NOTE TO USERS

The original manuscript received by UMI contains pages with indistinct, light, broken, and/or slanted print. Pages were microfilmed as received.

This reproduction is the best copy available

UMI
Tolerating Latency in Software Distributed Shared Memory Systems Through Multithreading

by

Adley Kam Wing Lo

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 Adley Kam Wing Lo 1998
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.
Abstract

Tolerating Latency in Software Distributed Shared Memory Systems Through Multithreading

Adley Kam Wing Lo
Master of Science
Graduate Department of Computer Science
University of Toronto
1998

Large communication latency is a key obstacle to achieving higher performance on software distributed shared memory (DSM) systems, which provide a shared memory abstraction on message passing hardware. Multithreading is one of the techniques that can tolerate the communication latency on software DSM. It tolerates the latency by overlapping communication and computation with multiple threads on each processor.

We investigate the performance of multithreading on software DSMs empirically. Our results demonstrate that multithreading can improve the speedups of the applications by 3% to 84%, and we can further improve the performance of some applications by modifying the source codes or making threads switch context on synchronization only. This thesis also compares the performance of multithreading with that of prefetching and investigates their combined effects. The best overall approach to hiding latency depends on the predictability of memory access patterns and the extent to which lock stalls dominate synchronization time.
Acknowledgements

I would like to thank my supervisor, Todd Mowry, who has been offering me great guidance to my research work. Also, I would like to give honour to my second reader, Ken Sevcik, for his time and knowledge. In addition, I would like to thank the POW research team, especially Eric Parsons, Paul Lu, and Ben Gamsa for configuring a network of workstations to allow the experiments to be performed.

I would like to acknowledge the Information Technology Research Centre of Ontario (ITRC) and thank the Centre for Advanced Studies, IBM Toronto Laboratory for providing the network of workstations on which the experiments were done. I would also like to thank the Natural Sciences and Engineering Research Council of Canada for the financial support.

Finally, I would like to thank my colleagues, especially Charles Chan and C. K. Luk for their technical advice. Besides, I would like to acknowledge the department graduate secretary, Kathy Yen, for her great help. Last but not least, I must thank my family, especially my parents, and my friends for their support and encouragement. Without them, I cannot finish this work.
## Contents

1 Introduction ............................................. 1  

1.1 Software DSM Performance .............................. 2  
1.2 Latency Tolerance Techniques .......................... 3  
  1.2.1 Relaxed Memory Consistency Models ................. 4  
  1.2.2 Prefetching ......................................... 5  
  1.2.3 Multithreading ...................................... 6  
  1.2.4 Overall Approach ................................... 6  
1.3 Research Goals ......................................... 7  
1.4 Related Work .......................................... 8  
1.5 Organization of the Thesis ............................. 9  

2 Multithreading Issues .................................... 10  

2.1 TreadMarks ........................................... 10  
  2.1.1 Lazy Release Consistency in TreadMarks ............ 11  
  2.1.2 False Sharing and Multiple-Writer Protocol ......... 12  
2.2 Pthreads ............................................... 13  
  2.2.1 Interactions between TreadMarks and Pthreads ...... 13  
2.3 The Design ........................................... 14  
  2.3.1 Using Pthreads Condition Variable for Scheduling .. 14  
  2.3.2 Thread Creation .................................... 16
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.3</td>
<td>Context Switches</td>
<td>18</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Synchronization and Memory Coherence</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>Reducing Network Traffic</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>Performance with Multithreading Support</td>
<td>25</td>
</tr>
<tr>
<td>3.1</td>
<td>Architecture and Applications</td>
<td>25</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Experimental Testbed</td>
<td>25</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Benchmark Applications</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Switching Context on Remote Memory and Synchronization Accesses</td>
<td>29</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Overall Performance</td>
<td>29</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Further Details</td>
<td>31</td>
</tr>
<tr>
<td>3.3</td>
<td>Context Switching on Synchronization Only</td>
<td>40</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Performance Results</td>
<td>43</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Network Traffic</td>
<td>45</td>
</tr>
<tr>
<td>3.4</td>
<td>Performance Enhancement with Source Code Modification</td>
<td>46</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Application Enhancements</td>
<td>48</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Performance Results</td>
<td>49</td>
</tr>
<tr>
<td>3.5</td>
<td>Impact of Multithreading on Scalability</td>
<td>54</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Results</td>
<td>55</td>
</tr>
<tr>
<td>3.6</td>
<td>Chapter Summary</td>
<td>58</td>
</tr>
<tr>
<td>4</td>
<td>Combining Prefetching and Multithreading</td>
<td>61</td>
</tr>
<tr>
<td>4.1</td>
<td>Non-Binding Software-Controlled Prefetching</td>
<td>62</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Results with Prefetching Alone</td>
<td>63</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Performance Comparison Between Multithreading and Prefetching</td>
<td>65</td>
</tr>
<tr>
<td>4.2</td>
<td>Combining Both Latency Tolerance Schemes</td>
<td>66</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Optimizing Prefetching for Multithreading</td>
<td>67</td>
</tr>
</tbody>
</table>
4.2.2 Performance Results ........................................... 68

4.3 Prefetching and Multithreading with Context Switch on Synchronization

Only ................................................................. 71

4.3.1 Performance Results ........................................... 72

4.3.2 Further Details ............................................... 72

4.4 Chapter Summary ................................................ 76

5 Conclusion ......................................................... 78

5.1 Future Directions ............................................... 79

Bibliography .......................................................... 81
List of Tables

3.1 Benchmarks and Problem Sizes ........................................... 27
3.2 Multithreading statistics (O = original, nT = n threads) .............. 33
3.3 Expected number of threads required to effectively tolerate latency ... 35
3.4 Network traffic statistics (O = original, nT = n threads) .............. 38
3.5 Network traffic characteristic with switch-on-sync-only (O = original, nT = n threads) ...................................................... 45
3.6 Multithreading statistics with source code modifications (O = original, nT = n threads) ...................................................... 52
3.7 Network traffic characteristic with source code modifications (O = original, nT = n threads) ...................................................... 53
## List of Figures

1.1 Execution time breakdown using TreadMarks on eight ATM-connected workstations ................................ 2

2.1 Wait condition and signal condition .................................. 15
2.2 Thread creation pseudocode ........................................ 17
2.3 Naive context switch algorithm .................................... 18
2.4 Revised context switch algorithm ................................... 19
2.5 Example of lock stealing ............................................. 23

3.1 Normalized execution time on eight processors \((O = \text{original}, nT = n \text{ threads})\) ................................................ 30
3.2 Normalized stalling time \((O = \text{original}, nT = n \text{ threads})\) ............... 32
3.3 Normalized number of messages \((O = \text{original}, nT = n \text{ threads})\) ...... 36
3.4 Normalized total number of bytes sent \((O = \text{original}, nT = n \text{ threads})\) ... 37
3.5 Examples of lock waiting times on remote processor ............... 42
3.6 Normalized execution time \((O = \text{Original}, nT = n \text{ threads with context switch on all latency operations}, nTS = n \text{ threads with switch-on-sync-only})\) 43
3.7 Normalized stalling time \((O = \text{Original}, nT = n \text{ threads with context switch on all latency operations}, nTS = n \text{ threads with switch-on-sync-only})\) ........................................... 44
3.8 Normalized number of messages ($O = \text{Original}$, $nT = n$ threads with context switch on all latency operations, $nTS = n$ threads with \textit{switch-on-sync-only}) ........................................ 45

3.9 The FFT algorithm ................................................................. 47

3.10 The WATER-N$^2$ Algorithm ..................................................... 48

3.11 Examples of using local barriers ............................................. 50

3.12 Normalized execution time ($O = \text{original}$, $nT = n$ threads without source code modification, $nTM = n$ threads with source code modification) .... 51

3.13 Normalized stalling time ($O = \text{original}$, $nT = n$ threads without source code modification, $nTM = n$ threads with source code modification) .... 52

3.14 Normalized number of messages with source code modifications ($O = \text{original}$, $nT = n$ threads without source code modification, $nTM = n$ threads with source code modification) ........................................ 53

3.15 The speedup graphs (Continued on next page) .......................... 56

3.15 The speedup graphs (Continued from previous page) .................. 57

4.1 Normalized execution time ($O = \text{original}$, $nT = n$ threads, $P = \text{prefetch}$) 63

4.2 Breakdown of the original remote misses .................................... 64

4.3 Eliminate unnecessary prefetches in multithreading by inserting a conditional test ......................................................... 67

4.4 Execution time of combined techniques ($O = \text{original}$, $P = \text{prefetch}$, $nT = n$ threads without prefetching, $nTP = n$ threads with both techniques to hide memory stalls) ........................................ 69

4.5 Normalized number of messages ($O = \text{original}$, $P = \text{prefetch}$, $nT = n$ threads without prefetching, $nTP = n$ threads with both techniques to hide memory stalls) ........................................ 70

ix
4.6 Normalized execution times ($O =$ original, $P =$ prefetch, $nT =$ $n$ threads without prefetching, $nTP =$ $n$ threads with prefetching to hide memory stalls) ................................................................. 73

4.7 Normalized number of messages ($O =$ original, $P =$ prefetch, $nT =$ $n$ threads without prefetching, $nTP =$ $n$ threads with prefetching to hide memory stalls) ................................................................. 74
Chapter 1

Introduction

There has been considerable interest in exploiting collections of workstations or PCs connected by commodity networks as less expensive alternatives to tightly-coupled multiprocessors. Since memory on each machine is not shared across the cluster, a processor can only directly reference the memory residing with the same processor. Programmers must send and receive messages explicitly for a processor to access the memories on other machines. However, managing message passing is quite difficult in some applications with irregular data access patterns (e.g., visualization applications [35]). With the help of the virtual memory system [28], software Distributed Shared Memory (DSM) systems provide a shared memory abstraction across machines and simplify the task of writing parallel programs on message passing machines. For certain classes of applications, software DSM systems can deliver performance that is comparable to hardware cache-coherent machines of a similar scale [8, 16]. However, for applications with large communication demands, the performance can be disappointing [8, 10].

A key bottleneck to achieving higher performance on software DSMs is the relatively large communication latency. In contrast to tightly-coupled multiprocessors, where remote miss latencies are on the order of half a microsecond [26], the remote miss latencies for software DSM on moderately aggressive hardware are closer to half a millisecond [9]—
i.e., roughly three orders of magnitude slower. This large communication latency affects not only remote memory accesses, but also synchronization operations [10]. Since communication latency is already known to be a significant bottleneck even in tightly-coupled multiprocessors, one might expect the much higher latencies in software DSMs to make the problem even worse.

1.1 Software DSM Performance

To illustrate the impact of communication latency on software DSM performance, we ran a collection of applications taken primarily from the SPLASH-2 suite [42]¹ using TreadMarks [19] (a software DSM implementation) on eight 133 MHz IBM RS/6000 workstations connected by a 155 Mbps FORE Systems ATM LAN. Further details of the hardware platform and the applications are given later in Chapter 3. Figure 1.1

¹The exceptions are SOR and TSP, which are taken from the TreadMarks distribution.
shows a detailed breakdown of the resulting execution times, as measured on the real hardware using high resolution timers under AIX 4.1. The normalized execution times are broken down into the following four categories, from top to bottom: time spent stalled waiting for (i) synchronization and (ii) remote memory misses, respectively; (iii) time spent executing DSM operations (e.g., the memory coherence protocol); and (iv) time spent doing useful computation.\(^2\)

As we see in Figure 1.1, seven of the ten applications spend over 40% of their time stalled waiting for either remote memory miss or synchronization. Clearly, we must cope with the large communication latencies to achieve higher performance on software DSMs.

### 1.2 Latency Tolerance Techniques

We can minimize the effect of the communication latency on software DSMs by (i) reducing the number of long latency operations, (ii) reducing communication overhead, and (iii) tolerating the stall time of long latency operations. Caching is an effective technique to reduce shared memory reference misses in multiprocessors [13]. Software DSMs exploit caching by using local memory as the cache for remote locations. Several techniques [5, 20] have been proposed to reduce consistency-related communications on software DSMs, and there have been several studies on techniques for reducing the communication overheads (e.g., efficient support on networks [36, 37, 40] and special hardware support [2, 15, 34]). However, it is impossible to completely eliminate all long-latency operations and their message overhead. Therefore latency tolerance techniques are essential to achieving higher performance on software DSMs.

\(^2\)The "Busy" time also includes interrupt times associated with the software DSM, since we cannot isolate them otherwise.
1.2.1 Relaxed Memory Consistency Models

One way to tolerate latency of remote memory reference is to pipeline and buffer the write accesses to shared memory. The degrees of pipelining and buffering depend on the underlying memory model. The Sequential Consistency (SC) model [23] is the strictest memory model, and it requires an outstanding memory reference to be completed before another memory reference is performed. Although SC provides a natural and intuitive concept of accessing shared memory, it severely limits the amount of pipelining and buffering.

The Release Consistency (RC) model [12] is a relaxed memory model. RC classifies every synchronization access as either an acquire or a release. It requires that the modifications by processor $p$ to shared memory be visible to another processor $q$ when a subsequent release by $p$ is visible to $q$. In this way, shared memory writes can be pipelined and their latencies can be tolerated. The DASH [27] implementation of RC combats memory latency by pipelining writes to shared memory. A processor stalls when reading remote data or executing a release (at which time it must wait for all previous shared memory accesses to perform). In software DSMs, it is also important to reduce the number of messages exchanged because sending a message in software DSM is more expensive than in hardware DSM. In the memory model of eager release consistency (ERC) [17, 20], each processor buffers shared memory modifications and exchanges the information about the modifications only at the point of a release. Therefore, writes to the same shared location can be buffered, and the ERC can use fewer messages than the RC implementation in DASH to maintain memory consistency. In addition, lazy release consistency (LRC) [17, 20] delays the information exchanges further until the point of an acquire. Each processor gets an up-to-date shared memory image only when it is neces-

\footnote{For an invalidation-based protocol, invalidations are sent the point of a release; for an update protocol, an up-to-date memory image is sent instead.}
This can further reduce the number of messages. Although RC allows pipelining and buffering to tolerate the latency of writing remote memory, it does not solve the problem of tolerating the latency of reading remote memory. We can see in Figure 1.1 that the remaining latency is still painful even with LRC (which is used by TreadMarks). Hence we look further for techniques to tolerate this remaining latency.

