operation that fetches *diffs* (runlength encodings of modifications of shared data) at the point where the operation is issued. To cope with the limitation imposed by lazy write notice propagation (which we will discuss in detail in Section 2.4), the `Validate` operation propagates modifications to the prefetched data at synchronization points. However, the bulk of their study focuses on compilation techniques to automatically insert prefetches into Fortran applications. The major difference between their prefetching scheme and ours is that our prefetching operation does not propagate modifications of the prefetched data at synchronization points. Instead, it queries the most recent modification at prefetch time by creating new intervals (which we will discuss in Section 2.4.1). In addition, our study focuses on evaluating the effects of prefetching on the memory miss latency and other software DSM overheads.

Finally, Gupta *et al.* compared and evaluated different latency reducing and tolerating techniques in a tightly-coupled multiprocessor environment [13]. In contrast, our hardware platform is a cluster of workstations and the memory system is a software DSM system. Similar to their study, we also conclude that combining both prefetching and multithreading produces mixed results.

1.5 Organization of Thesis

Chapter 2 describes several memory consistency models including lazy release consistency (LRC). We also discuss the issues of implementing prefetching within LRC and describe our implementation on our software DSM layer, TreadMarks. Chapter 3 studies the performance benefits of prefetching for software DSMs. We first describe our experimental framework and our prefetch insertion strategies. Then, we present our experimental results and discuss the limiting factors of our non-binding prefetching implementation.

At the end of the chapter, we also evaluate the performance of automatically inserted prefetches.

Chapter 4 discusses the possibilities of using additional runtime information to enhance the performance of prefetching. Using our prefetching implementation as a foundation, we develop two versions of prefetching with runtime information and evaluate their performance. Chapter 5 examines multithreading within software DSMs and compares it with prefetching. To deepen our understanding of these two latency tolerance techniques, we also evaluate the performance of combining the two techniques. Finally, Chapter 6 presents the conclusions of this thesis.

Chapter 2

Non-Binding Prefetching Implementation

In this chapter, we focus on the implementation of our non-binding prefetching operation. In Section 2.1, we briefly introduce several memory consistency models that have been proposed for DSM systems. In Section 2.2, we discuss lazy release consistency (LRC), which is the basis of our software DSM layer. Section 2.3 discusses different coherence protocols, which will provide some insights on how prefetching can be implemented. In Section 2.4, we propose several different prefetching techniques based on LRC and discuss how they differ from each other. We also justify our choice of a particular implementation over the others. Finally, in Section 2.5, we describe our software DSM layer, TreadMarks, and present our non-binding prefetching implementation.

2.1 Memory Consistency Models

In a shared memory environment, keeping memory consistent is a major issue. In this section, we briefly describe several memory consistency models that have been proposed for DSM systems.

Sequential consistency (SC) [20] is the most conservative memory consistency model. It is formally defined by Lamport as follows [20]:

> **Definition:** [A multiprocessor system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

The first implementation of SC on DSM system is IVY [25] where the virtual memory hardware is used to maintain memory consistency. Since SC provides a programming model of a single memory uniprocessor on a shared memory system, it is intuitive to programming and is generally viewed as a "natural" consistency model. However, Zucker *et al.* have shown that SC requires a substantial amount of communication [38].

Gharachorloo *et al.* developed a less conservative model called release consistency (RC) [12]. The intuition behind release consistency is that parallel programs should not have data races because they lead to non-deterministic results. It is therefore the responsibility of the parallel programs to provide sufficient synchronization to prevent data races. Updates to shared memory in one process do not need to be reflected in another process until they synchronize with each other. More formally, synchronization accesses in RC are divided into *acquire* and *release* operations. RC requires that the modifications made by processor p to shared memory are visible to another processor q when a subsequent release by p is visible to q. In this way, write accesses can be pipelined and hence their latency can be tolerated. Gharachorloo [11] have shown that RC reduces remote memory access penalties and outperforms SC on a simulation architecture based on the Stanford DASH multiprocessor [23].

In a software DSM implementation of release consistency, it is also important to reduce the number of network messages (since they involve traps into the operating system

kernel, interrupts, context switches, and the execution of several layers of networking software). Therefore, rather than pipelining writes as in the RC implementation of DASH [24], Munin's write-shared protocol [7] buffers them until a release. At release time, all writes going to the same destination are combined into a single message. To further reduce the number of messages and the amount of data exchanged, Keleher *et al.* proposed lazy release consistency (LRC) [18] which does not make shared accesses globally visible at the time of a release. Instead, shared accesses only have to be performed at other processes as they synchronize with the performing process (i.e. at the time of an *acquire*). This reduces unnecessary communication by not sending invalidation messages to processes that never access the invalidated data. In addition, as the invalidation message travels the same route as a subsequent lock transfer, a pair of messages can be eliminated by piggybacking the invalidation to the synchronization transfer. Therefore, LRC can reduce communication requirement over that needed by RC.

An alternative memory consistency model for software DSM is entry consistency (EC) [4]. The EC programming model requires programmers to explicitly associate shared data with synchronization objects. After a processor passes through an acquire, EC guarantees that it sees the most recent copy of the data associated with the acquired synchronization object. Although EC reduces false sharing by explicitly associating shared data with synchronization objects, finding these associations can be difficult in some occasions. Hence, LRC is more attractive than EC because it requires little or no change to the existing shared memory programs.

2.2 Lazy Release Consistency

Since LRC is an important consistency model for software DSMs, we implement our prefetching operation within LRC. In this section, we provide enough background in-

formation about LRC for the discussion of our prefetching implementation later in this chapter.

2.2.1 Write Notices and Timestamps

Since LRC is based on release consistency, synchronization primitives are classified as either acquires or releases. In LRC [18], the execution of each process is divided into *intervals*, each denoted by a *timestamp*. Each processor has a vector timestamp that contains the timestamps of all the processors that it knows. Each time a processor performs a release or an acquire, a new interval is created with a larger timestamp. For all of the pages that have been modified within an interval, *write notices* are created. A write notice is an indication that a page has been modified by a particular processor in a particular interval, but it does not contain the actual modifications. During synchronization, the acquiring processor piggybacks its vector timestamp on the request message to the releasing processor. From the received vector timestamp, the releasing processor can determine the write notices that are not present in the acquiring processor, and need to be propagated. In the reply message, the releasing processor replies with its vector timestamp and piggybacks the write notices that are not present in the acquiring processor. When the reply message is received, the acquiring processor computes its new vector timestamp according to the pair-wise maximum of its vector timestamp, and the vector timestamp it gets from the releasing processor. The acquiring processor then invalidates all pages for which write notices are received.

To maintain memory consistency, LRC enforces the *happen-before-1* [1] partial order among intervals. The partial order is encapsulated in: (i) write notices, which denote modifications and (ii) vector timestamps, which provide ordering information for these modifications.

2.2.2 Multiple-Writer Protocol

False sharing occurs when two or more processes are accessing different parts of the same page of data and at least one of the accesses is a *write.* The effect of false sharing is most intuitive in a single-writer protocol. With a single writer protocol, a writer is required to have sole access to a given page before performing any modification. After the modification is done, the protocol invalidates all copies of the page in other processors. Subsequent access to an invalidated copy of the page results in a remote access miss regardless of whether the access and the modification are related or not. The worst situation occurs when two processors are simultaneously updating independent parts of the same page. With this protocol, the page may "ping-pong" across the network and lead to enormous amount of network traffic. While this problem exists in both snoopy-cache and directory-based coherence multiprocessors, its effects are more prevalent in software DSM because the consistency protocol on software DSM tracks data accesses at the granularity of virtual memory pages rather than small cache blocks.

To eliminate the ping-pong effects in false sharing, LRC employs a *multiple-writer protocol* [7]. With a multiple-writer protocol, processors can independently write to different parts of the same page simultaneously. To accomplish this, the protocol write-protects all shared pages initially. When a write occurs, a copy of the page, or a *twin*, is created. The protocol then unprotects the page to allow further accesses to occur without software intervention. If another processor attempts to modify the same page, the same sequence of events occurs. Notice that all of these are local operations, and do not require any message exchange, unlike in the case of a single-writer protocol. When synchronized access is required, a word-by-word comparison of the two versions of the page is performed to create a *diff*, which is a runlength encoding of the modification to the page. Once the diff is created, it is sent to the acquiring processor so that it

can be applied to bring the page up-to-date. At the same time time, the twin is also discarded. Since actual communication occurs only when synchronized access is required, multiple-writer protocol eliminates the ping-pong effects in false sharing. In addition, it significantly reduces overall bandwidth requirement because diffs are typically smaller than a page.

2.3 Coherence Protocols

So far, we have not mentioned how modifications are propagated among processors. The actual data movement depends on which coherence protocol is being employed. In this section, we focus on both the *update* and the *invalidate* protocols.

An update protocol minimizes the number of remote memory read accesses by actively propagating updates of shared variables to every processor that has a copy of the original data. One of its drawbacks is that these updates are propagated to all of its copies, including those that are no longer being used. It can therefore increase network traffic. However, by keeping the copies up-to-date, an update protocol reduces the number of remote memory misses.

An invalidate protocol, on the other hand, does not propagate updates to all processors that have copies. Instead, the copies are merely invalidated and the modifying processor obtains exclusive access to the page, so that further modifications do not require communication. Communication occurs only when a processor requires synchronized accesses to an invalidated page. Therefore, if the degree of sharing of a page is not substantial, an invalidate protocol usually sends fewer coherence messages than an update protocol does. However, if the degree of sharing is high, it might incur more remote memory read accesses than an update protocol does.

Between the two protocols, an invalidate protocol typically requires the least network

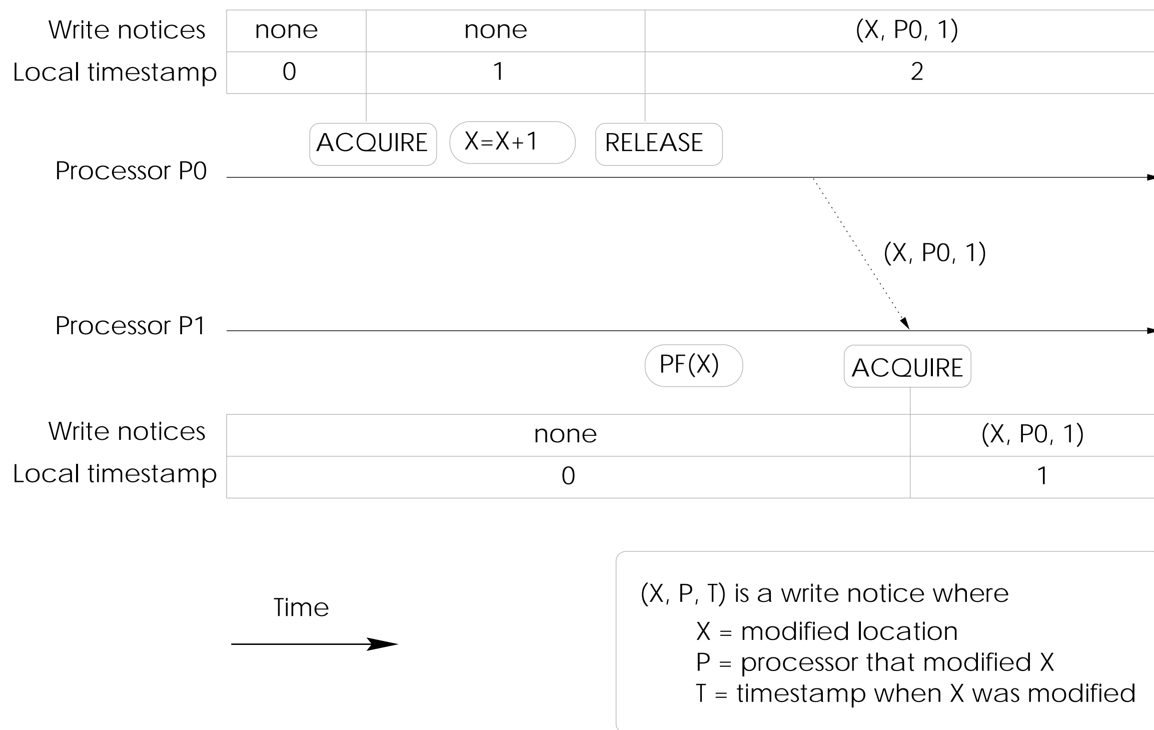

Figure 2.1: Illustration of a limitation of prefetching in lazy release consistency.

bandwidth [10]. Because of the high per-message cost in software DSMs, the lower network bandwidth requirement of an invalidate protocol is especially attractive. However, it also has rather high memory latency since many shared memory accesses are remote accesses. To achieve high performance, we need to find ways to cope with the large memory latency in an invalidation-based protocol.

2.4 Prefetching Techniques

Prefetching is a latency tolerance technique that is shown to be effective in many studies [3, 6, 27, 29, 30, 31]. In this section, we discuss the implementation of our prefetching operation within a LRC-based DSM system.

