Design and Implementation of an Efficient and Scalable Software Distributed Shared Memory System

Byung-Hyun Yu

A thesis submitted for the degree of

Doctor of Philosophy
at the University of Otago, Dunedin,
New Zealand.

7 December 2011
Abstract

This thesis presents the design and implementation of our novel hybrid software DSM system. We call our system hybrid home-based EAC (HHEAC) since the system implements our novel exclusive access consistency model (EAC) based on the hybrid protocol of the homeless and home-based protocols. HHEAC guarantees only that shared variables inside a critical section are up to date before the accesses. Other shared variables outside a critical section are guaranteed to be up to date after the next barrier synchronisation.

Our home-based DSM implementation is different from the previous implementations in that a home node does not receive any diffs from non-home nodes until the next barrier synchronisation. It is also different in that during a lock synchronisation required diffs are prefetched before the critical section, which reduces not only data traffic but also page faults inside the critical section.

We also present a diff integration technique that can further unnecessary data traffic during lock synchronisation. This technique is especially effective in reducing data traffic for migratory applications.

Finally, we develop a home migration technique that solves the wrong home assignment problem in the home-based protocol. Our technique is different from others in that an optimum home node is decided before updating a home node.

To evaluate our system, we performed various experiments with well-known benchmark applications, including a novel parallel neural network application. The performance evaluation shows that HHEAC is more scalable than
other DSM systems such as TreadMarks and removes the home assignment problem in the conventional home-based protocol.
Acknowledgements

Being a Ph.D student has not been easy for me. When I started a Ph.D, I was used to being passive in everything. Unless my environment forced me to do something, I was very reluctant to do anything actively and passionately. During Ph.D study, I learned that does not work and gave me more troubles. To survive, I learned that I have to push myself to achieve my goal.

I also learned that even though there are many different views on how to see an object or event, it is possible that those different views can all be legitimate and not contradict with one another. So, through the experiences and interactions with others, I learned to see other views that I could not see before. Now I know that the ability to see things more objectively is somehow a very important skill that I have to acquire for the rest of my life.

There are many people that helped me to see this lesson during this difficult time. First of all, I would like to thank my three supervisors: Paul Werstein, Martin Purvis and Stephen Cranefield. I am indebted to them for the completion of Ph.D in many ways.

Paul Werstein became my new supervisor when I needed a new supervisor. Since then, he has helped me to complete my Ph.D study as the primary supervisor. It has been a pleasure for me under his supervision.

Martin Purvis has supported me with his research grant for many years. Without the funding, it would be much more difficult to finish my study. He always generously responds to my requests.

Stephen Cranefield has given me many helpful suggestions, including endless proof reading whenever I published a paper. I will not forget his kind
consideration towards me. I will remember him as a very empathic man.

Zhiyi Huang is the one who led me into this DSM research area. I would like to express my gratitude that he employed me as his research fellow in 2002 and granted me further financial support throughout my study.

I would like to thank Mark Pethick, a MSc student in the Computer Science department, for taking care of the network, his contribution to my Ph.D work such as PNN coding, and finally valuable comments after the proof reading.

Finally, I would like to thank my wife, Yu-Jeong, who took care of me and our two children, Sam and Hannah, throughout my Ph.D study. During my study, Sam and Hannah were born. Sam is now attending the Maori hill Primary School and Hannah is two and a half year’s old. Our two children gave me heavenly joy throughout my study.
# Contents

1 Introduction ............................................. 1
   1.1 Cluster Computing .................................. 1
   1.2 Distributed Shared Memory .......................... 3
   1.3 Challenges and Research Objectives .................. 4
   1.4 Contributions ...................................... 7
   1.5 Overview .......................................... 9

2 Evolution of Software Distributed Shared Memory ......... 10
   2.1 Introduction ....................................... 10
   2.2 Memory Consistency Models ......................... 12
      2.2.1 Sequential Consistency ......................... 13
      2.2.2 Processor Consistency .......................... 14
      2.2.3 Weak Consistency ................................ 16
      2.2.4 Release Consistency ............................. 18
      2.2.5 Entry Consistency ............................... 22
      2.2.6 Scope Consistency ............................... 23
   2.3 Memory Coherence Protocols ......................... 26
      2.3.1 Cache Coherence Protocol ....................... 27
      2.3.2 Coherence Protocols used in Software DSM Systems .... 28
      2.3.3 Homeless versus Home-based Coherence Protocols .... 29
   2.4 Performance Improvement Techniques .................. 29
      2.4.1 Time Window .................................... 29
      2.4.2 Multiple Writer Protocol and Diffing ............ 30
      2.4.3 Prefetching ..................................... 30
      2.4.4 Adaptation Protocol ............................. 31
      2.4.5 Multithreading DSM system ....................... 32
   2.5 Conclusions ........................................ 32

3 Development of a Software Page-based DSM System ....... 33
   3.1 Memory Consistency Model ........................... 34
   3.2 The Size of a Memory Unit ........................... 37
   3.3 Maintaining Coherent Memory ......................... 38
      3.3.1 Is accessed data stale? ......................... 38
      3.3.2 Where is the new data? ........................ 39
      3.3.3 How does a system preserve modifications of data? .... 40
   3.4 Page Fault Handling ................................. 40
3.4.1 Page State Transition ............................................. 41
3.5 Synchronisation Primitives ........................................... 41
3.5.1 Lock Synchronisation .............................................. 42
3.5.2 Barrier Synchronisation ............................................ 43
3.6 Data Packet Handling ................................................. 43

4 Homeless and Home-based LRC Protocols .................................. 45
4.1 Homeless Protocol .................................................... 46
4.1.1 Protocol Design .................................................... 46
4.1.2 Protocol Implementation .......................................... 47
4.1.3 Protocol Analysis .................................................. 48
4.2 Home-based Protocol .................................................. 51
4.2.1 Protocol Design .................................................... 51
4.2.2 Protocol Implementation .......................................... 52
4.2.3 Protocol Analysis .................................................. 53
4.3 Performance Evaluation ................................................ 54
4.3.1 Benchmark Applications ......................................... 54
4.3.2 Environment for Performance Evaluation ......................... 56
4.3.3 Overall Performance Results ..................................... 57
4.3.4 Hot Spot ............................................................ 58
4.3.5 Garbage Collection ................................................. 61
4.3.6 Diff Accumulation .................................................. 64
4.3.7 Home Effect ........................................................ 66
4.3.8 Data Communication Traffic ...................................... 67
4.3.9 Limitation of the Performance Analysis ......................... 68
4.4 Conclusions ............................................................. 68

5 Hybrid Home-based EAC Protocol .......................................... 70
5.1 Introduction ............................................................. 70
5.2 Protocol Design ........................................................ 71
5.2.1 Exclusive Access Memory Consistency Model ................... 71
5.2.2 Memory Coherence Protocol ...................................... 75
5.3 Implementation ........................................................ 77
5.3.1 Main Data Structures .............................................. 77
5.3.2 Page Fault Handling ............................................... 81
5.3.3 Lock and Barrier ................................................... 84
5.3.4 Home Update ....................................................... 89
5.3.5 Packet Formats ..................................................... 90
5.4 Programming Model ................................................... 92
5.5 Expected Benefits and Protocol Overheads ........................... 95
5.6 Related Work ........................................................... 98

6 Performance Improvement Techniques ...................................... 101
6.1 Introduction ............................................................. 101
6.2 Diff Integration ........................................................ 102
6.2.1 Implementation ..................................................... 106
List of Tables

4.1 Problem Sizes and Sequential Execution Times ............................. 57
4.2 Comparison of Overall Speed-ups between HL and HB ..................... 59
4.3 Garbage Collection Effect for Barnes-Hut between TM and HB ........... 64
4.4 Comparison of Data Communication Traffic between HL and HB .......... 66

5.1 Comparison of DSM Systems .................................................. 100

7.1 Basic Network Costs .......................................................... 114
7.2 Basic System Operation Costs between TM, HLRC and HHEAC .......... 114
7.3 Problem Sizes, Iterations and Sequential Execution Times (secs.) ....... 115
7.4 Speed-ups between TM, HLRC and HHEAC ................................. 116
7.5 Data Communication Traffic between TM, HLRC and HHEAC ............ 118
7.6 Page Faults, Twinning and Diffing between TM, HLRC and HHEAC ...... 121
7.7 Statistical data for PNN over CHScC (CHB) and HHEAC (HHE) .......... 125
7.8 Performance Results with 24 nodes for HLRC and HHEAC ................. 127
7.9 Speed-up Improvements between 16 and 32 Nodes ......................... 136
# List of Figures

1.1 Trend of Supercomputer Architectures ........................................... 2  
1.2 Distributed Shared Memory ................................................................. 3  
2.1 Synchronisation under SC and WC ...................................................... 17  
2.2 Data Communication under Release Consistency .................................. 19  
2.3 Data Communication under Entry Consistency .................................... 23  
2.4 Difference between RC and ScC ............................................................ 25  
4.1 ERC versus LRC ..................................................................................... 47  
4.2 Vector Timestamps in TreadMarks ....................................................... 49  
4.3 Hot Spot in the Homeless Protocol ...................................................... 50  
4.4 Comparison between Homeless and Home-based Protocols .................. 52  
4.5 Our Network Configuration .................................................................... 58  
4.6 Comparison of Time Breakdown for NN between TM and HB ................ 61  
4.7 Time Breakdown for Neural Network in 32 nodes over TreadMarks .......... 62  
4.8 Time Breakdown for NN in 32 nodes over the HB DSM System .............. 62  
4.9 Comparison of the Number of Resent Request Packets in NN ................. 63  
4.10 Diff Accumulation in IS-B ................................................................. 64  
5.1 Memory Classification Snapshot between Two Consecutive Barriers ........ 74  
5.2 Difference between Previous HB and Our HH Implementations ............ 78  
5.3 NCS diffs and CS diffs ........................................................................... 79  
5.4 diff_info Data Structure (C programming language) ............................. 80  
5.5 Example of Diff Storage ........................................................................ 81  
5.6 Page State Transition Diagram ............................................................... 83  
5.7 Illustration of the Lock Synchronisation Process in HHEAC ................. 86  
5.8 Barrier Insertion Example ................................................................. 93  
5.9 Nested Lock Example .......................................................................... 93  
6.1 Diff Integration ...................................................................................... 104  
6.2 Incorrect Diff Integration ...................................................................... 105  
6.3 New Code preventing Bad Diff Integration ....................................... 106  
6.4 Pseudo Code for Diff Integration in Lock Acquirer .............................. 107  
6.5 Pseudo Code for Diff Integration in Lock Releaser .............................. 108  
6.6 Home Migration Protocol .................................................................... 111  
7.1 Speed-ups over 4 Nodes ........................................................................ 117  
7.2 Speed-ups over 8 Nodes ........................................................................ 117
7.3 Speed-ups over 16 Nodes ......................................................... 117
7.4 Speed-ups over 32 Nodes ......................................................... 117
7.5 Data Traffic over 16 Nodes ....................................................... 119
7.6 Data Traffic over 32 Nodes ....................................................... 119
7.7 Number of Messages over 16 Nodes .......................................... 120
7.8 Number of Messages over 32 Nodes .......................................... 120
7.9 Number of Messages in Barnes-Hut over 16 and 32 Nodes .......... 120
7.10 The Number of Total Page Faults over 32 Nodes ...................... 122
7.11 The Number of Remote Page Faults over 32 Nodes ................. 122
7.12 The Number of Twins created over 32 Nodes ......................... 123
7.13 The Number of Diffs created over 32 Nodes ......................... 123
7.14 Hybrid Home-based Protocol Effect on PNN’s Speedup ............ 124
7.15 The Number of Messages created in PNN over CHScC and HHEAC 125
7.16 Data Traffic created in PNN over CHScC and HHEAC ............... 125
7.17 Number of Total Page Faults in PNN over CHScC and HHEAC .... 126
7.18 Number of Remote Page Faults in PNN over CHScC and HHEAC ... 126
7.19 The Number of Diffs created in PNN over CHScC and HHEAC .... 126
7.20 The Number of Twins created in PNN over CHScC and HHEAC .... 126
7.21 Time Measurement in PNN ...................................................... 127
7.22 Lock Synchronisation and CS Times in PNN over TreadMarks ... 128
7.23 Lock Synchronisation and CS Times in PNN over HLRC ............ 129
7.24 Lock Synchronisation and CS Times in PNN over HHEAC ......... 129
7.25 Main Code of the Simple Fine-grained Lock Application .......... 130
7.26 Diff Integration Effect on PNN ................................................. 131
7.27 Home Migration Effect on SOR and Barnes-Hut ...................... 133
7.28 Home Migration Effect on IS-B .............................................. 133
7.29 Improvements by the Initial Dynamic Home Migration Technique .. 134
7.30 Exclusive Read Write Effect .................................................. 135
7.31 HHEAC versus LAM/MPI ...................................................... 137
Chapter 1

Introduction

1.1 Cluster Computing

Currently, nearly 60% of the world’s powerful top 500 supercomputers are based on a cluster architecture. According to the statistics from the TOP500 organisation, this trend of cluster-based systems will continue in the supercomputer arena. As can be seen in Figure 1.1, cluster-based supercomputers (upper right area of the figure) were not among the 500 most powerful supercomputers until 1998. From 1999, the number of cluster-based supercomputers has increased significantly. The popularity of cluster-based systems is not surprising since they are cost effective and versatile compared with other supercomputer architectures such as Massively Parallel Processing (MPP) systems or Vector Computers. Using so-called COTS (commodity off-the-shelf) hardware and freely available software, people can now produce high performance computing power with less cost and more flexibility as can be seen in many Beowulf-style clusters (Sterling, Savarese, Becker, Dorband, Ranawake and Packer, 1995; Ridge, Becker, Merkey and Sterling, 1997) or Network of Workstations (NOW) (Anderson, Culler, Patterson, and the NOW team, 1995; Culler, Arpaci-Dusseau, Arpaci-Dusseau, Chun, Lumetta, Mainwaring, Martin, Yoshikawa and Wong, 1997).

Though high performance computing power can be obtained with cluster-based systems, exploiting the power is another matter. Parallel computing is the way to exploit this power. Parallel computing divides a single task into many subtasks which can be solved in parallel by many nodes at the same time.

To solve a problem in parallel with physically separated nodes, two different paradigms can be used: message passing or shared memory. With message passing, nodes

\footnote{The data is obtained from http://www.top500.org November, 2004}
send or receive results of subtasks obtained from other nodes explicitly. A programmer controls all the data traffic movement such as the time of communication, which data is to be sent or received and which nodes are involved in data communication. Since a programmer can exploit this freedom of flexible data communication in message passing, this improves the performance of parallel applications by optimizing the data traffic. On the other hand, programming with message passing can be a time-consuming and cumbersome task, especially if an algorithm involves frequent complicated fine-grained data transfer of complex data structures (Lu, Dwarkadas, Cox and Zwaenepoel, 1997).

On the other hand, parallel programming via shared memory is very close to multithreaded programming in an uniprocessor system. Most programmers are familiar with multithreaded programming and do not worry about explicit data communication between threads and main memory. With the abstraction of shared memory, they implement a parallel algorithm without worrying about the low-level data transfer that is needed to ensure memory consistency, although they are concerned with synchronised data access on shared variables to prevent data races. If a parallel algorithm involves complicated fine-grained data transfer such as multiple levels of a pointer or multi-dimensional array, it is much easier to implement the algorithm with shared memory than with message passing.
Our research objective is to provide an efficient and scalable shared memory system over cluster-based architectures. Since nodes in a cluster system have their own local memories, efficient provision of one logical coherent shared memory across all nodes is more important compared to bus-based multiprocessing systems due to the high latency and lower bandwidth in cluster-based systems. A Distributed Shared Memory (DSM) system provides an illusion of a single address space in physically dispersed nodes which have their own local memories. However, as can be seen in Figure 1.2, all nodes are not required to have all the most up-to-date memory since they do not need it all at a given point in the execution. This relaxed memory view is possible with weaker memory consistency models, which will be explained in Chapter 2. For example, in Figure 1.2, all nodes actually have the blue memory which is replicated in all nodes. On the other hand, all the other colours, except green which is replicated in two nodes (Node 0 and 31), reside only in one node. A DSM system somehow preserves the most up-to-date shared memory as illustrated in the single logical view of DSM in Figure 1.2.

In a uniprocessor system, a cache is used to reduce memory access latency between main memory and the processor. The rationale of using a cache is simple: memory access through a cache is faster than through main memory. However this simple
idea of using caches in a multiprocessor system produces a so-called cache coherence problem because the same datum can be held in different caches at the same time. Therefore, when a datum has changed in a node, the same datum cached in other nodes may have different values than the one that has just changed. Even though this same datum, held in many different caches, can hide memory access latency, it can cause this coherence problem in multiprocessor systems. To prevent the coherence problem, a cache coherence protocol is needed. A cache coherence protocol ensures that all shared data, even if it is replicated in many nodes, has a single logical value, which should represent the most up-to-date value.

The classical cache coherence problem in a multiprocessor system remains in DSM, in which shared data is replicated in many nodes. The difference between multiprocessor systems and cluster-based software DSM systems lies in the memory system architecture. In a multiprocessor system such as Symmetric Multiprocessor (SMP) or cache coherent Non Uniform Memory Access (cc-NUMA), caches are connected by a bus in an SMP system, or a low latency interconnection bus in a cc-NUMA system. In a cluster-based software DSM system, the main local memory of each node has the DSM and communication between nodes is performed by standard network I/O functions using TCP or UDP.

An efficient cache coherence implementation in a cluster-based software DSM system is more difficult and challenging than in hardware DSM systems. There are two reasons. First, I/O speed is much slower because I/O is performed by costly network functions such as send and receive. High I/O latency and low bandwidth are typically present in a software cluster-based DSM system. Second, the size of the minimum memory unit is much larger. In a page-based software DSM system, the minimum memory unit is a virtual page.

### 1.3 Challenges and Research Objectives

As stated previously, our research objective is to provide an efficient and scalable page-based DSM system for shared memory parallel programming over a cluster-based supercomputer. A software DSM implementation using COTS hardware and freely available software has advantages such as cost-effective performance and flexible hardware scalability. On the other hand, apart from hardware scalability, developing an efficient and scalable DSM protocol is more challenging because the protocol should efficiently control the interaction between nodes, which is unpredictable.
By software scalability we mean an efficient protocol that exploits scalable hardware capability. An efficient scalable protocol in a software DSM system means that it produces not only less data traffic but also less load imbalance between nodes in order to prevent a bottleneck node while processing an application. It should be noted that preventing burst data traffic is important since the burst data traffic not only saturates a network capacity quickly but also causes frequent dropping of packets when the buffers fill due to the flood of data, requiring retransmission of the packets dropped.

The burst data traffic can happen more subtly and indirectly in a DSM system than in a message passing system since data communication in DSM is governed by a DSM system and memory access patterns of an application. Since a DSM system is designed for solving all kinds of problems that can produce many different memory access patterns, it can produce unexpected data traffic flows. The coherence-related data traffic that is required to maintain coherent memory throughout all nodes is the main contribution to hard-to-control data traffic. On the other hand, data communication in message passing is controlled by a programmer and can be optimized for any specific problem by the programmer.

Since all nodes should have a coherent view of shared memory through physically separated nodes, it is important for a DSM system to provide one coherent memory view efficiently throughout all nodes. Generally, a parallel algorithm implementation using software DSM has more data traffic and a higher message count than one using message passing, since it adds coherence-related data traffic in addition to application data traffic. Also the data traffic in message passing systems allows more coarse-grained transfer in contrast with the fine-grained data transfer in DSM systems (Lu et al., 1997).

Furthermore, page-based DSM systems produce more unnecessary data traffic and page faults due to the large granularity memory unit. The size of a page in Linux is 4 KB. Compared with the typical cache line size in a hardware DSM implementation, for example, 16 bytes in DASH (Lenoski, Laudon, Gharachorloo, Weber, Gupta, Hennessy, Horowitz and Lam, 1992), the large granularity memory unit can cause the problem of so-called false sharing more often. Since a memory system considers a page as a minimum unit, all data inside a page is treated as one unit. If only one byte of shared data inside a page (4096 bytes in Linux) is modified, all the data inside the page is considered as modified by the memory system, even though the other 4095 bytes were actually not modified. Therefore even if a node accesses one of the shared variables located in the unmodified 4095 bytes, which all contain the up-to-date values, a memory
coherence protocol regards the whole page as stale due to the modified one byte that has nothing to do with the intended memory access, and causes an unnecessary page fault.

False sharing not only causes an unnecessary page fault but also produces unnecessary data traffic in order to fetch the up-to-date copy of the faulting page. The false sharing problem can be more severe when we apply strict memory consistency models such as the sequential consistency model (Lamport, 1979), which will be explained in Chapter 2.

As stated earlier, scalability in a software DSM system is challenging. Scalability in DSM can be defined as a consistently improved speed-up as the number of nodes increases. However, unless an application is embarrassingly parallel, data communication traffic eventually hinders the scalability of DSM as the number of nodes increases. Therefore, it is very difficult to have an ideal speed-up in a scalable environment. In this thesis, we consider the scalability of software DSM systems to be its resistance to rapid slowdown as its size increases. The rapid degradation of DSM performance is certainly the major hindrance to DSM scalability.

There are many reasons why DSM applications show rapid performance slowdown. We claim that some of the reasons are load imbalance and synchronisation contention. Load imbalance occurs when any particular node has much more protocol processing load than other nodes. Load imbalance can happen due to inherent characteristics of a DSM protocol or inherent memory access patterns induced by a problem solving algorithm. In Chapter 4, we explain that the problem of load imbalance creates a hot spot node, which causes significant performance degradation.

Lock synchronisation contention happens when many nodes attempt to get into the same critical section at the same time. In software DSM systems, this is exacerbated since coherence information is exchanged at the same time. In particular, a migratory memory access pattern during synchronisation makes a DSM system very difficult to be scaled since coherence data tends to be accumulated as a sequential process of synchronisation continues. An efficient and scalable DSM protocol can be judged by the reduction of synchronisation contention.

Because of these challenges to developing an efficient and scalable software DSM system, our research objectives are set as follows:

1. Study the state of the art DSM protocols and find out their strengths and weaknesses, particularly in terms of scalability.
2. Based on the findings, choose the best DSM protocol or combine strengths of each protocol.

3. Develop performance improvement techniques for the chosen DSM protocol.

4. Provide a performance evaluation of different DSM protocols in order to improve DSM performance further.

### 1.4 Contributions

This thesis presents our contributions to the page-based software DSM research area. However, our contributions are not limited to that area. They can also apply, in a broad sense, to any protocols that are related to efficient and scalable shared memory implementations.

We summarise our contributions as follows:

1. An in-depth performance evaluation of and comparison between homeless and home-based protocols, which are two state of the art software DSM protocols that are used to maintain coherence in DSM systems (Yu, Huang, Cranefield and Purvis, 2004).

2. The design of a hybrid DSM protocol which combines the strengths of the home-based protocol and the efficient diff-based updating of the homeless protocol (Yu, Werstein, Purvis and Cranefield, 2005).

3. A dynamic home migration technique which solves the wrong home assignment problem in the home-based protocol (Yu, Werstein, Cranefield and Purvis, 2005).

4. A diff integration technique which solves the diff accumulation problem in migratory DSM applications (Yu, Werstein, Cranefield and Purvis, 2005).

The first contribution is to evaluate homeless and home-based protocols which are used in the state-of-the-art DSM implementations, TreadMarks from Rice University (Keleher, Dwarkadas, Cox and Zwaenepoel, 1994; Amza, Cox, Dwarkadas, Keleher, Lu, Rajamony, Yu and Zwaenepoel, 1996) and a home-based LRC (HLRC) implementation from Princeton University (Zhou, Iftode and Li, 1996). TreadMarks and HLRC are fundamentally different in how they maintain coherency in DSM. It is important to understand the differences, and find out the strengths and weaknesses of each protocol. We call the protocols of TreadMarks a homeless protocol and the other a home-based
protocol. In this thesis, we present an in-depth analysis and performance evaluation of both protocols in Chapter 4.

We found the two protocols have serious deficiencies in scalability even though the home-based protocol has more positive aspects. The homeless protocol is not scalable due to load imbalance, diff accumulation and the global garbage collection process required. The home-based protocol does not have the problems that the homeless protocol has, but it needs dynamic home assignment to be more scalable.

The second contribution is a new DSM protocol which combines the strengths of both the homeless and home-based protocols. The new protocol is called a hybrid home-based protocol and it further delays coherence-related data communication compared to the original home-based protocol. The hybrid home-based protocol essentially follows the coherence protocol used in the original home-based protocol but selectively uses the memory coherence protocol in the homeless protocol only inside a critical section. The hybrid home-based protocol exploits relaxed constraints on maintaining memory coherency provided by relaxed memory consistency models such as the entry (Bershad and Zekauskas, 1991) and scope (Iftode, Singh and Li, 1996) consistency models.

The third contribution is a technique to remove one of the weaknesses of the home-based protocol. Since the original home-based protocol did not provide an optimized home node allocation, it places the burden on application programmers of finding an optimized home allocation. With our dynamic home allocation technique, data locality is maximized by dynamically assigning the home node of a page to the node that most actively accesses that page.

The fourth contribution is also to add one of the strengths of the homeless protocol to the home-based protocol, which is a diff-based memory update. Upon a page fault, the diff-based memory update in the homeless protocol is more efficient than the page-based update in the home-based protocol since only the diff of a page is transferred. However, in some applications that show a migratory memory access pattern, diffs are accumulated so that the size of the accumulated diffs can exceed the size of a page. In the case of severe diff accumulation, it will degrade performance significantly since data communication traffic increases as diffs accumulate. Our diff integration technique reduces data communication traffic by removing the diff accumulation problem.

Our experiments and analysis demonstrate that these contributions improve DSM performance and scalability in many different ways. For example, data communication traffic and the number of messages would be reduced not only by dynamic home allocation and diff integration but also by the hybrid home-based protocol. In terms of
DSM scalability, the hybrid home-based protocol is less likely to produce a load imbalance compared to the homeless protocol. It also relieves synchronisation contention by making critical sections execute faster. Our contributions help get one step closer to a scalable and optimized DSM coherency protocol for a page-based software DSM system.

1.5 Overview

The rest of the thesis is structured as follows: Chapter 2 introduces an evolutionary history of software DSM. Then, it explains the important concepts that have been used in DSM systems in terms of memory consistency models and performance improvement techniques. It also explains why relaxed memory consistency models give more opportunity to implement an efficient DSM system. In Chapter 3, the considerations involved when designing and implementing a software DSM system are presented, such as memory consistency models, the size of a memory unit, page fault handling and synchronisation primitive implementations. In Chapter 4, two state-of-the-art DSM protocols — homeless and home-based protocols — are compared with each other in terms of performance and scalability. A performance evaluation of and comparison between the two protocols are presented using several benchmark DSM applications. In Chapter 5, the design and implementation of our novel “hybrid” home-based EaC protocol is presented. In Chapter 6, performance improvement techniques such as diff integration, dynamic home migration and the use of the exclusive_read_write page state are described. In Chapter 7, a performance evaluation of our system is presented. The effects of the HHEAC protocol and the three techniques on real DSM applications are discussed in detail. Finally, Chapter 8 presents the conclusions and areas for future research.
Chapter 2

Evolution of Software Distributed Shared Memory

2.1 Introduction

IVY (Li and Hudak, 1986; Li, 1988; Li and Hudak, 1989) is considered the first software implementation of DSM. The main contribution of IVY is to provide a shared memory platform using a loosely-coupled distributed memory system. When IVY was proposed, message passing was the predominant programming model on loosely-coupled distributed memory systems. The authors argued that parallel programming using message passing would be much more difficult when passing complex data structures is required. Also, they argued that process migration would be much easier when a parallel program is implemented with shared memory.