1.2.2 Prefetching

To hide the latency of reading remote memory, we must separate the request for data from the use of that data, while finding enough useful parallelism to keep the processor busy in between. Prefetching [3, 30, 31] is a technique for tolerating both read and write latencies. It attempts to hide the latency by loading data into local memory before accessing the data. Multithreading is another technique for tolerating both kinds of latencies, and we will discuss it later in Section 1.2.3. The distinction between prefetching and multithreading is that prefetching finds parallelism within a single thread of execution, while multithreading exploits parallelism across multiple threads.

In a multiprocessor environment, prefetches can be classified as either binding or non-binding. With binding prefetching, the value seen by a subsequent read access is bound at the time when the prefetch operation completes. While binding prefetching is often easy to implement, the data may become stale if another processor modifies the same location during the interval between the prefetch and the read access. This places significant restrictions on where prefetches can safely be inserted. In contrast, non-binding prefetching [30, 31] allows the prefetched data to remain visible to the memory-coherence mechanism, and modifications by the remote processors are still visible to the local processor when the data are being accessed. Hence non-binding prefetching gives the programmer or the compiler the flexibility to insert prefetches more aggressively without violating program correctness. With a sophisticated analysis of the data access pat-
tern, non-binding software-controlled prefetching can hide the latency of shared memory access [30, 31].

1.2.3 Multithreading

Multithreading [11, 21, 22, 24, 25, 41] is a technique for tolerating read and write latency, as well as synchronization latency. The idea behind multithreading is to switch from one parallel thread to another upon a long latency operation, thereby keeping the processor busy with useful work until the remote operation completes. In this manner, one can hide the communication latency of one thread by the computation of other threads.

The performance of multithreading depends on several factors. First, enough parallelism must be available in the application, so that there is always a ready-to-run thread to tolerate latency. Second, the context switch overhead must be small enough that a significant amount of time is not wasted during switching contexts. Finally, the locality effects of multiple threads sharing the same portion of the memory hierarchy can be negative.

Multithreading has several advantages. It can tolerate the latencies of memory misses with complex or unpredictable data access patterns, since it does not rely on prediction. Multithreading can also tolerate synchronization latency, while other latency tolerance techniques fail in this regard. Finally, if there is significant locality within the clusters of threads on each processor, we can see a prefetching benefit relative to running the threads on distinct processors. One thread can access the shared data (or locks) which have already been brought into the local node by another thread.

1.2.4 Overall Approach

Before considering latency tolerance techniques, we first address the latency problem by reducing the number of long latency events as much as possible. This is important since it reduces the demand for network bandwidth, which is crucial on software DSM
systems. Besides exploiting caching to reduce the number of remote accesses, software DSMs can buffer and pipeline remote accesses through a relaxed memory consistency model. TreadMarks uses the lazy release consistency protocol to exploit buffering and pipelining, and it also uses a multiple writer protocol [5] to minimize the effects of false sharing. While these techniques improve performance, the remaining latency is still substantial (as we saw in Figure 1.1), and hence we would like to cope with it as well.

To tolerate the remaining latency, we focus primarily on multithreading in this study. An open question is whether the performance gain offered by multithreading justifies its resulting run-time overheads. Addressing this open question is one of the goals of this thesis, as we discuss further in next section.

1.3 Research Goals

This section discusses the research goals of this thesis. We attempt to evaluate and improve upon the latency-hiding benefits of multithreading for software DSMs on real hardware. First, we implement our multithreading support within TreadMarks (the software DSM platform used throughout our experiments) and evaluate the resulting performance improvement on a collection of applications. Our second goal is maximizing the performance improvement for the applications we study. We study an alternative context switching strategy: context switching on synchronization only. Since lock contention can be severe in software DSMs, we investigate whether this context switching strategy helps improve the performance of those applications with lock contention. We also investigate the modifications to the application source codes which help to eliminate the redundant computation. In addition, we also evaluate the effects of combining non-binding software-controlled prefetching and multithreading. We compare the performance results of using each technique individually and study whether combining both techniques achieves better performance than each individual technique alone.
1.4 Related Work

Many researchers have considered multithreading as a latency tolerance technique on tightly-coupled multiprocessor machines [13, 22, 24, 25, 41] and have shown that it is an effective way to tolerate latency. In the context of software DSM systems, Thitikamol and Keleher [38] have studied multithreading on CVM [18] (another software DSM system) concurrently with our work. They have shown that applications can perform well with a small number of threads, and they argue that source modification is sometimes important for multithreaded applications to perform well on software DSMs. They argue that the speedup of multithreading is limited by (i) the contentions of synchronization variables and shared data on local nodes, (ii) the locality effects of multiple threads sharing the same portion of the memory hierarchy, (iii) the pressure on both the cache and TLBs, and (iv) the cost of context switching. However, their work does not clearly show the effect of multithreading on the overhead of the underlying software DSM. Furthermore, their work does not examine the network traffic in the initialization step, which accounts for a significant amount of time and network bandwidth consumption in some applications.

Freen et al. [11] studied Distributed Filaments (DF), which is an implementation of fine-grain parallelism on top of off-the-shelf hardware and software. Each thread on a processor executes a pool of filaments, which ideally reference the same set of pages. Upon each memory miss, the thread (with the entire pool of filaments) is suspended and there is a switch to another thread. This minimizes the number of memory misses on the same page and achieves the greatest overlap of computation and communication. However, the assignment of filaments to a processor relies on sophistication, and an application suffers poor performance if the assignment causes load imbalance. Finally, Freen et al. do not fully discuss the DSM overhead and network traffic that affect the performance of the applications.
Chapter 1. Introduction

In contrast to the two multithreading studies on software DSMs discussed above, our work also focuses on improving the speedups of the applications as well as evaluating the effectiveness of multithreading for tolerating latency. In addition, we consider combining prefetching and multithreading to tolerate latency. Combining both techniques to tolerate latency on hardware cache-coherent machines has been considered by Gupta et al. [13], and we evaluate whether the combination of the two latency tolerance schemes has the same effect on software DSMs.

1.5 Organization of the Thesis

This thesis is organized as follows. Chapter 2 describes multithreading implementation issues that arise with software DSMs. We discuss several issues concerning correctness and the techniques used for reducing interprocessor communication in our implementation. Chapter 3 describes the performance of applications with multithreading and evaluates the effectiveness of multithreading on the software DSM. To further improve the application performance, we also present some enhancements to either the context switching strategy or the source codes of the applications. Chapter 4 discusses the effect of combining prefetching and multithreading. We first compare the performance results using prefetching or multithreading individually. Then we study whether combining prefetching with multithreading can further improve the performance of the applications. Chapter 5 concludes this thesis and summarizes the important results. It also discusses directions for future investigation.
Chapter 2

Multithreading Issues

In this chapter, we discuss issues related to adding multithreading support to software DSMs. Section 2.1 describes the implementation of LRC and the multiple-writer protocol within TreadMarks [19], which is used throughout our experiments. Since we use the Pthreads [14] library to implement multithreading on TreadMarks, we describe some related issues about Pthreads in Section 2.2. We emphasize data consistency and signal-safety for adding multithreading support to TreadMarks with Pthreads API. Section 2.3 presents our design of the context switching algorithm using the Pthreads' synchronization primitives and discusses how we prevent deadlock. We also discuss how messages are handled after adding multithreading support and how to maintain correctness of shared memory and synchronization operations. Section 2.4 describes how we combine remote requests to minimize network traffic.

2.1 TreadMarks

TreadMarks [19] is a state-of-the-art software DSM implementation, and we use it as the basis of our experiments. TreadMarks does not require any kernel modifications, but instead uses the operating system's user-level memory management facilities. The processors synchronize by sending messages on the general purpose network, and TreadMarks manipulates the message traffic at run time.
2.1.1 Lazy Release Consistency in TreadMarks

TreadMarks uses the invalidation-based lazy release consistency protocol [20] to maintain memory coherence. The protocol divides the execution of each process into intervals. A new interval begins when a process executes an acquire or a release. Intervals between different processes are ordered according to the happen-before-1 partial order [1], and a vector timestamp is assigned to each interval. For a given processor $p$, its entry at position $p$ in its copy of the vector timestamp is set to $i$ where $i$ is its current interval number. The entry for a processor $q \neq p$ denotes the most recent interval of the processor $q$ that precedes the current interval of the processor $p$ according to the partial order. When a processor performs an acquire, it sends its vector timestamp to the previous releasing processor. The releasing processor piggybacks the write-notices to the acquiring processor according to the vector timestamp. A write-notice contains an indication that a page has been modified in a particular interval, and the acquiring processor can use the write-notices to invalidate its shared pages accordingly.

In TreadMarks, there are two types of synchronization objects: locks and barriers. Acquiring and releasing a lock is modeled as acquire and release operations, respectively; arrival at a barrier is modeled as a combination of an acquire and a release. Both locks and barriers are assigned static managers. A lock request message is sent to the manager, and the manager is responsible for forwarding the request to the last releaser of the lock. The releasing processor compares the received vector timestamp with its own vector timestamp, and it then piggybacks the up-to-date vector timestamp along with write-notices in its reply to the acquiring processor. Similarly, the processor sends a message to the barrier manager when it arrives at a barrier. To indicate that all processes have arrived, the barrier manager collects the messages and piggybacks the updated vector timestamp with write-notices in its messages.
According to LRC, a new interval is created when the processor (i) acquires or releases a lock or (ii) arrives at a barrier. In TreadMarks, however, an interval is created only at a barrier or a lock that is released to another processor. Hence the overhead of creating an interval is avoided if a lock is re-acquired locally. When a processor suffers a protection fault on a shared page, it first checks to see whether it has a copy of the page. If not, then the processor requests a copy of the page from a member of the page’s approximate copyset. If there are outstanding write-notices for the page, then the processor obtains the diffs \(^1\) from remote processors according to the write-notices. When the processor receives all necessary diffs, it applies them to the page in increasing vector timestamp order.

### 2.1.2 False Sharing and Multiple-Writer Protocol

False sharing can potentially hurt the performance of software DSMs [5]. False sharing occurs when the processors reference disjoint portions of a page, and at least one of the references is a write access. Under the common single-writer protocols, false sharing leads to unnecessary communication. A write to a variable of a page causes the entire page to become invalid on all other processors that cache the page. Any other processor will incur a memory miss even if it accesses another variable on the same page. This causes the modifications to be brought across the network and generates unnecessary network traffic.

TreadMarks uses a multiple-writer protocol to cope with the problem of false sharing. When two or more processors modify different portions of a shared page, the modification of each processor is not visible to the other processors immediately. Initially, all of the shared pages are write-protected. When a processor attempts to write to a page, TreadMarks creates a *twin* (a copy of the page). The processor can then continue writ-

\(^1\) A diff is a runlength encoded record of the modifications to the page.
ing the page without any software DSM intervention. For Munin [4] (another software implementation of DSM that also uses multiple-writer protocol), the page and the twin are compared to form a diff when there is a release. The diff (rather than the whole page) are sent across the network to update the pages on remote memories. TreadMarks also uses this multiple-writer protocol, except it creates diffs lazily. The diff creation is further postponed until the remote processors need the modifications of the page. In this manner, both the software overhead and number of diffs can be minimized.

2.2 Pthreads

We implement multithreading on top of TreadMarks using the Pthreads [14] user-level thread library. Pthreads is defined in POSIX (Portable Operating System Interface), the family of IEEE operating system interface standards. It is a standardized model for dividing a program into subtasks whose execution can be interleaved or run in parallel. In the Pthreads model, every thread within a process shares a common virtual address space; each thread has a private program counter, register set, and stack. Therefore context switching between threads involves less state to save, and it is less expensive than switching between processes. In addition, there is less overhead managing the local portion of shared memory. For example, if one thread brings a page into the local portion of shared memory, another thread can also use it directly without incurring any software trap to the kernel.

2.2.1 Interactions between TreadMarks and Pthreads

In the UNIX process model, a process is forked by copying the whole address space of its parent process. With this semantic, a process can have private global variables and only the process itself writes its own global data. In the Pthreads environment, however, all the threads within the same process share a common global address space after they are
created. Since an application may be written without the awareness of multiple threads whose global variables are shared, this leads to data inconsistency if the programmer assumes each thread has a private copy of global variables. To avoid this problem, we replicate the "private" data on the heap whenever appropriate for the subroutines in both TreadMarks and the applications.

In TreadMarks, we require signaling for memory management and message passing, so we must make TreadMarks signal-safe when we add multithreading support. In general, any thread that does not block an asynchronous signal can be the recipient of the signal within the same process. Hence we cannot guarantee the signal-safety of a multithreaded application by simply blocking the asynchronous signal inside the signal handler, as is the case with a single-threaded application. To ensure signal-safety, we must block the specified asynchronous signal from all threads within the same process. Since a thread cannot change the signal mask of other threads, we may use an explicit thread to wait for the asynchronous signal, thus leaving all other working threads with the signal blocked. However, this approach adds more context switches and overhead to TreadMarks. To guarantee signal-safety, we require that only one thread can be active at a given time and all other threads must be swapped out with the asynchronous signals blocked. With this restriction, TreadMarks is signal-safe since the only thread that can receive asynchronous signals is the active thread. Section 2.3 presents our method of creating and switching between threads such that we can ensure this restriction.