On a directory-based coherence multiprocessor machine like the SGI Origin, each

memory block is associated with a *home* node that keeps the location of the up-to-date copy of the memory block [22]. In this case, prefetch instructions can be issued at any time to obtain the latest copy of the cache line. For a timestamp-based LRC, however, there is no notion of a page's home node, and a processor does not know where to find the up-to-date copy of a page until it has performed an acquire. If prefetches are delayed until after the immediately preceding acquire operation, there may not be enough time to hide the prefetch latency. To find enough time to hide the latency, we have to cope with the lazy propagation of write notices without violating program correctness. Therefore, implementing prefetching within LRC is non-trivial.

To illustrate this more clearly, we show an execution trace of a LRC-based DSM application in Figure 2.1. The dotted line between processors represents a synchronization message in which a write notice is propagated. In this case, the write notice is piggybacked on the first acquire message in processor *P1*. Now, let us consider the prefetch operation **PF(X)** which we inserted in processor *P1*. It fails to fetch the required modification because at that point in the program, although processor *P0* has already modified the variable **X**, the associated write notice has not been propagated to processor *P1* yet. Therefore, the prefetch operation does not know where to find the required modification for **X**, unless it broadcasts the request to all of the processors. Since software DSM has high network overhead, broadcasting prefetches (which increases the communication requirement) may hurt application performance. To make the prefetch operation successful, it has to be postponed until after the first acquire operation **ACQUIRE** in processor *P1* is completed. By then, it can use the received write notice to determine where to find the modification for the variable **X**. From this simple example, we see that the lazy propagation of write notices in LRC can limit the prefetching effectiveness.

---

prefetch (&X); /* *prefetch shared variable outside critical section* */
...
LOCK ($L1$); /* *enter critical section* */
$X = X + 1$; /* *modify shared variable* */
UNLOCK ($L1$); /* *exit critical section* */

Figure 2.2: Example of when a binding prefetching is illegal.

2.4.1 Binding vs. Non-binding Prefetches

In a multiprocessor environment, prefetches can be classified as being either *binding* or *non-binding*. They differ regarding when the prefetched value is bound.

Binding Prefetching

Binding prefetching binds the data value at the point when the prefetch instruction is issued. The problem with binding prefetching is that if other processors modify the data in the interval between when the prefetch instruction is issued and the data is used, the prefetched data will become stale. This places significant restrictions on when binding prefetches can be issued. Figure 2.2 shows a case where binding prefetching is illegal because if another processor enters the critical section first, the prefetched data will become stale and an incorrect value of X will be produced. Therefore, with binding prefetching, program correctness depends on where prefetches are issued. In LRC, the safest place to issue prefetches would be inside a critical section since we can guarantee that no other processors are modifying the prefetched data. For the example in Figure 2.2, the prefetch operation must be inserted after the **LOCK** statement.

However, we can foresee that postponing prefetching until after the immediately preceding acquire operation has at least two drawbacks: (i) it may result in too little time to hide the latency (particularly inside small critical sections); and (ii) it delays handling normal synchronization messages and thus increases synchronization idle time. The re-

sults published by Bianchini *et al.* [5] confirm these drawbacks. Hence, we will focus on supporting non-binding prefetching.

Non-binding Prefetching

With non-binding prefetching, the prefetched data remains visible to the coherence mechanism. The data value is bound when it is actually used so that it is guaranteed to be up-to-date, unlike in the case of binding prefetching. This allows us to issue prefetches at any point in the program without violating program correctness [28, 31].

To further illustrate the advantage of non-binding prefetching over binding prefetching, we consider the code fragment in Figure 2.2 again where the shared variable X is modified within a very small critical section. If we issue a prefetch after the synchronization point, then there obviously is not enough time to hide the remote access latency. We must therefore move our prefetch operation ahead of synchronization. This would be illegal for binding prefetching, but it is perfectly legal in non-binding prefetching because the underlying coherence mechanism will keep the shared data coherent.

2.4.2 Implementing Non-binding Prefetching Within Lazy Release Consistency

In LRC, where can we obtain the necessary modifications to bring a given page up-to-date? Recall that a write notice is an indication that a page has been modified by a particular processor in a particular interval. By following the list of write notices, we can easily find out which processors have the modifications. With this information, we can issue prefetches for *the modifications associated with the write notices.*

However, due to the lazy propagation of write notices in LRC, prefetching modifications for the existing write notices can only cover the modifications made in previous

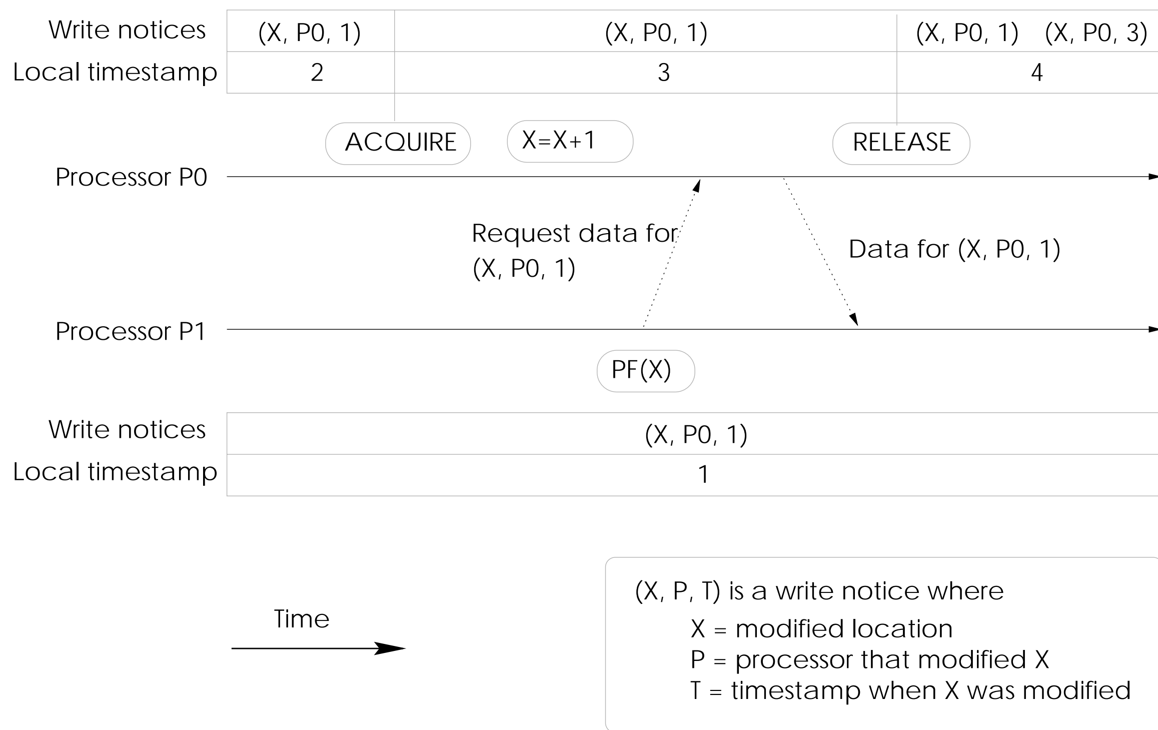

Figure 2.3: Simple non-binding prefetching within LRC (only parts of the modifications are retrieved).

intervals. To elaborate, let us consider the prefetch operation **PF(X)** in Figure 2.3. When processor *P1* issues the prefetch operation, it only knows (from the write notice it has) that the shared variable X has been modified in the first interval by processor *P0*, and therefore it only asks for that modification. With this simple scheme, processor *P0* replies exactly what *P1* has requested, although the prefetching variable has already been modified the second time in the third interval. This limitation can actually hurt performance. For example, in this case, an extra memory request message has to be sent to processor *P1* when processor *P0* actually accesses the page.

To try to avoid sending this extra message in this case, there are two possible techniques: (i) one can propagate updates to the prefetched pages at synchronization; and (ii) rather than simply requesting the modification associated with an existing write notice,

one can request the latest modification at prefetch time.

Propagating Updates of Prefetched Data at Synchronization

One way to keep the prefetched page up-to-date is to automatically update it during synchronization. To accomplish this, the processor must keep track of the list of prefetched pages and the associated modifications. At acquire time, the servicing processor replies with write notices as well as the more recent modifications for the prefetched pages in the synchronization messages. In the case of a lock acquire, the releaser of the lock checks if it has modified the prefetched pages and propagates modifications if necessary. In the case of a barrier arrival, all processors check the entire list of prefetched pages to see if they are modified and exchange modifications if necessary.

The problem with this technique is that the runtime overhead of tracking the modifications is significant when the list of prefetched pages is large (especially in the case of barrier arrival). In addition, if the prefetched data does not fit in a single synchronization message, multiple messages must be sent which increase runtime overheads. Dwarkadas *et al.* [9] have similar findings in their study.

Requesting the Most Up-to-date Modification at Prefetch Time

Rather than keeping track of the modifications made to all of the prefetched pages (which incurs high overhead), we can request the most recent modification at prefetch time. However, since we are requesting modifications ahead of synchronization, we have to be careful not to violate program correctness. For example, if a processor simply creates the most recent diff upon servicing a prefetch and sends it to the prefetching processor, it may violate program correctness. The reason for this is that after the prefetch is serviced, the servicing processor may modify the same page again. In this case, the prefetching processor must know how to apply the different modifications in the correct

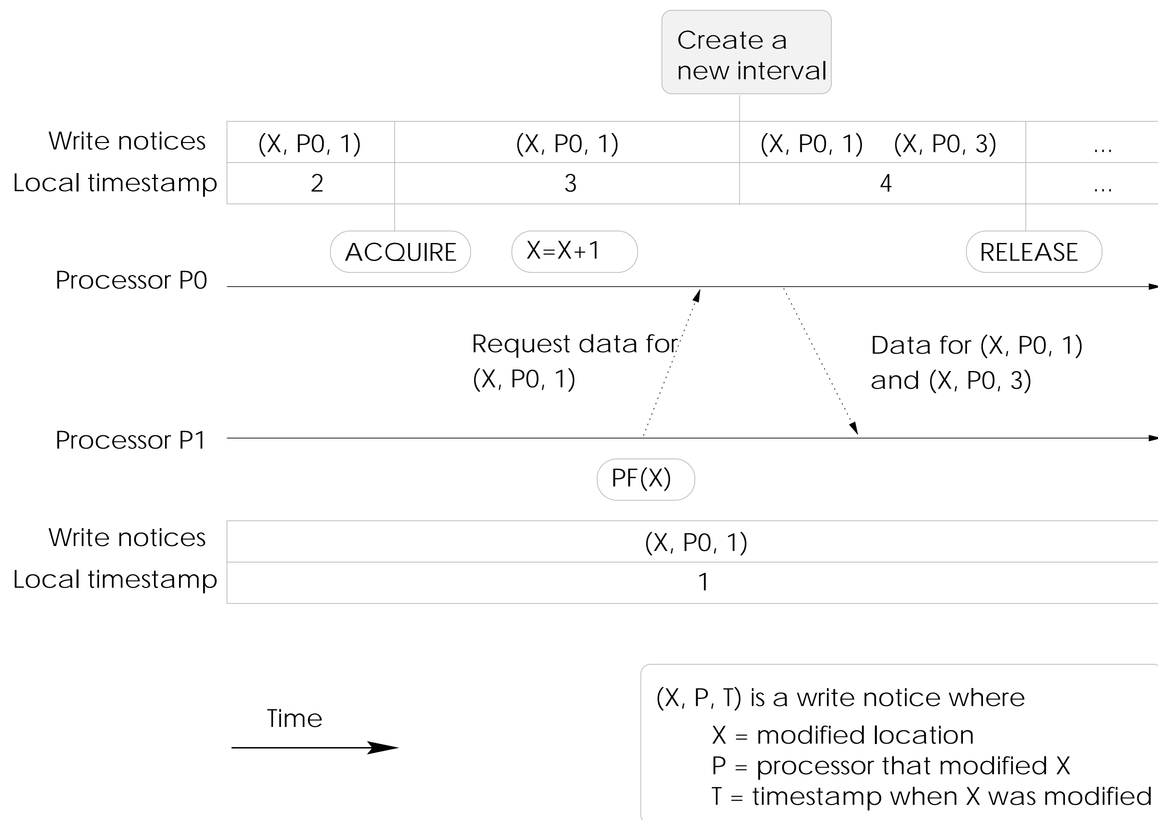

Figure 2.4: To request the most up-to-date modification within LRC, a new interval is created upon servicing a prefetch request.

order. Therefore, the major issue is how to order the prefetched data appropriately. In LRC, the ordering of modifications are kept in write notices and intervals. Hence, by creating new write notices and intervals at prefetch time, we can order our prefetched diffs such that they can be applied appropriately. The extra cost involved with this technique is the overhead of creating a new interval, which is smaller than the overhead of the previous technique. Figure 2.4 shows how this is accomplished. When processor *P0* receives the prefetch message, it creates a new interval to encapsulate the modification of variable *X* made in the third interval. It then replies to the requesting processor *P1* with both modifications.

Regarding the correctness of this technique, we know (as discussed earlier in Sec-

tion 2.2) that memory consistency is maintained as long as the modifications are applied according to the happened-before-1 partial order (i.e., in increasing vector timestamp order) while the time of actual data movement does not matter. When we create new intervals, we are monotonically increasing the timestamp of the local processor. Therefore, we will not violate the partial ordering among intervals and the modifications can still be applied in increasing vector timestamp order. This technique is thus truly non-binding and does not violate program correctness. Therefore, our prefetching implementation is built upon this technique.