The implementation of software DSM in IVY gives us a starting point to review how software DSM systems have been evolved. As described in Chapter 1, it is very important to understand computer architectures for supercomputers. Software DSM systems are developed over a distributed memory architecture. In a distributed memory architecture, each node has its own memory. Communication between nodes is achieved by message passing. A software DSM system provides a virtual single address space over distributed memories by message passing.

The idea of a single address space over physically distributed memories in IVY comes from combining two concepts: the virtual memory used in a uniprocessor memory system and a memory coherence protocol used in multiprocessor systems. The purpose of virtual memory is to expand the hardware memory space by software. The mapping between virtual memory addresses and physical memory addresses, which is
used in a virtual memory system, was employed in IVY in order to implement the mapping between local memories and the shared memory address space. In IVY, when a page fault triggered by an invalid shared page access happens, the mapping manager of the faulting node actually retrieves the page from the memory of a remote node via a network.

In order to make distributed local memories coherent, IVY’s coherence policy follows the Sequential Consistency (SC) model proposed by Lamport (Lamport, 1979). Intuitively, the SC model gives us a natural extension of the uniprocessor memory model to a multiprocessor memory system. In the uniprocessor memory model, provided a memory system does not violate data dependency or control dependency, arbitrary program order is permitted in order to hide memory access latency. In a multiprocessor system, this relaxed arbitrary program order is severely restricted since each node also has to take other nodes into consideration. Because of this parallel processes, Lamport saw unexpected program behaviour caused by arbitrary program order, which would be no problem in a uniprocessor system. To correct this problem, he proposed the SC model. In an SC memory system, though many arbitrary total memory access orders (all interleavings of the nodes’ memory access sequences) are possible, any total order should be seen by each node as the same and the total order should be consistent with each node’s program order.

Though the SC model gives correct interleaving of arbitrary parallel memory accesses, its straightforward implementation on a memory system, in particular a cache-based system, should provide a completion mechanism for each local access regardless of whether it is read or write (Adve, Cox, Dwarkadas, Rajamony and Zwaenepoel, 1996). A completion mechanism can be explained in terms of memory access being “performed with respect to” every node (Dubois, Scheurich and Briggs, 1986, p.437). In the case of a read access, when the read access to a shared variable is performed with respect to every node, this means that the return value of the shared variable is globally known to every node. Similarly, when a write access to a shared variable is performed with respect to every node, this means that the written value of the shared variable is globally known to every node.

Even in a hardware-based DSM system, the requirement of the completion mechanism can degrade its performance because it prohibits any optimization techniques to hide memory latency such as pipelining or coalescing (Gharachorloo, Gupta and Hennessy, 1991). In a cluster-based system, the penalty for adopting SC for memory consistency is much more severe due to its high I/O latency.
The SC implementation of IVY basically uses a single writer protocol. In a single writer protocol, at any time only one node has the right to write to a particular page. Replicated pages in other nodes should be invalidated before the write proceeds. Therefore the size of a page is particularly important in order to achieve a balance between reducing false sharing and improving the prefetching effect. Even though IVY showed satisfactory performance in some applications which have relatively high computation-to-communication ratios, the single writer SC implementation of IVY cannot provide satisfactory performance with multiple writer or multiple reader applications in which multiple nodes access the same page at the same time. When multiple nodes access the same page at the same time, a so-called ping-pong effect can happen, since only one node has the right to write on the page exclusively. A ping-pong effect is a phenomenon where a false sharing page moves back and forth between multiple nodes without them finishing their accesses.

In order to remedy these problems in IVY, research in software DSM systems has aimed to provide more efficient DSM protocols. One of the important breakthroughs for the development of efficient software DSM protocols was obtained from exploiting relaxed memory consistency models. Even though an application programming model under relaxed memory consistency models becomes more complicated than a traditional sequential programming model, many memory access optimization techniques are enabled by exploiting relaxed models.

### 2.2 Memory Consistency Models

The problems of IVY are mainly due to adopting a strict memory consistency model such as Sequential Consistency. In addition, the large granularity of a memory unit aggravates the problem. As can be seen in IVY, even though SC provides correct parallel memory access orders, it is very inefficient to enforce SC in a cluster-based DSM system.

Traditionally, a memory consistency model defines correct access ordering in multiprocessors (Lamport, 1979). More relaxed memory models relax access ordering further, but without producing an incorrect program result. Relaxed access ordering was important because the use of memory latency hiding techniques such as pipelining and write buffering are made possible by relaxing constraints on correct memory access executions (Dubois et al., 1986; Gharachorloo, Lenoski, Laudon, Gibbons, Gupta and Hennessy, 1990; Adve and Hill, 1990b). To guarantee a correct program result under
more relaxed models, each model also proposes a shared memory programming model that must be followed in order to have a correct program execution under the memory system. In the next subsections, we briefly describe the major memory consistency models and explain why more relaxed memory models give more freedom to implement efficient DSM coherence protocols.

### 2.2.1 Sequential Consistency

The SC model was proposed by Lamport in 1979 (Lamport, 1979). The intention of the model was to provide correct memory access interleaving of each process's memory access in order to make a memory system execute as a programmer intended. Lamport defined *sequentially consistent* memory access order for multiprocess programs as follows:

A multiprocessor is said to be sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. (Lamport, 1979, p.690)

Significantly, the model pioneered the issue of correctness in multiprocess programs. In terms of performance, however, the model restricts many optimization techniques for memory latency reduction. The model could not exploit a shared memory programming practice in which most shared variables are not concurrently accessed except for some special-purpose variables. These special-purpose shared variables are called synchronisation variables because they are only used for synchronisation purposes. When SC is applied to a page-based software DSM system, as shown in IVY, there are many negative effects on DSM performance due to the large memory unit size and high memory access latency, which is typical in many software DSM systems.

Traditionally, SC is adopted in bus-based multiprocessors and hardware-based DSM systems (Delp, Sethi and Farber, 1988; Frank, Burkhardt and Rothnie, 1993). The snooping cache coherence protocol used in bus-based multiprocessors is a good example of enforcing SC. In the snooping cache coherence protocol, whenever a node performs a write memory access, a shared bus broadcasts the effect of the write memory access to all other nodes. Therefore all the write operations governed by the snooping cache coherence protocol appear to be serialized, which is the requirement of SC.
In hardware DSM systems that are more scalable than bus-based multiprocessors, a directory-based coherence protocol (Lenoski, Laudon, Gharachorloo, Weber, Gupta, Hennessy, Horowitz and Lam, 1992; Frank et al., 1993) or serialized memory access network facility such as a token ring (Delp et al., 1988), can be used to enforce SC. Also, optimized SC implementations in hardware DSM systems have been proposed using many different means such as hardware prefetching and speculative loads (Pai, Ranganathan, Adve and Harton, 1996), or allowing more memory access overlapping at the time of a write miss (Adve and Hill, 1990a). Enforcing SC in these systems does not cause much adverse effect on performance due to low communication latency and the fine-grained memory unit size compared to enforcing SC in software-based DSM systems.

SC has been adopted in many software DSM systems also, even though SC has more negative effects on software DSM systems. To alleviate the negative effects of SC under the software DSM environment, many new techniques have been proposed. For example, a fine-grained memory granularity was emulated under a page-based DSM system (Itzkovitz and Schuster, 1999; Iosevich and Schuster, 2004) and more optimized SC implementations have been developed to alleviate the false sharing problem under the software DSM environment (Fleisch and Popek, 1989; Kelcher, 1996). Also, SC has been employed for object-based DSM systems such as Blizzard (Schoinas, Falsafi, Lebeck, Reinhardt, Larus and Wood, 1994) and Shasta (Scales, Gharachorloo and Thekkath, 1996). Since object-based DSM systems use an object as a memory unit, the false sharing problem can be less severe even though SC is chosen as their memory consistency model.

2.2.2 Processor Consistency

A more relaxed memory consistency model than Sequential Consistency was proposed by Goodman and is called Processor Consistency (PC) (Goodman, 1989). The PC model was implemented in DASH (Gharachorloo et al., 1990) and was explained further in order to clarify the original definition of PC specified by Goodman (Ahamad, Bazzi, John, Kohli and Neiger, 1993).

The cost of implementing a memory system that satisfies SC is the need to enforce a completion mechanism for each access regardless of whether it is a read or write. This makes the memory latency problem worse, since each access has to be executed one at a time. A processor’s stall time increases due to the serial memory access, which also requires each access to be completed with respect to other nodes. Prefetching
or pipelining memory access optimization is prohibited under SC. PC was proposed to allow read prefetching without the previous write accesses to be completed, which reduces memory latency of a read access. Goodman defined PC as follows:

A multiprocessor is said to be processor consistent if the result of any execution is the same as if the operations of each individual processor appear in the sequential order specified by its program. (Goodman, 1989, p.2)

Compared to SC, PC allows one memory access reordering (write → read) out of four possible memory accesses (write → write, write → read, read → write and read → read) (Adve and Gharachorloo, 1996; Adve, Pai and Ranganathan, 1999). The total memory access ordering constrained by PC is now more relaxed than SC. In SC, all nodes should agree on some arbitrary total access order, but in PC, all nodes should agree on only the write access order issuing from each node. Therefore, only a completion mechanism for a write access following a write access is needed to enforce PC.

As Goodman suggested, PC can be used to relax SC’s strict memory access completion policy which requires an acknowledgement of each write access before advancing to the next read access (Goodman, 1989). Under SC’s strict memory access completion policy, a read access is blocked until the previous write access is acknowledged. However, a PC memory system relaxes this memory access order so that a read access can be overlapped with the previous write access. This relaxed memory access reordering can reduce memory access latency relating to a read access.

The programming model in PC is similar to SC in that explicit synchronisation primitives are not needed in application programs. Many commercial multiprocessors adopt PC rather than SC in order to reduce memory access latency. As Goodman indicated, PC is considered a bridge between strong ordering and weak ordering (discussed in the next section). The performance of PC was evaluated in the DASH system (Gharachorloo et al., 1991). In it, PC showed comparable performance with weak consistency in three applications tested. The authors concluded that a read access that bypasses the previous write, which is allowed under PC, is more important for memory access relaxation than write access pipelining allowed under weak consistency.

Pipelined RAM (PRAM), which is a similar memory consistency model to PC, was proposed by Lipton and Sandberg in 1988. The requirement of correct memory access ordering under PRAM is defined as follows:

The writes of a single processor should be seen in the order in which they were invoked at all other processors. (Lipton and Sandberg, 1988)
The difference between PC and PRAM is not clear though. Ahamad et al. (1993) argued that PC should meet the requirements of PRAM and also cache consistency in which the order of writes to every shared variable should be seen as the same in all nodes.

### 2.2.3 Weak Consistency

Initiated by SC which provides correct memory access ordering, PC primarily concentrate on possible memory access reordering in order to reduce memory access latency by pipelining memory access operations. This direction is very relevant to hardware DSM systems. However, in the environment of software DSM systems, memory access pipelining is not as effective as in hardware DSM systems due to high memory access latency (Carter, Bennett and Zwaenepoel, 1995).

A major breakthrough, which further reduces the high memory latency and false sharing problem in a software DSM system, actually came from a more relaxed memory access buffering in multiprocessors proposed by Dubois et al. (Dubois et al., 1986; Dubois and Scheurich, 1990). By buffering and merging memory accesses, the number of messages can be significantly reduced, which is very effective in a software DSM system.

The authors found that most shared variables are not concurrently accessed while parallel processes are executed, except for some special shared variables. They divided shared variables into two groups: a normal shared variable which does not control concurrent execution and a special shared variable which controls program order in order to implement the synchronisation of parallel processes. The purpose of the special shared variable is to make sure that parallel processes are executed as determined by a programmer’s intention. This is why the variable is called a synchronising variable. More explicit instructions to control program behaviour are required by a programmer by placing the synchronisation points explicitly.

Dubois et al. (1986) distinguished between strong ordering and weak ordering of memory access events. The strong ordering of events is another interpretation of SC in terms of legal memory access interleaving. The authors defined strong ordering of memory accesses as the legal interleaving of accesses obeying SC. In practice, it means each node executes memory accesses without any reordering and write memory accesses must appear to be visible to all the other nodes at the same time, which is called write atomicity. Because of the latter requirement of guaranteeing strong ordering, each write access should be completed with respect to all the other nodes. The main point
in implementing a strong ordering memory system is to make each memory access appear to be atomic. However this atomic property requires a complicated completion protocol, in particular in cache-based memory systems.

On the other hand, weak ordering relaxes the serial write access completion constraint in strong ordering. Weak ordering of events are defined as follows:

In a multiprocessor system, storage accesses are weakly ordered if

1. accesses to global synchronising variables are strongly ordered,
2. no access to a synchronising variable is issued in a processor before all previous global data accesses have been performed,
3. no access to global data is issued by a processor before a previous access to a synchronising variable has been performed. (Dubois et al., 1986, p.439)

The weak consistency (WC) model allows weak ordering of events, which relaxes the atomic requirement of each write memory access. Under WC, only memory accesses to synchronisation variables are strongly ordered. This means that memory accesses between synchronisation variables are performed without waiting for previous memory accesses to shared data variables, thereby enabling pipelining of accesses. A WC memory system, however, should implement properly a fence operation which blocks the next memory access after a synchronisation access until all outstanding accesses are acknowledged. Also, under WC a programmer should identify synchronisation points explicitly in order to exploit WC.

<table>
<thead>
<tr>
<th>node 1</th>
<th>node 2</th>
<th>node 1</th>
<th>node 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 1; \quad y = 2; \quad A = 1;$</td>
<td>$x = 1; \quad y = 2; \quad A = 1;$</td>
<td>$x = 1; \quad x = x + 1; \quad y = y + 1; \quad \text{barrier}(A);$</td>
<td>$x = 1; \quad x = x + 1; \quad y = y + 1; \quad \text{barrier}(A);$</td>
</tr>
</tbody>
</table>

(a) Implicit Synchronization under SC    (b) Explicit Synchronization under WC

Figure 2.1: Synchronisation under SC and WC
The programming model under WC is different from previous memory consistency models in that synchronisation variables should be declared explicitly. For example, Figure 2.1 illustrates the difference between synchronisation under SC and WC. Under SC, a shared variable A is used to synchronise two processes implicitly. Under WC, synchronisation primitives such as fence or barrier would be provided to do the same job but more explicitly. If we look at (b) in Figure 2.1 closely, it can be seen that writes of x and y in node 1 do not have to be seen by node 2 until node 2 executes the barrier synchronisation. This means program order in node 1 can be any arbitrary order until the barrier. A WC memory system can exploit this freedom which is obtained from a contract between a programmer and a memory system developer. For example, node 1 can use pipelining of x and y under WC. Under SC, completion processes for x and y should be performed one at a time. In a software DSM system in which memory access latency is high, it would be more beneficial to delay updates until synchronisation rather than use pipelining techniques, as used in a hardware DSM system, in order to hide memory access latency. For example, two writes of x and y in node 1 in (b) of Figure 2.1 can be updated with one message rather than two messages.

2.2.4 Release Consistency

To further relax the memory consistency constraints of WC, the release consistency (RC) model was proposed (Gharachorloo et al., 1990). In RC, a synchronisation operation is divided further into two distinctive synchronisation operations: acquire and release. This finer categorization of synchronisation was articulated in order to provide more memory access overlap, especially for applications that use frequent critical sections. Different critical sections can be executed concurrently if there is no local data or control dependency between the critical sections. This concurrent execution of critical sections is not allowed under WC, in which each critical section must be completed before the next critical section starts. The legal order of memory accesses according to RC is as follows:

1. before an ordinary LOAD or STORE access is allowed to be performed with respect to any other processor, all previous acquire accesses must be performed, and

2. before a release access is allowed to be performed with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed, and
3. special accesses are processor consistent with respect to one another.

(Gharachorloo et al., 1990, p.19)

Normally, synchronisation requires communication between nodes in a DSM system. In RC, at the time of lock acquire and release memory accesses, data communication between two nodes occurs as shown in Figure 2.2. Figure 2.2 also illustrates how the RC memory constraints affect the coherence implementation of RC. In Figure 2.2, acq(N) and rel(N) mean acquire and release of lock number N respectively. According to the first legal ordering condition for satisfying RC, an ordinary memory access A (an arrow marked with red colour) in node 0 should see all previous acquire accesses, which are acq(2) and acq(1). Due to the third condition of RC, rel(2) in node 1 should be performed before acq(2) in node 0. Also at the time of rel(2) in node 1, all previous ordinary LOAD and STORE accesses must be performed, which are marked with the red arrow in node 1. Subsequently, rel(2) in node 2 and previous accesses before rel(2) marked with the red arrow in node 2 should be visible at the time of a memory access A in node 0. Similarly, all the memory accesses marked with the blue arrows should be visible at the time of a memory access B.

In RC, for example, values of the previous write accesses indicated by a blue colour in nodes 0 and 1 are not required to be seen in node 2 until a program execution in node 2 reaches acq(1) before B. The relaxation of write atomicity provided by RC can be exploited in software DSM coherency protocol implementations. For example reduction of the number of messages, thanks to relaxed write atomicity, is more effective in improving the performance of applications in software DSM systems than the relaxed access event ordering that allows pipelining memory accesses in hardware DSM systems.

Figure 2.2: Data Communication under Release Consistency
The programming model under RC is very similar to WC. With WC and RC, a programmer should explicitly identify synchronisation points between parallel processes. In addition to the requirements of WC, a programmer should identify mutual exclusive executions guarded by acquire and release to exploit RC. An acquire-release pair forms a critical section which only one process should be executing at a time.

**Eager Release Consistency**

Originally, RC was proposed to make efficient memory access pipelining possible in a scalable hardware DSM system called DASH (Lenoski, Laudon, Gharachorloo, Weber, Gupta, Hennessy, Horowitz and Lam, 1992; Lenoski, Laudon, Joe, Nakahira, Stevens, Gupta and Hennessy, 1992) at Stanford University. The main contribution of RC in DASH is to enable pipeline memory accesses without an access completion protocol, which minimizes memory latency. After DASH, RC was first used in a software DSM system called Munin (Carter, Bennett and Zwaenepoel, 1991; Carter et al., 1995). Munin exploited RC in order to reduce the number of messages rather than to enable the pipelining of memory accesses as in DASH, since Munin was implemented on a network of workstations in which the overhead of sending messages is very costly. The implementation of a coherence protocol in Munin was greatly optimized by adopting RC. The memory coherence conditions for RC allow Munin to use write buffering, to merge updates, and to send them in a single message at the time of lock release. This delay and merging update property is very effective in reducing the number of messages as well as data traffic.

In Munin, a distributed directory-based cache coherency protocol (Tang, 1976; Censier and Feautrier, 1978; Lenoski, Laudon, Gharachorloo, Gupta and Hennessy, 1990) was used to implement RC. Each node manages its own directory containing independent state information of shared objects such as the states of objects and their replication information. When a coherency action is required at a release time, the directory is contacted in order to maintain the coherent memory. The directory provides the states of the shared objects that have been written to and information about nodes which have the replicated objects. An inefficient coherence action can occur at this time. For example, some of the nodes owning the replicated objects actually may not use the data updated in the last releasing node. Referring to Figure 2.2, for example, at the time of rel(1) in node 0 if node 1 and 2 have the replicated objects, node 0 has to update the objects in both nodes even though only node 1 actually needs them at that time.
Lazy Release Consistency

The lazy release consistency (LRC) protocol (Keleher, Cox and Zwaenepoel, 1992) was developed by Keleher to remove this unnecessary data transfer. Keleher found that data communication during lock synchronisation is one-to-one between a current lock owner and a next lock owner. Therefore, the directory-based protocol used in Munin, though it is relatively simple to implement and may benefit from the prefetch effect, causes unnecessary data traffic that is sent from a releasing node to all nodes having the replicated data. The LRC protocol used in TreadMarks (Keleher et al., 1994) sends coherency information only to the next lock owner. “Lazy” means that previous write updates become visible as lazily as possible, which is the time that the next lock owner has an access fault which is then followed by fetching the updates from appropriate nodes.

The lazy update requires a complicated write record scheme since all writes are not visible at the time of release to all the nodes. This lazy update in TreadMarks contrasts with the eager update in Munin. This is why Munin’s memory model is called “Eager Release Consistency”. To implement this laziness, TreadMarks uses an “interval” that contains all the write information that occurred between two synchronisation points. Each interval is identified with a vector timestamp (VTS) (Mattern, 1989) which extends Lamport’s logical time (Lamport, 1978). The position of the event number in a VTS corresponds to the node’s ID number. It represents an event in that node. The number increases by one whenever an event happens in the corresponding node. For example, if an interval’s VTS is \(\{4, 2, 2, 3\}\) at node 0, node 0 has performed four events, node 1 two events, node 2 two events and finally node 3 three events. Also this means that the owner of the interval, which is node 0, has knowledge about the memory updates up to events 4 in node 0, 2 in node 1, 2 in node 2 and 3 in node 3.

In TreadMarks, an interval includes the dirty page numbers which have been written in the interval period. An interval starts from a synchronisation point, normally a lock release or barrier. Sometimes creation of an interval starts from a lock acquire if concurrent writes on the same page by the acquiring node and other nodes are detected at the acquire process. This means that if the acquiring node and other nodes concurrently write on the same page, the concurrent writes occurred in the acquiring node should be preserved before they are invalidated according to LRC. An interval ends at a release or barrier. At the end of an interval, all the required information is stored in an interval such as dirty pages and VTS. By comparing intervals’ VTSs and their dirty pages, the happens-before partial order (Lamport, 1978) of memory accesses
is established to implement the conditions for LRC. The happens-before partial order defines the invariant partial ordering of events in a distributed system based on the causal relationships that exist between pairs of events:

- If events $a$ and $b$ occur on the same process, we say $a$ happened-before $b$ if the occurrence of event $a$ preceded the occurrence of event $b$, and

- If event $a$ is the sending of a message and event $b$ is the reception of the message sent in event $a$, we say $a$ happened-before $b$.

### 2.2.5 Entry Consistency

As described in the discussion of previous memory consistency models, how the relaxed memory consistency conditions are exploited differs between hardware and software DSM memory systems. In hardware DSM systems, it is more about relaxing event ordering to allow out-of-order or pipelining of memory accesses, which is effective in hiding memory latency in a hardware DSM system. In a software DSM system, it is more about delaying and merging events due to the high cost of data communication. In other words, write information in a node is not required to be visible to other nodes until they are actually needed, normally around synchronisation points.

Entry consistency (EC) (Bershad and Zekauskas, 1991) was proposed in order to delay memory updates further than in RC. In EC, write accesses outside a critical section are not required to be visible to other nodes until the next barrier synchronisation. A programmer under EC should provide a memory system with the relationship between a synchronisation variable and the data protected by the synchronisation variable. The development of EC was motivated by the behaviour of shared memory program execution inside a critical section, in which program execution is mutually exclusive and most communication occurs. In most DSM programs, up-to-date values of shared variables are required only in a critical section. Also, the number of shared variables accessed in a critical section is relatively small. Furthermore, all nodes execute similar memory accesses within the same critical section guarded by the same lock.

Figure 2.3 illustrates data communication under EC. Data communication under EC is driven by lock ownership transfer, in which coherency information is also transferred. The coherency information under EC is transferred only to the next lock owner with only the data that was updated in the same critical section previously. This delayed selective data update reduces the number of messages and data traffic. Note that
In Figure 2.3, all delayed memory access updates must be visible to all nodes after the barrier.

Midway is a software DSM system that implements EC (Bershad and Zekauskas, 1991; Bershad, Zekauskas and Sawdon, 1993). It is an object-based DSM system, not a page-based DSM system such as Munin or TreadMarks. In an object-based DSM system, the granularity of the memory unit is the size of an object, which is more advantageous to the reduction of false sharing. Its implementation techniques are quite different from ones used in a page-based DSM system. For example, compiler support is required to detect a write on shared memory. In a page-based DSM system, detection of writes on shared variables is performed by the virtual memory system.

2.2.6 Scope Consistency

The programming model under EC is quite different from the previous models. First, a programmer under EC must provide the relationship between a synchronisation variable and its data. That is, all shared data must be explicitly associated with at least one synchronisation variable, which guards data integrity. Also, some programs developed under previous memory consistency models are not correctly executed under EC due to the more relaxed memory consistency constraints of EC. Scope consistency (ScC) (Iftode et al., 1996) was proposed to relieve the first difference in EC. In ScC, the binding between a synchronisation variable and its data is automatic, thereby reducing a programmer’s burden.
The first DSM system to implement ScC was based on the Shrimp system at Princeton University (Blumrich, Li, Alpert, Dubnicki, Felten and Sandberg, 1994). The Princeton DSM system uses the automatic update facility which is provided by the virtual memory-mapped network interface in the Shrimp system (Iftode, Blumrich, Dubnicki, Oppenheimer, Singh and Li, 1999). Automatic update provides automatic propagation of local writes on shared memory to other remote memory locations. Automatic update eliminates write detection and write collection which are costly in a software DSM system. Thus, strictly speaking, the first ScC DSM system was not an all-software DSM system. Automatic update gave the opportunity to develop a home-based DSM protocol which is explained further in Section 4.2. Later, several all-software ScC-based DSM systems were implemented (Hu, Shi and Tang, 1999b; Speight and Bennett, 1997; Lee, Yun and Lee, 2000).

The basic idea of ScC is to assign shared variables into scopes. Under the contract with the programmer underlying ScC, whenever a node enters into a scope, only the shared variables in that scope are guaranteed to be up-to-date. There are two different scopes defined in ScC. One is a local scope and the other is a global scope. A critical section is a good example of a local scope in which shared variables are protected by a lock. The most up-to-date values of shared variables modified in a local scope should be visible to a node that later enters into that scope. A section between two consecutive barriers is a good example of a global scope in which all nodes should see the most up-to-date values of all shared variables after the barrier.

In terms of programming impact, under ScC programmers need to be more careful about accesses of shared variables which must not occur in parallel within different scopes. This makes it more difficult to make a correctly running program under ScC compared to under LRC. The reason is that under ScC only the shared variables in the same scope are propagated to the other nodes so that programmers must not only consider how to avoid a data race but must also ensure that there is no overlapping of variable accesses across scopes.

Most existing programs correctly running under the weak memory consistency models such as RC can be run without problems under ScC. However not all correctly running programming under the weak memory consistency models can be executed correctly as programmers expected.

This condition of correct running program under ScC gives programmers more burden. However, due to the restrictive memory access pattern required, it makes the performance under ScC better and more scalable. The reasons for the better perfor-
mance are that only the variables under the same scope are invalidated or propagated (Iftode et al., 1996). This makes less false sharing which is the major hindrance to the DSM performance.

In Figure 2.4, the difference between RC and ScC is illustrated. In the figure, the notation \( w(a)1 \) and \( r(a)1 \) mean a write 1 to shared variable \( a \) and a read 1 from shared variable \( a \) respectively. As shown in Figure 2.4 the result of a read from \( a \) in node 1 under RC is guaranteed to be 1, but under ScC the result of a read from \( a \) is not specified. Under ScC, the result of read \( a \) is guaranteed to be 1 after the next barrier.