2.3 The Design

2.3.1 Using Pthreads Condition Variable for Scheduling

Since we use the condition variables in Pthreads to implement both thread creation and context switching routines, we describe the semantic of using condition variable in Pthreads. In Pthreads, each thread is given a time quantum to use the processor
(a) wait on a condition

```c
/* thread_i waits on cond_i */
pthread_mutex_lock(&mutex_i);
status_i = 0;
while (!status_i)
    pthread_cond_wait(&cond_i, &mutex_i);
pthread_mutex_unlock(&mutex_i);
```

(b) signal a condition

```c
/* thread_j wakes up thread_i */
pthread_mutex_lock(&mutex_i);
status_i = 1;
pthread_cond_signal(&cond_i);
pthread_mutex_unlock(&mutex_i);
```

Figure 2.1: Wait condition and signal condition

according to the underlying scheduling algorithm.² Although each thread can yield the processor explicitly, the thread cannot choose the next active thread. Since we cannot control the order of the thread executions explicitly, we must rely instead on condition variables to help us order the execution during a context switch. In our design, each thread has a condition variable, and it sleeps when it waits on this condition variable. Therefore to wake up another thread, a thread signals the condition variable of the sleeping thread.

In the Pthreads model, each condition variable has an associated mutex lock. A thread must hold this mutex lock before it waits on the condition variable [14], so that the condition is protected against a data race. Figure 2.1(a) shows how a thread (thread_i in the figure) waits on a condition variable (cond_i in the figure). After locking the

---

²We are using the Pthreads library on AIX 4.1, which schedules threads instead of processes.
mutex and resetting the condition \((mutex_i \text{ and } status_i, \text{ respectively, in Figure 2.1(a)})\), \(thread_i\) calls \texttt{pthread\_cond\_wait} to block itself. When \(thread_i\) calls \texttt{pthread\_cond\_wait}, it atomically releases \(mutex_i\) before blocking on \(cond_i\). Once \(cond_i\) is signaled, \(thread_i\) atomically acquires \(mutex_i\) before returning from \texttt{pthread\_cond\_wait}. Therefore it must finally release \(mutex_i\).

Figure 2.1(b) shows how another thread \((thread_j \text{ in the figure})\) signals \(thread_i\) waiting on \(cond_i\). After locking \(mutex_i\) and setting \(cond_i\), \(thread_j\) calls \texttt{pthread\_cond\_signal} to wake up \(thread_i\). After that, \(thread_j\) must release \(mutex_i\), since \(thread_i\) atomically acquires \(mutex_i\) before waking up.

### 2.3.2 Thread Creation

As mentioned in Section 2.2.1, we must ensure that only one thread is active at a time. In Pthreads, however, threads are created and then become ready to run immediately. Therefore we must block all but one thread before continuing execution. Figure 2.2 illustrates pseudocode which ensures that only one thread is active and that all other threads either (i) wait on their own condition variables, or (ii) are swapped out by the underlying scheduling algorithm after locking the associated mutexes. In the next section, we will show that one of these conditions must be true to prevent deadlock during context switching.

In Figure 2.2, each thread locks the mutex \texttt{global} and increases \texttt{threads\_arrived} after creation. Except for the first (arbitrarily chosen) active thread, all other threads lock their own mutex before increasing \texttt{threads\_arrived}. The first active thread keeps spinning inside the loop until \texttt{threads\_arrived} is equal to the total number of threads. Therefore either condition (i) or (ii) mentioned above is true when all threads have increased \texttt{threads\_arrived} and the first active thread continues execution.
startup() {
    /* Block all asynchronous signals */
    /* The mutex 'global' is used for protecting threads.arrived. */

    if (this thread is the first active thread) {
        pthread_mutex_lock(global);
        threads_arrived++;
        pthread_mutex_unlock(global);

        while (threads_arrived < total number of threads)
            yield the processor;
    } else {
        /* thread, (which is not the first active thread) executes the code here. */

        pthread_mutex_lock(mutex);

        pthread_mutex_lock(global);
        threads_arrived++;
        pthread_mutex_unlock(global);

        status = SLEEP;

        while (status == SLEEP)
            pthread_cond_wait(cond, mutex);

        pthread_mutex_unlock(mutex);
    }
    /* Unblock all asynchronous signals */
}

do_work() {
    startup();
    perform_the_task();
}

main() {
    /* Initialization */
    for (i = 1; i < number of threads; i++) {
        create(do_work, i);
    }
    do_work();
    /* Finalization */
}
/* thread\textsubscript{i} wakes up thread\textsubscript{j} */
Block all asynchronous signals;

pthread_mutex_lock(mutex\textsubscript{j});
status\textsubscript{j} = AWAKE;
pthread_cond_signal(cond\textsubscript{j});
pthread_mutex_unlock(mutex\textsubscript{j});

/* Deadlock occurs if the thread is swapped out here */

/* thread\textsubscript{i} blocks itself */
pthread_mutex_lock(mutex\textsubscript{i});
status\textsubscript{i} = SLEEP;

while (status\textsubscript{i} is SLEEP)
    pthread_cond_wait(cond\textsubscript{i},mutex\textsubscript{i});

pthread_mutex_unlock(mutex\textsubscript{i});

Unblock all asynchronous signals;

---

Figure 2.3: Naive context switch algorithm

2.3.3 Context Switches

In our implementation, a context switch occurs whenever the thread encounters a long latency event—i.e., a remote memory miss or a remote synchronization operation. We do not restart a suspended thread immediately once the event it was waiting for completes—instead, we mark the thread as "ready-to-run", and potentially restart it whenever the current thread is swapped out. We do this because the context switching routine calls Pthreads API routines, and it is unsafe to do so inside any asynchronous signal handler. Thus the thread is only swapped in again at a point where the currently executing thread is ready to give up control (i.e., during the start of a remote miss or synchronization). Although this strategy increases the total waiting time for synchronization operations, it works well in practice.
Figure 2.3 shows a naive implementation of the context switch algorithm, in which thread$_i$ switches to thread$_j$. First, thread$_i$ signals cond$_j$ to wake up thread$_j$. Then thread$_i$ waits on its own condition variable to block itself. Although this implementation is simple, it may result in deadlock. Consider the following example, where thread$_i$ switches to thread$_j$, and thread$_j$ switches back to thread$_i$ later. After signaling cond$_j$, thread$_i$ is swapped out by the underlying scheduling algorithm before locking mutex$_i$ and blocking itself on cond$_i$. Then thread$_j$ is swapped in. After some computation, thread$_j$ switches back to thread$_i$. It signals cond$_i$ to wake up thread$_i$. Nothing happens because thread$_i$ has not performed a wait on cond$_i$. Then thread$_j$ blocks itself by waiting on cond$_j$, and thread$_i$ is swapped in again but waits on cond$_i$ immediately. Both threads are now waiting on their condition variables indefinitely.

Figure 2.4 shows the revised context switch algorithm. As shown in the figure, thread$_i$
locks $\text{mutex}_i$ for protecting $\text{cond}_i$ before signaling $\text{cond}_j$ to wake up $\text{thread}_j$. After waking up $\text{thread}_j$, $\text{thread}_i$ then waits on $\text{cond}_i$. If, before the context switch, (i) only one thread is active and (ii) all other threads are either waiting on their condition variables or else holding the associated mutex locks, then facts (i) and (ii) hold after the context switch. Since facts (i) and (ii) are guaranteed by the pseudocode codes (shown in Figure 2.2) after thread creation, they will hold after every context switch.

A deadlock may occur when the following four conditions simultaneously hold on a system: "Mutual Exclusion", "Hold and Wait", "No Preemption", and "Circular Wait" [7]. For our context switching algorithm, we consider waiting on a condition and signaling a condition to correspond to waiting on a resource and releasing a resource, respectively, as described in [7]. In the Pthreads model, a thread ignores all the condition signaling that occurs before the thread waits on the condition variable. Once a thread waits on a condition variable, the thread will wake up only if another thread signals this condition variable. Hence we consider that $\text{thread}_i$ still holds the resource if $\text{thread}_i$ signals $\text{cond}_j$ before $\text{thread}_j$ begins waiting on $\text{cond}_j$.

The revised context switch algorithm is deadlock free. For the algorithm shown in Figure 2.4, deadlock occurs only if $\text{thread}_i$ signals $\text{cond}_j$ but $\text{thread}_j$ is not waiting on $\text{cond}_j$ (i.e., when $\text{thread}_i$ sleeps, it still holds the resource needed by $\text{thread}_j$). This is because "Hold and Wait" cannot occur when all threads (before waiting on their own condition variables) signal the condition variables with other threads already waiting on. Under the circumstance where $\text{thread}_i$ signals $\text{cond}_j$ and $\text{thread}_j$ is not waiting on $\text{cond}_j$, it must be the case that $\text{thread}_j$ is swapped out by the kernel with $\text{mutex}_j$ locked. Before signaling $\text{cond}_j$, $\text{thread}_i$ must lock $\text{mutex}_j$. Since $\text{thread}_j$ has already locked $\text{mutex}_j$, the kernel puts $\text{thread}_i$ on the waiting queue of $\text{mutex}_j$ and swaps in $\text{thread}_j$ to release $\text{mutex}_j$. After that, $\text{thread}_j$ waits on $\text{cond}_j$ and releases $\text{mutex}_j$ atomically in

\footnote{The kernel may swap in other threads before $\text{thread}_j$, but all other threads are either waiting on}
pthread_cond_signal. Then thread\textsubscript{i} can lock mutex\textsubscript{j} and signal cond\textsubscript{j} on which thread\textsubscript{j} is waiting. This implies that thread\textsubscript{j} has already awoken when thread\textsubscript{i} sleeps on cond\textsubscript{i}, i.e., thread\textsubscript{i} does not hold the resource needed by thread\textsubscript{j} when thread\textsubscript{i} sleeps. Hence the condition “Hold and Wait” cannot be true and the revised context switch algorithm is deadlock free.

### 2.3.4 Synchronization and Memory Coherence

Once we implement context switching on long latency events, lock reply messages can now arrive asynchronously. This may result in a data inconsistency problem in the local portion of shared memory if the current thread is in the middle of requesting updates from remote processors. If any lock reply messages arrive containing invalidations with write-notices, TreadMarks applies these write-notices and invalidates the shared pages. Then TreadMarks unprotects the invalid page when all updates have arrived. Therefore all the threads on the local processor consider this invalid page as a valid one, and this causes data inconsistency between local and remote processors.

To alleviate this problem, we delay the processing of a lock reply message when the current thread is in the middle of requesting updates. We allocate a buffer to store lock reply messages if locks arrive but the current thread is requesting updates on a page. After receiving updates and applying them to the local memory, the processor can perform invalidations according to the write-notices in the lock reply message and then deallocate the buffer. This approach preserves data consistency without increasing communication.

---

their condition variables (i.e., the kernel would not swap them in) or else swapped out by the kernel with the associated mutexes locked (i.e., the kernel would swap them in but they finally wait on their condition variables). Therefore the kernel will swap in thread\textsubscript{j} finally.
2.4 Reducing Network Traffic

Multithreading increases the number of asynchronous messages that TreadMarks must handle. Each thread switches to other threads after it sends a remote request; and a processor must handle the reply, as well as servicing other remote requests, asynchronously. Without multithreading, however, a processor can spin on a particular message queue for the reply after it sends a remote request, and an asynchronous message only arrives when the local processor must service the remote requests. Therefore the overhead of handling asynchronous messages greatly increases with multithreading.

To help reduce this overhead, we combine outstanding requests within a given single processor whenever possible. A thread does not send a separate message if another thread has an outstanding request for the same data; instead, it performs a context switch to the thread with the matching request already outstanding. In this way, no extra message is required when there is an outstanding request for the same memory page. However, combining memory requests incurs kernel overhead. Before context switching, the thread with a memory miss on a page must protect the page first. Any other thread then receives a protection fault when it attempts to access the page. Therefore this thread incurs the software overhead of an interrupt for the protected page before combining the data request with the outstanding one. This overhead also increases when more memory requests are combined for the same memory page. In contrast, combining lock requests does not incur this overhead, since it does not require a software interrupt. Therefore a thread acquires a lock and switches to another "ready-to-run" thread when there is an outstanding request for the same lock. Once the thread that acquires the lock finishes using the lock, it can release this lock to all local threads that have combined requests for this lock. By doing this, we can minimize the number of messages for locks.

In addition to combining requests for locks, a thread may also "steal" the lock re-
Figure 2.5: Example of lock stealing
quested by a swapped-out thread on the same processor to avoid a context switch. This can happen only if the current executing thread tries to get the lock that is not already used by this swapped-out thread and the lock reply has arrived. Figure 2.5 illustrates this scenario. A thread (thread 1 in the figure) gets a lock without performing context switch to the thread (thread 0 in the figure) that acquires this lock if the lock has arrived and thread 0 has not started using the lock. Finally, thread 1 releases the lock and switches to thread 0, and this lets thread 0 use the lock. Therefore we can reduce a context switch, as shown in the figure.

Finally, the barrier arrival requests by all threads within a processor can be merged into a single remote request. In our multithreading implementation, having each thread send an individual arrival request creates enormous amounts of network traffic and generates a hot-spot, especially when there are many threads on a processor. Therefore we make all the threads (except the last arrived thread) switch to other threads at barrier arrival, and only the last thread to arrive communicates with the barrier manager.
Chapter 3
Performance with Multithreading Support

In this chapter, we quantitatively evaluate the performance benefits of multithreading on software DSM. Section 3.1 describes the hardware platform and the applications used in the experiments. Section 3.2 presents the performance results of the applications with context switches upon both remote memory misses and synchronization. Section 3.3 evaluates the performance with context switching on synchronization and spin-waiting on remote memory misses. We compare the performance of switching on synchronization only to switching on both remote memory misses and synchronization. Section 3.4 investigates enhancements to the application source codes which result in improved speedups. Section 3.5 examines whether the multithreading speedups of the applications scale well with different numbers of processors. Section 3.6 summarizes this chapter.