2.5 Implementation within TreadMarks

In this section, we first describe our software DSM layer called TreadMarks. Then we discuss its implementation of the lazy release consistency protocol and present the details of our non-binding prefetching mechanism.

2.5.1 TreadMarks

TreadMarks [17] is a software DSM system based on lazy release consistency with an invalidation-based protocol. It makes use of the existing virtual memory facilities [25] provided by the operating system to trap memory read/write operations to maintain memory consistency.

When TreadMarks initializes, it statically allocates 64MByte of physical memory in each workstation as shared memory. Each shared page has a designated *page manager* that holds a copy of the page, an *empty* flag, and a *valid* flag to indicate the different states of the page. Initially, all of the pages have processor 0 as their page managers and the pages on processor 0 are unprotected, valid, and non-empty while those on other processors are read-write protected, invalid, and empty. An empty page is made valid

by bringing the entire page from its page manager. An invalid page is made valid by requesting the missing modifications and applying them in increasing vector timestamp order.

Upon a memory read fault, the page fault handler makes the page valid and write protects the page. Upon a memory write fault, the page fault handler first makes the page valid, then creates a *twin*, which is the unmodified version of the page. Finally, it removes the write protection of the page. If another processor requests the modifications made to a page, the given (i.e. "servicing") processor compares the modified version of the page with its twin and creates a runlength encoded record (called a *diff*). Once the diff is created, the twin is discarded. The record will be sent as a reply to the requesting processor. At this point, the servicing processor protects the page from being written so that the next modification can be captured.

In TreadMarks, a *lock release* and a *barrier arrival* correspond to the release operation in LRC while a *lock acquire* and a *barrier departure* correspond to the acquire operation. Intervals are created when actual communication occurs (i.e. when a *remote* lock request is received). This can save some software overhead if a processor re-acquires a lock before another processor requests it.

TreadMarks's garbage collector is invoked when the memory used by twins, intervals, and write notices reaches a predefined level. After the memory is reclaimed, the garbage collector ensures that *all* of the pages are either valid or empty.

2.5.2 Pseudo-codes

We now present further details on our prefetching implementation within TreadMarks. Our prefetch operation appears as a normal procedure call in the source code where the prefetch address is passed as an argument. Figures 2.5 shows the pseudo-codes for the routines involved in implementing prefetches.

```
/* Prefetch request procedure */
prefetch (address) {
    /* Calculates which page the supplied address belongs to */
    page ← page_align (address);
    if (page.valid = true) {
        /* If the page is already valid, there is no need to prefetch */
        return;
    } else {
        if (page.empty = true) {
            /* If the page is empty, prefetch the page asynchronously */
            send prefetch page request;
        }
        if (page.write_notices ≠ empty) {
            /* Prefetch required modifications */
            writes ← set of write notices which record the most recent
                modifications that are not present locally;
            send prefetch diff requests to the processors listed in writes;
        }
    }
}

/* Service prefetch requests */
prefetch_service (message) {
    if (message.type = PF_PAGE_REQUEST) {
        /* If a page is requested, return a page */
        send page to requesting processor;
    } else if (message.type = PF_DIFF_REQUEST) {
        /* If diffs are requested, return diffs */
        if (page.dirty = true) {
            /* If the page is being modified, create a new interval
            to capture the latest modifications */
            create a new interval;
        }
        create required diffs;
        send diffs to requesting processor;
    }
}
```

Figure 2.5: Non-binding prefetching pseudo-code. (Continued on next page.)

```
/* Handles replies from prefetch_service() */
prefetch_reply (message) {
    if (page.valid = true) {
        /* If the page is already valid, drop the prefetched data */
        return;
    }
    /* Otherwise, handle the prefetch reply appropriately */
    if (message.type = PF_PAGE_REPLY) {
        copy message.data into page address;
        page.empty ← false;
    } else if (message.type = PF_DIFF_REPLY) {
        copy message.data into local diff storage;
    }
}

/* A modified page fault handler to incorporate prefetched data */
page_fault_handler (page) {
    if (page.valid = false) {
        if (page.empty = true) {
            /* If the page is not prefetched, perform a normal page request and
            wait for its return */
            perform a normal page request;
        }
        writes ← set of write notices (which do not have diffs)
            in increasing vector timestamp order;
        foreach (w ∈ writes) {
            if (w.diff has been prefetched)
                apply w.diff to page;
            else
                request diffs for page;
        }
        wait until all diffs have come back;
        page.valid ← true;
    }
    call default page fault handler to unprotect the page
        according to the required type of access;
}
```

Figure 2.5: Non-binding prefetching pseudo-code. (Continued from previous page.)

On the requesting processor, when **prefetch()** is called, it first checks the state of the page. Obviously, if the page is valid, no request is necessary and the function returns immediately. However, if the page is empty, an asynchronous request message is sent to the page manager to request the entire page. After the message has been sent, the function checks the write notices associated with the given page and finds the set of write notices whose vector timestamps dominate the vector timestamp of the local copy of the page. For this resulting set of write notices, requests are sent to the set of processors where there are outstanding modifications to the page. After the request messages are sent, the prefetch procedure returns to the application *immediately* (i.e. the messages are handled asynchronously.).

On the servicing side, once a prefetch request is received, the **prefetch_service()** routine is invoked. Depending on the type of the request, either an entire page or the required modifications are sent in the reply message. If the page has been modified (i.e. it is "dirty"), a new interval will be created to capture the most recent modification. This modification, together with the requested modifications, will be sent to the requesting processor.

When a prefetch reply is received by the requesting processor, the **prefetch_reply()** routine is invoked. If the page is already marked valid, the prefetched data will not be stored. Otherwise, the prefetched data will be stored. After the reply message is handled, the page is still visible to the coherence mechanism. To make this possible, the prefetched diffs are not applied locally until the page is actually referenced.

It is important to note that our prefetch requests are unreliable since they may be dropped by the network. One of the problems with supporting reliable prefetch requests is that an ATM network is likely to drop messages under high contention. When this happens, we do not want to retry sending prefetch messages through the network, since this make the contention problem even worse. Therefore if the prefetch request fails to

return before the actual reference, the **page_fault_handler()** routine does not wait for the prefetch to return (since it may never come back), but instead issues a normal remote memory request at that time. This doubles the number of messages normally required for a remote memory reference.

The **page_fault_handler()** routine also issues remote memory requests for data that cannot be found locally (the *else* statement in the pseudo-code). The prefetched data and the requested data are applied in increasing vector timestamp order to ensure that our prefetch operations are truly non-binding and never violate program correctness. The final step for the **page_fault_handler()** is to unprotect the now valid page accordingly to allow further accesses.

Chapter 3

Non-binding Prefetching Performance

In this chapter, we present the results of our experiments to illustrate the performance benefits of our non-binding prefetching implementation. Section 3.1 describes our experimental framework, including the hardware platform and the benchmark applications. Since programmer-inserted prefetching represents a rough estimate of the expected performance improvement offered by prefetching, we first discuss its performance in Section 3.2. Our experiments indicate that software-controlled non-binding prefetching can reduce memory stall time and yield substantial performance improvement. However, inserting these prefetches manually can be both difficult and time-consuming. It is therefore beneficial to investigate automatic prefetch insertion. In Section 3.3, we use a prefetching compiler to insert prefetches into application source codes. We hope to learn from our experiments the characteristics of a software DSM system that can affect the performance of a prefetching compiler. Finally, in Section 3.4, we summarize the results of this chapter.

3.1 Experimental Framework

This section briefly describes the applications and the hardware we used throughout our studies.

3.1.1 Hardware Platform

In our experiments, we used eight IBM RS/6000 workstations running AIX 4.1. Each workstation contains a 133 MHz PowerPC 604 processor, with split 16KB primary instruction and data caches, a 512KB unified secondary cache, and 96MB of physical memory. The machines are connected by a FORE Systems ASX-200WG ATM LAN switch using 155Mbps OC3 multimode fiber optic links. The TreadMarks processes communicate using a lightweight reliable communication protocol built on top of UDP. All timing measurements were done using the high-resolution timers provided by AIX 4.1.

3.1.2 Applications

The applications we chose to evaluate are FFT, LU-NCONT, LU-CONT, OCEAN, RADIX, SOR, WATER-NSQ, and WATER-SP. All but SOR are taken from the SPLASH-2 suite [37], and SOR is taken from the TreadMarks distribution. Table 3.1 briefly describes these applications, their input data sizes, and their parallel execution times. (Further details can be found in studies done by Woo *et al.* [37] and Liviu *et al.* [15].)

FFT performs a 1-D fast Fourier transform using a six-step FFT method. The communication in an n-point FFT is essentially a $\sqrt{n} \times \sqrt{n}$ matrix transposition. Each processor is assigned a contiguous set of $\frac{n}{p}$ rows, and the source and destination matrices are reversed for every transpose. When solving the problem, each processor reads an $\frac{\sqrt{n}}{p} \times \frac{\sqrt{n}}{p}$ sub-matrix from every other processor and writes to its local partition of rows.

Application	Description	Problem size	Parallel Execution Time
FFT	1D fast Fourier transform using six-step FFT method	256K data points	11.2 sec
LU-NCONT	blocked LU factorization of dense matrix where the blocks are allocated *non-contiguously*	1024 × 1024 matrix with block size 32	51.8 sec
LU-CONT	the same LU factorization as in LU-NCONT except that the blocks are allocated *contiguously*	1024 × 1024 matrix with block size 128	8.3 sec
OCEAN	simulation of the large-scale ocean movements based on eddy and boundary currents	258 × 258 grid	53.9 sec
RADIX	integer radix sort kernel	1M keys with maximum key value of 2M	27.2 sec
SOR	red-black successive over-relaxation on a grid	50 iterations on 2000 × 2000 array	9.5 sec
WATER-NSQ	simulation of the forces and potentials among water molecules in liquid states	9 steps on 512 molecules	13.8 sec
WATER-SP	same as in WATER-NSQ except that the molecules are represented by a *linked-list* instead of an *array*	9 steps on 4096 molecules	22.5 sec

Table 3.1: Application descriptions, problem sizes, and parallel execution times

LU-NCONT performs blocked LU factorization of a dense matrix. This version implements the $n \times n$ matrix to be factored with a 2-D $N \times N$ array of $B \times B$ blocks. Although the data structure is more natural, it forces the data to be allocated *non-contiguously* and thus results in substantial false sharing.

LU-CONT is a different implementation of dense LU factorization. It uses an optimized data structure (a 4-D array) for the matrix so that the blocks can be allocated *contiguously* in memory, which greatly reduces false sharing.

OCEAN simulates large-scale ocean movements based on eddy and boundary currents [34]. The dominant sharing pattern is an iterative near-neighbor one, complicated by the use of a multigrid solver. The data in each subgrid are allocated contiguously using a 4-D array. The sub-grids are modified by their owners and read by their neighboring processors.

RADIX is an integer radix sort application. During each iteration, a processor reads its locally allocated keys from a source array and writes them to a destination array in a highly scattered and irregular permutation.

SOR performs red-black successive over-relaxation on a grid. The program execution is divided into two phases separated by a barrier. Within each phase, a processor reads the boundary elements written by the neighboring processor in the previous phase.

WATER-NSQ uses an $O(n^2)$ algorithm to simulate the forces and potentials among water molecules in liquid states. The molecules are allocated contiguously and a private force array is maintained in each processor. After the force array has been calculated, a processor updates its own molecules and the shared molecules in other processors.

WATER-SP solves the same problem as WATER-NSQ by using a spatial directory instead of a brute-force method. It is therefore more suitable for large problems. The 3-D physical space is broken up into cells, which are assigned to each processor contiguously. Within each cell, there is a linked list of molecules in the cell. A processor reads data from its neighbor for molecules that are within the cut-off radius.

3.2 Programmer-Inserted Prefetching

One of the requirements for software-controlled prefetching is that prefetch operations must be explicitly inserted into the applications. In this section, we focus on programmer-inserted prefetching. Later in Section 3.3, we will discuss the performance of compiler-inserted prefetching.

3.2.1 Inserting Prefetches

We inserted explicit prefetch procedure calls into the source code of the applications as follows. With the exception of WATER-SP, all of the other applications use arrays as

their primary data structures (WATER-SP uses linked lists). Therefore we apply Mowry's prefetching algorithm [28] to these applications to isolate dynamic miss instances through loop splitting techniques (e.g., strip mining) and to schedule prefetches far enough ahead using software pipelining. Prefetching for software DSM is quite similar to prefetching page faults to hide the I/O latency of out-of-core applications [30].

The non-binding property of our prefetches is particularly important for WATER-NSQ. WATER-NSQ is a multiple-producer, multiple-consumer application where the major misses occur when updating shared locations protected by locks. With non-binding prefetching, we can insert prefetches *before* the locks, thereby giving us more time to hide the latency.