Therefore, a DSM programmer who adopts scope consistency should be aware that a program that can be executed correctly under LRC cannot be executed correctly under ScC, as shown in Figure 2.4. As the memory consistency models become more relaxed, the total ordering constraints on local operations are more relaxed. That is, ScC which is more relaxed than LRC provides more relaxed total ordering constraints. This can preclude some programs developed under LRC from benefitting from ScC (Iftode et al., 1996).
2.3 Memory Coherence Protocols

In the previous section, we explained how various memory consistency models define legal memory access interleaving differently in a more relaxed way. As the models become more relaxed, constraints on the legal interleaving order of memory accesses also become more relaxed. A memory coherence protocol is required to enforce the constraints imposed by a given memory consistency model. We explained how lazy enforcement of memory coherence is possible with more relaxed memory consistency models. Thus, the number of actions required by the coherence protocol following more relaxed models is reduced, and merging of memory access updates is possible.

In terms of memory system implementations, a memory consistency model defines when to update other nodes, what data is to be transferred and to which nodes. Generally, comparing strict and weak models, strict models enforce the memory update of other nodes immediately but weak models allow update of other nodes to be delayed until synchronisation points. Similarly, strict models enforce more nodes and data to be updated while weak models allow fewer nodes and less data to be updated without violation of the correctness in program execution.

Regardless of the different memory consistency models chosen, every DSM system has a memory coherence problem since it supports the shared memory programming model in a distributed environment. Memory coherence protocols include all the mechanisms to be used in order to enforce a chosen memory consistency model. When a memory system enforces the requirements of a chosen memory consistency model, there are many considerations in terms of how to implement the enforcement even though the purpose of memory coherence protocols is simply that any read access should return the latest value of the variable. The considerations became more complicated due to the lack of hardware support or inherent complexity of memory coherence in a distributed environment, in which it is not easy to define the latest value (Lamport, 1978).

Below, we briefly present cache coherence protocols since they are directly related to the coherence protocol used in DSM systems. Then, we describe the coherence protocols used in software DSM systems. Finally, two different coherence protocols, homeless and home-based protocols, which are used in the state-of-the-art software DSM systems, are briefly introduced.
2.3.1 Cache Coherence Protocol

A cache coherence protocol is used to maintain correct values of shared variables throughout physically distributed caches in a cache-based multiprocessor. A cache coherence protocol prevents a node from reading stale cache values, which is known as the cache coherence problem.

Two cache coherence mechanisms are widely used for cache coherence in cache-based systems: snooping and directory-based (Lilja, 1993; Hennessy, Heinrich and Gupta, 1999). The snooping coherence mechanism is primarily used in small-scale shared memory multiprocessors. By using a broadcasting facility in a bus, which provides so-called uniform memory access (UMA), the snooping mechanism is relatively easy to implement. However its scalability is limited only to small-scale multiprocessors since all memory accesses are performed through a bus which can cause a bottleneck easily.

To remedy the scalability problem in the snooping mechanism, a distributed directory-based coherence mechanism is used in DASH (Lenoski et al., 1990) and SCI (James, Laundrie, Gjessing and Sohi, 1990). In the distributed directory-based coherence mechanism, locations of shared variables are statically assigned and distributed throughout the nodes. When a coherence action is required, data communication is performed only among nodes that cache the shared variable. However implementing the directory-based mechanism is more complicated since the coherence information, such as a node list that represents nodes that cache data and the state of data, is distributed throughout all nodes, compared to the snooping mechanism in which the coherence information is centrally available through a bus. Naturally, the directory-based mechanism is widely used in scalable architectures such as the cc-NUMA or Cache Only Memory Access (COMA) (Frank et al., 1993; Hagersten, Landin and Haridi, 1992) architectures.

Though the mechanisms described above can provide information about whether cached values are stale or not by using snooping or the directory, they require another coherence action by which values in every cache are guaranteed to be up-to-date. The two ways to do this are: write-update and write-invalidate.

In the write-update protocol, a cache experiencing a write hit updates other cached copies directly by sending the new value. On the other hand, the write-invalidate protocol first invalidates the cached copies on the remote nodes before proceeding with the write. When a node accesses the invalidated page, it has a cache miss and asks for the up-to-date value from the other cache or the memory that has the up-to-date value.
2.3.2 Coherence Protocols used in Software DSM Systems

The coherence mechanism in IVY is mainly concentrated on two issues: 1) synchronised access to a page, and 2) page ownership (Li and Hudak, 1989). These two issues arise due to enforcing SC which IVY adopts. Consequently, a single writer multiple reader (SWMR) coherence protocol is used with a write-invalidate coherence policy. The synchronised access issue is solved by the SWMR protocol since only one node has a right to write to a particular page. The page ownership issue is related to the first issue. When a page is accessed one at a time by nodes, the current owner should have the most up-to-date copy of the page. The second issue is raised by the need to find the locations of the current owners of pages when a node experiencing a page fault needs the up-to-date copy of the page. In Li and Hudak (1989) presented many algorithms for finding a page owner efficiently using page managers, such as a centralised manager algorithm and a distributed manager algorithm.

In Munin, strict synchronised access to a page in IVY is eased due to its adoption of RC. The page ownership problem is solved by using the distributed directory-based mechanism. The write invalidation and update policies are adaptively used depending on a page’s memory access pattern, classified into one of five types: conventional, read-only, migratory, write-shared and synchronisation (Carter et al., 1995). For example, if a page is classified as migratory by a programmer, the write update policy is used for the page. The coherence protocol in Munin is more complicated than in IVY since RC and a multiple writer protocol allows multiple writes on the same page, though eager coherence actions relieve the complexity.

In Midway, which implements EC, the write-update coherence policy is chosen because it is more efficient for EC. In EC, coherence actions during lock synchronisation time are required only for shared variables associated with a particular lock. Also these actions are performed only between two nodes: the acquiring and releasing nodes. Furthermore, the shared variables required for coherence actions are likely to be accessed again in a lock-acquiring node. Therefore, if the write-update policy is used under these requirements due to EC, it would reduce data traffic and cache misses.

In TreadMarks, which implements LRC, the write-invalidate policy is enforced for memory coherence. The set of happens-before memory accesses required for memory coherency under LRC is larger than under EC, as illustrated in Figures 2.2 and 2.3. Under this coherence requirement in LRC, the write-invalidate policy can be more efficient, since it can reduce unnecessary data traffic more than the write-update policy, apart from a prefetch effect.
Princeton’s home-based LRC DSM system (HLRC) (Zhou et al., 1996; Iftode, 1998) uses the write-invalidate policy in order to ensure write atomicity. To improve the performance of coherence-related actions, automatic update hardware is employed, which is called automatic update release consistence (AURC) (Iftode et al., 1999). ScC is also implemented in the same system using the same write-invalidate policy.

2.3.3 Homeless versus Home-based Coherence Protocols

TreadMarks and HLRC are regarded as the state-of-the-art page-based software DSM implementations. Both implement LRC but use a different memory coherence protocol in terms of how to manage the most up-to-date pages. The implementation of LRC requires that before an acquire access is finished, all previous happens-before memory accesses should be visible at the acquiring node.

The two implementations are different in terms of how to make the previous accesses visible. In TreadMarks, the information of the previous accesses is obtained by comparing vector timestamps in intervals made locally or remotely, and collecting required modifications that are created later than those at a local node has according to LRC. The required modifications may be distributed over many nodes. In HLRC, the information is obtained from a so-called home node which always retains the most up-to-date copies of its own pages. To make a comparison, the protocols used in TreadMarks and HLRC are called homeless and home-based, respectively. An in-depth analysis of these protocols and their differences are presented in Chapter 4.

2.4 Performance Improvement Techniques

Since IVY showed itself to be a viable software DSM implementation, many performance improvement techniques for DSM have been developed along with relaxed memory consistency models. Most performance improvement techniques exploit relaxed memory access constraints provided by relaxed memory consistency models. Below, we summarize major performance improvement techniques in software DSM since IVY.

2.4.1 Time Window

As described earlier, the first software DSM implementation, IVY, had a ping-pong data transfer problem caused by false sharing. Since IVY’s memory coherence implementation was based on SC and the minimum memory unit was a page, if more than
two nodes access the same page at the same time and at least one of the accesses is a write, the page is sent back and forth between the nodes without finishing the memory accesses. The coherence protocol enforcing SC prevented all parallel program executions accessing the same page from advancing, thereby producing useless page transfers between the nodes.

In Mirage (Fleisch and Popek, 1989), the writer of a page retains access to that page for a fixed period of time, called the time window ($\Delta$), to solve this ping-pong problem. The optimum setting of $\Delta$ is dependent on an application’s memory access pattern. If a memory access pattern has poor locality, a small or zero $\Delta$ is recommended. On the other hand if a memory access pattern has good locality, a larger $\Delta$ is recommended.

### 2.4.2 Multiple Writer Protocol and Diffing

The time window technique used in Mirage did not solve the false sharing problem entirely. The solution proposed in Mirage has a limitation because the memory consistency model used in Mirage is too strong. The more relaxed model used in Munin gave an opportunity to solve the ping-pong effect. The multiple writer protocol was developed by exploiting the relaxed memory consistency, which allows the delay of memory updates. Since the write update is delayed until the next synchronisation point, concurrent writes on the different parts of the same page are possible even though more complicated memory write bookkeeping is required.

The multiple writer protocol created another important write collection technique called “diffing”. Diffing is a technique to create a “diff” which represents the difference between a twin that is created as a copy of a page before a write, and the final copy of the page. By only sending a diff, instead of the entire page, the data communication required for memory consistency can be reduced. The downside of this technique is more complicated protocol implementation, more processing in order to create twins and diffs, and more memory required to store temporary twins and diffs.

### 2.4.3 Prefetching

Since a performance bottleneck in software DSM systems is remote memory access latency, data prefetching techniques have been used to reduce this latency in many DSM implementations (Pinto and Bianchini, 2003; Lee, Yun, Lee and Maeng, 2001). The main idea is to prefetch data before they are accessed. To do this, prediction of shared variables involved in future memory accesses is required. The prediction is made
based on previous memory access patterns of an application. Therefore this technique is normally applied to applications which have regular memory access patterns between two consecutive barriers.

2.4.4 Adaptation Protocol

The idea behind the adaptation protocol stems from various memory access patterns in DSM applications. Even in the same application, a memory access pattern on each shared variable can be different. Therefore a technique to apply different protocols to applications or shared variables has been proposed in order to reduce data communication by adapting protocols dynamically.

Munin is a typical example of a system that uses this technique (Bennett, Carter and Zwaenepoel, 1990a; Bennett, Carter and Zwaenepoel, 1990b). In Munin, multiple consistency protocols can be assigned to shared objects according to their access pattern as declared by programmer annotation. This annotation adds another burden to a programmer in exchange for performance gains.

Another adaptation technique between single writer and multiple writer protocols was proposed in order to combine the advantages of these two protocols (Amza, Cox, Zwaenepoel and Dwarkadas, 1997). Since a single writer protocol allows only one node to have a right to write on a particular page, whenever another node wants to write on the page, the page ownership is transferred. If a single writer protocol is implemented in a page-based DSM system, even under LRC it suffers from false sharing when an application’s memory access pattern shows frequent write-write false sharing. However if an application’s memory access pattern shows less write-write false sharing, a single writer protocol has more advantages than a multiple writer protocol. Unlike the programmer annotation required in Munin, the adaptation used in Amza et al. (1997) is automatic. Keleher (1996) concluded that a choice between memory consistency models is more important than a choice between the multiple writer and single writer protocols.

Finally, an adaptation technique between the write-invalidate and write-update coherence protocols has been developed (Ng and Wong, 2000; Yun, Lee, Lee and Maeng, 2001). The main reason for the adaptation between these protocols is to increase the efficiency of coherence actions. Generally, the write-invalidate protocol produces less data traffic compared to the write-update protocol. However, if a memory access pattern is predictable, then using the write-update protocol is more efficient since the data accessed are prefetched, and this can also reduce page faults.


2.4.5 Multithreading DSM system

In order to reduce remote communication latency, and also improve parallelism in program execution, many multithreaded DSM systems are implemented, such as CVM (Keleher, 1995a; Thitikamol and Keleher, 1998), Brazos (Speight and Bennett, 1997), Millipede (Itzkovitz, Schuster and Wolfovich, 1996) and DSM-Threa ds (Mueller, 1997), to name a few. In multithreaded DSM systems, different threads can work on computation and communication processes independently in order to increase overlapping of different processes. This can lead to efficient resource use and true parallel computing. This is particularly true and effective in a computer architecture consisting of a cluster of SMPs (Itzkovitz et al., 1996; Kee, Kim and Ha, 2004). Even though it is more complicated to maintain memory coherence in multithreaded DSM systems, data locality of DSM applications can be exploited more over a cluster of SMPs for the following two reasons. First, the cost of maintaining memory coherence within an SMP is much lower. Second, there is more likely to be data locality in a SMP than in a uniprocessing system.

Apart from a complicated memory coherence protocol, the disadvantages of using multithreading in a DSM system are related to the complications of dealing with multiple threads in a local node, such as context switching between threads and more cache misses in a local node (Thitikamol and Keleher, 1998).

2.5 Conclusions

In this chapter, we describe how software DSM systems have evolved from IVY in order to improve performance in terms of memory consistency models and performance improvement techniques. We discuss how memory consistency models such as SC, PC, WC, RC, EC and ScC (ordered from strict to relaxed) play an important role in determining the correct actions of a memory coherence protocol in DSM.

Also, we show that the more a memory consistency model is relaxed, the more opportunity there is to have an efficient implementation of a coherence protocol, especially for a software DSM system. The relaxation of ScC provides the opportunity to implement our hybrid home-based DSM system efficiently. In the next chapter, instead of describing a DSM system from theoretical points of view as in this chapter, we describe the practical points of an DSM system implementation.
Chapter 3

Development of a Software Page-based DSM System

In this chapter, we present many important considerations and implementation details for developing an efficient software DSM system. We do not include how to set up the data communication between nodes, rather we concentrate on a protocol that implements memory coherence in DSM. Since the cache coherence problem exists in a DSM system, an efficient implementation of a cache coherence protocol is the main goal of an implementation of a software DSM system. However, as explained in Chapter 1, the challenges facing the development of an efficient software DSM system must be overcome. The challenges are due to the relatively large memory access latency and memory unit size. A memory consistency model can reduce the problem of memory access latency. Also, since a large sized memory unit is common in a software DSM system, this causes the false sharing problem which adds another complicating factor. Therefore, two issues, the memory consistency model and the size of a sharing unit, need to be considered in order to develop an efficient DSM protocol.

Below, we first explain in detail how the memory consistency model and the size of a memory unit affect the implementation of a coherence protocol in DSM. Then we describe many implementation details which need to be considered when implementing a coherence protocol that handles the complexity of maintaining memory consistency. Since a page-based DSM system uses the virtual memory system, it is vital to have an efficient page fault handling algorithm.

Furthermore, a weaker memory system is required to have system-provided synchronisation primitives that not only guarantee synchronised program behaviour but also perform actions for coherence-related data communication in the weaker memory
consistency model. In contrast with message passing, shared memory programming requires frequent explicit synchronisations in order to protect data integrity. A performance bottleneck can happen during this synchronisation process since synchronisation contention and excessive communication in synchronisation processes prevent parallel execution from running efficiently, as evident in some applications such as PNN, Barnes-Hut, and IS over the homeless protocol as shown in Chapter 4. An efficient implementation of the synchronisation primitives results in less contention thereby promoting fast sequential synchronisation processes.

3.1 Memory Consistency Model

As discussed in Chapter 2, a memory consistency model governs a coherence protocol by which data are transferred. The more relaxed a memory model, the less data are transferred. This is because as a memory consistency model becomes more relaxed, a system following the model is more tolerant in allowing an inconsistent state of shared memory. Allowing an inconsistent state of shared memory for longer means more data is buffered and sent in aggregation at the proper time. Choosing a relaxed model is more important to software DSM systems since data communication is more costly.

There are many experiments which evaluate the effects of the relaxed memory models on DSM performance (Gharachorloo et al., 1991; Keleher, 1996; Adve et al., 1996; Iftode et al., 1996; Zhou, Iftode, Li, Singh, Toonen, Schoinas, Hill and Wood, 1997; Iosevich and Schuster, 2004). SC and RC were compared over DASH (Lenoski, Laudon, Gharachorloo, Weber, Gupta, Hennessy, Horowitz and Lam, 1992) which has a relatively large memory access latency compared to a small bus-based multiprocessing system. Relaxed models are more effective on systems with large memory access latency since their main purpose is to reduce or hide memory access latency. Gharachorloo et al. (1991) reported that under RC, there was a 10 to 40 percent increase in performance over SC. As the authors indicated, as a model becomes more relaxed, the implementation of the model is more complicated since more outstanding accesses are allowed and these should be managed at a synchronisation point. This holds even more in a software DSM system due to the added problem of the large granularity memory unit size.

Keleher (1996) compared SC and LRC over CVM, which is a page-based software DSM system. In it, the relaxed model, LRC, showed an average 34 percent better performance than SC in the eight applications tested. Among the applications, LRC had a
more positive effect on fine-grained lock-based applications. In particular, applications showing write-write false sharing had more benefit from LRC. With the SC protocol, those write-write false sharing applications would suffer from frequent page ownership changes. On the other hand, in applications that have coarse-grained memory access patterns, the two models had comparable performance. Even though LRC is better at hiding memory access latency, applications that have coarse-grained regular memory access patterns do not require each node to transfer its own writes to other nodes as often as in irregular fine-grained applications. Also LRC’s diff-related costs, which involve creating twins and diffs, and applying lazy diffs, offset its advantages over the simpler action of the SC protocol. Other disadvantages of LRC were also reported such as more complicated implementation and a higher memory space requirement for outstanding memory accesses.

Home-based LRC (HLRC) and SC were compared over a multithreaded software DSM system (Iosevich and Schuster, 2004). They demonstrated that an optimized SC implementation with fine-grained memory access granularity emulated on a page-based DSM system can be comparable with an HLRC DSM implementation. However, as indicated by Keleher (1996), one of their test applications, Barnes-sp, suffers under SC due to its frequent fine-grained write-write false sharing.

Performance comparisons between pairs of relaxed models, LRC versus EC and LRC versus ScC, were presented by Adve et al. (1996) and Iftode et al. (1996), respectively. Though the performance depends on the implementation of the models, there should be no performance difference between EC and ScC based on the reason that the conditions for correct memory accesses for EC and ScC are same. From an implementation point of view, the main difference between LRC and EC or LRC and ScC can be found in the depth and range of the happens-before order. In EC or ScC, the happens-before order of memory accesses only concerns memory accesses guarded by a particular synchronisation primitive. On the other hand, in LRC, the happens-before order should be respected for all previous memory accesses in program execution history.

Choosing a memory consistency model affects the implementation of a coherence protocol. This also greatly affects the performance of a page-based software DSM system. For example, consider implementations of two coherence protocols that conform to SC and LRC in a page-based DSM system. The programming model in SC has no concept of system-provided synchronisation primitives. Synchronisation of processes is achieved implicitly through shared variables. The contract between a programmer and an SC memory system is that every read should return the latest value of the variable
to be read. Since the contract defined is intuitive but too general, and a node cannot predict when other nodes will read the writes, every write in a node should be visible to other nodes immediately, which makes it costly to implement SC. This means that if a node would like to write on a shared variable while the other nodes cache the same variable, before the node proceeds to write it should have exclusive ownership of the variable. Therefore, for a particular memory unit, either only one writer or multiple readers exist at any time in a SC protocol. Though this can be implemented in many different or efficient ways, fundamental restrictions of SC mean the inevitable serialization of each memory access.

If the minimum memory unit size is large, the performance of an SC coherence protocol will suffer more due to frequent false sharing and fragmentation. On the other hand, under LRC, multiple nodes can write to different parts of the same page at the same time. Under LRC, multiple writers and readers of a page can be permitted at any time. This difference affects how to implement the protocol significantly. In an SC protocol implementation, it is simpler to make DSM coherent since the coherence protocol for SC is basically the single writer protocol in which only one writer is allowed to write to a page at any given time. To implement the single writer protocol it is important to know the owner and location of shared data.

The communication traffic under an SC protocol is greatly affected by the size of the minimum memory unit. A whole memory unit is transferred even if only a part of the unit is needed by a remote node. Therefore in an SC protocol, choosing the right size of the physical memory unit is more important than in relaxed models. Under LRC, however, due to allowing many nodes to write on a memory unit, many partial writes on the same page, which may be dispersed over many nodes, should be combined in order to make a complete up-to-date memory unit. This makes the implementation of a LRC protocol more complicated.

Another example of comparisons of protocol implementations can be found between LRC and EC or ScC. As stated earlier, LRC should keep a wider range of memory accesses occurring in the same happens-before partial order than EC or ScC. The happens-before partial order consists of a program order and lock release to lock acquire transition order. As described in Section 2.2.4, implementing LRC requires more complicated page version control due to more outstanding memory accesses. This is why most LRC implementations use complicated vector timestamps or vector clocks in order to keep the complete previous write history according to the partial happens-before order. On the other hand, EC or ScC, as illustrated in Figure 2.3 only require
the system to keep the values of the shared variables inside a critical section up-to-date. Therefore, an efficient implementation of EC or ScC is possible by keeping the last versions of shared variables written inside a critical section, which are also grouped by their particular locks.

3.2 The Size of a Memory Unit

The size of a memory unit has a significant effect on the performance of a DSM system. It also affects how to efficiently implement a coherence protocol. We could say that the hardware of a page-based DSM system should provide a variable size of granularity for a memory unit. Then, there would be no false sharing and more efficient data communication would be possible. On the contrary, a smaller memory unit is not always favourable because the memory access behaviour of many programs tend to exhibit locality and spatiality. For those programs, a smaller memory unit size prevents data prefetching.

There are many reports of determining the effects of memory unit size on performance of a DSM system (Zhou et al., 1997; Itzkovitz and Schuster, 1999; Chandra, Gharachorloo, Soundararajan and Gupta, 1994). Zhou et al. (1997) found that as the memory unit size becomes larger, it is beneficial to implement a more relaxed memory model. This is understandable because a relaxed model is more effective in reducing false sharing. There was another project that experimented with emulating a fine-grained memory unit over a page-based DSM system (Itzkovitz and Schuster, 1999). This approach involved dividing shared variables in the same physical page into many different virtual views, as if each view is placed in a different page. In this work, they justified the use of SC in a page-based DSM system by providing fine-grained access control.

Other than page-based DSM systems, implementations of fine-grained access control were developed for other types of DSM systems (Schoinas et al., 1994; Scales et al., 1996). The implementations were based on complicated compilation techniques, which add another challenge to overcome. Even though these implementations provided fine-grained access control, which reduces the false sharing effect and provides a simple SC programming model, a low latency communication network is required to cope with the larger number of messages generated due to the fine-grained access control.

Practically, a large memory unit in most software DSM systems adds complexity which requires efficient memory update management within a page. A page can contain
many shared variables. A DSM system treats multiple shared variables in a page as one unit. This causes many unexpected effects such as the ping-pong effect (in a negative way) or prefetching effect (in a positive way). How to handle this large memory granularity is one of the important tasks for maintaining memory consistency.

### 3.3 Maintaining Coherent Memory

When we design and implement a software DSM system, the main objective is to provide an efficient implementation of memory consistency. As stated in the previous two sections, the memory consistency model and the granularity of the memory unit greatly affect the implementation of a memory coherence protocol. In terms of the implementation, the two influence the answers to the following questions. Answering these questions is also very important in order to implement an efficient DSM system.

1. How does the system detect whether accessed data is stale?
2. Upon knowing accessed data is stale, how does a node find the new data?
3. How does a node record its new writes for other nodes?

Below, we discuss these questions mainly from the view point of a page-based DSM system. However, in order to make a comparison, we also describe the answers to the questions from the view point of other types of DSM systems such as a hardware-based DSM system or an object-based DSM system.

#### 3.3.1 Is accessed data stale?

In a page-based DSM system, when a node accesses stale data, the system invokes a page fault by means of a memory protection mechanism provided by the virtual memory system. This page fault is triggered by accessing an invalidated memory page. In another sense, a memory consistency model dictates which memory units should be invalidated. The memory unit size affects the range of memory invalidated.

In a relaxed DSM system, if a system uses an invalidation protocol, invalidation of pages occurs during a synchronisation time. Assuming an invalidation protocol is used, an invalidation notice that contains a set of the numbers of the pages that were modified previously is sent to the next synchronised node(s) that require memory consistency. Basically all new writes (contents of the modifications) in every node
should be preserved and the page numbers of the pages overwritten should be known
to other nodes, in order to invalidate those pages in the other nodes before they access
the pages.

In an object-based DSM system, in which the memory unit size is the size of an
object, compiler instrumentation is used to detect a new write in order to enforce
cache coherence actions (Bershad and Zekauskas, 1991; Schoinas et al., 1994; Scales
et al., 1996). When a node modifies the value of an object, the compiler generates code
to mark the modified object as dirty. With this technique, the overhead is very large
and it requires complicated optimization techniques (Scales et al., 1996).

### 3.3.2 Where is the new data?

After a node finds out that the accessed data is stale, the new data should be fetched
from other nodes that have the up-to-date data. Finding the location of the data is
another implementation decision that has to be made. In a single writer protocol, it
can choose between centralized manager or distributed manager algorithms (Li and
Hudak, 1989). The centralized manager algorithm is not efficient since one central
manager can easily become a bottleneck in a distributed environment. There are two
implementation choices widely used for the distributed manager algorithm: a fixed
distributed manager algorithm and a dynamic distributed manager algorithm.

In a fixed manager algorithm, every node is assigned a number of pages that the
node manages. This is easy to implement but it does not consider adaptation for an
irregular memory access pattern. In a dynamic manager algorithm, ownership of a
page is freely transferred to the current writer of the page. This is adaptable to various
memory access patterns but it is not easy to find the owner who has the up-to-date
data since the owner is changeable. To find the location of the owner, a node forwards
the request to the probable owner until the real owner is found. For a page-based DSM
system, it has been reported that the fixed manager algorithm was more efficient than
the dynamic manager algorithm in finding the owner of the data (Keleher, 1996).

In a multiple writer protocol, since more than one node can freely write on the same
page at the same time, locating new data is not as simple as in the case of a single writer
protocol. Basically, as explained in Chapter 4, the homeless and home-based protocol
can be used to locate the new data. Conceptually, the home-based protocol is similar
to a single writer protocol in terms of finding the location of the new data. A fixed or
dynamic manager algorithm can be used in the home-based protocol in order to find
the location of the new data. In the homeless protocol, there is no concept of a fixed
or dynamic manager that takes care of assigned pages as in the home-based protocol. Rather, each node maintains information about data locations through communication with other nodes at the time of synchronisation by exchanging so-called intervals. An interval is a structure containing information about the creating node of a modification, a vector timestamp recording the time of the modification and the list of dirty pages during the interval. Compared to the home-based protocol in which the minimum memory unit transferred when fetching new data is a page, the homeless protocol can support fine-grained diff-based update.

3.3.3 How does a system preserve modifications of data?

Since a relaxed memory consistency model allows the propagation of modified data to be delayed up to a proper synchronisation time, a DSM system implementing a relaxed model should preserve the modifications of data until they are needed by other nodes. In a page-based DSM system, twinning, which creates a copy of a page before writing on the page, and diffing, which creates the difference between a twin and the final version of the page by comparing them word by word, are widely used to preserve the modifications of shared data in a multiple writer protocol. In a single writer protocol, twinning and diffing are not needed since a whole copy of a page is transferred.

In an object-based DSM system, because the granularity of a sharing unit is the size of an object, a simple ownership protocol is sufficient to maintain the consistent state of shared objects. Rather than using twinning and diffing as in a page-based DSM system, an object-based DSM system uses timestamping which indicates the latest modification time of an object (Bershad and Zekauskas, 1991).