3.1 Architecture and Applications

3.1.1 Experimental Testbed

We run our experiments with TreadMarks on a cluster of eight IBM RS/6000 workstations running AIX 4.1. Each machine contains a 133 MHz PowerPC 604 processor with split 16KB primary instruction and data caches, a 512 KB unified secondary cache, and 96MB of physical memory. The workstations are connected by a single FORE Systems
ASX-200WG ATM LAN switch using 155 Mbps OC3 multimode fiber optic links. The processes in TreadMarks use the UDP protocol for communication, and a lightweight protocol is provided by TreadMarks on the top of the UDP to guarantee the reliable transmission of messages. Under this setup, it takes an average of 2195 μseconds to obtain a 4096-byte page. The average time to obtain a diff is 2189 μseconds. It takes 1560 μseconds to request empty diffs and 2818 μseconds to request full-sized diffs. The minimum time to perform an eight-processor barrier operation is 1390 μseconds. To acquire a lock, it takes 874 μseconds if the manager is the last releaser and redirection of the lock request to the last releaser is not necessary, and 1377 μseconds on average otherwise.

The experiment results were measured using the high resolution timers under AIX 4.1. Since we want to show the best possible performance of multithreading, we took the best result in each experiment from 100 trials for both single-threaded and multithreaded applications.

3.1.2 Benchmark Applications

We use ten applications in our experiments: FFT, LU-NCONT, LU-CONT, OCEAN, WATER-N², WATER-SP, RADIX, RAYTRACE, SOR, and TSP. The last two applications are taken from the TreadMarks distribution [19], and the remaining applications are taken from the SPLASH-2 [42] benchmark suite. The problem sizes for each benchmark are shown in Table 3.1.

**FFT** performs a 1-D complex Fast Fourier Transform. The data set consists of two \( \sqrt{n} \times \sqrt{n} \) matrices: the \( n \) complex data points to be transformed and the \( n \) complex data points referred to as the root of unity. Since the first row of the root of unity matrix is heavily accessed and is read-only after initialization, it is replicated at the beginning of the program.
Table 3.1: Benchmarks and Problem Sizes

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Problem size</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>256K data points</td>
</tr>
<tr>
<td>LU-NCONT</td>
<td>1024 × 1024 matrix with block size 128</td>
</tr>
<tr>
<td>LU-CONT</td>
<td>1024 × 1024 matrix with block size 32</td>
</tr>
<tr>
<td>OCEAN</td>
<td>258 × 258 grid</td>
</tr>
<tr>
<td>WATER-N²</td>
<td>9 steps on 512 molecules</td>
</tr>
<tr>
<td>WATER-SP</td>
<td>9 steps on 4096 molecules</td>
</tr>
<tr>
<td>RADIX</td>
<td>262144 integers, radix 1024</td>
</tr>
<tr>
<td>RAYTRACE</td>
<td>Balls4</td>
</tr>
<tr>
<td>SOR</td>
<td>50 steps on 2000 × 2000 grid</td>
</tr>
<tr>
<td>TSP</td>
<td>19 cities</td>
</tr>
</tbody>
</table>

**LU-NCONT** performs blocked LU factorization of a dense matrix. It factors an \( n \times n \) matrix into the product of a lower triangular and upper triangular matrix. The factorization uses blocking to exploit temporal locality on individual submatrix elements, but the blocks are allocated *non-contiguously*.

**LU-CONT** is a different implementation of LU factorization. It makes use of an optimized data structure for the matrix such that the blocks are allocated *contiguously* in the shared memory. This greatly improves the data locality.

**OCEAN** studies large-scale ocean movements based on eddy and boundary currents. The main data structure is a four-dimensional array, which is partitioned into square-like subgrids, where each is allocated *contiguously* and modified by its owner. Communication is required when the processor reads the boundary elements from neighboring partitions.

**WATER-N²** 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 every 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-N²**, but with a more efficient algorithm. It uses a 3-D spatial data structure which is a grid of boxes. Each box contains a linked list of the molecules currently in that box (in the current time-step).
A processor that owns a box in the grid only needs to look at its neighboring boxes for molecules that might be within the cutoff radius. Hence the algorithm is $O(n)$ rather than $O(n^2)$.

**RADIX** implements an integer radix sort. In each iteration, each processor generates a local histogram based on one digit. Local histograms on all processors are then accumulated into a global histogram. Finally, each processor uses the global histogram to permute its keys into a new array for the next iteration.

**RAYTRACE** renders a 3-D scene onto a 2-D image plane using optimized ray tracing. A uniform hierarchical grid is used to represent the scene. A ray is traced through each pixel in the image plane. Multiple rays are generated through reflection or refraction, creating ray trees. The image plane is partitioned among processors in contiguous blocks of pixel groups, and distributed task queues are used for task stealing.

**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.

**TSP** solves the traveling salesman problem using a branch and bound algorithm. It maintains a priority queue with pointers to a pool of partially evaluated tours. Each processor removes a partially evaluated tour from the priority queue. If the removed path contains more than a threshold number of cities, then the processor will try all possible permutations of the remaining cities to complete the tour. It compares the shortest tour found by completing this path with the global shortest current tour, and updates the latter if necessary. If the removed path contains fewer than the threshold number of cities, then it is extended by one city in all possible ways, and those partial tours are re-inserted into the priority queue, in order of their partial costs.

Further details of the applications can be found in studies done by Woo et al. [42], Iftode et al. [16], and Lu et al. [29].
3.2 Switching Context on Remote Memory and Synchronization Accesses

In this section, we evaluate the performance of multithreading when context switching occurs during both remote memory and synchronization accesses. We begin with the overall performance and then focus in greater detail on the key aspects that affect multithreading performance: the amount of parallelism, overhead, and network traffic.

3.2.1 Overall Performance

Figure 3.1 presents the normalized execution time of each benchmark on eight processors with one, two, four, and eight threads per processor. Each bar is broken down into the following categories. The "Busy" time is the time spent during useful computation. It also includes the overhead of signaling associated with TreadMarks since we cannot isolate this overhead. The "DSM Overhead" represents the computation that TreadMarks performs on consistency information, including the time spent on garbage collection.\(^1\) The "Synchronization Idle" and "Memory Miss Idle" times represent the time that the processor stalls for the synchronization and remote memory miss, respectively. The "Multithreading Overhead" is the time spent on switching contexts and scheduling threads.

As we see in Figure 3.1, multithreading improves the performance of seven of the ten applications. The speedups of these six applications are 3% or more, with two applications speeding up by over 50%. As shown in Figure 3.1, the performance of the applications with multiple threads per processor improves mainly because of the reduction in the memory stall times (ranging from 20% to 80%). In some cases, multithreading also reduces the synchronization stalls significantly.

There are three applications which perform poorly with multithreading: FFT, SOR, and SOR.

\(^1\)During garbage collection, each processor validates its copy of every page that it has modified. To reclaim the memory, the processor then discards all other pages, interval records, write-notices, and diffs.
Figure 3.1: Normalized execution time on eight processors (O = original, nT = n threads)
and TSP. SOR does not yield significant improvement with more than one thread per processor because both DSM and multithreading overheads offset the performance improvement gained by using multithreading. In both TSP and FFT, the execution times of the multithreaded cases are larger than their single-threaded counterparts. As shown in Figure 3.1, TSP performs worse in the multithreading cases because of higher synchronization time. Since TSP uses a centralized work queue, lock contention is already high in the single-threaded case. Therefore adding more threads causes much higher lock contention. In contrast, FFT performs poorly because of higher busy time and DSM overhead. Since the first row of the root of the unity matrix is replicated for each thread (virtual processor), every thread performs the same initialization to its own replicated row. This causes the busy time of the multithreaded cases to be higher than that of their single-threaded counterpart.

### 3.2.2 Further Details

To explore the details of the multithreading performance, we break down the stall times of Figure 3.1 into several categories in Figure 3.2. For the memory miss idle time, we divide it into the stall times for cold misses ("Page Idle") and coherence misses ("Diff Idle"). For the synchronization idle time, we divide it into the stall times for acquiring locks ("Lock Idle") and at barrier arrivals ("Barrier Idle").

As we see in Figure 3.2, multithreading tolerates the stall times of coherence misses. The reduction in this category ranges from 23% to 90%. In contrast, we see that multithreading is less successful at tolerating the stall times of cold misses. For example, in FFT and SOR, the stall times of cold misses are still significant in the multithreaded cases when comparing with those in the single-threaded cases. A thread requests an empty page from the processor which is in the approximate copyset of the page, and TreadMarks initializes the approximate copyset of each page to contain the starting pro-
Figure 3.2: Normalized stalling time ($O = \text{original, } nT = n \text{ threads}$)
### Table 3.2: Multithreading statistics (O = original, nT = n threads)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Total Context Switches</th>
<th>Run Length (µsec)</th>
<th>Remote Misses</th>
<th>Average Latency (µsec)</th>
<th>Combined Requests</th>
<th>Total Remote Acquires</th>
<th>Average Latency (µsec)</th>
<th>Stolen Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>—</td>
<td>861</td>
<td>13013</td>
<td>3554</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2T</td>
<td>23410</td>
<td>614</td>
<td>13013</td>
<td>2790</td>
<td>10322</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4T</td>
<td>28914</td>
<td>570</td>
<td>13013</td>
<td>2692</td>
<td>13013</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8T</td>
<td>29170</td>
<td>574</td>
<td>13013</td>
<td>2674</td>
<td>13013</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LU-NCONT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>—</td>
<td>3492</td>
<td>33002</td>
<td>2350</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2T</td>
<td>23582</td>
<td>3465</td>
<td>19829</td>
<td>1790</td>
<td>11196</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4T</td>
<td>15102</td>
<td>5327</td>
<td>10623</td>
<td>1823</td>
<td>7295</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8T</td>
<td>14544</td>
<td>5492</td>
<td>10403</td>
<td>1334</td>
<td>6691</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LU-CONT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>—</td>
<td>3041</td>
<td>6008</td>
<td>3902</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2T</td>
<td>9488</td>
<td>2815</td>
<td>6004</td>
<td>2080</td>
<td>1064</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4T</td>
<td>12191</td>
<td>2602</td>
<td>5990</td>
<td>1738</td>
<td>1806</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8T</td>
<td>17552</td>
<td>2479</td>
<td>9030</td>
<td>1380</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OCEAN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>—</td>
<td>1029</td>
<td>71464</td>
<td>1970</td>
<td>0</td>
<td>1645</td>
<td>7506</td>
<td>0</td>
</tr>
<tr>
<td>2T</td>
<td>51166</td>
<td>1161</td>
<td>62281</td>
<td>1356</td>
<td>2390</td>
<td>1656</td>
<td>4003</td>
<td>2881</td>
</tr>
<tr>
<td>4T</td>
<td>104880</td>
<td>1263</td>
<td>64212</td>
<td>858</td>
<td>9088</td>
<td>1657</td>
<td>4155</td>
<td>5931</td>
</tr>
<tr>
<td>8T</td>
<td>202530</td>
<td>1355</td>
<td>112278</td>
<td>545</td>
<td>15122</td>
<td>1662</td>
<td>6287</td>
<td>650</td>
</tr>
<tr>
<td>RADIX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>—</td>
<td>987</td>
<td>16437</td>
<td>3788</td>
<td>0</td>
<td>93</td>
<td>1680</td>
<td>0</td>
</tr>
<tr>
<td>2T</td>
<td>21138</td>
<td>1056</td>
<td>17490</td>
<td>1483</td>
<td>171</td>
<td>105</td>
<td>1077</td>
<td>0</td>
</tr>
<tr>
<td>4T</td>
<td>22977</td>
<td>1102</td>
<td>17650</td>
<td>1430</td>
<td>356</td>
<td>126</td>
<td>369</td>
<td>8</td>
</tr>
<tr>
<td>8T</td>
<td>24548</td>
<td>1152</td>
<td>17970</td>
<td>1508</td>
<td>733</td>
<td>167</td>
<td>266</td>
<td>7</td>
</tr>
<tr>
<td>RAYTRACE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>—</td>
<td>12583</td>
<td>11492</td>
<td>1730</td>
<td>0</td>
<td>2504</td>
<td>8804</td>
<td>0</td>
</tr>
<tr>
<td>2T</td>
<td>9966</td>
<td>13732</td>
<td>8725</td>
<td>1422</td>
<td>1807</td>
<td>1150</td>
<td>28553</td>
<td>19</td>
</tr>
<tr>
<td>4T</td>
<td>12229</td>
<td>14521</td>
<td>7189</td>
<td>1315</td>
<td>2920</td>
<td>628</td>
<td>39129</td>
<td>48</td>
</tr>
<tr>
<td>8T</td>
<td>19987</td>
<td>13425</td>
<td>6807</td>
<td>1377</td>
<td>4184</td>
<td>431</td>
<td>46080</td>
<td>27</td>
</tr>
<tr>
<td>SOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>—</td>
<td>10015</td>
<td>4917</td>
<td>3184</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2T</td>
<td>5736</td>
<td>10147</td>
<td>4917</td>
<td>2550</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4T</td>
<td>7354</td>
<td>10179</td>
<td>4917</td>
<td>2144</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8T</td>
<td>10586</td>
<td>10253</td>
<td>4917</td>
<td>2185</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TSP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>—</td>
<td>6772</td>
<td>7661</td>
<td>1209</td>
<td>0</td>
<td>656</td>
<td>78194</td>
<td>0</td>
</tr>
<tr>
<td>2T</td>
<td>3112</td>
<td>14921</td>
<td>4240</td>
<td>1254</td>
<td>7</td>
<td>199</td>
<td>451358</td>
<td>10</td>
</tr>
<tr>
<td>4T</td>
<td>2572</td>
<td>22826</td>
<td>2271</td>
<td>1265</td>
<td>8</td>
<td>57</td>
<td>1941896</td>
<td>6</td>
</tr>
<tr>
<td>8T</td>
<td>43699</td>
<td>39920</td>
<td>1669</td>
<td>1478</td>
<td>11</td>
<td>47</td>
<td>2445451</td>
<td>8</td>
</tr>
<tr>
<td>WATER-N²</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>—</td>
<td>857</td>
<td>9483</td>
<td>1610</td>
<td>0</td>
<td>25608</td>
<td>1470</td>
<td>0</td>
</tr>
<tr>
<td>2T</td>
<td>81164</td>
<td>965</td>
<td>9425</td>
<td>1035</td>
<td>3970</td>
<td>25811</td>
<td>491</td>
<td>862</td>
</tr>
<tr>
<td>4T</td>
<td>142132</td>
<td>1047</td>
<td>9543</td>
<td>985</td>
<td>4554</td>
<td>26000</td>
<td>315</td>
<td>3103</td>
</tr>
<tr>
<td>8T</td>
<td>149460</td>
<td>861</td>
<td>14729</td>
<td>812</td>
<td>5460</td>
<td>39148</td>
<td>114</td>
<td>2822</td>
</tr>
<tr>
<td>WATER-SP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>—</td>
<td>4704</td>
<td>21724</td>
<td>1721</td>
<td>0</td>
<td>386</td>
<td>10432</td>
<td>0</td>
</tr>
<tr>
<td>2T</td>
<td>23160</td>
<td>5116</td>
<td>21655</td>
<td>487</td>
<td>279</td>
<td>253</td>
<td>3284</td>
<td>0</td>
</tr>
<tr>
<td>4T</td>
<td>24656</td>
<td>5222</td>
<td>21677</td>
<td>306</td>
<td>692</td>
<td>249</td>
<td>3124</td>
<td>0</td>
</tr>
<tr>
<td>8T</td>
<td>29188</td>
<td>4733</td>
<td>21659</td>
<td>325</td>
<td>3514</td>
<td>252</td>
<td>2849</td>
<td>0</td>
</tr>
</tbody>
</table>
Cessor [19]. This results in a hot-spot on the starting processor during initialization. Since the performance is limited by the throughput of the network link of the processor, this causes higher stall times for cold misses. Therefore we see that multithreading is less successful at tolerating the stall time during initialization. Finally, multithreading tolerates lock stall times successfully, except for the cases in RAYTRACE and TSP. This is due to the lock contention introduced by the “centralized” data structures in these two applications.