We inserted prefetches into WATER-SP (a pointer-based program) by hand using a variation of the *history prefetching* scheme proposed by Luk and Mowry [27]. Since the recursive data structures do not change once they are allocated, we create a new local array and use it to record pointers to the elements in the traversal order. When prefetches are issued, they simply use the addresses stored in this array. As a result, we eliminate the pointer-chasing problem and cover 97% of the misses.

3.2.2 Overall Performance

Figure 3.1 shows the performance improvement of applying prefetching on the eight applications we studied. For each application shown on the graph, the column on the left (**O**) represents the parallel execution time without prefetching, and the one on the right (**P**) represents the parallel execution time with prefetching. Both are normalized to the original parallel execution time. For the prefetching column, the topmost "*prefetching overhead*" category represents the overheads of issuing prefetch requests. In addition, "*busy time*" includes prefetch overheads associated with loop transformation and unnec-

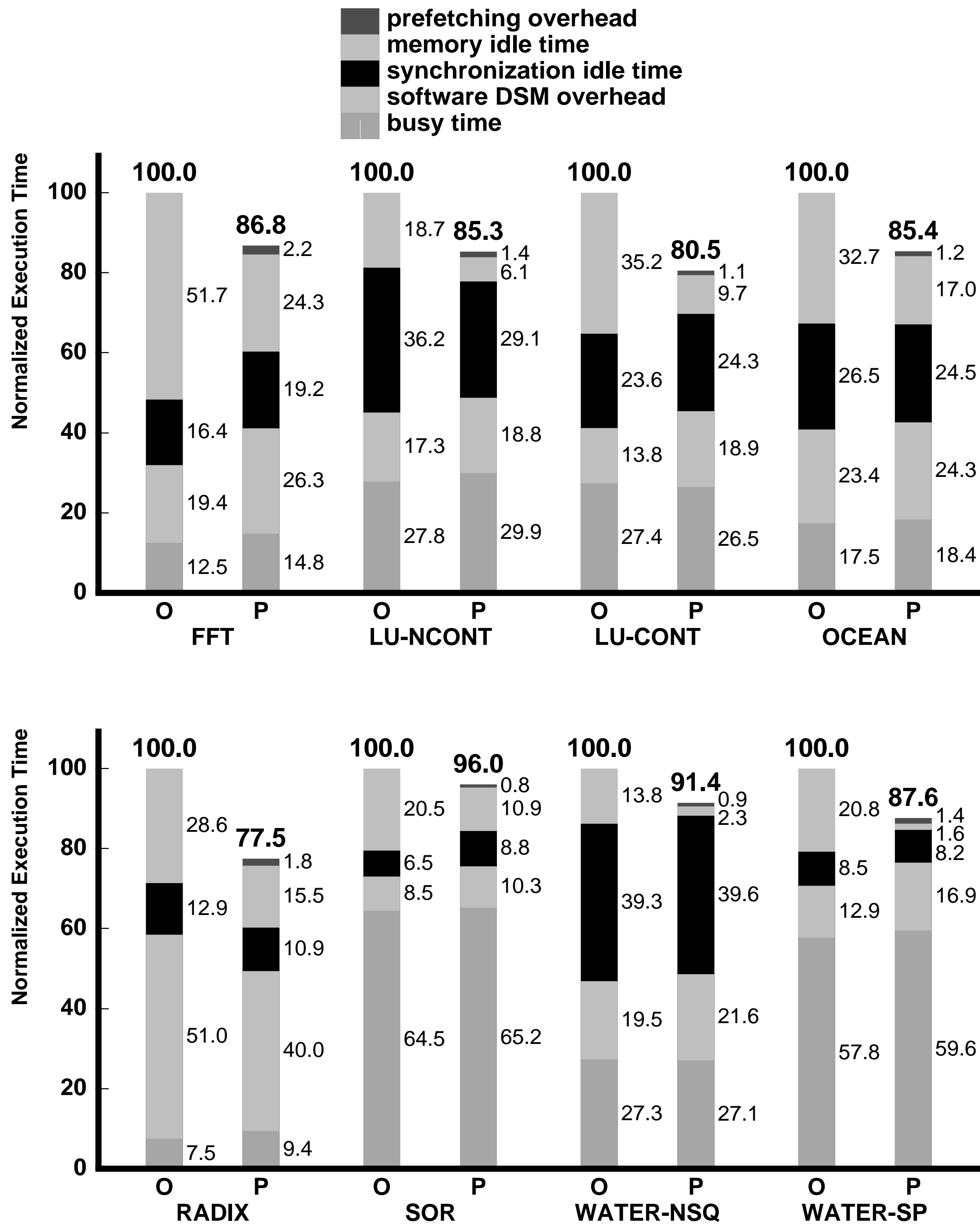

Figure 3.1: Performance improvement from programmer-inserted prefetch (**O** = original, **P** = programmer-inserted prefetch)

essary prefetches[1]. The remaining categories are the same as in Figure 1.1.

As we see in Figure 3.1, non-binding prefetching reduces memory idle times in all applications. In six of the eight applications, the reduction in memory idle times ranges from 45% to 92%. Despite the fact that each prefetch that generates remote messages (i.e. those that are not dropped immediately because they are unnecessary) requires roughly 140 μsec of software overhead, the "*prefetching overhead*" category remains quite low (under 3%) in all cases. The "*software DSM overhead*" increases for the following two reasons: (i) prefetch messages are more expensive than normal memory request messages because they involve creating new intervals if the prefetching pages are dirty; and (ii) prefetches that fail to return in time result in retry messages (since prefetch messages are unreliable). Some applications have reduced software DSM overhead because of reduced garbage collection time[2]

Although prefetching does not have a direct impact upon synchronization idle time, this time was reduced in LU-NCONT, OCEAN, RADIX, and WATER-SP as a result of improved load balance.

"*Busy time*" generally increases because of the extra prefetch overhead to generate prefetch addresses in the applications and the overhead of unnecessary prefetches. The slight reduction in busy times in WATER-NSQ is due to small measurement variations, since our experiments were all performed on real hardware.

Finally, with an overall speedup ranging from 4% to 29%, we can see that the benefit of prefetching is large enough to more than offset its runtime overhead.

[1]The overhead associated with an unnecessary prefetch involves an address calculation and checking the valid flag for the given page locally.

[2]With prefetching, the garbage collector can validate dirty pages more quickly because some of the modifications required to bring the pages up-to-date have already been prefetched.

Benchmarks	Unnecesary Prefetches (%)	Coverage Factor (%)
FFT	3.33	99.84
LU-NCONT	28.74	89.45
LU-CONT	11.48	94.54
OCEAN	35.44	74.41
RADIX	11.39	94.27
SOR	3.89	99.96
WATER-NSQ	77.91	91.36
WATER-SP	79.32	97.61

Table 3.2: The percentages of unnecessary prefetches and the percentages of remote misses covered by prefetches (i.e. coverage factors)

3.2.3 Prefetching Effectiveness

To understand why prefetching fails to hide all of the memory latency, we focus on two issues: (i) the success of our prefetch insertion algorithm in selecting appropriate references to prefetch and in scheduling prefetches early enough to hide the latency; and (ii) the effects of network traffic.

The Success of Our Prefetch Insertion Algorithm

For comparison with the ideal prefetch insertion case, two concepts are essential: the number of *unnecessary prefetches* and the *coverage factor*. Unnecessary prefetches are prefetches that find their data locally. The coverage factor is the fraction of original remote misses[3] that are prefetched. An ideal prefetch insertion scheme will have a 100% coverage factor and no unnecessary prefetches. Table 3.2 shows the corresponding values from our experiments.

If we focus on unnecessary prefetches, we find that they do not have a large negative impact on application performance. For example, in WATER-SP, although more than 79% of the prefetches are unnecessary, the increase in overhead is insignificant. Clearly,

[3]The number of detectable memory references in the prefetching case may be slightly different from that in the original one. The reason for this is that the time at which the TreadMarks' garbage collector is invoked in the prefetching case may be different from that when it is invoked in the original case.

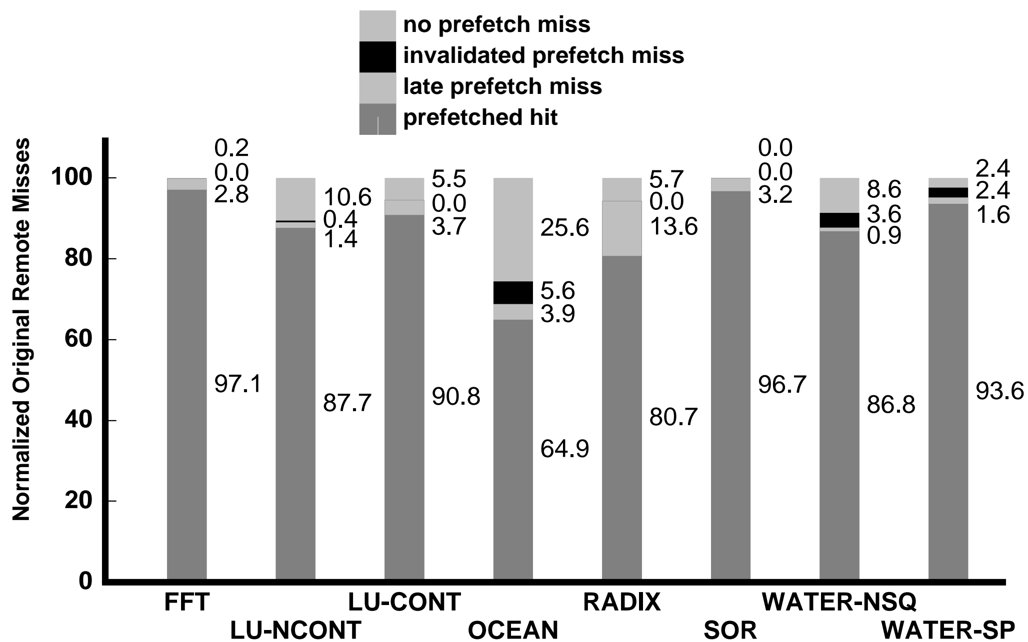

Figure 3.2: Breakdown of impact of programmer-inserted prefetching on the original remote misses

wasted overhead is not the real problem.

The real question is why the *upside* of prefetching is not larger. Based on the large coverage factors in Table 3.2, we would expect to see larger reductions in the memory idle times. For example, although 99.8% of the remote misses in FFT were prefetched, nearly half of the original memory stall time remains. To help resolve this question, Figure 3.2 shows a more detailed breakdown of what happened to the original remote misses. The topmost section ("*no prefetch miss*") is the fraction that was not prefetched. The remaining three cases constitute the coverage factor. The "*prefetched hit*" case is the ideal case, where the prefetches fully hide the latency—notice that this is the largest case for all applications.

The "*invalidated prefetch miss*" category is for cases where the data has been prefetched but is invalidated before it is used. This only happens in non-binding prefetching because non-binding prefetches can be issued before synchronization. There are two situations

that can lead to this type of miss. First, it happens when a prefetched page is modified *after* it has been prefetched. Second, it also happens when there is insufficient information to issue prefetches for all of the required modifications due to the lazy propagation of write notices in LRC. The "invalidated prefetch miss" case is generally small and is most noticeable in OCEAN, WATER-NSQ, and WATER-SP. We observe that for these cases, the invalidated prefetch misses are primarily caused by the lazy propagation of write notices in LRC. Since write notices are propagated at synchronization points only, prefetches issued before a barrier may not have enough write notices to determine the locations of all of the required modifications. In this case, prefetch requests can only be issued to retrieve the most recent copies of all of the known modifications. Any missing write notice will be propagated to the prefetching processor at the subsequent synchronization point to invalidate the prefetched page. However, the fact that these invalidations allow prefetches to be non-binding results in performance improvement that outweighs the cost of these occasionally unsuccessful prefetches.

Finally, the "*late prefetch miss*" category is for cases where prefetches do not return in time to satisfy the references. With the exceptions of OCEAN and WATER-NSQ, this case accounts for most of the unsuccessful prefetches. There are two reasons for this problem: either the prefetches were not issued early enough to hide the latency, or else the prefetch request messages were dropped in the network. Both cases are affected by the network traffic, which we now consider in greater detail.

Network Traffic

Let us first consider the total number of network messages and the total number of bytes sent through the network, as shown in the first two pairs of columns in Table 3.3. Except for some applications where the number of network messages decreases because of a reduced number of garbage collections (RADIX) and better load balance (WATER-NSQ),

Benchmarks	Number of Messages		Bytes sent (KBytes)		Number of Misses		Avg. Miss Stall (μsec)	
	O	P	O	P	O	P	O	P
FFT	31520	32189	63112	64678	13013	379	3600	57300
LU-NCONT	156067	157260	145024	123701	33002	4071	2400	6200
LU-CONT	18777	19645	35800	36509	6008	550	3900	11700
OCEAN	167918	178260	210688	218658	71464	25399	2000	2900
RADIX	262400	186724	141477	146402	16437	3359	3800	10000
SOR	11258	11573	14683	15383	4917	166	3200	50698
WATER-NSQ	91997	91772	23671	24140	9483	1230	1600	2100
WATER-SP	47948	48044	64640	65932	21724	1387	1700	2000

Table 3.3: Communication characteristics of programmer-inserted prefetching (**O** = original, and **P** = programmer-inserted prefetching)

we see a general increase in the number of network messages. There is also a general increase in the number of network messages (except for LU-NCONT where the reason for the reduction is that prefetching reduces the amount of remote diffs required for garbage collection and thus reduces the number of network bytes sent through the network). However, since the increase is typically quite small, it does not suggest a problem.