3.4 Page Fault Handling

Since a page-based DSM system uses the page protection mechanism to maintain memory coherence, page fault handling enforces coherence-related actions according to different types of page faults. Three types of page faults are possible: a read access on an invalidated page, a write access on an invalidated page and a write access on a write-protected page (or read-only page). An invalidated page indicates that the local copy of a page contains stale data. Any memory access on the page will trigger a SEGV handler invoked by a SIGSEGV signal. The SEGV handler takes responsibility for constructing the up-to-date copy of the page from the remote node(s). If the memory access is a write access, a node should prepare to keep the following modification
caused by the write. Then, the node can freely write on the page. If the memory access is a read access, changing the page state to read-only is sufficient.

A read-only page indicates that this page is safe to read but is ready to record modifications of the page if a write access occurs on the page, which is then followed by a SIGSEGV and invocation of a SEGV handler.

Generally speaking, there are two tasks for a node experiencing a page fault. One is to fetch an up-to-date copy of the page, and the other is to prepare for recording modifications of shared data on the page. The former is required for reading the correct value, and the latter is required for other nodes to use the modifications later.

### 3.4.1 Page State Transition

A page state is changed whenever a coherency action is required for memory consistency. A page state represents an allowable access to the page at a certain time. The reason for restricting the allowable page access is to maintain memory consistency.

Since a page-based DSM system uses the virtual memory system, it uses the memory protection mechanism that is provided by memory system hardware. For example in Linux, four page protection states are provided: PROT_NONE, PROT_READ, PROT_WRITE and PROT_EXEC. PROT_EXEC is not used in a coherence protocol since it is not relevant. PROT_NONE is used to set a page as invalid in order to indicate that the page is stale. When a node finds out that the page accessed is stale, the node should take an action to fetch the up-to-date copy of the page. Depending on the memory access (read or write), the following actions are carried out. If it is a read access, no action is needed for memory consistency. If it is a write access, a node creates a twin of the page in order to record subsequent writes on the page. PROT_READ is used to set a page as readable in order to indicate that the page is safe to read. Upon a write access on a PROT_READ page, a node also should create a twin of the page. Similarly, PROT_WRITE is used to set a page as writable. This protection is only set when there is no coherency problem after this.

### 3.5 Synchronisation Primitives

In a relaxed software DSM system, synchronisation primitives should be provided by the system as explicit synchronisation points should be known to the system in order to improve performance. In a relaxed memory system, the purpose of synchronisation
is not only synchronising parallel execution but also propagating outstanding memory accesses.

Two synchronisation primitives are widely used in DSM applications: locks and barriers. A lock synchronisation is needed to protect data integrity in a critical section by making the data accesses in the critical section mutually exclusive. A barrier synchronisation is used to have all nodes wait at the barrier point until all nodes reach the point.

3.5.1 Lock Synchronisation

Many different lock synchronisation implementations are employed in software DSM systems. Among them is the fixed lock manager scheme used in TreadMarks (Keleher et al., 1994). In this scheme, every lock has a manager to which the lock request is sent. The location of each lock manager is fixed. Whenever the manager receives a lock request from a node, it sets the current lock owner as the lock requester node. Then, the manager forwards the lock request to the previous lock owner if the manager is not the current owner. Upon receiving the forwarded lock request, the previous lock owner directly sends the lock ownership to the lock requester with coherence information, such as an invalidation notice containing invalidated page numbers. The requester, after receiving the lock ownership, can proceed with its executions inside a critical section protected by the requested lock.

The direct coherence information transfer from the current lock owner to the next lock owner via a lock manager is in contrast with other lock implementations in which coherence information is transferred from a lock manager to a lock requester. In these other implementations, the last lock owner sends all the coherence information to the manager which then decides which coherence actions are required for the next lock owner according to a chosen memory consistency model. This lock implementation was employed in JiaJia (Hu et al., 1999b) and KDSM (Lee et al., 2000).

An alternative implementation of a fixed lock manager scheme is a distributed queueing scheme employed in Midway (Bershad and Zekauskas, 1991). In this scheme, a lock acquire request is forwarded to a probable owner of the lock until the owner is found or the probable owner is waiting for the same lock. The lock request is granted if the node is the owner and has finished the critical section, or is queued at the node which is waiting for the lock. The node waiting for the lock will eventually send the lock ownership to the queued node after it finishes the use of the lock. Since Midway uses two types of lock modes, exclusive and non-exclusive, their implementation adds
another level of complexity in order to achieve write atomicity, which is complicated by the use of a non-exclusive lock mode.

The purpose of the lock synchronisation includes maintaining memory consistency, in particular, in a relaxed memory system. For example, in a LRC memory system, all the previous happens-before memory accesses should be visible after the lock acquire. In an EC or ScC memory system, all previous memory accesses occurring in previous critical sections guarded by the same lock should be visible after the lock acquire. An efficient implementation of coherence actions in a lock synchronisation point is therefore an important part of a lock synchronisation implementation.

3.5.2 Barrier Synchronisation

Many implementations of barrier synchronisation in a DSM system also use a barrier manager. When a node reaches a barrier point, it sends a barrier arrival notice to the barrier manager. When the barrier manager receives all the arrival notices, it sends all nodes a barrier departure permission. A barrier synchronisation also serves to maintain memory consistency in a relaxed memory system. Thus, with a barrier arrival notice, each node piggybacks a write notice containing page numbers modified between the last two barriers. Collecting all the write notices, the barrier manager decides which coherence actions are required for the other nodes. Immediately after exiting a barrier, each node should see the most up-to-date shared memory.

3.6 Data Packet Handling

Since a DSM system maintains memory consistency through network packet transfer, how to efficiently handle data packets at a low level such as the network layer is another important implementation issue. In a DSM system, asynchronous transfer of data packets is used to request important actions such as lock acquire request, page fetch request, or barrier arrival notice.

A DSM system uses generally two methods, polling and interrupt, in order to find out that asynchronous messages have arrived at a local node. Polling uses frequent checks to find out whether or not a data packet has arrived at a receive buffer. On the other hand, an interrupt involves sending a signal from the OS when a message has arrived.

As briefly described by Zhou et al. (1997), polling is more effective in applications that have a high communication/computation ratio because those applications require
much communication and polling interrupts fewer computation cycles compared to an
interrupt. The interrupt mechanism is more costly for those applications since frequent
interrupts means frequent context switches between kernel and user modes.

TreadMarks implements a SIGIO handler which handles incoming asynchronous
packets. A SIGIO handler is triggered by packet arrival at the receive socket. To
protect memory integrity, signal blocking is appropriately used while shared variables
are modified.
Chapter 4

Homeless and Home-based LRC Protocols

In this chapter, we compare the homeless and home-based LRC protocols. When we say the homeless LRC protocol, we refer to the protocol used to maintain memory coherence in TreadMarks (Keleher et al., 1994). The homeless protocol is named in order to make a comparison with the home-based LRC protocol (HLRC) (Zhou et al., 1996) used in a software DSM system developed at Princeton University. To make a fair comparison, we implemented a home-based protocol based on TreadMarks. The source code of the two implementations are the same except for differences due to the protocols.

The homeless protocol is a feature of the LRC DSM system implemented in TreadMarks that improves a previous ERC DSM system implemented in Munin (Bennett et al., 1990b). In order to remove unnecessary data traffic caused by RC and the directory-based coherence protocol in Munin, a more lazy implementation of RC was developed for TreadMarks.

The home-based protocol was developed to maximize the effect of the automatic update mechanism (Iftode et al., 1999) provided in the SHRIMP network at Princeton University. The automatic update mechanism provides transparent write-through communication between local and remote nodes independently of the computation process. Since communication and computation processes are independently executed, these two parallel executions can improve DSM performance. Using the automatic update, writes in a local node can be found in a designated remote node with much less communication cost. This network hardware facility opens the door to the software home-based DSM protocol.

A similar concept of a home node in a home-based protocol can be found in DASH
(Lenoski, Laudon, Gharachorloo, Weber, Gupta, Hennessy, Horowitz and Lam, 1992), a hardware DSM system, in which shared variables are distributed over the clusters of nodes. Each cluster, in the form of an SMP system, takes care of its own shared variables. In DASH, a home cluster maintains a directory which contains information about the state of its own blocks and a list of which cluster has the cached blocks. The only difference between the two implementations of the home-based protocol is that DASH implements RC, but HLRC implements LRC.

In the next two sections, we describe the design and implementation of the two protocols. There were already two papers that compared the performance between the two protocols (Zhou et al., 1996; Cox, de Lara, Hu and Zwaenepoel, 1999). The conclusions from those two papers showed different performance results: one found no significant difference between the two protocols (Cox et al., 1999) and the other found the home-based one significantly outperformed the homeless one, in particular in terms of scalability (Zhou et al., 1996). In Section 4.3, we present our performance comparison between the two protocols based on our performance measurements using real DSM applications. We use well-known DSM benchmark applications, as well as our parallel neural network application.

4.1 Homeless Protocol

4.1.1 Protocol Design

LRC is the heart of the homeless protocol design used to implement TreadMarks. The motivation for the protocol design was to reduce unnecessary data communication in order to achieve better performance compared to Munin, which was regarded as the best page-based software DSM system at that time. The developers of LRC discovered that the eager implementation of RC in Munin produces unnecessary data communication at the time of lock release. Since lock synchronisation is required to protect data integrity in a critical section by giving an exclusive access right to only one node, which is the next lock acquiring node, nodes other than the next lock owner would not access the invalidated data unless a data race has occurred. Therefore, eager coherence actions to nodes other than the next lock owner node are not required other than to improve the prefetch effect. The comparison between ERC and LRC is well illustrated in Figure 4.1. The figure assumes all three nodes cache pages 1 and 2 and use the write-invalidate cache coherence protocol. As illustrated with the same memory access
Eager Release Consistency

Lazy Release Consistency

Figure 4.1: ERC versus LRC

scenario in Figure 4.1 LRC, compared with ERC, removes four invalidation messages, two by piggybacking them in lock ownership grant messages and the other two by delaying the invalidation messages.

4.1.2 Protocol Implementation

TreadMarks has many innovative solutions which take advantage of the relaxed constraints that LRC provides. On the other hand, the laziness in TreadMarks requires a complicated write detection and collection mechanism. Each node creates its own diffs and keeps them until they are needed by other nodes according to LRC. Also, each node asks for diffs that were created by other nodes and also keeps them in its local
To implement the design of the protocol, an interval and a vector timestamp (VTS) are used. An interval is a linked data structure that records all the write activities between two synchronisation points, normally between a barrier and lock release, or between two consecutive barriers. Its main fields consist of a VTS that indicates the creation time, a node-id for the node that creates the interval and a list of page numbers that have been written during the interval.

During a lock acquire process, when a node detects concurrent writes by previous lock owners on a page that it has written to, it creates an interval for its concurrent writes before invalidating the page, in order to keep the concurrent writes in the form of a diff. The diff created at this time is eagerly created, which is in contrast with many other diffs that are created lazily.

A VTS indicates the creation time of an interval. Each number in a VTS represents an event sequence number in a node. By comparing VTSs, intervals can be sorted into the happens-before order (Lamport, 1978). The scenario, shown in Figure 4.2, explains how a VTS is used to implement the happens-before order. As can be seen in the figure, when communication happens between two nodes, such as lock synchronisation, each node updates its vector time so that each position in the vector has the maximum value from its current vector time and the other node’s one. Also, when an interval is created at release, its node element in a VTS increases by one. In this way, the happens-before order of partial intervals is maintained.

For example, a sequence of VTS(1, 0, 0) → VTS(1, 1, 0) → VTS(1, 2, 1) establishes the happens-before partial order consisting of three intervals, while VTS(1, 0, 0) and VTS(0, 0, 1) establishes two concurrent intervals. Since intervals can be arranged in the happens-before order and also concurrent writes can be found by comparing VTSs, the most up-to-date copy of the page can be correctly constructed later by applying diffs to the page in an order consistent with the partial order on their VTSs.

4.1.3 Protocol Analysis

When a page fault occurs because of an access to an invalidated page, the faulting node asks for the required diffs from the last writer(s) of the page, in order to construct the most up-to-date page. This diff request process is very complicated and could be a bottleneck. This problem becomes worse in migratory applications, where a diff
from the same page is accumulated so that the size of required diffs gets bigger and bigger through synchronisation processes. This is called the *diff accumulation problem*. Ironically, the diff accumulation problem is caused by the LRC’s main strength, which allows lazy update. Memory update by distributed diffs in the homeless protocol could be inefficient when the diff accumulation problem happens.

In TreadMarks, a diff request process, which retrieves the required diffs from a remote node(s) upon a page fault, is done by comparing the VTSs of the previous intervals that have written to the page before the fault. To facilitate this process, a page data structure has a linked list called a *write notice* field that points to the interval that contains the page as a dirty page. The linked list *write notice* is used to track down all the previous writes on the page. When tracking down is finished, then the diff request(s) is sent to the owner(s) of the required diffs. Note that, if among the required diffs, each number in a VTS is equal to or larger than the corresponding numbers in the other VTSs, the creator of the larger VTS diff should also hold the smaller VTS diffs, thus one request can fetch multiple diffs from one node. For example, in Figure 4.2 if three intervals with VTS(1, 0, 0), VTS(1, 1, 0) and VTS(1, 2, 1) all have the same dirty page, then, at VTS(1, 2, 1) node 1 should have all three diffs of the intervals.

Another problem the homeless protocol can create is a so-called *hot spot*. A hot spot node is a node that receives many coherence-related requests from other nodes simultaneously. A hot spot is susceptible to creating a bottleneck since the hot spot node holds back the advancement of program executions in other nodes. Figure 4.3 illustrates the situation. We assume that only one page and four nodes are involved in the figure. A different colour means a different diff on the same page. This is a
Figure 4.3: Hot Spot in the Homeless Protocol

typical memory access pattern in migratory applications. Node 0 only writes on the page between the two barriers. This sole write in node 0 is not uncommon in real DSM applications where only one node summarises the last barrier session or calculate an intermediate threshold value to end the execution. The figure also illustrates the diff accumulation problem where node 3 has accumulated three diffs made from one page. A diff request from node 0 should bring the three accumulated diffs from the last writer, in the figure node 3, after the first barrier. The problem of a hot spot is created after the last barrier when all nodes recognize that node 0 is the last writer of the page. The nearly simultaneous requests become worse due to diff accumulation. For example, node 1 will need three accumulated diffs made by the other three nodes, though node 2 and node 3 only need two diffs (red and cyan) and one diff (cyan), respectively. If more nodes are involved and memory access patterns inside a critical section are coarse, then the problem of combining hot spots and diff accumulation would degrade the DSM performance significantly.

The memory space requirement for keeping unnotified diffs becomes a problem as the size of those diffs become larger. When the size of unnotified diffs in any node exceeds a threshold value set by the system, garbage collection is triggered, in which all unnotified diffs in all nodes are forced to be visible to all nodes. Apart from the
larger memory space requirement, this garbage collection requires global and heavy communication, and can severely degrade DSM performance.

4.2 Home-based Protocol

4.2.1 Protocol Design

In the homeless LRC protocol, since its main objective is to delay write updates in a local node to be visible in remote nodes as long as possible without violating the conditions for LRC, the implementation involves a complicated diff management scheme. The home-based protocol makes this complicated scheme simple by eagerly processing write updates. When diffs representing write updates are created at lock release or barrier time, they are sent to their home nodes immediately. The home nodes receive the diffs and apply them to the corresponding pages to keep their pages up-to-date. That is, via a home node, the most up-to-date copies of the home-owned pages can be obtained. It is a much simpler strategy than the homeless protocol: *Keep the home node up-to-date and get the up-to-date memory from the home node if needed.*

In Figure 4.4, a comparison of coherence-related actions between homeless and home-based protocols is illustrated. The figure assumes that each protocol uses the write-invalidation protocol. As seen in the figure, the effects of remote write operations are only visible when they are required, which is after a lock acquire process followed by a page fault in both protocols according to LRC. The difference between them lies in how to preserve and fetch the up-to-date data.

In the homeless protocol, up-to-date shared variables in the form of diffs are not propagated unless other nodes request them. A node, however, notifies other nodes about the new update in the form of an invalidation notice at a synchronisation point. As a result of the invalidation, a memory access to invalidated data triggers a request for the diffs. The required diff(s) could be distributed over many nodes or one node could have all of them. In many cases, the last lock owner has all the diffs required by the next lock owner, as illustrated in Figure 4.4. Lazy diffing (Keleher, 1995b) further delays the creation of a diff until it is requested from the next acquiring node. Unless diffs are requested from others, diffs are not created.

In the home-based protocol, diffs are created and sent to the corresponding home node at a lock release or barrier time in order to keep the home node updated. Updating stale local data is much easier in the home-based protocol because a node simply
requests the data from the home node of the data. All diffs are applied in the home nodes as they are created so that the happens-before order of diff application is kept by the nature of the protocol. On the other hand, the homeless protocol has to sort out diff application in the happens-before order due to lazy diff application of diffs that are created at different times and by different nodes.

### 4.2.2 Protocol Implementation

As in the implementation of the homeless protocol, an *interval* data structure and a vector timestamp are used to order diffs in the happens-before order in the home-based protocol implementation. By comparing vector timestamps each node knows how far
the other nodes have been updated. Therefore when memory consistency is required, a node can determine which data are needed from other nodes according to LRC. The major difference between the two protocols can be found when a page fault occurs in order to fetch the most up-to-date copy of the faulting page. In the homeless protocol, the diff request is sent to the last writer(s) while in the home-based protocol the page request is sent to the home node.

As described in the discussion of the strategy for maintaining coherence in the home-based protocol earlier, two important requirements must be met.

- Keep the pages on the home node up-to-date when needed by the memory consistency model.
- Get the up-to-date memory from the home node if needed.

Since TreadMarks already implements the requirements of LRC by using the invalidation protocol based on the interval data structure, our home-based LRC implementation can use them too. The main objectives of the implementation are focused on how to efficiently implement the two requirements listed above. Achieving the second requirement is easy since we used static home assignment in which every node knows the home node of each page statically. Since the implementation follows the LRC model, meeting the first requirement is only required at the time of synchronisation, normally a lock release or barrier. Sometimes at the time of a lock acquire, a node has to send diffs to their corresponding homes when concurrent writes on the same page are detected. To efficiently send diffs to corresponding home nodes, diffs that belong to the same node are aggregated and sent to the home node as a single packet.

4.2.3 Protocol Analysis

Since diffs are sent to their corresponding homes as soon as they are created, many problems in the homeless protocol are solved such as diff accumulation and the large memory space requirement. Also a hot spot is less likely to happen because of the distributed page requests. Inefficient multi-diff requests upon a page fault in a multiple writer application in TreadMarks are eliminated. Instead, a single page fetch request is sent to the home node, which is always enough to make a faulting page up-to-date.

The other strengths of the home-based protocol mainly come from the home effect. The home effect means a home node is not required to fetch its own pages from a remote node because it always has the up-to-date copies of the pages. Thus, a home
node has no page fault on its pages. Also, a home node is not required to create diffs of its assigned pages in order to update others except for invalidating the cached copies of the pages in other nodes.

This home effect can be translated into a negative performance effect if a home node is assigned to pages that are infrequently accessed by the home node but frequently accessed by non-home nodes. If a non-home node accesses a page more often than its home node, then inefficient page update or fetch data traffic will occur between the home node and non-home nodes. The original implementation of the home-based protocol uses static home allocation in which a page is assigned a home node statically by the modulus operator or a programmer. However, when an application’s memory access pattern dynamically changes, the static home allocation cannot guarantee optimized data communication.

4.3 Performance Evaluation

4.3.1 Benchmark Applications

To evaluate our home-based DSM system, we compare it with TreadMarks. The objective of the evaluation is to compare the homeless and home-based protocols in terms of their effect on the performance of DSM applications. The applications are obtained from the TreadMarks application suite, except PNN which was implemented locally (Pethick, Liddle, Werstein and Huang, 2003). Below, we briefly describe the applications.

- **Parallel Neural Network (PNN)** is a parallel implementation of a neural network algorithm using forward and backward propagations (Werstein, Pethick and Huang, 2003). The data set trained is the shuttle set obtained from the University of California, Irvine machine learning repository (Hettich, Blake and Merz, 1998). The data set is divided with a subset allocated to each node. In the main loop, each node trains part of the data in parallel. Then the local weight changes, calculated in parallel previously, are summed sequentially through lock synchronisation in order to calculate the new global weight matrix. Since each node’s local weight changes are summed into the new global weight matrix during a lock synchronisation, the memory access pattern is migratory so that the partial global weight changes are transferred to the next lock owner.
• **Barnes-Hut** is a simulation of gravitational forces using the Barnes-Hut N-Body algorithm. The original implementation was in the SPLASH application suite (Woo, Ohara, Torrie, Singh and Gupta, 1995) but we have used a version modified by the TreadMarks development team. The Barnes-Hut application uses only barrier synchronisation. The parallelization of gravitational forces computation is achieved by each node computing forces exerted on its assigned bodies which are allocated from the N bodies. The memory access pattern of Barnes-Hut is known to be irregular and fine-grained since in every time step the hierarchical tree is changed and the size of a body is very small.

There are four sections divided by three barriers in each iteration within the main loop. The first and fourth sections are only executed by Node 0 and the second and third sections are executed in parallel. The first section loads bodies into the tree. The second section partitions bodies among processors and calculates forces on all bodies in parallel. The third section advances body positions and velocities, also in parallel. The fourth section outputs some data and computes the new root cell dimensions.

• **Integer Sort (IS)** ranks numbers represented as an array of keys by using a bucket sort. In IS, a local key density array obtained from each node in a parallel execution is added to a global key density array. To parallelize this process, each node accesses a different part of the global key density array in every iteration. IS is known to have a migratory memory access pattern. The version of IS we tested uses only barrier synchronization so we call this IS-B.

• **3D-Fast Fourier Transform (3D-FFT)** solves a partial differential equation using forward and inverse FFTs. The implementation is a part of the TreadMarks application suite. The memory access pattern of each loop is regular in that each node accesses the same area in every iteration. In each loop, each node writes exclusively on a specific part of the shared memory during the computation of the inverse 3D-FFT. After this, the updated memory is read by all nodes. At this time, simultaneous global communication occurs between nodes in order to fetch the memory that other nodes previously updated.

• **Travelling Salesman Problem (TSP)** finds the cheapest way of visiting a sequence of cities and returning to the starting point, given a finite number of cities along with the cost of travel between each pair of them. We used the
implementation in the TreadMarks application suite, which uses a branch-and-bound algorithm. It uses only lock synchronisation to protect the shared data to find out the minimum tour cost and the associated path.

- **Successive Over-Relaxation (SOR)** calculates for each cell in a matrix the average of the neighbouring four cells’ values (up, down, left and right). The shared matrix is divided into \( N \) blocks of rows on which \( N \) processors work. Only the values in boundary rows that two nodes share are sent to each other in every iteration. Therefore the memory access pattern is very regular, coarse-grained and exclusive.

- **Gauss** solves a matrix equation of the form \( Ax = b \). In the main loop, only one node finds a pivot element. After finding the pivot element, all the nodes run Gaussian elimination in parallel. The memory access pattern is very regular and exclusive.

The problem sizes and sequential execution times of the applications can be found in Table 4.1. The sequential execution times are measured using TreadMarks. We found that in SOR and Gauss the sequential execution times for the home-based implementation are slightly longer than for those using TreadMarks because of the different initial page state of node 0. The initial state of pages in node 0 is Read-Write in TreadMarks and Read-Only in the home-based system. The sequential times of SOR and Gauss over the home-based system are 10.32 and 25.86 seconds, respectively. The other applications are not affected by the initial page state difference. Note that each application tested over the different protocols is identical. Also the code implementing the two protocols is the same except for inherent differences due to the protocol difference. Therefore, performance evaluation of the applications between the two protocols tells us the effects of the protocols on the applications.

### 4.3.2 Environment for Performance Evaluation

As illustrated in Figure 4.5 our dedicated cluster network consists of 32 nodes, each one having a 350 MHz Pentium II CPU and running Red Hat Linux 7.2 (gcc 2.96). All nodes are connected by 100 Mbit switched Ethernet. Each node is equipped with a 100 Mbit network interface card and 192 MB of RAM except for node 0 which has 318 MB of RAM. Node 0 serves as a connection point from outside networks and also as a network file server.
<table>
<thead>
<tr>
<th>Application</th>
<th>Problem Size</th>
<th>Iterations</th>
<th>Execution Time (secs.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PNN</td>
<td>44,000</td>
<td>235</td>
<td>710.07</td>
</tr>
<tr>
<td>Barnes-Hut</td>
<td>64k Bodies</td>
<td>3</td>
<td>85.74</td>
</tr>
<tr>
<td>IS-B</td>
<td>$2^{24} \times 2^{15}$</td>
<td>20</td>
<td>72.07</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>64 x 64 x 64</td>
<td>50</td>
<td>44.44</td>
</tr>
<tr>
<td>TSP</td>
<td>19 cities</td>
<td>1</td>
<td>33.07</td>
</tr>
<tr>
<td>SOR</td>
<td>2000 x 1000</td>
<td>50</td>
<td>6.21</td>
</tr>
<tr>
<td>Gauss</td>
<td>1024 x 1024</td>
<td>1023</td>
<td>15.15</td>
</tr>
</tbody>
</table>

Table 4.1: Problem Sizes and Sequential Execution Times

Note that in this experiment, with 192MB of RAM in nodes 1 to 31 and 318MB of RAM in node 0, we monitored for page swaps and observed none. Page swaps occur when running applications need more memory and an operating system moves data between RAM and disk to free up more memory. While running the performance test, we found no page swaps and this shows that the amount of memory is not a primary contributor to the performance degradation and high level of garbage collection.

4.3.3 Overall Performance Results

As can be seen in Table 4.2, the home-based protocol outperforms the homeless protocol in three applications (PNN, Barnes-Hut and IS-B). However, the home-based protocol shows worse performance in TSP, SOR and Gauss compared to the homeless protocol. In 3D-FFT, the homeless protocol shows better performance except for the 32 node computation.

The overall results show clearly the strengths and the weaknesses of the two protocols. The abnormal performance degradations in PNN, Barnes-Hut and IS-B over the homeless protocol, in particular over more than 16 nodes, can be explained by the combination of a hot spot, garbage collection, and diff accumulation in the applications. These weak points in the homeless protocol make it difficult to provide scalability in many applications.

On the contrary, the mediocre performance in SOR and Gauss over the home-based protocol is due to the improper home assignment which produces unnecessary data traffic which could have been reduced if a home node is assigned its frequently accessed pages. We found that the lazy differencing technique and fine diff-based update is very effective in improving the performance in SOR and Gauss. With the lazy
diffing, due to the exclusive single writer memory access pattern, most pages, except for boundary pages that are accessed by two nodes at the same time, are not required to create diffs in order to let other nodes know the memory updates each node made since every node accesses only its exclusive shared memory.

Note that in our thesis, HL and HB mean that the protocols used in TreadMarks and our home-based DSM system respectively.

### 4.3.4 Hot Spot

The hot spot phenomenon in DSM applications can severely degrade performance. The homeless protocol is likely to have a hot spot more frequently than the home-based protocol, as can be seen in PNN. To know why the hot spot occurs in the neural network application, we should understand the parallel algorithm to train the neural network. The details of the training algorithm can be found in (Werstein et al., 2003). The algorithm is:
1. Check if the job\textunderscore flag is not ‘done’.

2. Each node trains the network on its part of the training set.

3. The local weight and node changes obtained from a node are summed through lock synchronisation to produce the global weight and node changes.

4. Barrier synchronisation.

5. Node 0 applies the two changes and calculates a final error. If the error is smaller than the specified error value, set the job\textunderscore flag to ‘done’. Other nodes just wait.