In general, whether multithreading can tolerate stall times depends on the amount of parallelism. The overheads are also quite significant in some cases and can partially or fully offset the gain from the reduction in the stall times. Finally, network traffic may affect multithreading performance to a large extent. In the following sections, we examine these important issues in more detail: amount of parallelism, overhead, and network traffic.

Parallelism

As we see in Figure 3.2, the number of threads required for tolerating the stall times varies. To provide some insight into these results, Table 3.2 shows the average run length and the average latency of remote memory accesses and lock operations. A rough estimate of the minimum number of threads for hiding latency is given by equation 3.1.

\[
\text{Number of threads} = 1 + \frac{\text{Avg. stall time}}{\text{Avg. context switch overhead} + \text{Avg. run length}} \quad (3.1)
\]

In Equation 3.1, the run length is the time between two consecutive remote operations that make a processor stall. The context switch overhead includes both the times for scheduling threads and calling Pthreads API, and we observe that the average context switch overhead is approximately 110 µsec in our experiments. Using these measurements, we compute the estimated minimum number of threads that is necessary to
Table 3.3: Expected number of threads required to effectively tolerate latency

<table>
<thead>
<tr>
<th>Application</th>
<th>Number of threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>6</td>
</tr>
<tr>
<td>LU-NCONT</td>
<td>2</td>
</tr>
<tr>
<td>LU-CONT</td>
<td>3</td>
</tr>
<tr>
<td>OCEAN</td>
<td>3</td>
</tr>
<tr>
<td>RADIX</td>
<td>4</td>
</tr>
<tr>
<td>RAYTRACE</td>
<td>2</td>
</tr>
<tr>
<td>SOR</td>
<td>2</td>
</tr>
<tr>
<td>TSP</td>
<td>2</td>
</tr>
<tr>
<td>WATER-N²</td>
<td>3</td>
</tr>
<tr>
<td>WATER-SP</td>
<td>2</td>
</tr>
</tbody>
</table>

effectively tolerate latency and show the results in Table 3.3.

As we see in Figure 3.2 and Table 3.3, multithreading cannot tolerate the stall times effectively when we do not have a sufficient number of threads. The reason is that there is not enough parallelism to hide the communication latency when all the threads are waiting for outstanding remote requests. For example, OCEAN and WATER-N² require three threads from the estimation, and we can only partially tolerate the stall times using two threads in these two applications. This ratio also suggests that two threads should suffice for WATER-SP—as we see in Figure 3.1, WATER-SP does achieve its best performance with two threads. In most cases, however, the performance is also affected by other factors.

Overheads

While multithreading provides a latency-hiding benefit, it also incurs overhead due to the context-switching code. When this overhead is large, it can offset the benefits of multithreading in some cases. For example, we can observe from Figure 3.1 that TSP and WATER-N² have significantly large multithreading overheads. In most of the applications, this overhead also increases as the number of threads per processor increases. This is because the number of context switches increases (as shown in Table 3.2).
The DSM overheads are also significant in our applications. As shown in Figure 3.1, the increases in DSM overheads range from 2% to 135% with multiple threads per processor (relative to the single-threaded case). Since both remote requests and replies arrive asynchronously in the multithreaded cases but only remote requests arrive asynchronously in the single-threaded cases, the DSM overheads of the multithreaded cases are higher than those of their single-threaded counterparts. For example, we can see that most of the benefit with multithreading is offset by the increased DSM overhead in WATER-SP.
Figure 3.4: Normalized total number of bytes sent ($O =$ original, $nT = n$ threads)
Table 3.4: Network traffic statistics (O = original, nT = n threads)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Total Messages</th>
<th>Total Bytes Sent (kbytes)</th>
<th>Average Byte per Message</th>
<th>Average Bandwidth (kbytes/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>O 31520</td>
<td>63112</td>
<td>2050</td>
<td>5642</td>
</tr>
<tr>
<td></td>
<td>2T 31518</td>
<td>63278</td>
<td>2055</td>
<td>5494</td>
</tr>
<tr>
<td></td>
<td>4T 31518</td>
<td>63278</td>
<td>2055</td>
<td>5461</td>
</tr>
<tr>
<td></td>
<td>8T 31518</td>
<td>63278</td>
<td>2055</td>
<td>5454</td>
</tr>
<tr>
<td>LU-NCONT</td>
<td>O 156067</td>
<td>145024</td>
<td>951</td>
<td>2801</td>
</tr>
<tr>
<td></td>
<td>2T 66508</td>
<td>99605</td>
<td>1533</td>
<td>2400</td>
</tr>
<tr>
<td></td>
<td>4T 25902</td>
<td>44438</td>
<td>1756</td>
<td>1373</td>
</tr>
<tr>
<td></td>
<td>8T 25455</td>
<td>46802</td>
<td>1882</td>
<td>1654</td>
</tr>
<tr>
<td>LU-CONT</td>
<td>O 18777</td>
<td>35690</td>
<td>1952</td>
<td>4301</td>
</tr>
<tr>
<td></td>
<td>2T 18012</td>
<td>36159</td>
<td>1958</td>
<td>4418</td>
</tr>
<tr>
<td></td>
<td>4T 18980</td>
<td>36366</td>
<td>1958</td>
<td>4694</td>
</tr>
<tr>
<td></td>
<td>8T 29540</td>
<td>57598</td>
<td>1956</td>
<td>6188</td>
</tr>
<tr>
<td>OCEAN</td>
<td>O 167918</td>
<td>210688</td>
<td>1284</td>
<td>3911</td>
</tr>
<tr>
<td></td>
<td>2T 149178</td>
<td>160241</td>
<td>1099</td>
<td>3447</td>
</tr>
<tr>
<td></td>
<td>4T 151816</td>
<td>125359</td>
<td>845</td>
<td>2735</td>
</tr>
<tr>
<td></td>
<td>8T 250718</td>
<td>198124</td>
<td>809</td>
<td>2722</td>
</tr>
<tr>
<td>RADIX</td>
<td>O 262400</td>
<td>141477</td>
<td>552</td>
<td>5202</td>
</tr>
<tr>
<td></td>
<td>2T 160008</td>
<td>97029</td>
<td>620</td>
<td>5404</td>
</tr>
<tr>
<td></td>
<td>4T 160386</td>
<td>97671</td>
<td>623</td>
<td>5239</td>
</tr>
<tr>
<td></td>
<td>8T 161142</td>
<td>98963</td>
<td>628</td>
<td>5019</td>
</tr>
<tr>
<td>RAYTRACE</td>
<td>O 37735</td>
<td>45381</td>
<td>1231</td>
<td>1558</td>
</tr>
<tr>
<td></td>
<td>2T 22368</td>
<td>22972</td>
<td>1051</td>
<td>819</td>
</tr>
<tr>
<td></td>
<td>4T 16790</td>
<td>22474</td>
<td>1370</td>
<td>857</td>
</tr>
<tr>
<td></td>
<td>8T 15922</td>
<td>23283</td>
<td>1496</td>
<td>880</td>
</tr>
<tr>
<td>SOR</td>
<td>O 11258</td>
<td>14683</td>
<td>1335</td>
<td>1538</td>
</tr>
<tr>
<td></td>
<td>2T 11258</td>
<td>14723</td>
<td>1339</td>
<td>1504</td>
</tr>
<tr>
<td></td>
<td>4T 11258</td>
<td>14723</td>
<td>1339</td>
<td>1544</td>
</tr>
<tr>
<td></td>
<td>8T 11258</td>
<td>14723</td>
<td>1339</td>
<td>1532</td>
</tr>
<tr>
<td>TSP</td>
<td>O 18796</td>
<td>8014</td>
<td>436</td>
<td>517</td>
</tr>
<tr>
<td></td>
<td>2T 10243</td>
<td>5298</td>
<td>529</td>
<td>245</td>
</tr>
<tr>
<td></td>
<td>4T 3769</td>
<td>4285</td>
<td>760</td>
<td>188</td>
</tr>
<tr>
<td></td>
<td>8T 4750</td>
<td>4762</td>
<td>1026</td>
<td>178</td>
</tr>
<tr>
<td>WATER-N²</td>
<td>O 91997</td>
<td>23671</td>
<td>263</td>
<td>1710</td>
</tr>
<tr>
<td></td>
<td>2T 91675</td>
<td>24103</td>
<td>269</td>
<td>1820</td>
</tr>
<tr>
<td></td>
<td>4T 92322</td>
<td>24237</td>
<td>268</td>
<td>1708</td>
</tr>
<tr>
<td></td>
<td>8T 139843</td>
<td>30541</td>
<td>223</td>
<td>1439</td>
</tr>
<tr>
<td>WATER-SP</td>
<td>O 47948</td>
<td>66191</td>
<td>1380</td>
<td>367</td>
</tr>
<tr>
<td></td>
<td>2T 47363</td>
<td>66670</td>
<td>1407</td>
<td>378</td>
</tr>
<tr>
<td></td>
<td>4T 47474</td>
<td>66881</td>
<td>1408</td>
<td>369</td>
</tr>
<tr>
<td></td>
<td>8T 47363</td>
<td>66702</td>
<td>1408</td>
<td>360</td>
</tr>
</tbody>
</table>
CHAPTER 3. PERFORMANCE WITH MULTITHREADING SUPPORT

Network Traffic

Besides the amount of parallelism and overheads, network traffic also affects the performance of multithreading on software DSMs. To illustrate the effects of network traffic, Figure 3.3 shows the number of messages (normalized to the case with one thread) generated by each application. Each bar is broken down into the following four categories. The "Lock Messages" represent the communications required for locking and unlocking. The "Barrier Messages" are the communications required at the barrier arrival. The "Page Messages" are the communications for bringing a new page into local memory. The "Diff Messages" represent the communications for preserving local memory coherent with the distributed shared memory. This category also includes some messages during garbage collection. Figure 3.4 shows the total number of bytes sent (normalized to the single-threaded case) with analogous categories.

First, we observe that the number of messages is a more important factor than message size to multithreading performance. As shown in Table 3.4, some multithreaded applications send more messages with a smaller average size than their single-threaded counterparts (e.g., OCEAN and WATER-N² with eight threads), and other multithreaded applications send fewer messages with a larger average size than their single-threaded counterparts (e.g., RAYTRACE and WATER-SP with eight threads). As we see in Figure 3.2, multithreading is less effective at tolerating the stall times in the former case. This is because multithreading can tolerate the "wire time" for a large message but cannot tolerate the non-trivial software overhead of sending requests and handling replies.

In six out of the ten applications, the number of messages decreases with two and four threads per processor. As we see in Figure 3.3, the reduction in number of messages ranges from 0.5% to 85.2%. The number of messages for memory coherence is either

---

²Before performing garbage collection, every node has to get all the updates for maintaining the memory coherence.
unchanged or else decreases with two and four threads, when we compare it to the single-threaded case. For the eight-threaded cases, we see that both LU-CONT and OCEAN send more messages than their single-threaded counterparts. The variations in network traffic (shown in Table 3.4) are a result of the impact of multithreading on locality. For example, LU-NCONT enjoys improved locality with multiple threads, while locality degrades in LU-CONT and OCEAN beyond four threads. Since each processor’s local memory is large enough to hold the entire data set of each application, multithreading affects communication due to false sharing. On the one hand, multithreading can result in better task assignments which improve spatial locality (e.g., LU-NCONT); on the other hand, reducing the block size too much to accommodate additional threads can induce false sharing (e.g., OCEAN).

Although seven out of ten applications with multithreading outperform their single-threaded counterparts (with speedup ranging from 3% to 84%), three applications still perform worse with multithreading. As we saw in the experiments, some applications do not perform well with multithreading due to high lock stall times. Since these applications use centralized data structures for dynamic scheduling, lock contention exists. To help alleviate this problem, we focus on improving the performance of these applications in the next section.