The last two pairs of columns in Table 3.3 compare the effects of prefetching on the number of remote misses and the average miss latency. In all cases, the number of remote misses has been reduced, which indicates that prefetching is indeed successful. However, the average remote miss latencies in cases like FFT, LU-CONT, RADIX, and SOR have increased *enormously*. For example, although prefetching has reduced the number of remote misses in FFT by a factor of 34, it has *increased* the average miss latency by a factor of 15. For these applications, we observe that the problem is the result of bursty network traffic, which causes extreme queuing delays and also dropped network messages. In particular, we see hot-spotting effects during program initialization (when all processors are trying to communicate with the master processor to get the latest copy of the memory image) which are particularly acute.

3.2.4 Summary

Our experiments demonstrate that software-controlled non-binding prefetching reduces remote memory stall time significantly (from 45% to 92%) and can yield substantial performance improvements (from 4% to 29%). There are two main issues that affect the efficacy of our prefetches. The first is the success of the prefetch insertion strategy in selecting the appropriate references to prefetch and scheduling them early enough. The second concerns the effects of prefetching on network traffic. Regarding the first issue, we find that in most cases, we are able to cover more than 90% of the remote misses. However, we observe that the lazy propagation of write notices in LRC can limit the effectiveness of prefetching in a few cases. In Chapter 4, we will discuss an alternative prefetching scheme, which aims at overcoming this limitation. As for the effects of prefetching on network traffic, we see hot-spotting effects during program initialization. These hot-spotting effects cause extreme queuing delay and also dropped network messages which increases memory latency significantly.

3.3 Compiler-Inserted Prefetching

Since inserting prefetches into the application source code manually can be difficult and time-consuming, we now discuss the performance of compiler-inserted prefetching. The compiler we used implements Mowry's prefetching algorithm [28, 32] within the SUIF compiler [36]. The compiler also includes the I/O prefetching algorithm proposed by Mowry, Demke, and Krieger [30].

3.3.1 Overall Performance

Figure 3.3 compares the performance of compiler-inserted prefetching and programmer-inserted prefetching for three of our applications. (We will discuss the other five cases

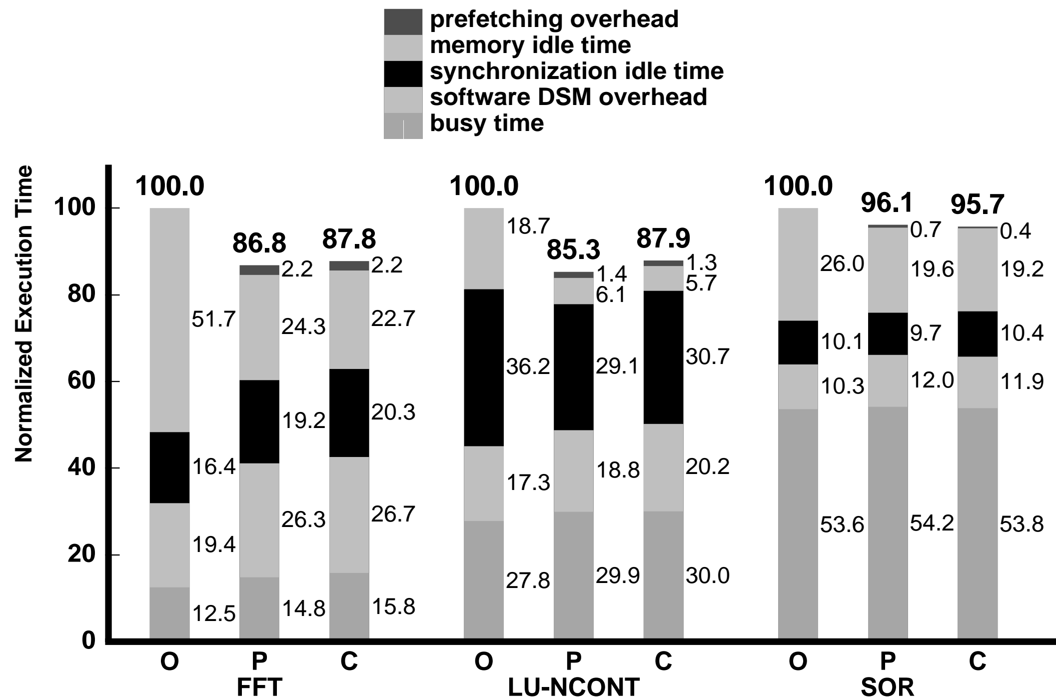

Figure 3.3: Performance improvement from compiler-inserted prefetching (**O** = original, **P** = programmer-inserted prefetching, **C** = compiler-inserted prefetching)

in Section 3.3.2). For each application, the two bars on the left are taken from Figure 3.1[4]. The right most bar labeled **C** represents the parallel execution time with compiler-inserted prefetching normalized to the original parallel execution time. As we see in Figure 3.3, the speedups achieved with compiler-inserted prefetching in these three applications are comparable to that with programmer-inserted prefetching.

For these three applications, the biggest difference between programmer-inserted prefetching and compiler-inserted prefetching is the number of unnecessary prefetches, as shown in Table 3.4. Compared with hand inserted prefetching, we observe that the compiler issues a larger number of unnecessary prefetches. The reason for this is that our compiler does not distinguish between global and private variables, so it issues prefetches

[4]SOR is an exception because we have modified its data allocation scheme to make it suitable for compiler-inserted prefetching.

Benchmarks	Unnecesary Prefetches (%)	
	Hand-inserted Prefetching	Compiler-inserted Prefetching
FFT	3.33	98.38
LU-NCONT	28.74	47.93
SOR	5.71	14.45

Table 3.4: The percentage of unnecessary prefetches with programmer-inserted prefetching and with compiler-inserted prefetching

for both of them. Prefetches for private variables are clearly unnecessary and are dropped immediately. Since these unnecessary prefetches are dropped immediately, we do not observe a significant increase in busy time in Figure 3.3.

3.3.2 Cases where the Compiler Failed

Ideally, the prefetching compiler should be able to insert prefetches into our original software DSM applications automatically. However, we find that the compiler did fail in a few cases. In this subsection, we discuss several issues concerning compiler-inserted prefetching that arose during our experiments.

Language Specific Problems

First of all, since the compiler we used was initially written to optimize Fortran codes, it does not handle some C constructs very well. In particular, the compiler cannot identify a multi-dimensional array within a loop if the array is allocated dynamically. Also, if the loop bounds of a "for" loop are more complicated expressions (as in the case of WATER-NSQ), the compiler treats the loop as a "do" loop and does not insert prefetches. In some cases, we are able to overcome these limitations by slightly modifying the application source codes (e.g. we can fix the array dimensions statically or replace the expressions within loop bounds with variables whenever possible). Therefore, it is a matter of engineering the compiler to better handle C constructs—a non-trivial task but

one that is feasible.

Algorithm Limitations

Two of our applications use algorithms that require more sophisticated compiler analysis to insert prefetches. In RADIX, the major remote miss reference is a reference to an array that is not indexed by an induction variable. In this case, static data analysis cannot be applied to issue prefetches. On the other hand, WATER-SP uses *linked-lists* to store its data, and our prefetching compiler currently does not handle linked-list. There are on-going researches regarding how to make use of profiling information to issue prefetches [33] and how to prefetch recursive data structures [27].

Software DSM Specific Issues

We also find that scheduling prefetches in software DSM applications is a challenging task. In software DSMs, acquiring a *remote* lock and arriving at a barrier are long latency operations. Since the compiler does not take the long latency into consideration when scheduling prefetches, it may schedule prefetches too early. This happens in WATER-NSQ because its critical sections are very small and the processors compete to enter the critical sections in each loop iteration. In this particular case, the compiler schedules prefetches so early that when the prefetches are issued, some of the required write notices are not yet propagated to the processor. Therefore, when the missing write notices are propagated, they invalidate the prefetched data.

In addition, the long prefetch latency makes prefetching for small loops prohibitive. For example, in LU-CONT and OCEAN, the loop bounds of the affected loops are very small. Although the compiler is able to issue all of the prefetches in the prolog stage [28], there is not enough computation to hide the prefetch latency. The problem is made worse because our prefetches are unreliable. Upon a memory miss, a normal memory request

is issued onto the network. This doubles the number of messages normally required for a remote memory reference. To address these issues, the compiler and the prefetch insertion algorithm have to be sensitive to the parameters of a software DSM system.

3.4 Chapter Summary

In this chapter, we focus on evaluating the effectiveness of our implementation of software-controlled non-binding prefetching. We evaluate the performance of eight applications (seven from the SPLASH-2 suite and one from the TreadMarks's distribution) and observe the following:

1. Software-controlled non-binding prefetching can yield substantial performance improvements (from 4% to 29%) by reducing memory idle times (from 45% to 92%).

2. The lazy propagation of write notices in LRC can limit the effectiveness of prefetching in a number of cases. In particular, when prefetches are issued before synchronization, there may not be enough write notices to determine the locations of all of the required modifications. In Chapter 4, we will discuss how runtime information can be used to predict the locations of the missing modifications.

3. Despite the high coverage factors, one of the limitations that prevents prefetching from achieving better performance is network contention caused by bursty network traffic. In a few cases, we see hot-spotting effects during program initialization which are particularly acute.

4. With a prefetching compiler, we successfully insert prefetches into three of the applications. They achieve speedups comparable to the best that we have obtained from programmer-inserted prefetching. For the remaining five cases, we have identified several limitations of the compiler. The two issues specific to scheduling

prefetches in software DSMs are: (i) the compiler does not realize the long latency of a synchronization operation and therefore issues prefetches too early; and (ii) the compiler cannot find enough computation within a small loop to hide the high prefetch latency. To address these issues, the compiler and the prefetch insertion algorithm have to be sensitive to the parameters of a software DSM.

Chapter 4

Alternative Prefetching Approaches

From our experience with programmer-inserted prefetching in the previous chapter, we observe that there are two challenges faced by statically inserted prefetches: (i) the addresses must be predictable, which may not be true (e.g., RADIX); (ii) the lazy propagation of write notices in LRC can limit the effectiveness of prefetching. Since LRC propagates write notices at synchronization points only, a processor does not necessarily know the locations of the required modifications to validate a page until after it has performed an acquire. It would be interesting to see if additional runtime information can help overcome these two challenges to further improve prefetching performance. In this chapter, we discuss two different prefetching schemes that make use of more runtime information to issue prefetches, namely *history prefetching* and *hybrid prefetching.*

History prefetching relies entirely on dynamic information to determine what to prefetch. Bianchini *et al.* [5] have also evaluated history prefetching in their protocol controller for hiding communication latency in TreadMarks. Hybrid prefetching, on the other hand, makes use of runtime information to try to enhance the effectiveness of statically inserted prefetches. In Sections 4.1 and 4.2, we discuss history prefetching and hybrid prefetching respectively and compare them with our programmer-inserted prefetching scheme. Our experiments with history prefetching indicates that issuing

prefetches at synchronization points increases synchronization idle time significantly because of the bursty network traffic during synchronization. In addition, it is difficult to find a prefetching heuristic that is general enough to tolerate memory latency in all cases. With hybrid prefetching, we observe that although we are able to cope with the lazy propagation of write notices, application performance does not improve due to network contention.

4.1 History Prefetching

History prefetching works as follows. During program execution, dynamic information which characterizes memory access patterns is collected. History prefetching makes use of this information to predict the future access patterns and issues prefetches for the expected remote references. In contrast with software-controlled prefetching, which relies on either the programmer or the compiler to explicitly insert prefetches, history prefetching relies entirely on runtime information to issue prefetches. Therefore, if history prefetching works well, it will be an attractive alternative to software-controlled prefetching.

Bianchini *et al.* [5] proposed a history prefetching scheme in their protocol controller for hiding communication latency in TreadMarks. To predict future access patterns, they employ a prefetching heuristic which assumes that a computation processor will likely need pages it used to cache and reference, but were invalidated by another processor. In their scheme, prefetches are issued at the time of lock acquisition only.

We implemented the above history prefetching scheme and extended it to issue prefetches at both lock acquire and barrier arrival times (since four of our applications do not acquire remote locks). In addition, our prefetching heuristic is evaluated at every synchronization point before prefetches are issued.

4.1.1 History Prefetching Performance Analysis

Figure 4.1 shows the performance improvement from history prefetching. For each benchmark, the **HIS** column represents the parallel execution time of history prefetching normalized to the original execution time (**O**). The rest of the columns are taken from Figure 3.1. For the history prefetching column, the topmost "*prefetching overhead*" category represents the overheads of history prefetching, which include evaluating the prefetching heuristic to predict future access patterns and the overhead of issuing prefetches. The remaining categories are the same as we have shown in Figure 3.1.