6. Barrier synchronisation

7. Go to Step 1.

During the parallel computation in Step 2, each node gets access to the pages of the shared global weights which were invalidated in the prior Step 6. Then all the nodes except node 0 will have a page fault. In response to the page fault, the two protocols behave differently.

In the homeless protocol, all the nodes except node 0 will send diff requests to node 0 to send the diffs since node 0 is the last writer because of the prior Step 5. These simultaneous multiple diff requests from all the other nodes can be a significant bottleneck as the number of the nodes increases. Even worse, if diffs from the same page are accumulated and become larger, this diff request service will take more time to finish. We confirmed this hot spot phenomenon by measuring the time breakdown

<table>
<thead>
<tr>
<th>Apps</th>
<th>2 nodes</th>
<th>4 nodes</th>
<th>8 nodes</th>
<th>16 nodes</th>
<th>32 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HL</td>
<td>HB</td>
<td>HL</td>
<td>HB</td>
<td>HL</td>
</tr>
<tr>
<td>PNN</td>
<td>2.0</td>
<td>2.0</td>
<td>3.9</td>
<td>4.0</td>
<td>6.5</td>
</tr>
<tr>
<td>B-H</td>
<td>1.6</td>
<td>1.5</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>IS-B</td>
<td>1.9</td>
<td>1.9</td>
<td>3.3</td>
<td>3.6</td>
<td>3.8</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>0.9</td>
<td>0.7</td>
<td>1.2</td>
<td>0.9</td>
<td>1.8</td>
</tr>
<tr>
<td>TSP</td>
<td>1.9</td>
<td>2.0</td>
<td>3.8</td>
<td>3.6</td>
<td>6.5</td>
</tr>
<tr>
<td>SOR</td>
<td>2.0</td>
<td>0.9</td>
<td>3.7</td>
<td>1.2</td>
<td>5.9</td>
</tr>
<tr>
<td>Gauss</td>
<td>1.6</td>
<td>0.2</td>
<td>1.8</td>
<td>0.3</td>
<td>1.4</td>
</tr>
</tbody>
</table>

Table 4.2: Comparison of Overall Speed-ups between Homeless (HL) and Home-based (HB) DSM Implementations
of the main loop. We measured the execution time for the following phases of the computation:

- Step 1 and Step 3 which is where the main computation occurs in all the nodes including lock synchronisation.
- Step 4 which is the first barrier.
- Step 5 which is the computation only in node 0.
- Step 6 which is the last barrier.

Figure 4.6 presents the data obtained from node 0, which shows an abnormal barrier time increase starting from 16 nodes over TreadMarks, but a minor barrier time increase over the home-based system. We also measured per-node time breakdown on 32 nodes over TreadMarks and the home-based system. As shown in Figure 4.7, the first barrier time in node 0 took more than 98% of the total execution time due to a hot spot. This data means that even though node 0 gets into the first barrier first, it can not get out of the barrier until all the faulting nodes have received diffs from node 0. As the number of nodes increases, PNN over TreadMarks is likely to have a hot spot in node 0 anywhere between Step 1 and Step 4. In our experiment, it occurred at Step 4.

Note that in Figure 4.7, the last barrier time is not missing but not very noticeable due to the very fracture of the time. Also, note that the barrier time in each node is taken between entry of barrier and exit of barrier. So some node can exit earlier than others. In that case, barrier time takes more since it enters barrier earlier and exit at the same time with all other nodes.

In the home-based protocol, in response to the page fault, each node requests the pages from the homes of the pages. The page requests are likely to be distributed over multiple nodes. Compared to the homeless protocol in which one node takes all the responsibility for diff request service, the page request service is distributed over multiple nodes. Figure 4.8 shows that node 0 does not suffer from a hot spot. This resulted in good performance. In the neural network application, the page request service is distributed over the first 5 nodes. Also page request service time is much shorter because sending a page can use a much smaller message than sending accumulated diffs in the homeless protocol.

---

1Label 1 to 32 in Figures 4.7 and 4.8 means node 0 to 31 respectively
Figure 4.6: Comparison of Time Breakdown for Neural Network in Node 0 between TreadMarks and the Home-based DSM System

Further proof that the hot spot affects the performance can be found by comparing the number of resent request packets. In the implementation of the two protocols, if a request is not serviced in one second, the request is resent. As shown in Figure 4.9, the number of resent request packets steeply increases with the number of nodes after 16 nodes in the neural network application over the homeless protocol, but there are none in the home-based protocol.

4.3.5 Garbage Collection

Another negative effect that the homeless protocol can have is the need for continuous garbage collection. In the Barnes-Hut application, we found that the garbage collection affected the performance significantly, in particular, over more than 16 nodes. To see how the garbage collection affects the performance negatively we measured garbage collection time over 8, 16 and 32 nodes.

In the implementation of TreadMarks, the process of garbage collection happens inside the barrier synchronisation implementation if memory space for diffs and intervals exceeds some limit. There are 9 barrier synchronisations during the main computation.
Figure 4.7: Time Breakdown for Neural Network in 32 nodes over TreadMarks

Figure 4.8: Time Breakdown for Neural Network in 32 nodes over the Home-based DSM System
in Barnes-Hut. Out of the 9 barriers, garbage collection was run for 8, 16 and 32 nodes 3, 3 and 5 times respectively over TreadMarks. Because garbage collection requires barrier-like communication, it becomes very costly as the number of nodes increase as shown in Table 4.3. For 32 nodes, the barrier time took more than 377 seconds out of the total execution time (411 seconds) over TreadMarks. Out of the 377.6 seconds of barrier time, garbage collection time took 274.38 seconds, which was almost 70% of the execution time.

On the other hand, Barnes-Hut over the home-based protocol showed no garbage collection and much better performance. The short life span of a diff in the home-based protocol allows small and temporary memory space for diff storage compared to large and permanent memory space in the homeless protocol. Thus in the home-based protocol, memory space for diffs are relatively constant and not getting bigger as can occur in the homeless protocol.
4.3.6 Diff Accumulation

The diff accumulation problem can be found in PNN and IS-B over the homeless protocol. The two applications are known to be migratory applications. In PNN, as explained in Subsection 4.3.4, local weight changes and node changes calculated in each node are summed one at a time through lock synchronisation. Through this serial summing process, each node completely overwrites the partial global changes. To get the partial global changes, all previous local changes in the form of diffs are applied in the happens-before order. In PNN, two local changes are accumulated to create the two global change matrices, the weight and node changes matrices, which occupy four pages and one page, respectively.

Similar to the illustration in Figure 4.3, diff accumulation occurs in PNN. In PNN, four pages are completely overwritten in the critical section in which each set of local weight changes is added into the global weight matrix. Thus, each node produces a total diff size of 16Kb (4Kb * 4 pages) in the critical section. Therefore, for example, in the worst case under 32 nodes, the last lock owner receives accumulated diffs of size 496Kb (16Kb * 31 nodes) from the previous lock owner within the critical section.

Diff accumulation in IS-B occurs differently than in PNN. In IS-B, a global key density array is divided into N components, one for each node. See Figure 4.10 for an illustration. Since Figure 4.10 assumes four nodes, the global array is divided into four.
In the main loop of the bucket sort function, each node first adds its local key density into the global key density array to make the complete global key density array. To make a complete global key density array, $N$ iterations are required. In the figure, the first to fourth iterations illustrate the construction of the global key density array. To prevent a data race in this construction process, each node accesses a different part of the global key density array in every iteration, as illustrated in Figure 4.10. Diffs created from each page are accumulated as the iteration continues. For example, at the fourth iteration, each node requires three previous diffs made by the other three nodes at the different iteration. Even if each node requests the three diffs from the neighbouring node which has all the required diffs, a diff request for accumulated diffs may require more than one packet because the size of the required diffs exceeds the size of the maximum transfer unit governed by the network layer. As the number of nodes increase, the diff accumulation problem becomes worse since a node has to fetch the updates made by the other nodes. For example, with 32 nodes, in the last iteration of the construction process of the global key density array, each node requires 31 diffs made in the other 31 nodes at the previous iterations.

Ironically, diff accumulation is a strength of the LRC implementation, if it is less severe. By sending a message containing accumulated diffs it can reduce the number of messages as well as data traffic. However, in migratory applications, the lazy protocol action causes coherence-related data to be accumulated rapidly, making the protocol less scalable. On the other hand, a prompt protocol action in the home-based protocol
Table 4.4: Comparison of Data Communication Traffic between Homeless and Home-based Protocols

at the synchronisation time prevents the diff accumulation.

4.3.7 Home Effect

One of the weaknesses of the home-based protocol can be the unnecessary data communication traffic if the home assignment of the pages is not matched well with the memory access pattern of the application. Our performance results for SOR and Gauss showed poor performance over the home-based protocol because our fixed home assignment was not well matched with the memory access patterns. The fixed home assignment was done using the following equation:

\[
\text{page's home} = \text{page number} \mod \text{number of nodes}
\]

where “\mod” is the modulus operator.

SOR and Gauss are typical matrix problems that can be parallelised by dividing the matrix. Each divided sub-matrix is processed on each node, and the sub-results are collected at the barrier time. The two applications show a regular memory access pattern meaning that each node always works on the same sub-matrix. This kind of memory access pattern should not generate much data traffic between nodes other than the synchronisation for the boundary of each sub-matrix. However this is not the case for the fixed manner of home assignment. According to the home-based protocol, each node still needs to update the homes of the pages that have been changed. This will generate unnecessary data traffic which could be avoided if the pages were assigned to the home nodes that exclusively access the pages.
4.3.8 Data Communication Traffic

In Table 4.4, the number of messages and the amount of data traffic can be compared between the two protocols. In Barnes-Hut we see that even though the amount of data transferred over the home-based protocol is more than for the homeless protocol, the performance over the home-based protocol is better than over the homeless protocol as shown in Table 4.2 except for the 32 node computation.

The homeless protocol generates many more messages in Barnes-Hut because the memory access pattern shows frequent write-write false sharing. In Barnes-Hut over the homeless protocol, upon a page fault a node sends many diff requests to nodes that previously wrote on the page. This fine-grained diff request has no positive aggregation communication. In the home-based protocol, diffs which belong to the same node are aggregated and sent as one message, which saves many messages.

As for PNN, the number of messages and the amount of data traffic between the two protocols are not much different. However, the performance over the home-based protocol is much better than over the homeless protocol. From the speed-up and data traffic of PNN, we can conclude that burst data traffic over a short time, and burst requests serviced by one node, can be critical to overall performance as well as the amount of the overall data traffic, as explained in Subsections 4.3.4 and 4.3.5.

Nevertheless, in general, the greater the amount of data and the number of messages, the worse the performance, as is clearly shown in SOR and Gauss over the fixed home assignment. This unnecessary data traffic, due to the wrong home assignment, shows that good home assignment is critical in the home-based protocol. In Chapter 6, we present a dynamic home assignment technique by which a home node is dynamically assigned to the most frequently accessed pages.

The homeless protocol showed more efficient data traffic in TSP and 3D-FFT. In TSP, a fine and frequent memory access pattern within critical sections generates more data traffic and messages over the home-based protocol. In TSP, there is no severe diff accumulation because its memory access pattern is very fine. Without severe diff accumulation, diff-based memory update in the homeless protocol is more efficient, since only required diffs are transferred. On the other hand, page-based update in the home-based protocol has to update a home node first with diffs. Then a home node has to send a whole copy of the page even though only part of the page is modified as in TSP. In 3D-FFT, since the size of the shared memory is not too large as in SOR and Gauss, the negative home effect is not too severe to degrade the performance.
4.3.9 Limitation of the Performance Analysis

In Section 1.3, we defined the scalability in DSM as a consistently improved speed-up as the number of nodes increases. However in our experiment results especially over more than 16 nodes, all the benchmark applications actually showed no improvement of speed-up except one application B-H over HB. With these results, it is very difficult to say that one protocol is more scalable than the other.

Our results do show that the home-based protocol exhibits resistance to rapid slowdown as the number of nodes increases. However, although resistance to rapid slowdown is certainly necessary to achieve scalability, this is not a good measure of scalability when the actual speed-up decreases while adding more nodes. The rationale of this is as follows:

In order to determine which protocol is more scalable, it is very important to define scalability. There have been many attempts to define it (Hill, 1990; Goodman, Hill and Woest, 1989).

One example of the definitions is as below by Goodman (Goodman et al., 1989).

"Scalability is an attribute of systems that can be implemented over a range sizes. Size is measured by system resources such as processors. Asymptotic (or theoretical) scalability compares the performance of a system of fixed size which one that is arbitrarily large, while limited (or practical) scalability is concerned with system growth by small factors, less than 100 . . .”

In this paper, Goodman suggested that a scalable algorithm is a parallel algorithm whose serial portion requires constant execution time, regardless of problem size, s, and whose parallel portion contains parallelism at least proportional to f(s), the algorithm’s serial complexity.

In our experiment results, the parallel portions in the benchmark applications do not contain enough parallelism proportional to the algorithm’s serial complexity when running over more than 16 nodes as indicated by negative speed-up. Therefore, with the performance results over 16 and 32 nodes, it is not possible to evaluate which protocol has better scalability.

4.4 Conclusions

In this chapter, the homeless and home-based protocols were compared, and the strengths and weaknesses of the two protocols were described. Our performance eval-
uation showed that the home-based protocol has the following strengths:

- less hot spot susceptibility,
- no garbage collection being required and
- no diff accumulation.

Also, the positive home effect reduces twinning and diffing at a home node if a home node is assigned to frequently accessed pages. However, to promote the positive home effect, the protocol needs to assign a home node dynamically according to a memory access pattern of an application.

We also found out that even though the homeless protocol has some weaknesses such as hot spot susceptibility, garbage collection being required and diff accumulation, the protocol can be more efficient with fine-grained lock-based applications without severe diff accumulation. This is due to the fine-grained diff-based update which only transfers required diffs rather than a whole page in the home-based protocol.

Based on our findings, it is desirable to have an optimized DSM system which maximizes the strengths and minimizes the weaknesses of the two protocols. In Chapters 5 and 6, a hybrid protocol which maximizes the strengths of the two protocols, and performance improvement techniques which minimize the weaknesses of the two protocols, are presented.
Chapter 5

Hybrid Home-based EAC Protocol

5.1 Introduction

In this chapter, we describe the design and implementation of our all-software page-based DSM system in detail. Our DSM system is different from currently available software DSM systems in many ways. First, we claim that we have implemented a hybrid DSM implementation using the homeless and home-based protocols, which is novel in this DSM research area (Yu, Werstein, Purvis and Cranefield, 2005). Our hybrid DSM implementation reduces the number of messages and page faults in a critical section, thus improving DSM performance. Second, we have developed two performance improvement techniques, diff integration and dynamic home migration (Yu, Werstein, Cranefield and Purvis, 2005), which are described in detail in Chapter 6. As shown in the performance evaluation in Chapter 7, we have achieved significant performance improvements in four out of the seven applications we tested compared with TreadMarks. The other three applications showed comparable performance between TreadMarks and our system.

In Chapter 2, we explained that a relaxed memory consistency model provides more opportunities for implementing an efficient coherence protocol. This is why many relaxed memory consistency models have been proposed since SC was proposed for defining the correct interleaving order of parallel executions in a multiprocessor system. The primary motivation behind these relaxations is to provide a more optimized and efficient implementation of a memory system while preserving SC by making parallel program execution data race free. This optimization and efficiency can be obtained in two ways: first, reduction of the constraints that a memory system must follow in order to provide the correct results of operations, and second, by efficiently implementing a
model such as using an efficient and scalable coherence protocol.

For our DSM system, we chose ScC as the base memory consistency model since it gives the least requirements for a memory system as well as automatic binding between a synchronisation object and its guarded data. Automatic binding of the relationship reduces the burden to a programmer. In essence, our novel hybrid DSM protocol is developed by exploiting the relaxed consistency constraints of ScC even though the conditions for programs to execute correctly differ slightly between ScC and our protocol as will be explained in Section 5.4. Because of this difference, our memory consistency model is different from ScC and we call ours the exclusive access memory consistency (EAC) model.

As a coherence protocol implementing the model, we chose the home-based protocol as a base protocol since the protocol has many advantages and simple to implement compared to the homeless protocol as discussed in Chapter 4. However, the diff-based update, which has been found to be one of the strengths of the homeless protocol, is added to our protocol to be employed in a coherence protocol used in lock synchronisation. We call our protocol Hybrid Home-based EAC (HHEAC) protocol.

The HHEAC protocol uses, in essence, the home-based coherence protocol that implements EAC. The HHEAC protocol is developed by taking advantage of the relaxed consistency constraints of EAC. As LRC (Keleher et al., 1992) lazily implemented RC, our hybrid home-based protocol lazily implements the home-based protocol. The benefits of our hybrid implementation are fewer page faults and fewer numbers of messages compared to the first implementation of ScC (Iftode et al., 1996), thereby improving DSM performance.

5.2 Protocol Design

5.2.1 Exclusive Access Memory Consistency Model

Applying a relaxed memory consistency model to DSM means not only enabling memory access optimization techniques such as pipelining and buffering but also reducing data communication between physically distributed nodes. The reduction of data communication in DSM is more important to a software DSM system than memory access optimization since the main bottleneck of a software DSM system is coherence-related data communication. The reduction of data traffic and the number of messages is possible with a relaxed memory consistency model due to lazy coherence actions. Also,
since only the last values of shared variables updated between two consecutive synchronisation points are transferred, the number of messages can be reduced.

EC (Bershad and Zekauskas, 1991) and ScC (Iftode et al., 1996) are two of the most relaxed memory consistency models currently known. These two models take advantage of the memory access pattern in a critical section, which is sequential, exclusive and relatively predictable compared to barrier synchronisation. According to EC or ScC, only the data inside a critical section must be the most up-to-date before entering into the critical section. The major difference between them is that under ScC, DSM systems automatically define the relationship between a synchronisation object and its data, but under EC a programmer should provide the relationship.

Our novel hybrid home-based protocol is developed by exploiting the relaxed consistency constraints of ScC. The relationship between data and a synchronisation object is detected automatically by the system, based on previous memory access patterns inside a critical section. Our implementation is different from currently known implementations of ScC. For example, the first ScC implementation (Iftode et al., 1996) was developed with the support of automatic update network hardware and used the invalidation protocol. Using automatic update network hardware is certainly useful to enhance DSM performance. However, this specialized hardware is not easily available in many places and is more expensive than COTS hardware. Our HHEAC has been implemented fully in software for use with COTS hardware. The coherence protocol of HHEAC uses a hybrid coherence protocol which uses the update protocol during lock synchronisation and the invalidation protocol during barrier synchronisation.

Assuming a program is data-race free, the conditions of a correct program under ScC are specified as follows (Iftode et al., 1996):

1. Lock-protected modifications to shared data are not expected to be visible at node N before at least one of the protecting locks is acquired by N.

2. Modifications to shared data that are not protected by a lock, are not expected to be visible at node N before the next barrier.

If a DSM application meets these conditions, it can be executed more efficiently under an memory coherence protocol that exploits the benefits of ScC. These conditions also implicitly restrict memory accesses executed in parallel. From this implicit restriction, we can deduce the following facts of program behaviour if a program maintains the conditions:
1. From the first condition, we can deduce that the last lock owners have the latest values of their guarded shared variables.

2. From the second condition, we can deduce that if shared variables are not protected by any lock, memory accesses to those variables are mutually exclusive between two consecutive barriers in all nodes.

The first deduced memory access pattern is obtained from the observation that when a node finishes a critical section, which is at the time of a lock release, the node should have the most up-to-date values of the shared variables accessed inside the critical section. When another node would like to enter into the same critical section, it sends a lock acquire request to the previous lock owner node, which has all the up-to-date values of the guarded variables. Therefore, communication between the previous and current lock owners is sufficient to guarantee coherence at least inside a critical section. That is, if the acquiring lock owner receives the most up-to-date guarded variables from the previous lock owner, this would be sufficient to guarantee coherence inside the critical section.

Assuming that shared variables guarded by different locks are mutually exclusive, it is simpler to implement memory coherence by grouping shared variables by the lock that guards them. Therefore, per-lock shared variables are grouped and considered as a logical memory unit that is propagated during the lock transition time. The implementation of a memory coherence protocol then becomes much simpler because it only needs to keep the logical memory unit up-to-date.

The second deduced memory access pattern is implicitly inferred from the second condition of a correct program under ScC. The second condition restricts each node to access exclusive shared variables during executions in non-critical sections. This means that modifications to shared variables occurring in non-critical sections are not expected to be transferred to other nodes within the current barrier session. Only modifications occurring in a critical section are transferred to other nodes that require the modifications.

From these observations, we can simplify an implementation of memory coherence by dividing all shared variables into two groups: critical section (CS) variables and non-critical section (NCS) variables. CS variables are the shared variables that are guarded by locks. Shared variables other than CS variables are NCS variables. CS variables are again divided into subgroups so that each subgroup contains shared variables associated with a specific lock.
EAC is different from ScC as exemplified in Figure 5.1. The upper diagram of Figure 5.1 illustrates that shared memory is divided into many groups whose shared variables are mutually exclusive under our EAC. There should be no memory overlap between groups, otherwise it will complicate a coherence protocol. This memory snapshot can be totally different after each barrier session.

This memory grouping gives us a simpler implementation of a coherence protocol. The memory system is required only to keep shared variables in each group up-to-date. This simpler implementation also means that possible future memory access patterns are more restricted, thus more predictable. This simpler approach to implementing memory coherence is not possible under LRC.
As shown in the lower diagram of Figure 5.1 in ScC, scope overlapping can be allowed even though it can make automatic variable to scope binding very difficult or impossible. However, in EAC as shown in the upper diagram of Figure 5.1, memory division cannot be overlapped strictly. This difference between ScC and EAC comes from using the memory coherence protocols differently. In ScC, modified shared variables inside the same scope should be visible to other nodes before they enter the same scope. To make sure this constraint is satisfied, memory systems ensuring ScC should update the home node immediately before other nodes enter the same scope.

Therefore, if the program is data race free and synchronization primitives are appropriately set to prevent a data race, this makes sure that the program is correctly running under ScC. On the other hand, under our memory consistency model EAC, memory systems can defer the update of shared variables modification until the next barrier time. Because of this deferred memory update, if exclusive memory division is not met, inconsistent values of shared data can be seen between nodes as explained later in Figures 5.8 and 5.9.

Note that the mutually exclusive memory division of shared variables into many subgroups, as shown in Figure 5.1, should not be considered as a totally new burden to a programmer, because data race free execution is already imposed on shared memory programming implicitly in order to avoid a data race. When a programmer develops an application in shared memory programming, the programmer should ensure that all shared variables have no data race. Our model simplifies the enforcement of data race free execution in shared memory programming.

5.2.2 Memory Coherence Protocol

We explained in the previous section that the ScC model not only imposes the least constraints on a memory system but also provides an opportunity to have a simpler implementation of the coherence protocol. The purpose of a coherence protocol is to make sure that whenever a node reads a shared variable, the returned value should comply with the conditions of the chosen memory consistency model.

In Chapter 4, we also discussed the strengths and weaknesses of the homeless and home-based protocols respectively. We concluded that the home-based protocol has many advantages compared to the homeless protocol, while the homeless protocol is more efficient for fine-grained lock-based applications. In other words, the diff-based fine-grained memory coherence protocol used in the homeless protocol can be more efficient if there is no severe diff accumulation or hot spot. Similarly, the home-based
protocol is not efficient if lock synchronisations are required frequently to mutually access shared data, since the protocol has to not only update the corresponding homes at every release, but also fetch a whole copy of a faulting page from the home node upon a page fault.

We also found that due to regular memory access patterns shown in a critical section, it is more advantageous to use the write-update coherence protocol rather than using the write-invalidation protocol. As a result, we have developed a hybrid coherence protocol to combine the advantages of the homeless and home-based protocols. The hybrid coherence protocol means that the invalidation and update coherence protocols are employed at different times. During lock synchronisation, the update protocol is employed, while during barrier synchronisation the invalidation protocol is employed.

Barrier synchronization is used to stop all the nodes at the particular barrier point until all the nodes reach that barrier point. This is global synchronization meaning that all the nodes are involved. Generally, a barrier is used to let all the nodes know the most up-to-date values that were updated by other nodes. On the other hand, lock synchronization is used for preventing a data race and the guarded data are generally fine-grained compared to the more coarse grained values involved in a barrier. Moreover lock synchronization is not global meaning that a lock is acquired and released between two nodes.

Based on the above explanation about barrier and lock synchronizations, the update protocol with homeless diff-based protocol during lock is effective because of the predictability of the memory access pattern of guarded data. On the other hand, the invalidation protocol with the home-based protocol is effective for barriers since a barrier is a global process, so the invalidation protocol can minimize the data communication traffic.

In summary, during lock synchronisation a data access pattern is relatively predictable and fine-grained, but during barrier synchronisation, it is more unpredictable and global. The advantage of using the write-update protocol during lock synchronisation is the generation of relatively accurate data updates for the next lock owner, which will eventually reduce the number of messages and page faults at the next lock owner. More efficiently, the required data is piggybacked with the lock ownership transfer. During barrier synchronisation, stale data is invalidated so that the most up-to-date data can be obtained from a home node.

Figure 5.2 illustrates a comparison between previous home-based SeC protocol implementations and our hybrid home-based EAC protocol implementation. Compared
with previous implementations of the home-based protocol, our protocol is more “lazy”,
because it does not send diffs at a lock release time in order to update home nodes,
but delays home update until the next barrier time. Our implementation takes advant-
geage of ScC more aggressively than previous ones in the sense of “laziness”. The lazy
home update is still correct under the ScC model since other nodes should not read
diffs made in non-critical sections before the next barrier and sufficient diffs created in
critical sections are transferred to the next lock owner by the update protocol.

5.3 Implementation

5.3.1 Main Data Structures

To efficiently implement the HHEAC protocol, we distinguish a non-critical section
(NCS) diff and a critical section (CS) diff as stated in the previous section. Figure 5.3
illustrates NCS and CS diff creations during parallel execution. A NCS diff is a diff
created in a non-critical section, and a CS diff is a diff created in a critical section.
Intuitively, according to our programming model, NCS diffs made between two consec-
utive barriers should be mutually exclusive to one another. NCS diffs are kept until the
next barrier when non-home nodes send their non-home NCS diffs to the corresponding
home nodes. In this way, all NCS diffs are safely preserved at the corresponding home
nodes. As for CS diffs, they are sent to the next lock owner during a lock ownership
change. Intuitively, the last lock owner before a barrier should have the most up-to-
date data that the lock protects. Therefore, the CS diffs from the last lock owners
are sufficient to construct the most up-to-date CS data. Upon arrival at a barrier, the
last lock owner for each lock sends its CS diffs to the corresponding home nodes unless
the last lock owner is the home of the last CS diff. In the case that a node owns the
same lock consecutively, diffs created from the same page are numbered so that they
are applied at the next lock owner in the happens-before partial order.

To suit all of these requirements, the \texttt{diff\_info} data structure was developed, which
is a linked list (Figure 5.4). A \texttt{page} data structure used in TreadMarks is extended to
link to the \texttt{diff\_info} data structure. The page data structure was used to represent a
shared virtual page. The fields of the \texttt{page} data structure indicate virtual addresses
of a page and a twin, a page’s current state and a page’s home node. The \texttt{page} data
structure adds a \texttt{diff\_info\_t} array in which two elements point to the latest CS and
NCS diffs of the page respectively, as shown in Figure 5.5. Whenever a copy of a page
Figure 5.2: Difference between Previous Home-based and Our Hybrid Home-based Implementations
is modified, a diff is created to preserve the modification. The new diff is set at the start of the list before the previously created diff. In this way, each shared page can preserve its CS and NCS diffs in an order consistent with the happens-before partial order.

Below, we briefly explain the fields of the diff structure.