### 3.3 Context Switching on Synchronization Only

Although multithreading helps tolerate the latency of acquiring locks, the lock waiting time of a remote processor may increase. Figure 3.5(a) shows an example. Thread 0 gets a lock and then performs a context switch to thread 1 when it encounters a remote memory miss. Therefore a remote processor in the multithreaded case may wait longer for the lock than in the single-threaded case if the run length of thread 1 on the local processor is also long. In addition, the lock waiting time rises further when thread 1
acquires the same lock, as shown in Figure 3.5(b). As we discuss in Section 2.4, local threads have higher priority than remote threads for locks. This can result in long delays for remote lock accesses.

This problem becomes obvious when an application uses locks to protect centralized data structures for dynamic scheduling (e.g., TSP and RAYTRACE). As we saw in Section 3.2.1, TSP performs poorly with more than one thread per processor. Each thread in TSP accesses a centralized work queue, which becomes a hot-spot. Since a thread must acquire the lock before accessing the work queue, lock contention is severe. In contrast, RAYTRACE uses distributed task queues, which reduce the likelihood of contention. Nevertheless, contention can still occur near the end of the computation when there is only one non-empty task queue. As we saw in Table 3.2, the average lock latencies of these two applications increase when the number of threads increases.

To help alleviate this problem, we investigate an alternative context switching strategy: switch-on-sync-only. Under the switch-on-sync-only strategy, a thread switches to another thread on synchronization but spins for replies on remote memory misses. The switch-on-sync-only approach has several advantages over switching on all long latency events. First, it reduces the context switching overhead since it does not switch on remote memory misses. Second, a processor will receive fewer asynchronous messages (which involve non-trivial overhead) by using switch-on-sync-only, since threads spin for the replies on memory misses. This reduces the overhead of handling asynchronous messages. Finally, switch-on-sync-only can minimize the lock waiting time on remote processors by reducing the chance that too many local threads acquire the same lock at the same time.

To evaluate the performance impact of the switch-on-sync-only strategy on applications with lock contention, we perform experiments on two of our applications which use centralized queues for scheduling: RAYTRACE and TSP.
Figure 3.5: Examples of lock waiting times on remote processor
3.3.1 Performance Results

Figure 3.6 shows the execution times (normalized to the single-threaded case) of the four applications. The bars labeled $nTS$ represent the cases with $n$ threads and using the strategy *switch-on-sync-only*. As we see in the figure, both RAYTRACE and TSP perform better than their counterparts with context switching on all long latency operations. For RAYTRACE, the multithreaded cases with *switch-on-sync-only* gain 2% to 10% more speedup than their counterparts without using *switch-on-sync-only* (when comparing to their single-threaded case). For TSP, the speedups of the multithreaded cases with *switch-on-sync-only* range from 26% to 39%, while their counterparts with switching on all latency operations perform worse than the single-threaded case.

Figure 3.7 shows a more detailed breakdown of the stall times. The top two categories are the same as those in Figure 3.2. The category "Memory Idle" represents the total
CHAPTER 3. PERFORMANCE WITH MULTITHREADING SUPPORT

Figure 3.7: Normalized stalling time (O = Original, nT = n threads with context switch on all latency operations, nTS = n threads with switch-on-sync-only)
Figure 3.8: Normalized number of messages (O = Original, nT = n threads with context switch on all latency operations, nTS = n threads with switch-on-sync-only)

Table 3.5: Network traffic characteristic with switch-on-sync-only (O = original, nT = n threads)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Total Messages</th>
<th>Total Bytes Sent (kbytes)</th>
<th>Average Byte per Message</th>
<th>Average Bandwidth (kbytes/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAYTRACE</td>
<td>O</td>
<td>37735</td>
<td>45381</td>
<td>1231</td>
</tr>
<tr>
<td></td>
<td>2T</td>
<td>29108</td>
<td>44195</td>
<td>1554</td>
</tr>
<tr>
<td></td>
<td>4T</td>
<td>23859</td>
<td>44129</td>
<td>1893</td>
</tr>
<tr>
<td></td>
<td>8T</td>
<td>23472</td>
<td>48219</td>
<td>2016</td>
</tr>
<tr>
<td>TSP</td>
<td>O</td>
<td>18796</td>
<td>8014</td>
<td>436</td>
</tr>
<tr>
<td></td>
<td>2T</td>
<td>14888</td>
<td>8546</td>
<td>587</td>
</tr>
<tr>
<td></td>
<td>4T</td>
<td>12383</td>
<td>9238</td>
<td>763</td>
</tr>
<tr>
<td></td>
<td>8T</td>
<td>11649</td>
<td>11474</td>
<td>1008</td>
</tr>
</tbody>
</table>

idle time for cold misses and coherence misses. As we see in the figure, the performance improvements of both RAYTRACE and TSP mainly come from the reduction of lock stall times (ranging from 6% to 80%, relative to their counterparts with context switching on all latency operations). In contrast, the memory stall times increase in these two applications with the context switching strategy switch-on-sync-only. Since threads do not switch on memory miss but spin for the replies only, the memory miss stall times cannot be tolerated.

3.3.2 Network Traffic

Figure 3.8 shows the number of messages (normalized to the single-threaded case) sent by the applications under both context switching strategies, and the categories are the same
as those in Figure 3.3. As we see in Figure 3.8, both RAYTRACE and TSP send 3% to 26% more messages for memory coherence in the multithreaded cases than in the single-threaded cases. Since the strategy *switch-on-sync-only* reduces the chance of combining lock requests, it is more likely that different threads on the same processor access the same page at different intervals. This results in more diffs and communication. In addition, these two applications send (3% to 15%) more messages with *switch-on-sync-only* for cold misses than their counterparts with switching on all long latency operations. Since each thread get the data dynamically on the centralized data structures, the reduction in the likelihood of combining lock requests increases the chance that the threads on the same processor get the data which scatter on different memory pages. This increases the number of memory pages accessed by a processor and hence results in more message traffic for cold misses.

While the context switching strategy *switch-on-sync-only* helps improve the performance of those applications that have lock contention, it does not improve the performance of other applications without lock contention. In the next section, we examine the source codes of the applications to see whether they can be further improved to enhance multithreading performance.

### 3.4 Performance Enhancement with Source Code Modification

Most applications do not require any modifications to run in a multithreading fashion, other than replicating "private" data on the heap whenever appropriate such that each thread has its own copy. In addition to these minor modifications for the sake of program correctness, we have found that other modifications are quite helpful for enhancing *performance*.

Based on the performance results in Section 3.2.1, we found that FFT does not have
The Six-Step FFT Algorithm

1. Transpose data matrix
2. Perform 1-D FFT on each row of data matrix
3. Apply roots of unity to data matrix
4. Transpose data matrix
5. Perform 1-D FFT on each row of data matrix
6. Transpose data matrix

Figure 3.9: The FFT algorithm

any performance improvement from multithreading. This is due to the redundancy during initialization and lack of parallelism to tolerate latency during the transpose phase. As described in Section 3.1, the first row of the root of the unity matrix is replicated for each process, and it is read-only after initialization; therefore each process must initialize its own replicated row before executing the algorithm (in Figure 3.9). In the multithreaded case, however, each thread is a virtual process and it has its own replicated row; therefore the initialization of all replicated rows causes redundancy on the same processor. In addition, the threads on the same processor tend to access the same set of rows during the transpose step. In FFT, each row of the matrices is allocated contiguously on a page, and the threads on the same processor perform a transpose on different blocks within the same row during the transpose step. This increases the chance that the threads wait on the same memory miss and hence reduces the amount of parallelism available to hide latency.

As we saw in Section 3.2.1, WATER-N² also performs worse beyond two threads. The algorithm of WATER-N² is shown in Figure 3.10. As we observed in the experiments, WATER-N² spends most of the time in the computation of inter-molecular forces. In this phase, a private force array is used in each process to avoid communication, and it
Set up scaling factors and constants.
Read in displacement values and randomize velocities.
Compute forces once to estimate accelerations.
Repeat NSTEP times {
    Calculate predicted values for atomic variables.
    Compute intra-molecular forces for all atoms.
    Compute inter-molecular forces.
    Calculate corrected values of variables from the predicted values and computed forces.
    Boundary conditions: put molecules back inside the box if they are not inside the box.
    Compute kinetic energy of system.
    if (this is an output time-step) {
        Calculate potential energy.
        Print the output.
    }
}

Figure 3.10: The WATER-N² Algorithm

is reinitialized in each time step. Although this helps avoid communication in the single-threaded case, it causes redundant computation in the multithreaded cases because each thread keeps a copy of a private force array. In addition, each thread computes the interactions between molecules independently, and two different threads on the same processor may access a molecule at different times. Since a thread must get the lock to protect a molecule before computing interactions, this may result in more remote lock acquires and remote data accesses.

3.4.1 Application Enhancements

To avoid the redundancy, we modify the applications in the following way. Rather than acting as a virtual process, each thread on the same processor shares the data set assigned to a single processor. In this manner, the replicated data structure on each processor appears only once. Therefore, we can minimize the redundant initialization in both
To ensure program correctness after modification, we use local barriers to synchronize the threads on the same processor. A local barrier is similar to a global barrier, with the exception that local barrier synchronizes the threads on the same processor only and does not require any interprocessor communication. In the original case (without this modification), each thread performs the task of a virtual process, and its computation is independent of those of other threads on the same processor (as we have already privatized the global variables). In contrast, the computations of the threads become dependent when we apply this modification. Figure 3.11 shows an example of sharing computation in a for loop.

In Figure 3.11(a), a thread can safely execute the second loop after finishing the first loop without endangering program correctness. In contrast, this is not the case in Figure 3.11(b), because a thread can be executing the second loop while some other threads may still be executing the first loop. Hence this may result in data inconsistency. To alleviate this problem, we insert a local barrier between the two loops.

### 3.4.2 Performance Results

In our experiments, we find that the source code modifications can improve the performance of FFT and WATER-N². Hence, we show the experiment results of these two applications. Figure 3.12 shows the normalized execution times of the two applications (both before and after modification). Both applications with modifications outperform their counterparts without modifications. The speedups of the modified FFT range from 5% to 7% (relative to the single-threaded case), while there is no speedup in the original FFT with multithreading. The speedups of the modified WATER-N² range from 16% to 20%, while there is only 4.6% speedup with two threads in the original WATER-N². As we see in the figure, the busy times of both applications decrease. This is because the
(a) Before modification

// gtid is the global thread id and chunk size is
// defined by total size / total number of threads on all processors

for (i = gtid \times chunk size; i < (gtid + 1) \times chunk size; i++) {
    A[i] = i;
}

for (j = gtid \times chunk size; j < (gtid + 1) \times chunk size; j++) {
    sum += A[j] + A[gtid \times chunk size];
}

(b) After modification

// pid is processor id, tid is the local
// thread id, and chunk size is defined by
// total size / number of processors.

for (i = pid \times chunk size + tid; i < (pid + 1) \times chunk size; i += number of threads) {
    A[i] = i;
}

/*
All threads will be synchronize here before executing the
second loop
*/
local_barrier();

for (j = pid \times chunk size + tid; j < (pid + 1) \times chunk size; j += number of threads) {
    sum += A[j] + A[pid \times chunk size];
}

Figure 3.11: Examples of using local barriers
enhancements eliminate the redundant computations of the applications by keeping only a single private data structure (the first row of the root of the unity matrix in FFT and the private force array in WATER-N²).

Figure 3.13 shows the breakdown of the idle times of the two applications, both with and without modifications. The categories are the same as those in Figure 3.2. We can observe from the figure that the modifications also help reduce the stall times in both FFT and WATER-N². As we compare the results in Tables 3.2 and 3.6, we see that both numbers of misses and lock requests decrease in WATER-N² in the cases with modification. Since the modifications make each thread share the task within the same processor, it is less likely that the threads perform redundant operations on the molecules owned by this processor. This reduces not only the numbers of remote accesses and lock requests but also their stall times. Finally, the modifications can reduce the stall time
Figure 3.13: Normalized stalling time ($O = \text{original}, nT = n \text{ threads without source code modification}, nTM = n \text{ threads with source code modification}$)

Table 3.6: Multithreading statistics with source code modifications ($O = \text{original}, nT = n \text{ threads}$)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Total Context Switches</th>
<th>Run Length (\mu sec)</th>
<th>Total Remote Misses</th>
<th>Average Remote Latency (\mu sec)</th>
<th>Combined Requests</th>
<th>Total Remote Acquires</th>
<th>Average Remote Latency (\mu sec)</th>
<th>Stolen Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td></td>
<td></td>
<td>12013</td>
<td>3554</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2T</td>
<td>16258</td>
<td>972</td>
<td>13013</td>
<td>2189</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4T</td>
<td>17639</td>
<td>964</td>
<td>13013</td>
<td>2020</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8T</td>
<td>19340</td>
<td>948</td>
<td>13013</td>
<td>2072</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>WATER-N2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td></td>
<td></td>
<td>9483</td>
<td>1610</td>
<td>0</td>
<td>25808</td>
<td>1470</td>
<td>0</td>
</tr>
<tr>
<td>2T</td>
<td>90400</td>
<td>938</td>
<td>9425</td>
<td>1244</td>
<td>1182</td>
<td>25806</td>
<td>241</td>
<td>0</td>
</tr>
<tr>
<td>4T</td>
<td>66567</td>
<td>950</td>
<td>9323</td>
<td>1195</td>
<td>1180</td>
<td>25808</td>
<td>102</td>
<td>0</td>
</tr>
<tr>
<td>8T</td>
<td>70763</td>
<td>981</td>
<td>9419</td>
<td>997</td>
<td>1139</td>
<td>25810</td>
<td>36</td>
<td>0</td>
</tr>
</tbody>
</table>

of remote memory accesses in FFT. Since the threads perform transposition on different rows within the same block after the source code modification and FFT allocates every row on a separate shared memory page, it minimizes the chance that all threads on the same processor block on the same memory miss. This increases the available parallelism for tolerating memory stall times.