As we see in Figure 4.1, all of our applications perform worse with history prefetching than with programmer-inserted prefetching. The reduction in memory idle time ranges from 11% to 43% (compared to a range of 45% to 92% in programmer-inserted prefetching). There are two reasons for this: (i) our prefetching heuristic fails to predict future data access patterns in some of our applications, and hence does not issue enough prefetches to cover their remote misses (as in the cases of FFT, LU-NCONT, LU-CONT, RADIX, and SOR); and (ii) since history prefetching does not know when the data is actually used, it issues prefetches as soon as the access patterns meet the prefetching heuristic criteria. Therefore, prefetches may be issued too early such that the prefetched data is invalidated before it is used. For example, in WATER-NSQ, although more than 96% of its remote misses are covered by prefetches, roughly 70% of the prefetched data is invalidated by other processors before it is used.

Besides the above memory latency problem, we observe that the increase in synchronization idle time dominates application performance in some cases. For example, although history prefetching tolerates a significant amount of remote memory latency in OCEAN, it increases OCEAN's synchronization idle time by roughly 50%, and thereby increases its execution time. In fact, LU-CONT, SOR, and WATER-SP also exhibit

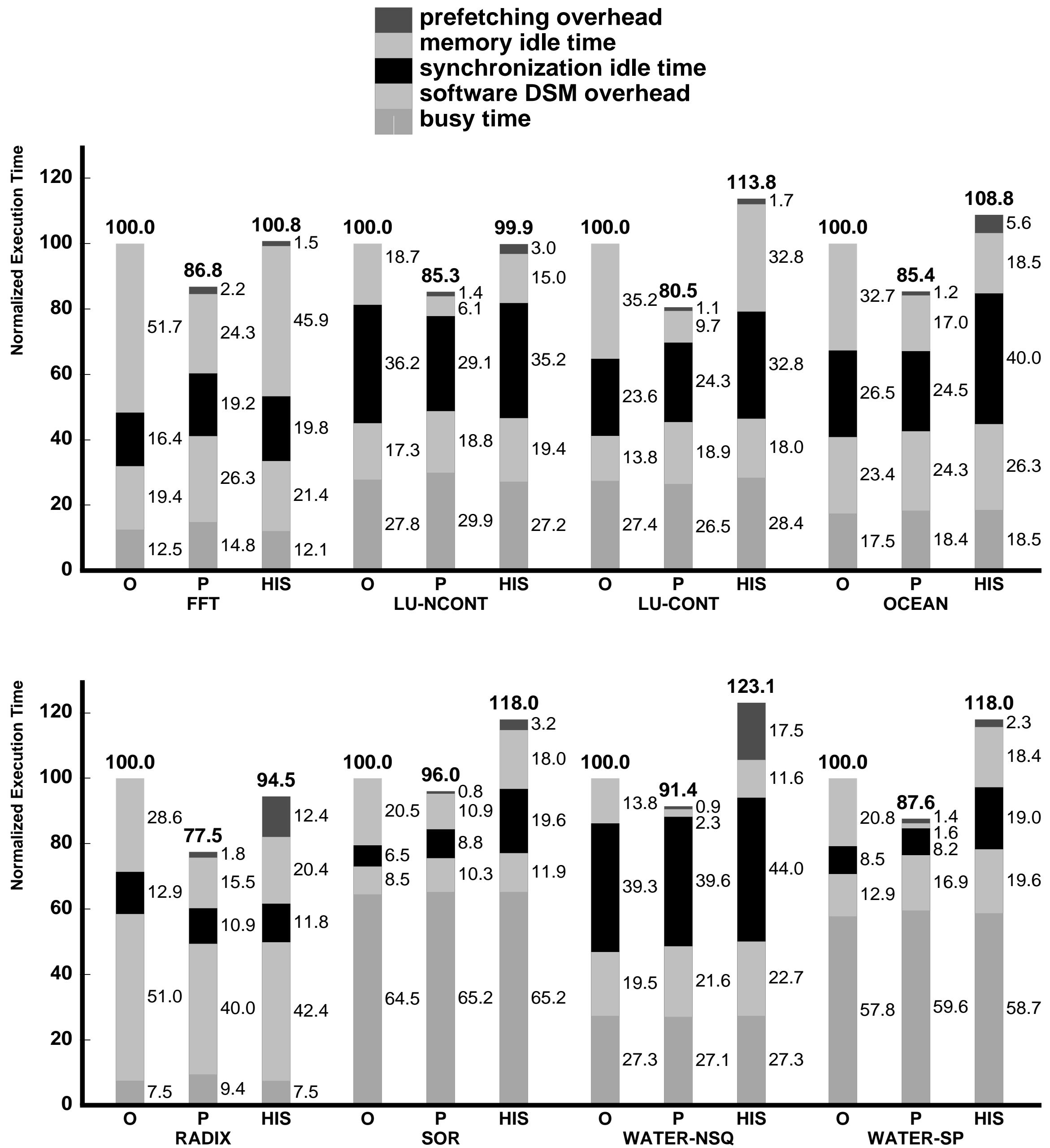

Figure 4.1: Performance improvement from history prefetching (**O** = original, **P** = programmer-inserted prefetching, and **HIS** = history prefetching)

similar behavior. It is because our history prefetching scheme issues prefetches only at synchronization points. The large number of prefetches issued at synchronization points induces bursty network traffic in some cases. The bursty network traffic during synchronization causes extreme queuing delays and increases synchronization idle time significantly. Bianchini *et al.* also have similar findings in their experiments with history prefetching [5].

Compared with programmer-inserted prefetching, history prefetching generally has a higher "*prefetching overhead*". In RADIX and WATER-NSQ, more than 13% of their execution times are prefetching overheads. This is due to the highly irregular access pattern (RADIX) and the large number of synchronization points (WATER-NSQ) (since our prefetch heuristic is evaluated at synchronization points) in these two applications.

4.1.2 Summary

In this section, we investigated history prefetching. Our history prefetching scheme collects runtime information to predict future access patterns and to schedule prefetches. We observe that the time when prefetches are issued onto the network plays an important role in application performance. With history prefetching, it is difficult to decide when prefetches should be issued such that the most remote misses can be covered without incurring too many overheads. Our experiments show that that issuing prefetches at synchronization points usually increases synchronization idle time significantly due to bursty network traffic during synchronization. We also find that our prefetching heuristic is not general enough to tolerate the memory latency in all cases and is computationally expensive if the access pattern is highly irregular.

Benchmarks	Number of invalidated prefetch misses		
	Lazy propagation		Issuing prefetches too early
	No request	One or more request(s)	
LU-NCONT	136	1	0
OCEAN	2664	1177	221
WATER-NSQ	260	66	0
WATER-SP	547	3	0

Table 4.1: The number of invalidated prefetch misses due to lazy propagation of write notices and issuing prefetches too early

4.2 Hybrid Prefetching

We discussed earlier in Section 2.4 that the lazy propagation of write notices in lazy release consistency can limit the effectiveness of non-binding prefetching. Our experiments with programmer-inserted prefetching in Section 3.2 also confirm this limitation. With LRC, a processor does not necessarily know which other processors have modified a given page until after the required write notices are propagated, which happens at synchronization points only. Therefore, if non-binding prefetches are issued before synchronization, there may not be enough write notices to determine the locations of the required modifications. In this case, any missing write notice will be propagated at the subsequent synchronization point to invalidate the prefetched data. Therefore, lazy propagation of write notices can increase the number of *invalidated prefetch misses* (i.e., cases where the data has been prefetched but is invalidated before being used).

To better understand this problem, Table 4.1 breaks down of the number of invalidated prefetch misses in four of our applications that are affected by this problem. The first pair of columns show the number of invalidated prefetch misses due to lazy propagation of write notices, which is the issue we are concerned with. The "*No request*" column indicates that no prefetch request is sent onto the network while the "One or more request(s)" column indicates that one or more prefetch requests are sent onto the network. As we see, these two columns constitute most of the invalidated prefetch misses.

Finally, the last column on the table shows the number of invalidated misses due to issuing prefetches too early. This problem only appears in OCEAN and can be fixed by scheduling prefetches properly. Therefore, to reduce the number of invalidated prefetch misses, we must cope with the lazy propagation of write notices.

In an extreme approach, we can broadcast prefetch requests to gather all of the required modifications. However, since software DSM has a high communication overhead, broadcasting prefetch messages may hurt application performance. Rather than broadcasting prefetches, we proposed a technique called *hybrid prefetching*. Similar to our programmer-inserted prefetching scheme, with hybrid prefetching, prefetches are explicitly inserted into the applications by programmers or by compilers. However, rather than merely assuming that the processor knows the locations of all the modifications, this scheme also makes use of runtime information to predict which other processors (besides the ones that are indicated in the write notices) are modifying the prefetching data and issues prefetches to these processors for the latest modifications.

Our prefetching heuristic for hybrid prefetching assumes that if a page is invalidated by write notices that are not present at prefetch time, the processors associated with these write notices are likely to modify the same page in the future. Therefore, at the time of subsequent prefetching, additional prefetches are issued to these processors to obtain the latest modifications. With this scheme, we would expect to see a reduction in the number of invalidated prefetch misses. In addition, since the collected runtime information is not cleared periodically, the worst scenario is that hybrid prefetching broadcasts prefetches to all processors. We would like to see if this happens in our experiments.

4.2.1 Hybrid Performance Analysis

Figure 4.2 shows the performance of the affected applications with hybrid prefetching. The prefetch insertion strategy we used is the same for both the programmer-inserted

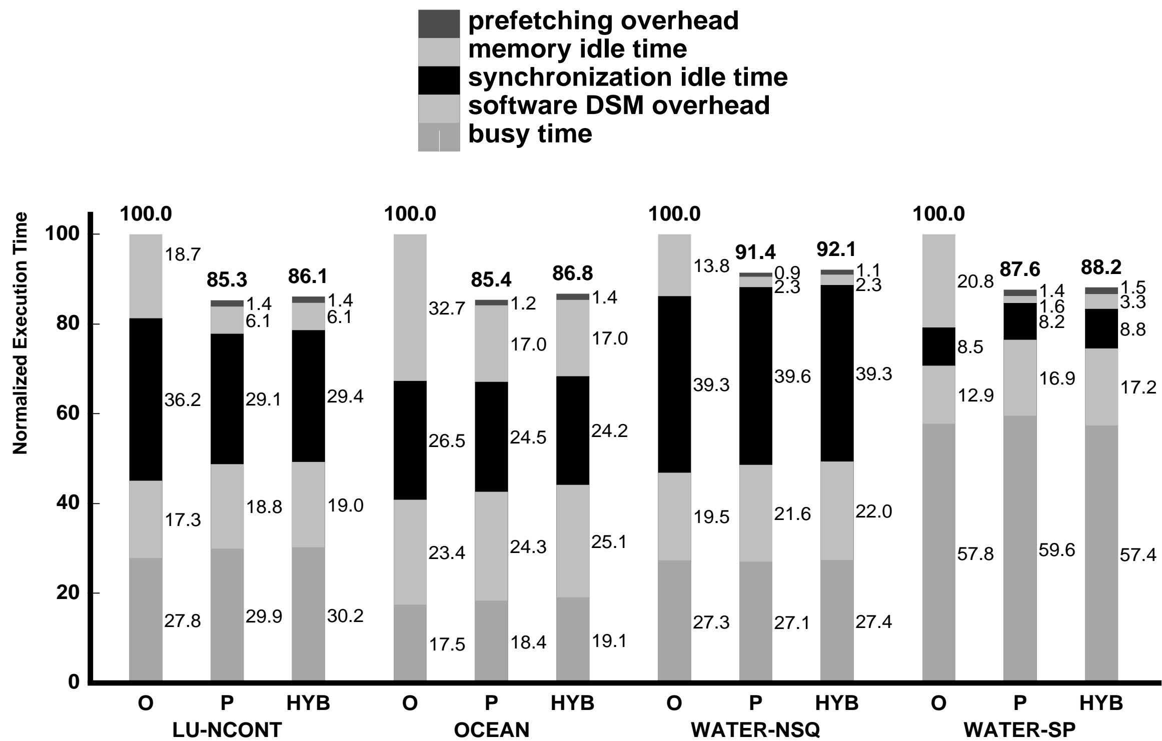

Figure 4.2: Performance improvement from hybrid prefetching (**O** = original, **P** = programmer-inserted prefetching, and **HYB** = hybrid prefetching)

Benchmarks	Number of invalidated prefetch misses					
	Programmer-inserted prefetching			Hybrid prefetching		
	Lazy propagation		Issuing prefetches too early	Lazy propagation		Issuing prefetches too early
	No request	One or more request(s)		No request	One or more request(s)	
LU-NCONT	136	1	0	3	1	0
OCEAN	2664	1177	221	27	363	200
WATER-NSQ	260	66	0	6	71	3
WATER-SP	547	3	0	501	1	0

Table 4.2: The impact of hybrid prefetching on the number of invalidated prefetch misses

prefetching cases and the hybrid prefetching cases. For each benchmark, the first two columns are taken from Figure 3.1 which represent the original and the programmer-inserted prefetching parallel execution times respectively. The **HYB** column represents the hybrid prefetching parallel execution time. All of them are normalized to the original parallel execution time. As we see in the figure, our applications do not benefit from hybrid prefetching.