- **next** is a pointer that points to the diff previously created from the same page.

- **lock_id** indicates the ID number of the lock that protected the critical section when the diff was made. In the case of a NCS diff, lock_id is not required.

- **again_no** is needed to distinguish possible multiple diffs created from the same page during the same critical section in the same node without transferring the lock ownership to other nodes. For example, we can distinguish two diffs from the same page if their lock_ids are different. But if lock_ids of two diffs are same, the two diffs are distinguished by again_no. Also, again_no is needed to send diffs that were created consecutively in the same critical section in accordance with the happens-before order. That is, the larger the number, the later the diff was applied at the next lock owner.

  When a node acquires a lock from another node and make a diff at the subsequent release time, again_no is set as one. It increases by one whenever a node reacquires the same lock locally and make another diff from the page that was diffed at the previous critical sections. Whenever a node receives lock ownership from another node and creates a diff at the release time, again_no is reset to one.

- **diff** is a pointer to point the actual diff address.

- **diff_size** is the size of the diff.
typedef

struct diff_info *diff_info_t;

struct diff_info {
    diff_info_t next;
    unsigned short lock_id;
    unsigned short again_no;
    caddr_t diff;
    unsigned short diff_size;
    unsigned char last_received;
};

Figure 5.4: diff_info Data Structure (C programming language)

- *last_received* is a flag that indicates whether the diff was received from other nodes (TRUE) or created by the local node (FALSE). This flag is needed to distinguish whether *again_no* was increased by a remote node or a local node. When consecutively created diffs from the same page are received from the previous lock owner, they are applied to the local page in the happens-before order. That is, an earlier created diff is applied first. These diffs are not destroyed until a new diff from the same page is created by the local node. When a new diff from the same page is created, then all the previous diffs are integrated into the new diff thanks to our diff integration technique, which is explained in detail in Chapter 6. When a local node acquires the same lock consecutively and finally writes on the page, the local node could assume wrongly that the previous diffs received from the previous lock owner were created by the local node. However due to seeing the *last_received* flag as TRUE, the local node starts *again_no* as one.

Figure 5.5 illustrates a possible diff storage of a page. In the figure, the CS diffs consist of four diffs; two of them were created within a critical section guarded by lock 1 and the other two were created in two critical sections guarded by locks 2 and 3, respectively. The top two CS diffs were received from the previous lock owner.
during lock 1 acquire time, which shows that the previous lock owner acquired lock 1 two times consecutively. In a lock-based DSM application, if there is no write-write false sharing there should be only one NCS diff between two consecutive barriers. Also, one NCS diff will be created at a barrier time. In the figure, two NCS diffs are already created meaning that write-write false sharing happened and was detected during lock synchronisation. Thus false sharing writes should be recorded as an NCS diff. Otherwise those writes can be lost or mixed within a CS diff, which would cause memory inconsistency. Therefore in the figure, if it is not the time of a barrier, the two NCS diffs represent two false sharing writes on the page.

5.3.2 Page Fault Handling

In a page-based software DSM system, page fault handling is the means to maintain memory consistency. Page fault handling includes not only how to handle a page
fault according to the memory consistency requirements, but also how to set the page protection to invoke an appropriate coherence action later.

In the implementation of our system, the purpose of page fault handling is two-fold. The first purpose is to fetch up-to-date data due to an access of stale data. This is accomplished by invalidating all the dirty pages that occurred between two consecutive barriers, before departing from a barrier. The exception is that pages in their home nodes are not required to be invalidated since home nodes should have the most up-to-date copies of their pages.

The second purpose is to record local writes on shared memory. This is accomplished by making a page's protection write-protected. A write access on a write-protected page causes the system to record future writes. Note that even though our system employs the home-based protocol, a home node is required to record local writes on its home page during a critical section. This is because the homeless and write-update protocols are applied during lock synchronisation. To meet the home-based protocol requirements, each last lock owner in a barrier session should send its CS diffs to the corresponding homes to make the homes up-to-date.

Since our implementation distinguishes CS writes and NCS writes, these two different types of writes are recorded differently. In order to detect all CS writes, before entering into a critical section, a node should make all writable copies of pages write-protected. Otherwise, modifications on writable pages would not be detected and would be lost.

Figure 5.6 shows the page state transition in our implementation. This diagram is to explain some consideration for the HHEAC implementation, which is needed to make the complexity of the page state transition simpler. Even though only three page protection states are supported by the virtual memory system (INVALID, READ_ONLY, READ_WRITE), we divide them into 14 page states to suit our needs. In the figure, if the page state is in one of the states with a “read-only” suffix (2, 3, 4, 6, 11, 12, 13), then READ_ONLY protection is applied. Similarly, if the page state is in one of the states with a “read_write” suffix (1, 5, 7, 8, 9, 10), the page’s protection is READ_WRITE. The reason for having many fine-grained page states which have the same page protection is due to applying different page fault handling processes depending on a variety of page state variables.

The need for different page fault handling arose because we need to record NCS dirty page lists and each CS page list for every lock. We also created a page state

\[\text{1The numbers in the boxes are used for explanation only}\]
ncsRPR: Receiving page request during non critical section execution

csRPR: Receiving page request during critical section execution

RPFH & RF: Receiving a page from home and read fault

RPFH & csWF: Receiving a page from home and critical section write fault

RPFH & ncsWF: Receiving a page from home and non critical section write fault

LA: Lock Acquire

LR: Lock Release

lpLA: Lock CS pages received during LA

csRF: Critical section read fault

csWF: Critical section write fault

Figure 5.6: Page State Transition Diagram
called exclusive_read_write to optimize coherence actions for single writer applications in which each node exclusively accesses a different part of shared memory. More details about exclusive_read_write are given in Chapter 6. All of these needs are contributed to creating the 14 page states. Upon exiting a barrier, all home pages are started as exclusive_read_write. On the other hand, non-home pages are started as invalid.

In our page fault handler, we divided all page faults into two in order to simplify coherence actions:

- page faults occurring within CSs.
- page faults occurring within NCSs.

The page fault handler can determine whether a page fault occurred within a CS or a NCS by a flag which is set as CS when a lock is acquired or NCS when a lock is released. If a page fault occurred within a CS, the handler first finds the lock-id that protects the CS. Each lock-id is associated with the pages protected by that lock. The handler collects information about the relationship between a lock-id and pages when a write page fault happens inside a CS. This information is also obtained from a previous lock owner sending previous modifications that occurred inside the CS due to the write-update protocol used in lock synchronisation.

### 5.3.3 Lock and Barrier

Our implementation of lock synchronisation centers on managing CS diffs by lock-id. Whenever lock ownership is transferred, its CS diffs are also transferred. A group of CS diffs protected by the same lock represents the most up-to-date memory in the corresponding critical section. Thus, it is important to keep all the modifications inside a critical section in order for the next lock owner to access the most up-to-date CS variables.

Efficient diff reduction is also implemented due to the diff integration technique. The diff integration technique replaces many old happens-before diffs created from the same page with the latest diff without violating the correctness of a program result. The diff integration technique is described in detail in Chapter 6.

As CS diffs are grouped by lock-id, NCS diffs are grouped by node-id. Each node only concentrates on maintaining its own exclusive NCS variables as restricted by our programming model. During barrier synchronisation, all the modifications made by all the nodes should be known to one another. These modifications include NCS diffs.
as well as CS diffs. Our programming model, as stated earlier, effectively allows for an efficient implementation of the management of modifications by means of memory grouping as shown in Figure 5.3.

We do not describe general lock and barrier processes algorithms or implementation details used in a software DSM system. They are briefly described in Sections 3.5.1 and 3.5.2 respectively. Below, we only explain our lock and barrier implementations specific to HHEAC by describing coherence actions during lock acquire, lock release and barrier operations.

**Lock Implementation in HHEAC**

We do not describe general lock process algorithms or implementation details used in a software DSM system. They are briefly described in Section 3.5.1. Below, we only explain our lock implementation specific to HHEAC by describing coherence actions during lock acquire and release operations.

We start from a lock acquire when a node tries to enter into a critical section. We assume that the lock is owned by another node. We also assume that during the lock acquire, before getting into a critical section, all the required diffs are transferred from the previous lock owner. When the acquiring node receives the diffs, it stores them, and applies them to the corresponding pages in order to make all the previous modifications that occurred inside the CS visible before entering the CS. At this time, page numbers of the diffs are recorded in a lock page list which is implemented as a two dimensional array. The first dimension refers to the lock-id number and the second dimension refers to the corresponding lock-id’s page numbers. Each node, independent of other nodes, keeps a lock page list array.

Before the diff is applied to the corresponding page, the page state of the page should be checked to see whether the page has READ_WRITE protection. If it has, then, before applying the diff to the page, modifications to the page should be preserved as an NCS diff. Otherwise the modifications before the critical section are lost. Twins of the pages corresponding to the transferred diffs are created. Also the pages should be write-protected in order to preserve any writes on the pages.

During executions inside a critical section, if a page other than already known lock pages transferred from the previous lock owner is modified, the page number is also inserted into the lock page list.

At the following lock release time, a releasing node creates the required diffs for the next lock owner regardless of whether the next lock owner is determined or not. If a
Figure 5.7: Illustration of the Lock Synchronisation Process in HHEAC

current lock owner received a lock request from another node before the release, the node simply records the requesting node’s number and contacts the requesting node directly at the release time. Since each node maintains a lock page list for every lock object, a releasing node can easily determine which page’s diff(s) is required by the next lock owner by looking at the lock page list. After identifying the lock pages, the CS diffs of the pages are also easily found by lock_id and again_no. Not all the accumulated diffs of the pages are transferred to the next lock owner due to our diff integration. Transferring diffs are determined as follows.

- New diffs created inside the CS.
- Diffs of pages that were not modified inside the CS but included in the lock page list.

It is obvious that every new diff created inside the CS should be transferred since they represent the most up-to-date CS variables. A newly created diff might consist of
new values as well as old values that were written by the previous lock owners at the time of the acquire due to the diff integration technique. For example, in Figure 5.7, a newly created diff of page 0 at the time of the release in N1 consists of value 1 for a shared variable $a$ that was written somewhere at the previous lock owners, and value 0 for a shared variable $x$ that has just been written at N1. On the other hand, a newly created diff of page 1 at the time of the release in N0 consists of only the new value of $y$ that has just been written at N0. The second type of diff also should be transferred since these diffs also represent the most up-to-date CS variables which were modified somewhere by the previous lock owner(s). For example, in Figure 5.7, page 2 is not modified inside the critical sections. However, an old diff of page 2 transferred from the old lock owner still represents one of the required CS variables, $z$. Therefore, a diff containing $z$ should be transferred to the next lock owner. In many practical cases, CS variables that are protected by the same lock are accessed in every node. This is why the diff integration is effective for diffs created inside a CS.

Note that when a lock acquiring node requests a lock from the current lock owner, the lock-id of the lock is sent with the request, as shown in Figure 5.7. The lock-id is sufficient for the releasing node to find out which diffs are required for the acquiring node according to HHEAC. This is in contrast with a lock request in LRC, which should send a vector timestamp of the acquiring node in order to find out which shared variables are required for memory consistency. This implementation of finding required diffs by means of comparing vector timestamps is more complicated than finding required diffs by using the diff data structure, as explained in Subsection 5.3.1.

To determine which node is the last lock owner when a lock is acquired, the system sets the last lock owner flag of the lock to TRUE in the acquiring node. Similarly, when a lock is released, the system sets the last lock owner flag of the lock to FALSE in the releasing node.

**Barrier Implementation in HHEAC**

At the barrier time, all the nodes are synchronised and all modifications of shared variables occurring between the last and current barriers are made visible to all nodes. To help with this coherence requirement, shared variables are divided and grouped by lock-id in the case of CS variables or node-id in the case of NCS variables. As described earlier, whenever a lock is transferred, its CS diffs representing the most up-to-date CS variables are also transferred. Therefore, the last lock owner before the current barrier should have the most up-to-date values of the CS variables guarded by the lock. As for
NCS variables, each node preserves its NCS diffs in order to let other nodes see them after the barrier.

Since our coherence protocol is a home-based protocol, before exiting the current barrier, a home node should have the most up-to-date values of its own shared variables. To meet this requirement, at a barrier time, all nodes should update the home nodes if they have the most up-to-date values of shared variables assigned to other nodes.

Since we also add the dynamic home migration scheme at a barrier time, the barrier implementation becomes more complicated. The dynamic home migration scheme is described in detail in Chapter 6. Below, we present our barrier implementation algorithm without the dynamic home migration scheme.

1. Upon a barrier arrival, create NCS diffs of pages that are modified in a NCS.

2. A barrier manager node and other nodes act differently as follows:
   - A barrier manager node receives barrier arrival notices from others until all notices have arrived.
   - Other nodes send a barrier arrival notice to the manager. A barrier notice should also contain the page numbers of dirty pages that were modified in NCSs. If a node is the last lock owner of lock $i$, the page numbers in $lock_{\text{page list}}[i]$ are also included in the barrier notice.

3. Again, a barrier manager node and other nodes act differently as follows:
   - A barrier manager node, after gathering all the barrier notices from the non-manager nodes, creates a global dirty page list that contains the numbers of the modified pages in all nodes including the manager node. It sends an invalidation notice to every non-manager node based on the global dirty page list. It invalidates pages based on the global dirty page list. Home pages are not invalidated but become exclusive_read_write.
   - Other nodes receive an invalidation notice from the manager. They invalidate pages based on the invalidation notice. However, home pages are not invalidated but become exclusive_read_write.

4. Based on NCS and CS diffs, each node updates the home nodes of the diffs by sending the diffs. However, in the case of CS diffs, only those stored in the last lock owner for each lock are sent to their corresponding home nodes.
5. Once again, a barrier manager node and other nodes act differently as follows:

- A barrier manager node collects all the barrier departure requests from other nodes and sends the acknowledgements to them.
- Other nodes send a barrier departure request to the manager. They depart from the barrier once the acknowledgement is received from the manager.

The fourth step, the home update process, should be done after all nodes enter the barrier. Otherwise, multiple nodes could consider themselves to be the last lock owner of the same lock because a lock request could be sent to a node that had already entered the barrier and updated the home nodes. That is why this step is performed after completing a barrier arrival process. The last step is needed to prevent any node from accessing stale data in a home node after departing the barrier, due to the incompletion of the previous home update process by some nodes. Thus, all nodes must finish their home updates before any node departs from a barrier.

5.3.4 Home Update

The home update process is an essential part of a home-based protocol implementation. The most important requirement of the home-based protocol is that a home node should have the most up-to-date shared variables assigned to that home. Our lazy home-based protocol relaxes this requirement further than other home-based protocol implementations. Our protocol updates home nodes only at barrier time. This differs from other home-based protocol implementations that update the home nodes at every lock and barrier synchronisation time.

Due to more lazy updates, our implementation has to store more diffs before a barrier. For example, conventional home-based protocol implementations update home nodes immediately at a lock or barrier point. In these implementations, diffs sent to corresponding home nodes are minimal since they are created between the last and the current synchronisation points. On the other hand, our implementation stores diffs created between two consecutive barriers until the next barrier point. This means that if the number of NCS diffs created from the same page between two consecutive barriers is more than one, then they should be applied to a copy of the page in the home node in accordance with the happens-before order. In the case of CS diffs, as stated earlier, only the last lock owner sends CS diffs protected by the lock to the corresponding home nodes. Due to mutual exclusiveness between groups as shown in
Figure 5.1 the application order between groups does not affect correctness in terms of memory coherency. However, diff application order within a group must consider the happens-before order.

### 5.3.5 Packet Formats

All the requests used in the implementation are performed by means of a *SIGIO signal handler*. When the handler is triggered by a packet arriving on a receive socket, it calls the appropriate request handler according to the packet request. The major handlers are `lock_sigio_handler`, `barrier_sigio_handler`, `page_sigio_handler` and `home_update_sigio_handler`.

- The `lock_sigio_handler` receives a lock ownership request and takes care of sending the lock ownership as well as the required diffs to the requester.

- The `barrier_sigio_handler` which is always called in a barrier manager node, receives a barrier arrival notice. After receiving all the notices from other nodes, the barrier manager node sends an acknowledgement to every non-manager node with the required coherence information.

- The `page_sigio_handler` which is always called in a home node, receives a page request and sends the up-to-date copy of the page to the requester.

- The `home_update_sigio_handler`, which is always called in a home node, receives the diffs that keep the pages of the home node up-to-date and sends an acknowledgement to the sender.

Below, we illustrate the packet formats used in lock and barrier synchronisations, page requests and home update requests. All request packets include a packet head indicating the sequence number of the packet, a request type and a requester node number. The packet format for a lock ownership request requires the lock-id only. The lock ownership request is first sent to the lock manager. The lock manager determines whether it should be forwarded to the current lock owner or serviced by the manager depending on whether or not the manager is the current lock owner. An example of the format of the reply packet to the lock ownership request is as follows:

<table>
<thead>
<tr>
<th>head</th>
<th>page no (0)</th>
<th>number of diffs(2)</th>
<th>diff</th>
<th>diff</th>
</tr>
</thead>
</table>

90
In the example above, we assume that only one page (page 0) is modified and the lock releaser sends two diffs created from the same page to the lock acquirer. If a node does not acquire the same lock consecutively, there should be only a single diff from a page due to the diff integration. In the above example, two diffs from page 0 implies that the previous lock owner(s) acquired the same lock consecutively.

An example of the packet format for a barrier arrival notice is as follows:

```
| head | page no (0) | diff size (512) | page no (1) | diff size (1028) |
```

In the example above, we assume that the home migration technique is used and the non-barrier manager node has two dirty pages (page 0 and 1). The aggregated diff size is calculated, as explained earlier, by summing up the size of all NCS diffs of the page and last lock page CS diff(s).

An example of the format of the reply packet to the barrier arrival notice is as follows:

```
| head | page(0) | home(0) | page(1) | home(1) |
```

In the example above, two pages, pages 0 and 1, are assigned their homes, node 0 and node 1, respectively. The reply packet is sent to every non-manager node in order to notify them of new home assignments made by the barrier manager by comparing the aggregated diff sizes received from all nodes and choosing the node having the maximum aggregated diff size.

An example of the packet format for the home update request is as follows:

```
| head | page no (7) | no of diffs (2) | diff | diff |
```

In the example above, we also assume that only one page (page 7) is modified and two diffs of the page are sent to its home node. However, in a real DSM application, the total size of diffs to be sent could exceed the size of the maximum transfer unit. In that case, multiple home update requests will be sent to the home node. Upon sending and receiving multiple home updates, the two nodes involved should be careful of the boundary of each packet since from the packet format above the packing of a packet can end anywhere such as page number, the number of diffs or in the middle of the diffs. For example, if packing of a packet ends in the middle of the diffs, the number
of diffs should reflect only the number of diffs packed. The rest of the diffs of the same page should be packed at the subsequent packet and the number of diffs should reflect the number of diffs left after the previous packet.

We do not illustrate a page request packet format since it only includes the page number requested. Also its reply packet only includes a copy of the requested page.

### 5.4 Programming Model

A memory consistency model affects the programming model. For example, if a program is to be executed correctly under an EC-adopted memory system, the program should provide the relationship between a synchronisation object and data protected by the synchronisation object explicitly. In the same way, under ScC, even though the relationships are automatically detected by an ScC-adopted memory system, a programmer should take care of program correctness since the memory system guarantees memory consistency only for shared variables that belong to the current scope. As we stated earlier, HHEAC adopts the basic principles of ScC. In HHEAC, the relationship between shared data and lock objects is automatically detected by the system. Also, memory consistency is guaranteed only within critical sections. However under HHEAC, DSM programs should comply with slightly more restrictive conditions than programs under ScC. We define the conditions of a correct program under HHEAC as follows.

1. There should be no data race between consecutive barriers.

2. To prevent a potential data race, a lock or barrier synchronisation primitive should be used.

3. Shared data that are protected by a particular lock between two consecutive barriers should be mutually exclusive to other shared data that are protected by different locks.

Conditions 1 and 2 are the same under RC, EC, ScC and ours. But condition 3 is more restrictive than other models due to the update coherence protocol during lock synchronisation and the increased laziness in HHEAC. Even though condition 3 is more restrictive in terms of the programming model, generally it is advisable to follow condition 3 in order to remove potential data races due to nondeterministic lock transfer orders. Similar locking disciplines are described in data race detection techniques in

To make our programming model clear, consider Figures 5.8 and 5.9. In (a) of Figure 5.8 lock 2 in N1 protects x and y, but lock 1 and 2 in N0 protect x and y, respectively. Therefore, this violates condition 3 above. The solution is to insert a barrier as shown in (b) of the example. The intention of program (a) is that node 1 waits until node 0 executes \( z = 1 \). The barrier insertion as shown in (b) does this job.
In (a) of Figure 5.8, shared variable z is implicitly used as the synchronisation variable. However, this implicit synchronisation is prohibited in relaxed memory consistency models. Therefore, alternatively, we could provide another form of a synchronisation primitive, similar to a conditional variable by which a consumer node can see whether a producer node has finished a job.

In Figure 5.9, t and t1 are local variables while x and y are shared variables. We assume that their initial values are all zero. In Figure 5.9, (a) violates condition 3 as locks 0 and 1 in N0 protect x and y at the same time. Since diff-based updates during lock synchronisation in HHEAC are performed based on lock-id numbers, the CS variables, x and y, updated under the protection of lock 0 and lock 1 in N0 at the same time can have inconsistent values between nodes. For example in (a), if N0 executes the critical section under the nested lock earlier than the critical sections in N1 and N2, then N1 has x = 1 and y = 0 after lock_{rel}(0) while N2 has x = 0 and y = 2 after lock_{rel}(1).

Under HHEAC, a programmer should ensure that shared data protected by many different locks should have no overlap between two consecutive barriers as modified in (b) of this example in which lock 0 and 1 protect exclusively data x and y, respectively. Therefore x and y will always have a single consistent value in the last lock owners of lock 0 and 1, respectively. Alternatively, a programmer makes sure that the data integrity of x and y is protected by either lock 0 or lock 1.

Note that the HHEAC programming model does not prohibit all forms of a nested lock. However, in case the use of a nested lock that violates the third constraint is inevitable, HHEAC could detect the nested lock in an application and adapt to a more complicated implementation in order to avoid memory inconsistency under the current implementation of HHEAC. The added implementation would need diff version numbers for the diffs created under the nested lock and a nest flag indicating a nest lock. It also involves slightly complicated diff application in the home node if the undesirable nested lock is detected. Currently, our system does not implement this.

We also suggest as a future research that the violation of the third condition can be found by the protocol at runtime and immediately reported. Another solution would be using a tool that automatically rewrites a LRC/ScC program to comply with the EAC programming model.

A general recommendation for correct programming under HHEAC is as follows: first, a programmer should decide what shared data are required to implement a parallel algorithm. Second, a programmer should identify potentially conflicting accesses to
the shared data. If there are any, a programmer should use synchronisation primitives
such as a lock or barrier as appropriate. Finally, between two consecutive barriers a
programmer should check data inside each critical section to see whether the data are
consistently exclusive to other shared data protected by different locks.

5.5 Expected Benefits and Protocol Overheads

Since our design goal is to combine the homeless and home-based protocols, the ad-

dvantages of both protocols should be combined as well as the weaknesses of the two

protocols being removed. As described in Chapter 4, fine-grained diff update in the

homeless protocol is more efficient for lock-based fine-grained applications. However,

scalability of the homeless protocol is limited by large diff storage, hot spot, and diff

accumulation problems. The home-based protocol can solve the scalability problems.

However its performance is sensitive to a good home assignment which needs a dynamic

assignment scheme.

Our HHEAC protocol retains most strengths of the home-based protocol. Further-

more, it adds a dynamic home migration scheme. It also adopts fine-grained diff-based

update during lock synchronisation without diff accumulation thanks to the diff in-

tegration technique. Apart from the existing advantages of the two protocols, added

advantages of our implementation of the HHEAC protocol are as follows. Note that

the first and second benefits are mainly due to HHEAC, and third and fourth benefits

are due to the dynamic home migration and diff integration techniques respectively,

which are described in detail in the next chapter.

1. Compared with the conventional home-based protocol, an application’s compu-

tation is less often interrupted by home update requests from other non-home

nodes during lock release operations.

2. Execution time inside a critical section will be faster since there are fewer page

faults thanks to the update coherence protocol.

3. Performance slow-down due to a wrong home assignment will be removed by the

dynamic home migration technique.

4. Diff accumulation in migratory applications will be removed by the diff integra-

tion technique.
First, faster lock synchronisation is important since this process is sequentially executed one node at a time under contention. Reducing unnecessary home update requests results in not only a reduction of the number of messages but also less interruption of the main computation in the node to which the request is sent. For example, as described earlier, in the homeless protocol a node in a hot spot will be a bottleneck node which will receive all the diff requests from others at the same time. Similarly, in the conventional home-based protocol, though a hot spot is less significant, a node which owns pages frequently accessed by other nodes will have frequent interruptions due to page requests or page update diffs from non-home nodes. Our HHEAC protocol removes sending page update diffs to their homes at a lock release time. It also reduces many page requests at home nodes triggered by page faults inside critical sections of non-home nodes.

Second, our protocol ensures that all data needed in a critical section are prefetched from the previous lock owner before the node gets into the critical section. This diff-based update borrowed from the homeless protocol is very efficient since only diffs needed in a critical section are transferred. This is in contrast to the conventional home-based protocol, which sends a whole page and triggers page faults inside a critical section. The prefetch during lock synchronisation removes not only unnecessary data transfer but also page faults which trigger costly OS system calls such as a SEGV signal handler and the following network operations such as send or receive.

Third, the dynamic home migration scheme is possible because of the lazy home update of HHEAC. The lazy home update, which is delayed until the next barrier, allows a barrier manager to choose the best home nodes for shared pages.

Finally, diff integration provides the maximum benefit for DSM applications that use lock synchronisation and have a coarse-grained migratory memory access pattern inside their critical sections.

Many scientific parallel applications use an iterative algorithm in which the same instructions are performed on a different data set iteratively. According to Fox et al. (Fox, Williams and Messina, 1994), more than 70% of parallel applications are classified as synchronous or loosely synchronous applications in which the main parallel operations are simultaneous identical updates to a set of partitioned data. Our HHEAC is efficient for synchronous applications using lock synchronisation as shown in the parallel neural network application in Section 7.4.

On the other hand, added protocol overheads are as follows:

1. Twinning and diffing of home pages are required even in a home node.
2. In addition to the global barrier arrival process, a global barrier departure process is needed to prevent a node from having a stale copy of a page due to “lazy home update” as stated in Subsection 5.3.3.

The first added overhead weakens the positive “home effect” in the home-based protocol. The home effect means that a home node is not required to create twins or diffs for its home pages since all writes on its home pages are always visible to other non-home nodes according to the protocol. In a page-based DSM system, twinning and difffing is the method of preserving writes on shared memory. Since our HHEAC implementation employs the homeless protocol and the write-update coherence protocol only in lock synchronisation, CS diffs made within a critical section should be transferred to the next lock owner regardless of whether the diffs are made from the home pages or not.

In the case of NCS diffs, since the implementation employs the dynamic home migration scheme, in order to renew optimum home nodes at every barrier time, even a home node needs to create twins and diffs for its home pages. At the next barrier, a barrier manager will decide the dirty pages’ homes based on the size of the accumulated diffs of a page made between two consecutive barriers. The new home node of a page will be chosen if the node has the largest size of accumulated diffs of the page in order to minimize home update data traffic.