In Figure 3.14, we see that the number of messages in the modified FFT stays the
Figure 3.14: Normalized number of messages with source code modifications (O = original, nT = n threads without source code modification, nTM = n threads with source code modification)

Table 3.7: Network traffic characteristic with source code modifications (O = original, nT = n threads)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Total Messages</th>
<th>Total Bytes Sent (kbytes)</th>
<th>Average Byte per Message</th>
<th>Average Bandwidth (kbytes/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>31520</td>
<td>63112</td>
<td>2050</td>
<td>5642</td>
</tr>
<tr>
<td>2T</td>
<td>31520</td>
<td>63279</td>
<td>2055</td>
<td>5933</td>
</tr>
<tr>
<td>4T</td>
<td>31520</td>
<td>63279</td>
<td>2055</td>
<td>6078</td>
</tr>
<tr>
<td>8T</td>
<td>31519</td>
<td>63278</td>
<td>2055</td>
<td>6056</td>
</tr>
<tr>
<td>WATER-N²</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>91997</td>
<td>23671</td>
<td>283</td>
<td>1710</td>
</tr>
<tr>
<td>2T</td>
<td>91576</td>
<td>24146</td>
<td>270</td>
<td>2024</td>
</tr>
<tr>
<td>4T</td>
<td>91476</td>
<td>24265</td>
<td>271</td>
<td>2107</td>
</tr>
<tr>
<td>8T</td>
<td>91578</td>
<td>24429</td>
<td>273</td>
<td>2625</td>
</tr>
</tbody>
</table>
same as those in the original FFT. The modifications to FFT eliminate the redundancy (which does not cause any communication) in the initialization phase and minimize the chance of blocking all threads on the same memory miss (which does not affect the message traffic) during the transpose step. In contrast, the number of messages required in WATER-N² drops when we use more threads per processor. The differences between the original version and the modified version are not significant with two and four threads per processor. However, the modified WATER-N² has 35% fewer messages than its counterpart without modifications in the eight-threaded case. Since the modifications make all threads share the task of a processor, it decreases the chance that these threads perform redundant operations (both locally and remotely) on the molecules. This can greatly reduce the network traffic by eliminating some remote lock requests and data accesses (as we see from Tables 3.2 and 3.6).

3.5 Impact of Multithreading on Scalability

The previous sections demonstrate that multithreading is quite effective at tolerating the latency encountered by software DSMs. In this section, we study whether the speedups of the applications with multithreading scale well. We examine whether multithreading can improve the performance of the applications running on few or many processors.

We run our experiments (in both single-threaded and multithreaded cases) on one, two, four, eight, and twelve processors.³ In Sections 3.3 and 3.4, we found that (i) switching on synchronization only improves the performance of those applications with dynamic scheduling, and (ii) source code modification improves the performance of applications with redundant computation. Since we want to show the best possible performance, we make RAYTRACE and TSP perform context switching on synchronization only and eliminate the redundancy of FFT and WATER-N² by modifying their source codes.

³FFT, OCEAN, and RADIX run only when the number of processors is an integer power of two, so there are no speedup results for these three applications with twelve processors.
Figure 3.15 shows the speedup curves of the applications in both single-threaded and multithreaded cases. As we see in the figure, only four applications (LU-NCONT, RAYTRACE, SOR, and WATER-SP) scale well with respect to speedup with a single thread per processor. We also see that three applications (FFT, OCEAN, and RADIX) do not speedup with a single thread per processor. Finally, the speedup of TSP scales well with two or four processors in the single-threaded cases but drops from 3.5 to 2.7 when number of processors increases from eight to twelve.

For most of the applications, multithreading does not improve the speedups significantly when we use two processors. This is because the stall times are not large in the two-processor cases (both with a single thread and multiple threads). Since each processor's local memory is large enough to hold the entire data set of each application, the communication latencies in the two-processor cases are not significant because there is little data sharing (both true sharing and false sharing).

When the number of processors grows, however, there is more data sharing and hence communication latency becomes much larger than that in the two-processor case with a single thread. Hence we can see noticeable performance improvements with multithreading beyond four processors. In addition, the benefits of using multiple threads per processor become more obvious in those applications (e.g., RAYTRACE and TSP) where performance is dominated by lock stall times. Since a lock manager redirects a lock request message to the previous releaser (if the manager itself is not the previous releaser), the lock stall times are more significant in the cases with more than two processors than those in the two-processor cases (for which redirection of lock request message is not necessary). For TSP, the speedups of the single-threaded cases are comparable to those of the multithreaded cases when we use two and four processors. In contrast,
Figure 3.15: The speedup graphs (Continued on next page)
Figure 3.15: The speedup graphs (Continued from previous page)
we see that the speedups of the single-threaded cases start deteriorating but those of the multithreaded cases still increase beyond four processors. This is mainly because multithreading minimizes lock contention (which is due to the centralized task queue) in TSP. Finally, we can see that the applications fail to have scalable speedups with multithreading if they already fail to have scalable speedups with a single thread per processor (although the multithreaded speedups are still higher than the corresponding single-threaded speedups).

3.6 Chapter Summary

From our experiments, we observe that six out of ten applications improve in performance under multithreading. The speedups of these six applications are 3% or more, with two applications speeding up by over 50%. Multithreading improves the performance of the applications by tolerating the memory miss stall times (which decrease by 20% to 80%). In some cases, multithreading also reduces the synchronization stalls significantly. We see from the experiments that the degree of performance improvement with multithreading depends mainly on the following factors.

1. The degree to which misses occur in a short period of time such that they create a hot-spot on a few processors. This is particularly problematic during initialization, where hot-spotting occurs as the processors read their initial data sets from the master processor. Since performance is limited by the throughput of the master processor's network link in these cases, multithreading shows less improvement than one might otherwise expect.

2. There is a sufficient number of threads available to tolerate the latencies. We find that multithreading can only partially hide the latency if there is not enough parallelism.
3. Whether multithreading and DSM overheads are large enough that they either partially or fully offset the performance benefit of using multithreading. The multithreading overhead becomes non-trivial if the number of context switches is large. Since a thread no longer spins waiting on the reply of an outgoing message, the overhead of handling the asynchronous replying message becomes significant in the multithreaded cases.

4. The impact of multithreading on locality has both positive and negative effects. On the one hand, multithreading can result in better task assignments which improve spatial locality and reduce false sharing; on the other hand, reducing the data size too much to accommodate additional threads can induce false sharing.

We also studied enhancements to the applications with multithreading and make the following observations:

1. Context switching on synchronization only helps improve the speedups of those applications with lock contention. In the single-threaded applications with a centralized data structure for task scheduling, we can see that the stall times for locks are already large due to contention on the locks for protecting the centralized data. The lock contention becomes more severe in the multithreaded case, since the local thread has higher priority than the remote one in getting a lock, thus making the remote thread wait longer for the lock. Context switching on synchronization only minimizes the chance that local threads acquire a lock within an interval. From our experiments, we see that the applications have 6% to 39% speedups with this context switching strategy.

2. Modifying the source code of the applications (that require “private” data structure to avoid communication) enhances performance. Since each thread on the same processor performs the task of a (virtual) processor, the initialization to its “private”
data structure results in redundant computation and hence poor performance. To eliminate these redundancies, we modify the source codes as follows. Each thread does not act as a virtual processor; instead, all of the threads on the same processor share the task and the data set assigned to this processor. From our experiments, we observe that modifying the source codes of FFT and WATER-N^2 results in large improvement to these two applications. The speedups of the modified FFT range from 5% to 7%, whereas the original FFT does not have any speedup; the speedups of the modified WATER-N^2 range from 16% to 20%, while the original WATER-N^2 has 4.6% speedup in the two-threaded case.

Finally, we found that the multithreading speedups scale well beyond two processors if the speedups in the single-threaded cases also scale well. For the application whose lock stalls account for a significant fraction of execution time, the speedups of the multithreaded cases become more scalable than those of the single-threaded cases.
Chapter 4

Combining Prefetching and Multithreading

Non-binding software-controlled prefetching is another technique, besides multithreading, for tolerating communication latency. In this chapter, we evaluate the impact of combining multithreading and non-binding software-controlled prefetching on TreadMarks. We perform experiments on eight benchmarks: FFT, LU-NCONT, LU-CONT, OCEAN, RADIX, SOR, WATER-N², and WATER-SP.¹ In Section 4.1, we describe some relevant details of prefetching [6] and compare the performance results between prefetching and multithreading. In Section 4.2, we discuss the details of combining both multithreading and prefetching in a software DSM. We then examine the performance of the combined technique. Later, in Section 4.3, we consider an alternative strategy for combining multithreading and prefetching: using multithreading to handle synchronization latency and prefetching to hide memory latency. We discuss the performance differences of multithreading alone, prefetching alone and this alternative combined technique. In Section 4.4, we summarize the whole chapter.

¹Both RAYTRACE and TSP are not included because they use global data structures to schedule both data set and there is not enough time to prefetch the data.
4.1 Non-Binding Software-Controlled Prefetching

Before studying the effects of combining prefetching and multithreading, we describe some issues concerning non-binding software-controlled prefetching on TreadMarks. This work was done by Chan [6], and we are just summarizing the relevant details here. The idea behind prefetching is to use knowledge of future access patterns to bring remote data into the local memory before it is actually needed. In particular, software-controlled prefetching requires either the programmer or the compiler to insert explicit prefetch calls into the code. Moreover, non-binding prefetching gives the programmer or the compiler the flexibility to insert prefetches more aggressively without worrying about violating program correctness [30, 31]. Therefore we consider combining multithreading with non-binding software-controlled prefetching in this chapter.

Explicit prefetch calls are inserted into the source code of the applications as follows. Except for WATER-SP, which uses linked lists, all applications use arrays as their primary data structures. Therefore Mowry's prefetching algorithm [30] is applied 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 a software DSM is quite similar to prefetching page faults to hide the latency of out-of-core I/O [33]. For two of the seven array-based applications—FFT and LU-NCONT—compiler-inserted prefetching (with an implementation of prefetching in the SUIF compiler [39]) is used, and the performance is within 3% of that with programmer inserted prefetching. In the other five cases (LU-CONT, OCEAN, RADIX, SOR and WATER-N²), the hand-tuned prefetching code achieved better performance. Hence, to show the full potential of prefetching, we look at the results of hand-inserted prefetching in the latter five cases and compiler-inserted prefetching for FFT and LU-NCONT. Further details about non-binding software-controlled prefetching on software DSM can
Figure 4.1: Normalized execution time (O = original, nT = n threads, P = prefetch) be found in [6, 32].

4.1.1 Results with Prefetching Alone

The experimental results of prefetching were taken from Chan’s work [6], and their experimental framework were the same as ours. Figure 4.1 shows the impact of prefetching on the execution time for all of the applications. The leftmost four bars are taken from our experiments in Chapter 3. For each application, the “prefetch overhead” represents the overhead of issuing the prefetches. The remaining categories are the same as in Figure 3.1, except that for the prefetching case, the “busy” time includes prefetch overheads.
associated with loop transformation and unnecessary prefetches. Moreover, the "DSM overhead" includes the amount of time spent on servicing prefetch requests and replies.

As we observe in Figure 4.1, prefetching improves the execution time of all applications, with speedups ranging from 4% to 29%. This improvement results from significant reductions in the memory miss stall times, ranging from 45% to 89%. This benefit is large enough that it can offset the runtime overheads of prefetching. The DSM overhead increases because of two effects: (i) prefetch requests are more expensive to service than normal memory requests, and (ii) prefetches which fail to fully hide the latency result in a retry request upon the real access [6]. In general, the busy times increase due to the extra prefetch computation embedded in the application source codes and due to unnecessary prefetches. The slight decrease in busy time in some cases is largely due to small measurement variations, since the experiments were performed on real hardware.
4.1.2 Performance Comparison Between Multithreading and Prefetching

To gain further insights into the differences between the performance results of multithreading and prefetching, Figure 4.2 shows a detailed breakdown of what happened to the original remote misses with prefetching. The topmost section ("no pf") is the fraction that was not prefetched. The "pf-miss: invalidated" case is where prefetched data is brought into the local memory but is invalidated before it can be used, and the "pf-miss: late" category are cases where the data has been prefetched but does not return in time to satisfy the reference. Finally, the "pf-hi" represents those prefetches which fully hide the latency.

Comparing the performance of multithreading and prefetching in Figure 4.1, one finds that the technique giving the best performance results depends on the application. For example, RADIX achieves better performance with multithreading. The primary problem here for prefetching is that the loop structure makes it difficult to schedule prefetches early enough to hide the large network latencies—notice in Figure 4.2 that RADIX has the largest fraction of late prefetches. In addition, RADIX suffers from network contention delays because of its high rate of communication. These two effects interact negatively: increased network contention slows down prefetches, causing them to arrive even later, and as more prefetches become late, they result in more retry messages, thus resulting in more network contention. Hence multithreading outperforms prefetching in RADIX, since it can address memory latency without requiring the ability to predict addresses far in advance (thus avoiding the late prefetch problem). Multithreading also outperforms prefetching when lock stalls account for a significant fraction of execution time in an application. For example, WATER-N² in the single-threaded case has 34% of its execution time spent on stalling for locks. While prefetching is more effective than multithreading at tolerating the memory stall time in this application, multithreading
still outperforms prefetching since it eliminates 98% of the original lock stall time in the eight-threaded case while prefetching does not address the lock stalls.