To understand why hybrid prefetching is not working, we begin by focusing on how well hybrid prefetching can reduce the number of invalidated prefetch misses. In Table 4.2, the three columns on the left represent the number of invalidated prefetch misses with programmer-inserted prefetching while the three columns on the right represent that with hybrid prefetching. As we see, hybrid prefetching has indeed reduced a large number of invalidated prefetch misses in most cases. For example, in LU-NCONT and OCEAN, roughly 90% of this type of miss is eliminated. In WATER-NSQ, 75% of these misses are eliminated. The question is why does the performance of these applications not improve with the reduced number of invalidated prefetch misses?

We mentioned in the beginning of this section that the worst scenario of our hybrid prefetching scheme occurs when prefetches are broadcast to all of the processors. Broadcasting prefetching can degrade network performance significantly. Our experiments show that only OCEAN occasionally broadcasts prefetches. In fact, OCEAN is the only application that issues useless prefetches due to mispredicting the locations of the required

modifications. For other applications, the additional prefetch messages issued to cover the invalidated prefetch misses increase network traffic. Therefore, we observe that although hybrid prefetching reduces the number of invalidated prefetch misses, it increases the number of late prefetch misses (i.e., cases where prefetches do not return in time to satisfy the references), which is mostly due to high network traffic. Late prefetch misses are particular problematic in software DSM because they incur retry messages (since prefetches are unreliable), which make the network contention problem worse. Therefore, application performance does not improve.

4.2.2 Summary

Lazy propagation of write notices in LRC can limit the effectiveness of non-binding prefetching and increase the number of invalidated prefetch misses. To cope with this limitation, we propose hybrid prefetching. Hybrid prefetching makes use of runtime information to predict the locations of the required modifications and issues additional prefetches to get the modifications at prefetch time. Our experiments show that hybrid prefetching can eliminate most of the invalidated prefetch misses. In addition, only OCEAN occasionally broadcasts prefetches and issues useless prefetches. However, the additional prefetches issued by the hybrid prefetching scheme congest the network and slow down messages in the network, including prefetch messages. When a prefetch fails to return in time for the actual reference, a retry occurs and results in even more network contention. As a result, application performance does not improve.

Chapter 5

Prefetching and Multithreading

Although prefetching can reduce memory idle time significantly, it does not directly address synchronization latency, which is also significant in software DSMs (as we see in Figure 3.1). To tolerate the latency of both remote memory accesses and synchronization, we also consider multithreading. We begin in this chapter by evaluating multithreading on its own in Section 5.1. From our experiments, we observe that multithreading reduces both memory idle time and synchronization idle time in our applications. In some cases, however, we find that prefetching can tolerate memory latency just as good (if not better than) multithreading can. Therefore, it would be interesting to see whether the two techniques can be combined to achieve the best performance. In Section 5.2, we discuss the effects of combining these two techniques. Finally, in Section 5.2.3, we summarize this chapter with a discussion on the overall approach of applying latency tolerance techniques.

5.1 Multithreading

To help explain the comparison between prefetching and multithreading, we present some background information about multithreading from Lo's thesis [26]. The idea behind

multithreading is to switch from one parallel thread to another thread within the same program upon a long latency operation. In this way, the processor is kept busy until the remote operation completes. Therefore, the latency of the operation in one thread is hidden by the computation of another. The advantages of multithreading over prefetching are that it can handle arbitrarily complicated data access patterns and tolerate both memory latency and synchronization latency. The performance of multithreading depends on several factors: (i) whether there is enough parallelism in the application such that a ready-to-run thread is always available upon remote operation; (ii) whether the overhead of context-switching is high relative to the latency of a remote operation; and (iii) whether the locality effects of sharing the same local portion of the memory hierarchy are positive or negative.

5.1.1 Multithreading Implementation and Application Modifications

The implementation of multithreading on TreadMarks is taken from Lo [26]. In his implementation, a thread switch occurs whenever the current thread performs a synchronization operation or a remote memory operation. Once the thread is swapped out, it is marked as "ready to run" and will potentially be restarted when another thread switch occurs. To avoid unnecessary communication, outstanding requests are combined whenever possible.

Most applications do not require any modification to run correctly with multithreading, other than replicating "private" data on the heap whenever appropriate such that each thread get its own copy. However, in cases like FFT and WATER-NSQ where a significant amount of redundant computation is performed to initialize the data structure, a performance optimization is applied to keep a single shared copy of the data structure

per processor, thus avoiding wasted computation and synchronization. (Further details can be found in Lo's thesis.)

5.1.2 Multithreading Performance

Figure 5.1 shows the performance improvement gained by multithreading. For each application, the leftmost two bars are taken from our experiments with programmer-inserted prefetching in Figure 3.1. The remaining three bars are the performance results of multithreading using 2, 4, and 8 threads. The "*multithreading overhead*" component of the normalized execution time represents the overhead for switching between threads—all other components remain the same as in Figure 3.1. In addition, the "*software DSM overhead*" category includes any time spent servicing asynchronous message arrivals and combining remote requests (in the multithreading cases).

As we see in Figure 5.1, multithreading improves the performance of six of the eight applications by 6% or more, with two applications speeding up by over 50%. The number of threads that offers the best performance for an application varies greatly. For example, RADIX and WATER-SP are best with two threads, three cases are best with four, and three cases are best with eight.

Similar to prefetching, multithreading reduces memory idle time in most cases. For all applications, the reduction in memory idle time ranges from 32% to 81%. In three of the applications, multithreading can reduce more memory idle time than prefetching can. The reasons for this are: (i) multithreading can result in better task assignment which improve spatial locality (LU-NCONT and OCEAN); and (ii) multithreading can tolerate memory latency without requiring the ability to predict addresses far in advance (RADIX), and hence eliminate the problem of late prefetches. Late prefetches in software DSM are particularly costly because they require retry messages upon the actual memory references. These retry messages increase network contention and hence increase

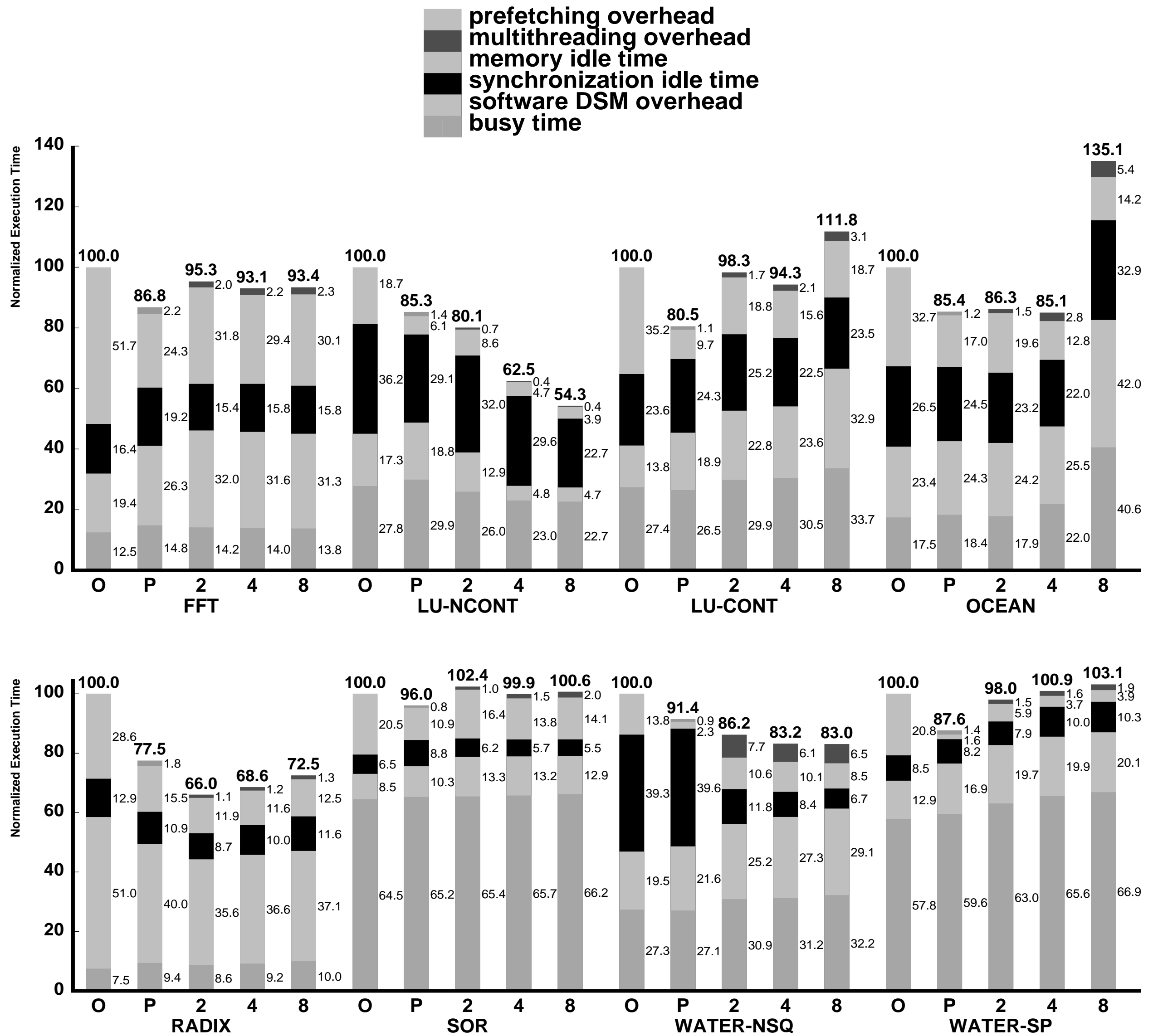

Figure 5.1: Performance comparison between prefetching and multithreading using 2, 4, and 8 threads (**O** = original, and **P** = programmer-inserted prefetching)

memory latency. One of the causes of these late prefetches is that the addresses of the remote memory references are not known far enough in advance to allow prefetches to be scheduled earlier (e.g., RADIX). Multithreading does not have this problem because it does not require the ability to predict addresses in advance.

While prefetching can only reduce synchronization idle time by improving load balance, multithreading reduces synchronization idle time by improving load balance as well as tolerating lock idle time. This is evident in WATER-NSQ which requests remote locks extensively. In WATER-NSQ, roughly 82% of the synchronization idle time is eliminated.

Regarding the runtime overheads of these two techniques, multithreading involves higher overhead. This is mainly due to handling more asynchronous messages (besides remote memory requests, synchronization messages are also handled asynchronously).

5.1.3 Summary

The idea behind multithreading is to switch from one parallel thread to another upon a long latency operation so that the latency in one thread can be hidden by the computation of another. The advantages of multithreading over prefetching are that it can handle arbitrarily complicated data access patterns and tolerate both memory latency and synchronization latency.

As we compare the performance of prefetching and multithreading, we find that in cases where addresses are fairly predictable, prefetching is generally as good (if not better than) multithreading at tolerating memory latency. On the other hand, if addresses cannot be predicted far enough in advance to schedule prefetches, multithreading appears to be a better choice because it does not have the late prefetch problem, which increases network contention and hence increases memory latency. In addition, if a significant portion of an application's synchronization idle time is due to acquiring remote locks, multithreading can reduce its synchronization idle time significantly.

Since each technique has different advantages and disadvantages, a natural question is whether the combination of these two techniques offers even more performance enhancement. In the next section, we will discuss the combined effects of multithreading and prefetching.

5.2 Combining Multithreading and Prefetching

We begin our investigation by first combining the two techniques such that synchronization latency is tolerated by multithreading and memory latency is tolerated by both multithreading and prefetching at the same time. In our experiments with this approach, we found that despite all of our efforts to improve its performance, it never achieved better performance than either prefetching or multithreading alone. The problem with this approach is that switching between threads tends to results in bursty miss patterns, which slow down messages in the network, including prefetch messages. When a prefetch fails to return in time because of network congestion, a retry occurs (since prefetches are unreliable) which is followed by a thread switch. The newly swapped in thread may issue more remote requests and thus exacerbate the network congestion problem. Consequently, we pay the overheads of both techniques (includes handling more asynchronous messages) without improving our ability in tolerating both latency.

In the remainder of this section, we therefore focus on a different approach to combining the two techniques. We use multithreading to tolerate only synchronization latency and we use prefetching to tolerate memory latency. This reduces the number of thread switches significantly. The rationale of this approach is that it tries to avoid the bursty miss patterns found in the previous approach by reducing the chance of frequent thread switch within a short period of time.

<u>SINGLE-THREADED CASE</u>

for ($i = 0$ to N) {
    prefetch (&$a[i]$);
}

<u>MULTITHREADED CASE</u>

/ first_time is a global variable*
*within a processor initialized to be true */*
if (*first_time*) {
    first_time = false;
    for ($i = 0$ to N) {
        prefetch (&$a[i]$);
    }
}

Figure 5.2: Eliminating unnecessary prefetches in multithreading by inserting a conditional test

5.2.1 Prefetch Insertion Strategy

We find that if we apply multithreading naively to our existing prefetched applications, the performance often worsens for the reasons discussed below.