The second added overhead is a global barrier departure process which is not required in the homeless and conventional home-based protocols. A global barrier departure process is needed in HHEAC because the home update process which makes all home nodes up-to-date should be completed at the same time. Otherwise, it is possible for a node to request a page before any of the non-home nodes finish on the page update. This may cause a home node to send an incomplete copy of the page. A barrier departure process makes sure that all nodes finish their home update before they leave the barrier. However, a barrier manager can remove the barrier departure process if the memory access pattern between two consecutive barriers showed no write-write false sharing between nodes because all nodes already have the complete up-to-date copies of the pages before the barrier. A barrier manager can know whether there was no write-write false sharing on the same page by comparing write notices that contain dirty page numbers collected from all nodes. That is, if there are no pages that are dirty in more than one node, then the memory access pattern in the last barrier session had no write-write false sharing between nodes.

To minimize the protocol overheads and maximize the home effect, we developed an initial dynamic home migration technique (IDHMT) in which a home node is dy-
nationally assigned only within some initial execution period. The rationale behind the initial dynamic home assignment is that the memory access pattern of an application would be relatively well known to the system after assigning home nodes dynamically several times. In the current implementation of IDHMT, we stopped the dynamic home assignment after the third barrier. After the third barrier, twinning and diffing for home pages are removed and the global barrier departure process is also removed.

5.6 Related Work

As presented in previous chapters, TreadMarks and Princeton’s home-based ScC system are two original implementations of LRC and ScC, respectively. Since then, many other contributions to the software DSM have been published. In this section, we compare other work with HHEAC, and present the difference between HHEAC and other work.

As far as we know, there are two similar all-software ScC DSM implementations to HHEAC: Brazos and JiaJia. JiaJia (Hu et al., 1999b; Hu, Shi and Tang, 1999c) employs a home-based protocol. On the other hand, Brazos (Speight and Bennett, 1997; Speight and Bennett, 1998) is essentially a homeless DSM system. There are also several LRC DSM implementations that have similar ideas to HHEAC. Below, we present the comparisons between those systems and HHEAC.

Brazos is a homeless page-based all-software ScC DSM system. In Brazos, stale data are made up-to-date by receiving diffs from other nodes efficiently by exploiting multicast communication, compared to HHEAC that uses both diffs and pages in order to update stale data without multicast support. Since Brazos uses multicasting for memory coherence during lock and barrier synchronisation, it reduces many complexities of implementing a ScC homeless DSM system. Even though Brazos claims that it uses an adaptation technique to update stale data between homeless and home-based protocols, it is dependent on a page’s memory access pattern, and it still has to pay the adaptation and page ownership finding overheads since it uses essentially a homeless protocol. On the contrary, HHEAC is much more efficient in combining the two protocols, as it is essentially a home-based protocol with the lazy home update and there is no overhead of combining the two protocols.

JiaJia is a home-based all-software ScC DSM system. However, it has no concept of the lazy home update and only uses the write-invalidate coherence protocol. Also, the implementation of ScC is different between JiaJia and HHEAC. In JiaJia, a lock manager manages ScC coherence information so that the manager determines which
pages are invalidated in the next lock owner, whereas in HHEAC each local node determines the required diffs for the next lock owner. In this way, HHEAC can prevent an added burden on a lock manager and is a more fine-grained implementation of ScC. The implementation of ScC in JiaJia is not only inefficient compared to HHEAC but also cannot prevent write-write false sharing inside a critical section due to the use of the write-invalidate protocol and the large page granularity.

ADSM (Monnerat and Bianchini, 1998) is a homeless all-software LRC DSM system in which two adaptations between single and multiple writer protocols, and write-update and write-invalidate protocols, are selectively used based on the page’s memory access pattern. Basically it uses the invalidation protocol. However the update protocol is used for migratory pages inside a critical section and producer/multiple consumer pages during barrier synchronisation. Compared to the update protocol used in HHEAC, CS data transferred by the update protocol in ADSM are limited to migratory pages only. Also the granularity of the update is the size of a page, whereas there is a more fine-grained diff size in HHEAC.

There have been similar ideas of using the two coherence protocols selectively in order to implement a more efficient coherence protocol in a software DSM system. In KDSM (Yun et al., 2001), instead of using only the invalidation coherence protocol as in most home-based systems, the update coherence protocol is also used only in lock synchronisation times to solve an inefficient page fetch process occurring in a critical section. However, the efficiency obtained by their implementation is still limited due to the LRC model implementation. For example, at the time of release a node has to send modified data to the corresponding homes, which is unnecessary in HHEAC. Also, in case of diff accumulation, the efficiency of their protocol can be severely diminished.

Another similar use of a hybrid protocol is found in the Affinity Entry Consistency (AEC) system (Seidel, Bianchini and de Amorim, 1997) even though the AEC system employs the homeless protocol only. In their system, a Lock Acquirer Prediction (LAP) technique is used to predict the next lock owner in order to prefetch required CS data to the next lock owner. We believe that the LAP technique is not required for most lock-based DSM applications since the next lock owner is already determined many times before the release time. When the next lock owner is not determined at the release time, employing the LAP technique leads to unnecessary updating if the prediction is wrong. Rather than updating eagerly based on prediction, it would be better to wait until the next lock owner is determined. In a similar scenario, at the release time, our implementation first creates diffs modified inside a critical section but waits until
Table 5.1: Comparison of DSM Systems

<table>
<thead>
<tr>
<th>DSM System</th>
<th>Memory Model</th>
<th>Coherence Protocol</th>
<th>Lock-related Coherence Action</th>
<th>Barrier-related Coherence Action</th>
<th>Major Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADSM</td>
<td>LRC</td>
<td>Homeless</td>
<td>hybrid</td>
<td>hybrid</td>
<td>Adaptation</td>
</tr>
<tr>
<td>AEC</td>
<td>EC</td>
<td>Homeless</td>
<td>update</td>
<td>invalidate</td>
<td>LAP</td>
</tr>
<tr>
<td>Brazos</td>
<td>ScC</td>
<td>Homeless</td>
<td>update</td>
<td>invalidate</td>
<td>Multicast</td>
</tr>
<tr>
<td>HHEAC</td>
<td>EAC</td>
<td>Homeless</td>
<td>update</td>
<td>invalidate</td>
<td>Hybrid of HL and HB</td>
</tr>
<tr>
<td>JiaJia</td>
<td>ScC</td>
<td>Home-based</td>
<td>invalidate</td>
<td>invalidate</td>
<td>Lock-based cache coherence</td>
</tr>
<tr>
<td>KDSM</td>
<td>LRC</td>
<td>Home-based</td>
<td>update</td>
<td>invalidate</td>
<td>Efficient lock update</td>
</tr>
<tr>
<td>Orion</td>
<td>LRC</td>
<td>Home-based</td>
<td>hybrid</td>
<td>hybrid</td>
<td>data collection by home node</td>
</tr>
</tbody>
</table>

the next lock owner is determined. Upon receiving a lock request, the diffs previously created and stored are sent to the lock requester with the lock ownership. That is, HHEAC eagerly creates and stores required CS diffs, but can lazily transfer those diffs when the next lock request is received after the diff creation.

Finally, Orion (Ng and Wong, 2000) is a home-based LRC DSM system. It has a different approach to other adaptation techniques. It exploits a home node which collects all data access information from other non-home nodes. When other non-home nodes are detected as the frequent readers of its home pages, the home node notifies all non-home nodes about the frequent reader nodes. Next, when a non-home node sends diffs to the home node, it also updates the frequent reader nodes, hoping that the diffs are accessed by them. In Table 5.1, we present a summary of the DSM systems described above.
Chapter 6

Performance Improvement Techniques

6.1 Introduction

In this chapter, we present in detail three performance improvement techniques: diff integration, dynamic home migration and the use of a page state called exclusive\_read\_write. These techniques are employed in the implementation of HHEAC to further improve the performance of DSM applications. Below, we briefly introduce the rationales for developing the three techniques and their effects on DSM performance, respectively.

- The diff integration technique: The homeless DSM implementation can cause the diff accumulation problem in migratory applications, as discussed in Chapter 4. On the other hand, there is no diff accumulation problem in the home-based protocol. However, our hybrid home-based protocol can also accumulate unnecessary diffs as in the homeless protocol. Unnecessary diffs, for example, include old values of a shared variable that were changed before during previous critical sections. Since we use the homeless protocol during lock synchronisation, the same diff accumulation problem as in the homeless protocol can happen in HHEAC due to the diff-based update during lock synchronisation.

In a homeless LRC DSM implementation such as TreadMarks however, since it implemented LRC, diff integration is much more complicated as briefly explained by Keleher (1995b) due to LRC’s more liberal programming model. In HHEAC, an implementation of diff integration is simpler because it only concerns CS diffs made in the same critical section. The diff integration technique incorporates...
multiple diffs created from the same page during the same critical section into one diff that is representative of all the previous diffs.

- The dynamic home migration technique: More lazy coherence actions such as the lazy home update allowed by the HHEAC protocol give more time to dynamically assign a home node to the most frequently accessed pages, which can solve the home assignment problem in the home-based protocol.

Our technique is different from others (Hu, Shi and Tang, 1999a; Fang, Wang, Zhu and Lau, 2004; Cheung, Wang and Hwang, 1999). Our protocol updates home nodes after all nodes have a knowledge of optimum home locations. This guarantees minimum data traffic related to home page updates by non-home nodes. On the other hand, other dynamic home assignment schemes predict optimum home locations based on previous memory access patterns. This prediction would work well for applications showing regular and coarse-grained memory access patterns without a migratory pattern. However, for applications showing irregular, fine-grained or migratory memory access pattern, their schemes do not work since a future memory access pattern would be different from the previous patterns.

- The use of the page state exclusive_read_write: In the home-based protocol, before exiting a barrier, home pages are made write-protected (read_only) in order to notify other non-home nodes that the pages written are stale. However, in a single writer application, since other non-home nodes do not access home pages and know the location of the pages according to the home-based protocol, write-protection of home pages are not necessary but only cause unnecessary page faults. A new page state, exclusive_read_write, prevents unnecessary page faults in a home node when a home node exclusively accesses its pages between two consecutive barriers.

The exclusive_read_write state is very effective in single writer applications. Even in multiple writer applications, the exclusive_read_write state is useful because it can reduce the size of a diff of a false sharing page in a home node due to delayed twinning time.

6.2 Diff Integration

It is well known that there is a diff accumulation problem in migratory DSM applications in which a group of shared variables are accessed by every node sequentially by
means of synchronisation (Lu, 1995). This problem is found in the homeless protocol but not in the conventional home-based protocol since the home-based protocol updates home nodes as soon as it create diffs. This is one of the reasons why the conventional home-based protocol has shown better scalability for those migratory applications over the homeless protocol due to the significant reduction of diff communication as presented in Section 4.3.

Furthermore, the home-based protocol can be better utilized by adopting the EAC memory model as demonstrated in HHEAC. Previous home-based protocols used the invalidation coherence protocol during lock ownership change. However for the home-based protocol in the EAC implementation, use of the update coherence protocol would be better since the scope is relatively predictable in a critical section which is protected by the same lock. If the update protocol is used to implement the EAC model, the diff accumulation problem occurs even in the home-based protocol. For example, in the hybrid home-based EAC protocol in Figure 5.2, when N2 requests lock ownership from N1, N1 would send two diffs for page 1 and two diffs for page 2 (two diffs forwarded from N0 and the new two diffs created by N1) to N2. This diff accumulation seems trivial in this example. However if writes in a critical section are large and executed in many nodes sequentially, the size of accumulated diffs soon becomes large enough to severely degrade DSM performance.

Our implementation of the diff integration technique uses different timing of twinning and diff application than previous page-based DSM implementations. In previous DSM implementations, twinning occurred after a write page fault followed by diff application since they are only concerned with preserving the local node’s write, not a remote node’s write. That is, each node only preserves its own writes. For the writes of remote nodes, they are brought to a local node by requesting distributed diffs in the homeless protocol or pages in the home-based protocol.

However, twinning in our new technique occurs before diff application and not after a write page fault. By making a twin before applications of previous CS diffs made from the same page, all the previous CS diffs can be restored by comparing the twin and the final copy of the page at the following release time. In this way, all previous diffs on the same page made in the same critical section can be preserved and integrated into one integrated diff.

To understand how our diff integration technique can prevent diff accumulation,
Figure 6.1 is presented to explain the advantage of diff integration. With the diff integration, N1 and N2 will send only one diff: \{2,2,4\} and \{3,3,6\}, respectively. Without diff integration, in the same scenario as Figure 6.1, N1 would send the two diffs \{1,1,2\} and \{2,2,4\} which were made in N0 and N1, respectively to the next lock owner, N2. Also N2 would send the three diffs \{1,1,2\}, \{2,2,4\} and \{3,3,6\} to the next lock owner. This diff accumulation continues until the next barrier.

In case of a lock owner acquiring the same lock again, the lock owner accumulates diffs until it receives a lock request and sends accumulated diffs to the next lock owner. A diff created earlier is applied ahead of ones created later at the next lock owner. As described in Section 5.3.1, one of the fields of the diff data structure, again_no, is used in order to distinguish among the accumulated diffs in this situation. This local diff accumulation is not a serious problem since diffs accumulated in one node are integrated into one as soon as they are applied at the next lock owner.

However this form of diff integration would not be always successful. Previous write history would be lost if a value in a diff created by a node returns back to the same node without any change during lock ownership change and that value is not changed during the current critical section. This scenario is illustrated in Figure 6.2.
Figure 6.2 shows that a write on S[1] by N0 will be lost after the third lock release in N0 since there is no way to preserve that write due to the same value of S[1] in the twin and the final copy of the page, thus only producing a diff containing {2, 1}. Our current solution to this bad diff integration is that when this situation is detected, that is, a node owns the previous owned lock again between two consecutive barriers, diff integration stops and diff accumulation happens. Note that from Figure 6.2, if N2 acquires the lock from N0 after the third lock release in N0, N2 can integrate accumulated diffs into one diff safely since N2 has not owned the lock before between two consecutive barriers.

Many real DSM applications having the diff accumulation problem do not have reacquisition of previously owned locks. Most migratory DSM applications, in particular iterative scientific applications, own the same lock only once between two consecutive barriers. However, even if reacquisition of previously owned locks is detected so that the diff integration is not applied, it only affects performance, not the correctness of a program execution.

Another solution to bad diff integration can be sought by changing the application program itself. For example, if the code in the critical section in Figure 6.2 is changed...
for (i=0; i < Max; i++) {
    lock_acquire(i);
    S[i] += 1;
    lock_release(i);
}

Figure 6.3: New Code preventing Bad Diff Integration

into the code shown in Figure 6.3 (where the single lock number 1 is replaced by a separate lock for each element of the array S), then there would be no bad diff integration even though the new code would incur more lock synchronisation.

6.2.1 Implementation

Our diff integration technique is applied to CS diffs only during lock synchronisation. Put simply, the objective of the implementation is to have the current lock owner hold the latest values of all its CS variables but discard the old values. Since the memory sharing unit of our system is a page, it is difficult to record fine-grained CS variables in the system in order to group them by lock-id. Instead, we are only able to record a page number in which each CS variable is located. However, by using the diffing technique and our diff data structure described in Section 5.3.1, we can precisely construct the modifications of CS variables.

As the same critical section is executed in different nodes between two consecutive barriers, the set of CS variables of each lock is dynamically growing. However the growth of CS variables protected by the same lock should follow the HHEAC programming model, in particular, the third constraint presented in Section 5.4.

We describe the implementation, starting from a lock acquire operation in which all previous CS diffs are transferred from the lock releaser. The pseudo code of the implementation during lock acquire is shown in Figure 6.4. Refer to the reply packet format of a lock ownership request to understand the code better, which is illustrated in Subsection 5.3.5.

As can be seen in Figure 6.4, a twin is created only once for each CS page before the diff application. Thus, by comparing the twin and the final copy of the page after completing the diff application, all previous CS writes on the page can be produced. The reason for storing subsequent diffs of the same page is that they may be transferred to the next lock owner as accumulated diffs in the case that there is no write on the
while (there is a diff to be applied) {
    find out page number and the number of diffs;
    add the page number to the lock page list;
    do {
        if (the first diff of the page) {
            make a twin of the page;
            store the diff;
            apply the diff to the copy of the page;
        }
        else {
            store the diff;
            apply the diff to the copy of the page;
        }
    } while (−−the number of diffs);
}

Figure 6.4: Pseudo Code for Diff Integration in Lock Acquirer

page during the CS.

After applying all the CS diffs transferred from the lock releaser, the new lock acquirer can enter the CS. Note that the lock acquirer makes sure that every modification occurring in the CS is detected and reproduced before entering the CS by means of the page protection mechanism and twinning. During CS executions, only a page fault occurring on a page other than an already known lock page obtained from the lock acquiring operation causes a twinning operation, and the page number is added to the lock page list.

At a lock release operation, for the diff integration, the lock releaser performs the following, as described in Figure 6.5. New CS diffs made in the CS can be of two kinds:

- A CS diff made from a known lock page at the previous lock acquire
- A CS diff made from a newly added lock page.

The former CS diff is the integrated diff of the page. The packing follows the packet format specified in Subsection 5.3.5.

### 6.2.2 Related Work

Trials of diff integration can be found in the two publications. First, in his Ph.D thesis, Keleher (1995b) mentioned that combining multiple diffs of a single page into a single diff could improve DSM performance. Two possible implementations were described in ...
create new CS diffs made in the CS;
if (lock request received) {
    do {
        pack the CS diffs of the page and the number of diffs;
    } while (--the number of lock page);
}

Figure 6.5: Pseudo Code for Diff Integration in Lock Releaser

the thesis: (1) retaining a twin even after a diff has been created, then subsequent diffs are made from the twin. (2) discarding a diff if it is completely overlapped by a diff received. They did not use any of them because they found that these implementations did not improve overall performance. Since their DSM system is developed with the LRC programming model in which a combined diff of a page should represent all the previous writes on the page, diff integration is less effective and more complicated.

Similarly, diff squashing as proposed by Lu et al. (1997) deals with all the previous writes on a page due to adopting LRC. The authors reported that diff squashing had no positive effect on eight applications they tested except IS-L. Rather, they reported an adverse effect due to the processing overhead of diff squashing on other applications.

On the other hand, the implementation of our diff integration is simpler and more effective due to implementing EAC. In our implementation, a combined diff should represent only the writes occurring inside the same critical section, not all the previous writes on the page as implemented in the other two systems.

6.3 Dynamic Home Migration

6.3.1 Related Work

The home-based protocol has a weakness when a home node is allocated for pages that are not accessed or are accessed less frequently by the home node compared with other nodes. The wrong home allocation causes unnecessary data traffic. There is no or less data traffic if home nodes are chosen according to the application’s memory access pattern. The strength of the home-based protocol comes from the so-called “home effect” that pages owned by a home node are always up-to-date, thus a home node needs no data communication when accessing its pages.

Previously home-based protocol DSM implementations used static home allocation (Zhou et al., 1996) or initial home allocation (Keleher, 1998). For static home alloca-
tion, a good home allocation would be decided by the application programmer. In case of initial home allocation, it cannot guarantee continuous good home allocation if an application’s memory access pattern changes between barriers such as for a migratory application. Therefore dynamic home migration is needed for the home-based protocol.

To solve the home allocation problem, a number of home migration protocols have been proposed (Fang et al., 2004; Hu et al., 1999a; Chung, Seong, Park and Park, 1999; Cheung et al., 1999; Ng and Wong, 2000; Whately, Pinto, Rangarajan, Iftode, Bianchini and Amorim, 2001). All home migration protocols proposed provide a solution only for single writer DSM applications or are speculative when home migration decisions are made, hoping that the future memory access pattern will be the same as before.

We propose a novel home migration protocol which can decide optimum home nodes for multiple writer and migratory applications as well as single writer applications. Our home migration decision is not based on a speculative assumption but is based on statistical data of the memory access pattern between two consecutive barriers, such as the sizes of diffs that need to be transferred to corresponding home nodes.

Though previous migration protocols also exploit the memory access pattern, their home migration decisions are made after updating the home nodes and are based on the hope that the same memory access pattern will occur after the migration. Since their migration decision is based on the speculative assumption, effectiveness of the decisions is not guaranteed, especially for those applications which have a migratory or unpredictable memory access pattern. On the other hand, our home migration decision is made before updating home nodes and is efficient since it calculates and chooses a home node which produces the minimum data traffic for the current home node updates.

Another direction of previous home migration protocols is to migrate homes at the time of lock synchronisation. However this direction is very conservative and would cause a home-finding overhead (Chung et al., 1999; Cheung et al., 1999). On the other hand, our protocol uses a barrier process in which information about best home nodes are piggybacked with other coherence-related data, thus minimizing the home finding overhead.

### 6.3.2 Design and Implementation

The main motivation of the home migration protocol is, as discussed in Chapter 4, to solve the home assignment problem in the home-based protocol. Since our home migration protocol exploits the HHEAC model, there is no home node update by
non-home nodes between two consecutive barriers. This lazy home update gives an opportunity to choose optimum home nodes for shared pages at the time of barrier synchronisation. At the barrier time, according to the HHEAC protocol, each node has its own exclusive up-to-date shared variables.

For barrier-only applications, the exclusive up-to-date shared variables consist of only NCS diffs. For lock and barrier applications, they consist of NCS and CS diffs. All NCS diffs should be transferred to corresponding home nodes. Similarly, the last CS diffs of each lock should be transferred to corresponding home nodes. However, in contrast with other home-based DSM implementation, these diff transfers to corresponding home nodes are delayed until optimum new home nodes of shared pages are determined dynamically.

In Figure [6.6], an overview of the protocol is illustrated. Below, we describe how the home migration decision is made. All nodes record their dirty pages between two consecutive barriers. Upon arrival at a barrier, all nodes create final NCS diffs. All nodes except the barrier manager node send their final dirty page information including each dirty page diff size to the manager node. Whenever the manager receives these information from other nodes, it creates a global dirty page list by accumulating dirty page information from other nodes, and sets a home node which has the maximum diff size for each dirty page. The idea of choosing the maximum diff size is to minimize data traffic for the home node update which should be done according to the home-based protocol. If we set the home node to be the node which has the maximum diff size, it definitely minimizes home update data traffic since the chosen home node which has the maximum diff size is not required to update the page for others and other non-home nodes update the home with smaller-sized diffs.

For lock and barrier applications, the protocol is the same with the addition that only the last lock owner adds each CS diff size to the diff size of the corresponding dirty page. The idea is that the last lock owner will have the integrated diffs made inside the critical section. Therefore the diffs made in the last lock owner will cover all the writes occurring inside the critical section by all nodes between two consecutive barriers. For example in Figure [6.6], if N0 is the last lock owner of lock 1, then only N0 adds its CS diff size into the corresponding NCS diff size, if any, and sends them to the barrier manager.

After the manager decides home nodes for the dirty pages, it piggybacks the result to all nodes as a normal barrier departure procedure. Upon receiving the result of the home assignment, all nodes update home nodes with their diffs. Note that only
the last lock owner updates the home nodes with its integrated diffs made in the lock synchronisation if the last lock owner is not the home of the CS diffs. For example, in Fig 6.6 CS diffs made in N1 and N2 need not to be sent to the home(s).

However, to find an optimum home node, even the home node creates twins and diffs for its home pages which diminishes one of the home-based protocol advantages. Also a barrier will take slightly longer than when not using the dynamic migration protocol since the home migration decision is made during the barrier time. This overhead time can be minimized since all the required information is piggybacked during the normal barrier synchronisation. The performance gains due to the home migration would be dependent on the memory access pattern of an application. If the memory access pattern is regular and coarse without migration, the benefit of the home migration will increase overall performance which overcomes the cost of the protocol overhead. Even if the memory access pattern is not regular and migratory, still the home migration guarantees the minimum data traffic for the home node updates.
6.4 Exclusive_Read_Write State

According to the home-based protocol, a write in a home node should be notified to others. To this end, previous home-based protocol implementations make initial home pages read only after a barrier departure so that a home node can detect its writes. However, we have found that this scheme makes unnecessary page faults as well as unnecessary coherence data traffic for single writer applications. In addition, with our home migration protocol this read only page state as the initial page state for home pages is not efficient even for multiple writer applications.

To remedy this problem, we created a new initial home page state called “exclusive_read_write”. The exclusive_read_write page state is retained until other nodes request the home page and then becomes read_only.

The benefits of the new scheme are as follows: first, it removes home page write faults for single writer applications if the page is only accessed by the home node. Second, it can reduce a diff size since twinning timing is delayed until the page request from others. Note that the new scheme has no overhead cost so it only improves DSM performance and does not decrease in any applications. Also, memory consistency is maintained since the writes during the exclusive_read_write state are preserved when the copy of the page, including the writes, is transferred to a page requester who would be potentially a next home node of the page.

A technique called lazy home page write detection, which is essentially the same idea as the use of exclusive_read_write, has been developed independently (Hu, Shi and Tang, 2001).
Chapter 7

Performance Evaluation

We have evaluated the performance of our implementation of HHEAC to determine the benefits and side effects of HHEAC. In this chapter, we present the overall performance results of the HHEAC DSM system. Furthermore, we present a performance evaluation of the HHEAC protocol and the three techniques — diff integration, home migration and the use of the exclusive_read_write page state. These four factors affect the performance of the HHEAC system positively. Each factor is analysed and evaluated to find out the impact of each factor to the DSM performance.

7.1 Environment for Performance Evaluation

As shown in Figure 4.5, we again used our dedicated cluster network consisting of 32 nodes, each one having a 350 MHz Pentium II CPU. However, the environment of this performance evaluation changed slightly from the previous experiment in Section 4.3. In the previous experiments, N0 had a 100 Mbit NIC — the same as the rest of the nodes. After we found out that N0 can cause a hot spot due to the nature of the homeless protocol, we replaced the 100 Mbit NIC with a 1 Gbit NIC. This replacement will benefit the homeless protocol more than the home-based protocol since the home-based protocol is less susceptible to a hot spot. Also, we changed the operating system from Red Hat Linux 7.2 with gcc 2.96 to Gentoo Linux with gcc 3.3.2.

We measured the cost of basic network operations between nodes, and basic operations of the three systems as presented in Tables 7.1 and 7.2, respectively. We used \textit{iperf} which is a network testing tool to measure the throughput of a network, and \textit{ping} which is a test application that measures the round-trip time of a packet between

\footnote{For more information about iperf, see http://dast.nlanr.net/Projects/Iperf}
<table>
<thead>
<tr>
<th>Operation</th>
<th>TM</th>
<th>HLRC</th>
<th>HHEAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barrier over 32 Nodes (ms)</td>
<td>3.489</td>
<td>3.417</td>
<td>9.181</td>
</tr>
<tr>
<td>Lock Acquire Time over 2 nodes (ms)</td>
<td>0.223</td>
<td>0.218</td>
<td>0.214</td>
</tr>
<tr>
<td>Lock Acquire Time over 3 nodes (ms)</td>
<td>0.327</td>
<td>0.316</td>
<td>0.303</td>
</tr>
<tr>
<td>Remote Page Fetch (ms)</td>
<td>0.693</td>
<td>0.687</td>
<td>0.692</td>
</tr>
</tbody>
</table>

Table 7.2: Comparisons of Basic System Operation Costs between TreadMarks (TM), Home-based LRC (HLRC) and Hybrid Home-based EAC (HHEAC) systems
<table>
<thead>
<tr>
<th>Application</th>
<th>Problem Size</th>
<th>Iterations</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PNN</td>
<td>44,000</td>
<td>235</td>
<td>613.56</td>
</tr>
<tr>
<td>Barnes-Hut</td>
<td>64k Bodies</td>
<td>3</td>
<td>79.58</td>
</tr>
<tr>
<td>IS-B</td>
<td>$2^{24} \times 2^{15}$</td>
<td>20</td>
<td>71.61</td>
</tr>
<tr>
<td>IS-L</td>
<td>$2^{22} \times 2^{13}$</td>
<td>30</td>
<td>16.39</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>64x64x64</td>
<td>50</td>
<td>45.10</td>
</tr>
<tr>
<td>SOR</td>
<td>4000x4000</td>
<td>50</td>
<td>49.58</td>
</tr>
<tr>
<td>Gauss</td>
<td>1024x1024</td>
<td>1023</td>
<td>15.26</td>
</tr>
</tbody>
</table>

Table 7.3: Problem Sizes, Iterations and Sequential Execution Times (secs.)

sending and receiving entities in order to measure the effective throughput and the round trip time between nodes, respectively.