Prefetching is generally just as good (if not better) than multithreading at tolerating memory latency if the access pattern of the application is predictable. For the four applications (FFT, LU-CONT, SOR, and WATER-SP) with predictable access patterns, we can see from Figure 4.2 that prefetching can successfully hide more than 90% of the original miss latency. By comparing the performance results from multithreading and prefetching in these four applications, we observe that prefetching can hide more memory stall time than multithreading and hence outperforms multithreading. In both FFT and SOR, multithreading is less effective than prefetching at tolerating the stall time due to cold misses. As we discussed in Section 3.2, multithreading is not effective at tolerating memory stall time during initialization. Since the stall times for cold misses are quite significant in these two applications, we see that prefetching can reduce the original memory miss idle time by 40% more than multithreading.

### 4.2 Combining Both Latency Tolerance Schemes

In Chapter 3 and Section 4.1, we saw that multithreading and prefetching can improve application performance by tolerating the long network latency in a software DSM environment. An interesting question is whether the combination of the two techniques offers even better performance than applying either one individually. On the one hand, each technique might compensate for the other technique’s weaknesses, thereby hiding more latency; on the other hand, the techniques may interfere with each other, thus degrading performance. In this section, we examine whether combining both prefetching and multithreading further improves the performance. First, we study optimizations which making prefetching more suitable for multithreading. Then we study the performance of the combined technique.
Figure 4.3: Eliminate unnecessary prefetches in multithreading by inserting a conditional test

One might expect multithreading to be the right technique for hiding synchronization latency, since prefetching does not directly address this problem. To hide memory latency, however, the right approach is less clear, since both prefetching and multithreading can potentially hide this latency. In this section, we apply multithreading to hide synchronization latency and both prefetching and multithreading to tolerate memory latency.

4.2.1 Optimizing Prefetching for Multithreading

We discovered that naively applying the original prefetching scheme described in Chan's work [6] to multithreaded code resulted in disappointing performance. Since there is often significant overlap among the working sets of threads running on the same processor, the first thread which touches remote data often effectively "prefetches" it for other threads. Hence we would like to avoid having these subsequent threads issue unnecessary prefetches, but there are two complications: all threads execute the same static code, and we do not know a priori which thread will arrive at the data first (since we schedule threads dynamically). To address this problem, we identify cases where threads on the same processor would be redundantly prefetching the same data, and we protect these prefetches with a conditional test of a dynamic flag which is explicitly reset by the first
thread to arrive at the data. Figure 4.3 shows an example. The dynamic flag is set again after all threads on the same processor have accessed the code. This allows us to avoid unnecessary prefetches again if we have widely separated accesses to this same code.

A second optimization was useful in the case of RADIX. Since the access pattern of RADIX is highly unpredictable, the original prefetching scheme issues prefetches for the entire histogram. This would result in many unnecessary prefetches, as well as a high volume of network traffic. To help minimize the load on the network, we reduced the number of prefetches arbitrarily by eliminating every other dynamic prefetch.\(^2\) Although this resulted in a lower prefetching coverage factor, this was more than offset by reductions in network queuing delays in this particular case.

4.2.2 Performance Results

Figure 4.4 shows the performance results of applying prefetching alone, multithreading alone, and the combined technique. The last three bars for each application show the execution times for the combined cases with two, four, and eight threads per processor. As we see in the figure, most of the applications have the best performance with either prefetching alone or else multithreading alone. Only WATER-N\(^2\) and OCEAN work well with the combined technique in the two-threaded cases. We can observe that the memory miss idle times of the combined technique are smaller than those of prefetching alone and multithreading alone in some applications (LU-NCONT, LU-CONT, and RADIX). However, these reductions are offset by higher synchronization stall time and/or larger DSM overhead.

To provide more insights into why combining prefetching and multithreading resulted in poor performance, Figure 4.5 shows the normalized number of messages. The category "Sync Message" shows the messages required for synchronization, while the categories

\(^2\)Although this optimization is the best we could do, better results may still be achieved by reducing prefetches in other ways.
Figure 4.4: Execution time of combined techniques ($O$ = original, $P$ = prefetch, $nT$ = $n$ threads without prefetching, $nTP$ = $n$ threads with both techniques to hide memory stalls)
Figure 4.5: Normalized number of messages (O = original, P = prefetch, nT = n threads without prefetching, nTP = n threads with both techniques to hide memory stalls)
“PF Message” and “Data Message” represent the messages for prefetch and actual miss, respectively, of the applications.

The performance is poor because switching between threads tends to result in bursty miss patterns, which in turn slow down requests in the network, including prefetches. When prefetches fail to return in time, a retry occurs (since prefetches are unreliable) followed by a thread switch. This further exacerbates the problem since the newly swapped-in thread might also send messages on the already congested network. As we see from Figure 4.5, we need many normal memory request messages even with a significant number of prefetch messages in FFT, LU-CONT, and WATER-SP. Therefore the stall times of these applications increase due to the network queuing delay. We also tend to pay full overheads of both prefetching and multithreading. In fact, the overhead tends to go up since both techniques involve more asynchronous messages (prefetching handles the prefetch messages asynchronously, while multithreading handles all memory request and lock request messages asynchronously). This causes the large DSM overhead in the combined technique, without appreciably improving our ability to hide latency.

4.3 Prefetching and Multithreading with Context Switch on Synchronization Only

Since using both prefetching and multithreading to hide memory latency resulted in worse performance, one would expect that it is sufficient to use either technique to tolerate memory latency. Synchronization latency is also high in some cases, and we would like to combat this with the latency tolerance techniques. We have already shown in Chapter 3 that multithreading can tolerate both memory and synchronization latencies. While prefetching does not hide synchronization latency, we would like to see whether multithreading can complement prefetching as follows: prefetching hides memory latency and multithreading tolerates synchronization latency (only). To do this, we make all the
threads perform context switch on synchronization only and spin for replies of remote memory requests.

4.3.1 Performance Results

Figure 4.6 shows the performance results of applying prefetching alone, multithreading alone, and the combined technique with multithreading to hide synchronization only. The last three bars for each application on the graph show the parallel execution times for the combined cases with two, four, and eight threads. As we see in the Figure 4.6, three of the eight applications (FFT, OCEAN, and WATER-N^2) achieve the best performance through the combination of prefetching and multithreading, with speedups ranging from 4% to 26% over either technique alone. In two cases (LU-NCONT and RADIX) the best performance occurs with multithreading alone, and in three cases (LU-CONT, SOR, and WATER-SP) prefetching alone performs the best.

4.3.2 Further Details

Why does this combined approach fail to outperform the individual techniques in these latter five cases? In LU-NCONT, the combined approach achieves the best performance with fewer than eight threads per processor, and is comparable to the best case even with eight threads.

Since we use multithreading to hide synchronization latency only, the effectiveness of the combined technique at tolerating memory latency depends on the performance of prefetching. As we saw in Section 4.1, prefetching can hide memory latency if the access pattern is predictable. In RADIX, however, the loop structure makes it difficult to predict addresses and schedule prefetches early enough to hide the large network latencies. This is also true for the combined technique used in this section, since using multiple threads only makes it more difficult to predict memory addresses. In addition, RADIX also suffers from network contention delays due to its even higher rate of communication than that
Figure 4.6: Normalized execution times (O = original, P = prefetch, \( nT = n \) threads without prefetching, \( nTP = n \) threads with prefetching to hide memory stalls)
Figure 4.7: Normalized number of messages (O = optimal, P = preetched, N = n threads without preetching, n threads with preetching) to hide memory stalls.
in the prefetching alone case. As we discussed in Section 4.1, these two effects interact negatively. This would result in larger network traffic as shown in Figure 4.7. Hence multithreading alone is the clear winner for RADIX, since it can address memory latency without requiring the ability to predict memory addresses far in advance.

In LU-CONT, SOR, and WATER-SP, prefetching is more effective at hiding memory latency than multithreading, so clearly we would at least want to use prefetching. What may be surprising, however, is that we are better off doing nothing for synchronization latency rather than attempting to tolerate it through multithreading. The reason for this is twofold. First, the bulk of the synchronization latency in these cases is due to barriers, and multithreading improves barrier stall times only indirectly by improving load balancing. Second, there is a significant fixed cost involved in supporting multithreading (as shown in Chapter 3), since all message arrivals must then be handled asynchronously. As we see in Figure 4.6, this increased overhead (which mostly appears as “DSM Overhead”) more than offsets any gains in reduced synchronization stall times.

The largest performance gain from the combined scheme is in WATER-N². As we saw in Section 4.1, multithreading alone outperforms prefetching alone in WATER-N² because lock stalls in this application dominate the stall times. Therefore multithreading can improve the performance of WATER-N² by successfully hiding most of the synchronization idle time. However, the access pattern of this application is also predictable. One would expect that prefetching could help tolerate the memory stall time (as in the prefetching alone case which eliminates 83% of the memory stall time). Throughout the experimental results for the combined technique, we find that prefetching eliminates 85% of the memory stall time (as opposed to less than 50% with multithreading), and multithreading eliminates roughly 80% of the synchronization stall time (which is primarily due to locks). Hence WATER-N² is a good illustration of how prefetching and multithreading can be combined in a complementary fashion.
4.4 Chapter Summary

In this chapter, we have evaluated the impact of adding non-binding software-controlled prefetching to our existing multithreaded software DSM. The experiments on combining prefetching and multithreading demonstrate the following:

1. Using both prefetching and multithreading to tolerate memory latency is not a good idea. In our experiments, four of the applications perform worse for the combined technique than for either technique alone. This is because both techniques tend to send many asynchronous messages that increase the network load and DSM overhead for handling the messages.

2. Combining prefetching and multithreading such that multithreading hides synchronization latency and prefetching hides memory latency produces improved performance in three of the applications, with speedups ranging from 4% to 26% over either technique applied alone.

3. Applications can achieve the best performance with multithreading alone if it is difficult to predict memory addresses early enough to schedule prefetches effectively. Multithreading alone also outperforms prefetching alone if lock stalls account for a significant fraction of execution time.

4. Prefetching is more effective than multithreading at tolerating memory latency if the data access pattern is predictable. Also, if synchronization stall times are small or are dominated by barrier stalls, it is more likely that prefetching alone will outperform both multithreading alone and the combined techniques.

5. Applications can achieve the best performance with the combined technique (i.e., using prefetching to handle memory latency and using multithreading to handle
synchronization latency only) if the memory access pattern is predictable and lock stalls account for a significant fraction of execution time.

In general, the best performance can be achieved by prefetching alone if the application with array as primary data structure has a predictable data access pattern and does not frequently modify global variables. In addition, multithreading is likely to complement prefetching by hiding the remaining lock stall times if updates to global variables (i.e., accesses to critical sections) are also frequent. Finally, multithreading alone is likely to be the best strategy for hiding latency if the application modifies the data without a predictable pattern (e.g., dynamically getting task from a task queue).
Chapter 5

Conclusion

To achieve high performance on software DSMs, it is necessary to cope with high communication latency. Since we cannot completely eliminate communication and its overhead on software DSMs, we must tolerate its latency. First, we can increase the degrees of pipelining and buffering of memory writes by using a relaxed memory model. However, the latencies of read and synchronization still make software DSMs suffer. To tolerate read latency, multithreading is an attractive solution. Multithreading does not rely on predicting addresses, and therefore can handle complex and unpredictable data access patterns. In addition, it can directly address synchronization latency, unlike other latency tolerance techniques that fail in this regard. In this thesis, we evaluate multithreading on a software DSM system, namely TreadMarks. We investigate whether applying multithreading can improve performance of applications on software DSM.

The key results are the following:

1. Multithreading can be quite effective at tolerating communication latency on software DSM. Six of the ten applications we study have speedups of 3% or more, with two applications speeding up by over 50%. The improvement is due to reductions in memory stall times (ranging from 20% to 80%) and synchronization stall times. We found that whether multithreading can effectively tolerate communication latency generally depends on the following factors: sufficiency of parallelism, overheads
introduced by multithreading, and network traffic.

2. For those applications that use a centralized data structure for dynamic scheduling, context switching on synchronization only can further improve their performance (with speedups ranging from 6% to 39% compared to the single-threaded cases) by minimizing lock contention. By applying source code optimizations to minimize redundant computation on "private" data structures, we can improve the performance (with speedups ranging from 5% to 20%) of those applications that require private data structures to avoid communication.

3. The multithreading speedups scale well beyond two processors if the speedups in the single-threaded cases also scale well. For those applications with a significant amount of lock stalls in the single-threaded case, multithreading also helps to improve scalability.

4. Combining prefetching and multithreading such that multithreading hides synchronization latency and prefetching hides memory latency improves the performance of three of the eight applications compared to applying either technique individually. We do observe, however, that using both techniques together to tolerate memory latency is not a good idea, and hurts performance through redundant overhead and high network traffic in most cases.

5. The best overall approach to hiding latency depends on factors such as predictability of memory access patterns and the extent to which lock stalls dominate synchronization time.

5.1 Future Directions

The goal of our research has been to investigate the effectiveness of multithreading at tolerating communication latency on software DSMs. In this section, we briefly discuss
how our results might be extended in future work.

As we have shown in Chapter 3, *locality* affects network traffic, which in turn determines the performance gain of multithreading on software DSM. To further increase spatial locality, we can use the compiler to determine the distribution of threads. For example, one could distribute the threads that *write* the same shared memory pages within the same *interval* on the same processor. This approach would minimize the "false sharing" and hence communication across machines with software DSMs.

In addition to examining page-based DSM systems, we can extend our study of multithreading to object-based DSM systems. Object-based DSM systems allow a per-object granularity of sharing and solve the false sharing problem of page-based DSMs. Since false sharing affects multithreading when we reduce the data size too small to accommodate more threads, higher performance can be achieved if we can eliminate the problem of false sharing.

Finally, the run time system can be further extended to support thread migration. Thread migration can help redistribute the work load by migrating threads from a heavily loaded machine to a lightly loaded one and hence improve load balancing. The challenge of exploring thread migration would be to build a lightweight protocol for migrating threads such that the communication requirement and overhead would not overwhelm the benefit.
Bibliography