Firstly, since there is often significant overlap in the working sets among different threads on the same processor, the first thread on a given processor which touches the remote data is essentially "prefetching" the data for the remaining threads. It is therefore unnecessary to issue prefetches in the remaining threads. Although these unnecessary prefetches do not have a large impact on application performance, they can nonetheless be easily removed. To do so, we first identify cases where threads on the same processor would be redundantly prefetching the same data. We then protect these prefetches by inserting a conditional test of a dynamic flag, which is explicitly reset by the first thread to arrive at the data. Figure 5.2 shows an example of such a modification.

Secondly, as we have mentioned in the beginning of this section, frequent thread switches can result in bursty miss patterns, which increase the chance of network contention. For RADIX, which has a high rate of communication, we reduce its network traffic by removing a number of prefetches (in this case by eliminating every-other dynamic prefetch). Although this results in a lower coverage factor, it was more than

compensated by the reduction in network queuing delay in this particular case.

5.2.2 Overall Performance

Figure 5.3 shows the performance improvement from combining prefetching and multithreading. For each application, the four columns on the left labeled **NOPF** represent the parallel execution times of the application without prefetching. The four columns on the right represent the parallel execution times with prefetching. The parallel execution times are normalized to the original one (i.e. the single-threaded case without prefetching).

As we can see in Figure 5.3, the combined effects of prefetching and multithreading are mixed. Three of the eight applications (FFT, OCEAN, and WATER-NSQ) achieved better performance when both techniques were applied, with speedups ranging from 4% to 26% over either technique alone. In two cases (LU-NCONT and RADIX), the performance was best with multithreading alone, and in three cases (LU-CONT, SOR, and WATER-SP), the performance was best with prefetching alone.

Why does the combined approach fail to outperform either technique alone in these five cases? In LU-NCONT, the combined approach actually outperforms both techniques with fewer than eight threads per processor. In the eight-threaded case, its performance is also comparable with the best case from multithreading.

In RADIX, the primary problem is that the loop structure makes it difficult to schedule prefetches early enough to hide the large network latency. As we see in Figure 3.2 in Chapter 3, RADIX has the highest portion of late prefetch misses. In this case, multithreading alone is clearly the best choice, since it tolerates memory latency without requiring the ability to know the addresses far in advance (thus avoiding the late prefetch problem).

In LU-CONT, SOR, and WATER-SP, when we compare their prefetching alone

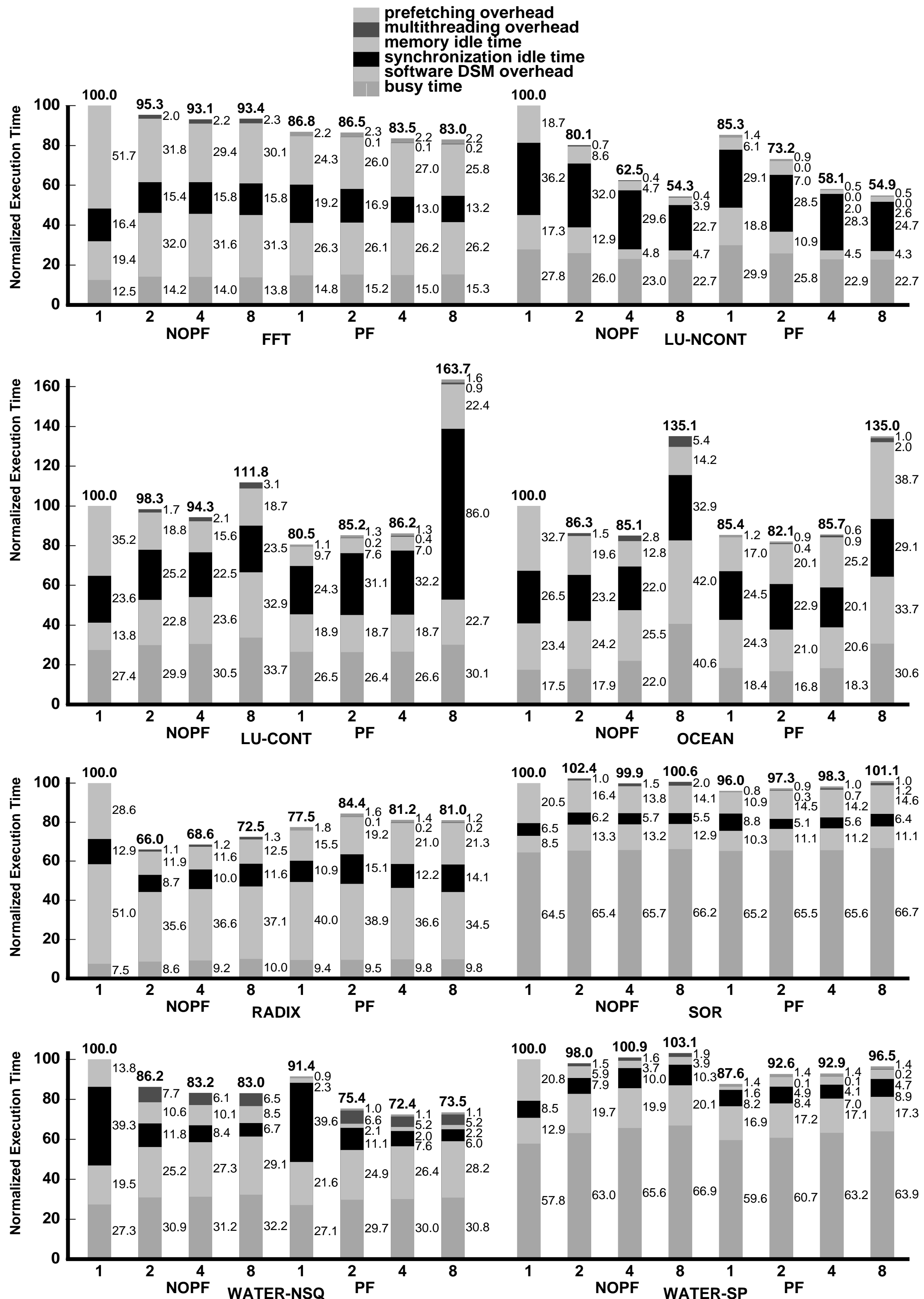

Figure 5.3: Performance improvement from combining prefetching and multithreading using 2, 4, and 8 threads (**NOPF** = without prefetching, and **PF** = with prefetching)

columns and their multithreading alone columns, we find that prefetching is more effective than multithreading in tolerating memory latency. Therefore, we would at least want to use prefetching. However, it may seem surprising that we are better off doing nothing for synchronization latency than attempting to tolerate it through multithreading. The reason for this is twofold. Firstly, the bulk of synchronization latency in these cases is due to barriers (100% in LU-CONT and SOR, 82% in WATER-SP), and multithreading reduces barrier stall times only indirectly through improving load balance. Secondly, there is a significant fixed cost in supporting multithreading, since all synchronization messages must be handled asynchronously. As we see in Figure 5.3, this increased overhead (which mostly appears as "*software DSM overhead*") more than offsets any gain in reduced synchronization idle time.

WATER-NSQ shows the greatest performance improvement from the combined approach. In this case, prefetching eliminates over 80% of the memory idle time (as opposed to less than 50% with multithreading), and multithreading eliminates roughly 80% of the synchronization idle time (which is mostly due to locks).

5.2.3 Chapter Summary

What is the best overall approach for tolerating long latency in software DSM? The choice between using a particular technique or a combination of both depends on the nature of the application. If the addresses cannot be predicted early enough to schedule prefetches (e.g., RADIX), it appears that using multithreading alone is the best solution. Once we have paid the price of supporting multithreading to hide memory latency, there is no reason not to apply it to synchronization latency as well.

If the memory addresses are predictable enough that prefetches can be scheduled far enough in advance, prefetching is generally as good (if not better) than multithreading at tolerating memory latency. Therefore, we would at least want to use prefetching.

However, we find that using both techniques to tolerate memory latency does not improve performance due to the induced bursty miss patterns, which increases the chance of network contention. The best choice appears to be using either multithreading or prefetching to tolerate memory.

Once prefetching is being used to tolerate memory latency, multithreading will most likely complement prefetching (by tolerating synchronization latency) if a significant fraction of the application execution time is lock idle time. On the other hand, if synchronization stall times are small or are mostly constituted of barrier idle time, it is less clear that the additional overhead of supporting multithreading is worthwhile.

Chapter 6

Conclusions

In this thesis, we investigated software-controlled non-binding prefetching within a LRC-based software DSM system called TreadMarks and evaluated the effectiveness of our implementation by inserting prefetches into eight parallel applications.

The key results of this thesis are as follows:

1. We have shown that software-controlled non-binding prefetching can reduce memory idle time significantly (from 45% to 92%) and can yield substantial performance improvements (from 4% to 29%). One of the limitations that prevents prefetching from achieving better performance is network contention caused by bursty network traffic. In a few cases, we see hot-spotting effects during program initialization which are particularly actue.

2. In three of our applications, we find that automatically inserted prefeching perform as well as programmer-inserted prefetching. From the cases where the compiler fails to match hand-inserted prefetching performance, we observe that scheduling prefetches in software DSMs is a challenging task, mainly due to the high communication latency in software DSMs. The automatic prefetch insertion algorithm does not take into consideration the long synchronization latency and therefore is-

sues prefetches too early. In addition, it is difficult for the compiler to find enough computation within small loops to hide the high prefetch latency.

3. From our experiments with history prefetching, we find that the time when prefetches are issued plays an important role in application performance. In particular, issuing prefetches at synchronization points often degrades application performance by increasing synchronization idle times. In addition, it is difficult to devise a prefetching heuristic that is general enough to handle arbitrarily complicated data access patterns without incurring too much overhead.

   We also observe that hybrid prefetching can cope with the lazy propagation of write notices in LRC and thus eliminates a significant portion of invalidated prefetch misses. However, this scheme does not improve application performance because the additional prefetch messages congest the network and increase memory latency.

4. By combining prefetching and multithreading such that memory latency is tolerated by prefetching and synchronization latency is tolerated by multithreading, we find that three of the eight applications achieve better performance improvement than applying either technique alone. However, none of the applications show performance improvement when both techniques are used to tolerate memory latency at the same time. This is mostly due to the redundant overheads in supporting these two techniques simultaneously, and the bursty miss patterns caused by frequent thread switching. The best overall approach to tolerating latency depends on factors such as the predictability of memory access patterns and the fraction of execution time lost due to lock stalls.

Bibliography

[1] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. Technical Report CS1051, University of Wisconsin, Madison, September 1991.

[2] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[3] J. L. Baer and T. F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proc. of Supercomputing '91*, pages 176–186, 1991.

[4] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of The 1993 COMPCON Conference*, pages 528–537, February 1993.

[5] R. Bianchini, L. I. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C. L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[6] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[7] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[8] A. L. Cox, S. Dwarkadas, and P. Keleher. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 106–117, April 1994.

[9] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[10] S. Dwarkadas, P. Keleher, A. L. Cox, and Willy Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *Proceedings of the 20th International Symposium on Computer Architecture*, 1993.

[11] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared Memory Multiprocessors. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, April 1991.

[12] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[13] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W. D. Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 254–263, May 1991.

[14] R. H. Halstead, Jr. and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, June 1988.

[15] L. Iftode, J. P. Singh, and K. Li. Understanding Application Performance on Shared Virtual Memory Systems. In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.

[16] P. Keleher. The Relative Importance of Concurrent Writers and Weak Consistency Models. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, 1996.

[17] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter Usenix Conference*, January 1994.

[18] P. Keleher, A. L. Cox, and W. Zwaepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[19] J. S. Kowalik, editor. *Parallel MIMD Computation : The HEP Supercomputer and Its Applications*. MIT Press, 1985.

[20] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transaction on Computers*, C-28(9):690–691, September 1979.

[21] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, October 1994.

[22] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.

[23] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 92–103, May 1992.

[24] D. E. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. L. 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.

[25] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transaction on Computer Systems*, 7(4):321–359, November 1989.

[26] A. K. W. Lo. Tolerating latency in software distributed shared memory systems through multithreading. Master's thesis, University of Toronto, forthcoming.

[27] C. K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, October 1996.

[28] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching.* PhD thesis, Stanford University, March 1994.

[29] T. C. Mowry, C. Q. C. Chan, and A. K. W. Lo. Comparative evaluation of latency tolerance techniques for software distributed shared memory. In *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture*, Feburary 1998.

[30] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, 1996.

[31] T. C. Mowry and A. Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared memory Multiprocessors. *Journal of Parallel and Distributed Computing*, June 1991.

[32] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[33] T. C. Mowry and C. K. Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *Proc. of Micro-30*, December 1997.

[34] J. P. Singh and J. L. Hennessy. Finding and exploiting parallelism in an ocean simulation program: Experience, results and implications. *Journal of Parallel and Distributed Computing*, 15(1):27–48, 1992.

[35] K. Thitikamol and P. Keleher. Multi-threading and Remote Latency in Software DSMs. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, 1997.

[36] S. Tjiang, M. Wolf, M. Lam, K. Pieper, and J. Hennessy. *Languages and Compilers for Parallel Computing*, pages 137–151. Springer-Verlag, Berlin, Germany, 1992.

[37] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[38] R. N. Zucker and J. L. Baer. A Performance Study of Memory Consistency Models. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 2–12, May 1992.