The effect of the 1 GB NIC in node 0 can be seen only when node 0 receives data from other nodes as shown in Table 7.1. But this is not a problem since node 0 was the hot spot which receives too much data at the same time from other nodes as shown in the previous experiment in Section 4.3. The relatively large barrier time over HHEAC is due to the added barrier departure process.

7.2 Applications

We chose seven applications to evaluate our HHEAC protocol. We used the same applications employed in Section 4.3 except that TSP was removed and lock-based Integer Sort was added. The problem sizes and sequential execution times of the applications are presented in Table 7.3. Compared with Table 4.1 in Chapter 4, SOR has a larger problem size. Also, we found that PNN and Barnes-Hut have better sequential execution times under the new environment due to the use of the different compiler. Again, as explained in Section 4.3, the sequential times of only SOR and Gauss over HLRC are significantly different compared to TreadMarks and HHEAC. Table 7.3 presents the execution times over TreadMarks and HHEAC. The sequential times of SOR and Gauss over HLRC are 71.36 and 25.86 seconds, respectively. Note that the source code for each application tested over the different protocols is identical.
7.3 Overall Performance Results

Compared with the performance results presented in Table 4.2 in Subsection 4.3.3, TreadMarks has gained significant performance improvements in PNN, Barnes-Hut and IS-B. The abnormal performance degradations of those applications over TreadMarks with 32 nodes shown in Table 4.2 disappeared due to added bandwidth in Node 0. Nonetheless, TreadMarks, which implements the homeless LRC protocol, still showed its inherent weakness of scalability due to the hot spot in PNN, garbage collection in Barnes-Hut and diff accumulation in the two IS.

As can be seen in Table 7.4 and Figures 7.1 to 7.4, HHEAC retains the scalability of the home-based protocol and avoids most of the poorer performances of HLRC for SOR and Gauss, even though Gauss over HHEAC with 32 nodes is slightly worse than over HLRC. HHEAC also avoids the poorer performances of TreadMarks in PNN, Barnes-Hut, IS-B and IS-L over more than 16 nodes. In particular, HHEAC has shown significantly better performance with applications showing coarse-grained migratory memory access patterns in a critical section such as PNN and IS-L.

HHEAC showed better scalability compared with the homeless protocol in TreadMarks. For example, over 4 nodes HHEAC has no clear performance superiority over the other two protocols as shown in Figure 7.1. However, over 32 nodes, the HHEAC implementation was 3.6 times, 1.7 times, 7.7 times, and 11 times faster than TreadMarks in PNN, Barnes-Hut, IS-B and IS-L, respectively, and 3.4 times faster than HLRC in SOR, as shown in Figure 7.4.

The better performance of SOR and Gauss in TreadMarks can be explained by the lazy diffing technique (Keleher et al., 1994). The lazy diffing technique, which does not create a diff unless it is requested, is very effective for single writer applications without

<table>
<thead>
<tr>
<th>Apps</th>
<th>4 nodes</th>
<th>8 nodes</th>
<th>16 nodes</th>
<th>32 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TM</td>
<td>HLRC</td>
<td>HHEAC</td>
<td>TM</td>
</tr>
<tr>
<td>PNN</td>
<td>3.9</td>
<td>3.9</td>
<td>3.9</td>
<td>7.1</td>
</tr>
<tr>
<td>B-H</td>
<td>2.1</td>
<td>2.0</td>
<td>2.1</td>
<td>2.4</td>
</tr>
<tr>
<td>IS-B</td>
<td>3.4</td>
<td>3.6</td>
<td>3.6</td>
<td>4.1</td>
</tr>
<tr>
<td>IS-L</td>
<td>2.3</td>
<td>2.9</td>
<td>2.7</td>
<td>1.4</td>
</tr>
<tr>
<td>FFT</td>
<td>1.4</td>
<td>0.9</td>
<td>1.3</td>
<td>2.1</td>
</tr>
<tr>
<td>SOR</td>
<td>3.4</td>
<td>1.9</td>
<td>3.2</td>
<td>6.0</td>
</tr>
<tr>
<td>Gauss</td>
<td>2.1</td>
<td>0.2</td>
<td>1.3</td>
<td>1.6</td>
</tr>
</tbody>
</table>

Table 7.4: Comparison of Speed-ups between TreadMarks (TM), Home-based LRC (HLRC) and Hybrid Home-based EAC (HHEAC)
migratory memory access patterns. But this low protocol overhead is only applied to applications such as SOR or Gauss which have a very regular and exclusive memory access pattern. Migratory applications such as PNN and IS showed much better performance under the home-based protocol. In particular, significant improvements can be achieved in PNN under HHEAC by diff integration and efficient implementation of the update protocol during lock synchronisation.

The super slow-down shown in PNN, Barnes-Hut and the two IS over a large number of nodes in TreadMarks proves that the homeless protocol is vulnerable to the scalability problem. Up to 8 nodes, the homeless protocol showed relatively comparable performance with the two home-based protocols. However over 32 nodes, TreadMarks showed rapid slow-down except FFT and SOR. As discussed in Chapter 4 a hot spot,
Table 7.5: Comparison of Data Communication Traffic between TreadMarks (TM) and Two Home-based Protocols — HLRC and HHEAC

<table>
<thead>
<tr>
<th>Apps.</th>
<th>Number of Messages(K)</th>
<th>Amount of Traffic(MB)</th>
<th>Number of Messages(K)</th>
<th>Amount of Traffic(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TM</td>
<td>HLRC</td>
<td>HHEAC</td>
<td>TM</td>
</tr>
<tr>
<td>PNN</td>
<td>114.2</td>
<td>159.4</td>
<td>102.4</td>
<td>277.6</td>
</tr>
<tr>
<td>B-H</td>
<td>2809.5</td>
<td>413.4</td>
<td>594.7</td>
<td>622.9</td>
</tr>
<tr>
<td>IS-B</td>
<td>62.0</td>
<td>65.5</td>
<td>66.8</td>
<td>604.3</td>
</tr>
<tr>
<td>IS-L</td>
<td>23.9</td>
<td>26.3</td>
<td>26.3</td>
<td>247.4</td>
</tr>
<tr>
<td>FFT</td>
<td>134.5</td>
<td>152.2</td>
<td>159.5</td>
<td>223.1</td>
</tr>
<tr>
<td>SOR</td>
<td>45.5</td>
<td>57.0</td>
<td>69.1</td>
<td>65.6</td>
</tr>
<tr>
<td>Gauss</td>
<td>120.7</td>
<td>321.5</td>
<td>123.7</td>
<td>124.4</td>
</tr>
</tbody>
</table>

garbage collection and diff accumulation are three major hindrances to scalability in the homeless protocol.

SOR and Gauss have the most benefit from the dynamic home migration technique, even though Gauss over 32 nodes showed the adverse effect of the technique due to its frequent barrier synchronisation use.

Generally, Table 7.5 and Figures 7.5 and 7.6 indicate that the more the data traffic, the poorer the performance, as strongly suggested in the two IS over TreadMarks, and SOR over HLRC. The exceptions are PNN between TreadMarks and HLRC, and Barnes-Hut between TreadMarks and two home-based protocols. The reason for the exception of PNN is a frequent occurrence of a hot spot as explained in Subsection 4.3.8. The reason for Barnes-Hut is that even though it produced less data traffic in the homeless protocol than in the home-based protocol, the number of messages produced was much more, for example nearly ten times more for 32 nodes compared with the home-based protocol, as shown in Figure 7.9. This shows that write-write false sharing applications such as Barnes-Hut over the homeless protocol will produce many diff requests sent to many nodes in order to construct the up-to-date copy of an invalid page. On the contrary, with the home-based protocol, only one page request sent to a home node of the page is sufficient to have the up-to-date copy of the page, which is much more efficient.

In the case of IS-B over 16 nodes, even though TreadMarks produces fewer messages compared to other two systems, it produces much more data traffic; more than 7.6 times compared with HHEAC. The data traffic statistics of IS-B over TreadMarks means that the average size of a message is quite large due to diff accumulation. For example, the
average size of a packet in TreadMarks over 32 nodes is 11605 bytes, compared to 1043 bytes in HHEAC. This data shows that in IS-B over TreadMarks, when a node requests the required diffs of a stale page from the last writers, the diffs received can be large due to diff accumulation as explained in Subsection 4.3.6.

As shown in Figures 7.7, 7.8 and 7.9, TreadMarks produced more messages in one (Barnes-Hut) over 16 nodes and three (Barnes-Hut, IS-B and IS-L) over 32 nodes. As for HLRC, it produced more messages in two (PNN and Gauss) over 16 nodes and four (PNN, FFT, SOR and Gauss) over 32 nodes. Finally, HHEAC produced more messages in three (IS-B, FFT and SOR) over 16 nodes and none over 32 nodes. Overall, TreadMarks and HHEAC produced fewer messages compared to HLRC. Frequent home updates due to the wrong home assignment produced more messages over HLRC, in particular in the case of Gauss.

Table 7.6 presents the numbers of page faults, twinning and diffing over three protocols with 32 nodes. In Figures 7.10 to 7.13, a comparison between three protocols are illustrated with bar graphs. As shown in Figure 7.11, HHEAC has the least number of remote page faults in lock-based applications such as PNN and IS-L. The reduction of remote page faults is important since they reduce data communication such as up-to-date memory fetches from others. In barrier-only applications, however, the number of page faults is dependent on either the memory access patterns of applications or good home assignment. For example, large numbers of page faults in SOR and Gauss over HLRC, as shown in Figure 7.10, are caused by wrong home assignment.

The added protocol overheads of HHEAC such as more twinning and diffing are not
Figure 7.7: Number of Messages over 16 Nodes

Figure 7.8: Number of Messages over 32 Nodes

Figure 7.9: Number of Messages in Barnes-Hut over 16 and 32 Nodes
Table 7.6: Comparison of Page Faults, Twinning and Diffing between TreadMarks (TM) and Two Home-based Protocols — HLRC and HHEAC — over 32 nodes

<table>
<thead>
<tr>
<th>Apps.</th>
<th>Total Page Faults</th>
<th>Remote Page Faults</th>
<th>Twinning</th>
<th>Diffing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TM</td>
<td>HLRC</td>
<td>HHEAC</td>
<td>TM</td>
</tr>
<tr>
<td>PNN</td>
<td>112536</td>
<td>121032</td>
<td>77775</td>
<td>74854</td>
</tr>
<tr>
<td>B-H</td>
<td>498739</td>
<td>685677</td>
<td>678909</td>
<td>196704</td>
</tr>
<tr>
<td>IS-B</td>
<td>60159</td>
<td>59521</td>
<td>60159</td>
<td>39711</td>
</tr>
<tr>
<td>IS-L</td>
<td>15532</td>
<td>15121</td>
<td>15532</td>
<td>15120</td>
</tr>
<tr>
<td>FFT</td>
<td>157004</td>
<td>205097</td>
<td>156960</td>
<td>105568</td>
</tr>
<tr>
<td>SOR</td>
<td>44061</td>
<td>829929</td>
<td>44061</td>
<td>15520</td>
</tr>
<tr>
<td>Gauss</td>
<td>66093</td>
<td>588974</td>
<td>66093</td>
<td>64046</td>
</tr>
</tbody>
</table>

significant as shown in Figures 7.12 and 7.13. We believe more twinning and diffing in HHEAC are not desirable — they are less significant to DSM performance as CPU speed increases rapidly.

7.4 HHEAC Protocol Effects

To better identify the benefits of the HHEAC protocol, we performed three experiments.

7.4.1 First Experiment

We implemented a conventional home-based ScC (CHScC) DSM system which employs only the write-invalidate coherence protocol and updates home nodes eagerly at every synchronisation point. Then, we ran PNN over this implementation and compared the execution time with the HHEAC implementation.

The result of the first experiment is shown in Figure 7.14 which displays the speed-up of PNN over the two different home-based implementations of ScC and EAC. As can be seen in Figure 7.14, PNN showed performance that was comparable between the two implementations until 4 nodes. However, from 8 nodes PNN over the HHEAC implementation began to show better performance than PNN over the CHScC implementation. For example, with 32 nodes, HHEAC showed more than 100% better speed-up than CHScC.
To better understand the reason for the performance differences between CHScC and HHEAC in PNN, we measured the number of messages, data traffic, the numbers of page faults, diffs and twins created, as shown in Table 7.7. Table 7.7 shows that HHEAC reduced the data communication cost and the number of total page faults as well as remote page faults that require page fetches from remote home nodes after the faults. The number of messages and data traffic is significantly reduced in HHEAC, as shown in Figures 7.15 and 7.16 due to lazy coherence actions and effective use of the write-update protocol during lock synchronisation. Similarly, the number of page faults is reduced in HHEAC, as shown in Figures 7.17 and 7.18 due to the effective use of the write-update protocol during lock synchronisation. On the other hand, HHEAC has more diffs and twins created than in CHScC as added protocol overheads, as illustrated in Figures 7.19 and 7.20. However, this added processing is well offset by the other benefits of HHEAC.

### 7.4.2 Second Experiment

We measured the times taken during lock synchronisation (lock acquire and release) and critical sections in PNN over three different protocol implementations: TreadMarks, HLRC and HHEAC. As illustrated in Figure 7.21, critical section times are measured by the time taken between 2 and 3, and lock synchronisation time is obtained by adding the times taken between 1 and 2, and 3 and 4. We chose PNN as it uses lock synchronisation heavily for its computation. Applications using only barrier synchronisation such as Barnes-Hut, IS-B, FFT, SOR and Gauss cannot demonstrate the benefit of the lazy...
As shown in Figure 7.22, Node 0 (N0) over TreadMarks suffers the most from lock contention due to a hot spot in N0. The hot spot in N0 occurs since only N0 writes on the shared pages at the end of barrier in each loop and those pages are accessed after the barrier by all nodes. Therefore all nodes request the pages from the last writer (N0) at the same time, which makes a hot spot. The hot spot becomes worse due to a migratory access pattern in PNN causing diff accumulation.

Meanwhile, a hot spot is relieved in HLRC since the shared pages are assigned evenly to the nodes. In PNN, pages 1 to 4 are most written by all nodes. Since home assignment in HLRC is statically performed, nodes 1 to 4 share the responsibility of sending the most up-to-date four pages. However, eager home update after lock synchronisation interrupts the main computation in nodes 1 to 4. Also nodes 1 and 2, which are two lock manager nodes, have another interruption due to lock requests from other nodes. That is why nodes 1 to 4, in particular nodes 1 and 2, have relatively long lock synchronisation times as shown in Figure 7.23. Note that the vertical scale in Figure 7.23 is different from that of Figure 7.22.

Finally, HHEAC greatly reduces the time taken for lock synchronisation and critical section execution compared with two other protocols as shown in Figure 7.24. Again, the vertical scale is reduced more in the figure. The reduction of the lock synchronisation is partly due to the diff integration technique since the technique greatly reduces the size of diffs transferred. The critical section execution time in HHEAC is negligible, just 0.04 seconds on average, compared with 4.46 seconds and 1.29 seconds in Tread-
Marks and HLRC, respectively, due to fewer page faults inside the critical sections. In PNN over HHEAC, N0 is a hot spot node as in TreadMarks, since N0 becomes the home node of the most frequently accessed five pages at the start of each training epoch due to the dynamic home migration technique. However, this time the hot spot effect is weakened thanks to our diff integration technique.

### 7.4.3 Third Experiment

We created a simple application that uses frequent fine-grained memory accesses in a critical section in order to see the difference between a conventional home-based (CHB) DSM implementation and HHEAC. For a CHB implementation, we used HLRC. Using HLRC as a CHB implementation is justified since the difference between LRC and ScC has nothing to do with the overall performance in this application. Also note that diff integration in HHEAC has a negligible effect on this application since very little diff accumulation occurs.

The main code is illustrated in Figure 7.25. In this application, all nodes add one to each element of an array consisting of 3,000 integers. Each element is protected by
<table>
<thead>
<tr>
<th></th>
<th>2 Nodes</th>
<th>4 Nodes</th>
<th>8 Nodes</th>
<th>16 Nodes</th>
<th>32 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHB</td>
<td>11965</td>
<td>11381</td>
<td>34394</td>
<td>74953</td>
<td>154311</td>
</tr>
<tr>
<td>HHE</td>
<td>25185</td>
<td>51273</td>
<td>102591</td>
<td>204777</td>
<td></td>
</tr>
<tr>
<td>Data Traffic (MB)</td>
<td>15.1</td>
<td>13.6</td>
<td>26.7</td>
<td>50.7</td>
<td>168.9</td>
</tr>
<tr>
<td>No. of Total Page Faults</td>
<td>6922</td>
<td>6918</td>
<td>11964</td>
<td>21596</td>
<td></td>
</tr>
<tr>
<td>No. of Remote Page Faults</td>
<td>2349</td>
<td>3085</td>
<td>7044</td>
<td>16433</td>
<td></td>
</tr>
<tr>
<td>No. of Diffs created</td>
<td>1715</td>
<td>2548</td>
<td>4311</td>
<td>9314</td>
<td></td>
</tr>
<tr>
<td>No. of Twins created</td>
<td>1715</td>
<td>2548</td>
<td>4311</td>
<td>9314</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.7: Statistical data for PNN over CHScC (CHB) and HHEAC (HHE)
Figure 7.17: The Number of Total Page Faults created in PNN over CHScC and HHEAC

Figure 7.18: The Number of Remote Page Faults created in PNN over CHScC and HHEAC

Figure 7.19: The Number of Diffs created in PNN over CHScC and HHEAC

Figure 7.20: The Number of Twins created in PNN over CHScC and HHEAC
a different lock so that 3,000 locks are required.

As shown in Table 7.8, a CHB DSM protocol implementation has a weakness in the application that has frequent critical sections within which fine-grained memory accesses are required. The problem is that even though one integer (4 bytes) is changed in a page, the whole page (4096 bytes) is required to be transferred between home and non-home nodes. This problem has been considered the weakness of the home-based protocol (Keleher, 1999; Hu et al., 2001). Furthermore, home updates after a lock release not only delay the releasing node but also interrupt home nodes frequently as indicated by the comparison of the number of messages between two protocols. HHEAC solved these two problems of CHB, which resulted in better performance.

### 7.4.4 Summary

In summary, HHEAC reduces lock synchronization contention, thereby improving the overall performance of PNN. On the other hand, lock synchronization contention over TreadMarks become worse due to the hot spot and diff accumulation problems. Compared with HLRC, HHEAC shows positive effects on the performance of PNN due to the write update protocol and no home update at a release time.

The positive HHEAC protocol effect on lock-based DSM applications are divided
into three factors:

- Fast lock synchronisation due to fewer page faults inside a critical section.

- Fewer messages and less traffic required compared to conventional ScC implementations.

- Fewer interruptions on home nodes.

Below, we identify other factors of HHEAC that improve DSM performance:

- Eager Diffing: Eager diffing at the release time could have a more desirable effect on DSM performance, compared to the lazy diffing in TreadMarks. The lazy diffing caused by page faults inside a critical section aggravates lock contention. There are two reasons for this — a diff request in a faulting node and the following lazy diffing in a remote node. On the contrary, eager diffing in HHEAC can promote faster lock transfer if the next lock owner is not itself or the next lock owner is not already known.

- Aggregate Diff Transfer: One packet containing many CS diffs made from multiple pages sent by a lock releaser is much more efficient than sending many small
Figure 7.23: Lock Synchronisation and CS Times in PNN over HLRC

Figure 7.24: Lock Synchronisation and CS Times in PNN over HHEAC
for(i=0; i < 3000; i++) {
  lock_acquire(i);
  shared->array[i] += 1;
  lock_release(i);
}

Figure 7.25: Main Code of the Simple Fine-grained Lock Application

packets, each having a diff(s) of a page as occurred in TreadMarks. This problem was identified by Lu et al. (1997).

7.5 Diff Integration

To understand how the diff integration technique improves performance, we did another implementation which is exactly the same as HHEAC except without the diff integration. Since the diff integration is useful only for applications using lock synchronisation, we did not test it for barrier-only applications.

As illustrated in Figure 7.26, the diff integration effect becomes clear as the number of nodes increase. In particular, the diff integration technique can reduce the amount of redundant data which is often created in migratory applications such as PNN. For example, without the diff integration, PNN produced 619 Mbytes of data traffic over 32 nodes, but with the diff integration, it produced 191 Mbytes.

7.6 Dynamic Home Migration

We evaluated the effect of the dynamic home migration technique by comparing the two performance results obtained from HLRC and HHEAC, respectively. The HLRC DSM implementation used the static home assignment and HHEAC used the dynamic home migration. If applications tested employ barrier synchronisation only, performance comparisons between the two implementations can tell us the dynamic home migration effect only. In other words, since SOR, Barnes-Hut and IS-B have no lock synchronisation, the HHEAC or diff integration factor has nothing to do with their performance results. Also, the difference of the memory consistency model between LRC and ScC has no effect on the performance results on those applications due to the same reason above.
The positive effect of the dynamic home migration can be found in SOR as shown in Figure 7.27. SOR is a single writer application without a migratory memory access pattern. Each node in SOR accesses its own shared memory in every computation iteration. Once a home node is decided for every page, the home assignment does not occur again due to the regular memory access pattern. The home migration technique is very effective for this kind of regular memory access pattern without a migratory access pattern.

The another positive effect of the technique can also be found in Barnes-Hut as shown in Figure 7.27. The memory access pattern in Barnes-Hut shows frequent write-write false sharing. In this pattern, many nodes write on the same page at the same time. Therefore, at the end of each barrier, even though the optimum node is chosen as the home page, the effect is not too positive since the many distributed diffs have to be gathered to the home node.

In the case of IS-B, which shows a regular but migratory access pattern, Figure 7.28 has shown a negative effect on the performance because the overhead of the technique...
is larger than the home migration effect.

In summary, these results tell us that the dynamic home migration is very effective for single writer applications without a migratory memory access pattern. The technique can improve the performance of write-write false sharing applications as shown in Barnes-Hut, even though the improvement is not significant as shown in SOR. On the other hand, for migratory applications such as IS-B, it could incur a negative effect on performance if the added overhead of the technique exceeds the benefit.

7.6.1 Initial Dynamic Home Assignment

The home migration did not work very well for migratory and multiple writer applications. The added protocol overhead diminishes the positive home effect of a home-based protocol. For example, diffing and twinning of home pages reduce the home effect. To diminish these negative effects in the home migration technique, we developed the initial dynamic home migration technique (IDHMT), in which the home migration protocol stopped after the system has acquired good home assignment. After that, fixed home assignment continues so as to remove the home migration overhead. Currently, our system stops the home migration after the third barrier. We chose the third barrier because after that, most DSM applications’ memory access patterns are known to the system except migratory or unpredictable memory access patterns between barriers.

This will enhance the home effect so that home pages are not required to create twins and diffs. Also, a barrier procedure will be simpler and faster since the home node decision that occurred in barrier synchronisation is removed. Performance improvements due to IDHMT can be found in Figure 7.29. We only present Barnes-Hut and Gauss since the others are almost the same with or without IDHMT. The improved speed-ups, even though the performance of Gauss in 4 nodes with IDHMT is slightly worse, are due to reduction of the HHEAC protocol overhead thanks to IDHMT.

7.7 Exclusive_Read_Write State

To find out how much the use of exclusive_read_write improves performances of the tested applications, we ran them without exclusive_read_write and compared the performance results that were obtained with exclusive_read_write.

As shown in Figure 7.30, the use of exclusive_read_write significantly improves performance in SOR and Gauss. Even though the speed-ups in Gauss actually slow down in the both cases, with exclusive_read_write, the execution time is much shorter than
Figure 7.27: Home Migration Effect on SOR and Barnes-Hut

Figure 7.28: Home Migration Effect on IS-B
without exclusive_read_write. These improvements stem from the reduction of home page faults due to exclusive_read_write. Without exclusive_read_write, SOR and Gauss will cause unnecessary page faults of home pages followed by unnecessary page state changes.

On the other hand, we found that PNN, Barnes-Hut and two IS have nil or less benefit from exclusive_read_write though there is no decrease in performance. Even though two IS are a single writer application, their migratory memory access pattern prevents the benefit of exclusive_read_write.

### 7.8 Limitations

The performance evaluation of HHEAC has limitations as follows:

- As described in Section 4.3.9, the benchmark applications we tested have not
enough parallel portions running over 16 and 32 nodes. In particular, IS-L and Gauss have negative speed-ups in all three protocols as the number of nodes increase. With the performance results of IS-L and Gauss, it is irrelevant to find out which protocol is more scalable than others.

- With much larger problem size of benchmark applications, more than 32 nodes should be used to evaluate the scalability of HHEAC with other protocols. This is due to the limitation of TreadMarks. Also, we only have 32 nodes in our cluster.

- More applications, in particular lock-based applications, should be tested in order to evaluate the benefits of HHEAC.

As shown in Table 7.9 when we compare between 16 and 32 nodes, even though HHEAC shows the speed-up improvements in four applications (PNN, Barnes-Hut, FFT and SOR) and less rapid slowdown in the other three applications except for Gauss, the data shows that doubling the number of nodes is not worthwhile with the
Table 7.9: Speed-up Improvements between 16 and 32 Nodes

<table>
<thead>
<tr>
<th>Apps</th>
<th>TreadMarks</th>
<th>HLRC</th>
<th>HHEAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>PNN</td>
<td>-85%</td>
<td>-15%</td>
<td>24%</td>
</tr>
<tr>
<td>Barnes-Hut</td>
<td>-26%</td>
<td>8%</td>
<td>10%</td>
</tr>
<tr>
<td>IS-B</td>
<td>-188%</td>
<td>-8%</td>
<td>-6%</td>
</tr>
<tr>
<td>IS-L</td>
<td>-265%</td>
<td>-90%</td>
<td>-80%</td>
</tr>
<tr>
<td>FFT</td>
<td>-2%</td>
<td>39%</td>
<td>10%</td>
</tr>
<tr>
<td>SOR</td>
<td>27%</td>
<td>12%</td>
<td>30%</td>
</tr>
<tr>
<td>Gauss</td>
<td>-99%</td>
<td>2%</td>
<td>-93%</td>
</tr>
</tbody>
</table>

current problem sizes. Mainly, the first limitation comes from the limited shared memory size allocated in TreadMarks, currently 64 MB. Since our HHEAC implementation is based on TreadMarks, this affects our system too. Another factor contributing to the first limitation is the limited size of transferred diffs during the lock ownership change. This limitation has prevented any further increase in the problem size of IS-L.

The second limitation is also due to the limitation of TreadMarks. Currently, up to 32 nodes can be used to run DSM applications. However, it would be interesting to see whether HHEAC with large problem sizes is scalable over more than 32 nodes.

As for the third limitation, we could not obtain as many lock-based applications as we can have barrier-only applications. In this thesis only two applications, PNN and IS-L, are used other than the simple lock application used in Section 7.4.3. Since lock-based applications have more benefits from HHEAC, in order to evaluate HHEAC completely, more lock-based applications should be tested over HHEAC.

### 7.9 Performance Comparison between message passing and DSM

We implemented message passing PNN in order to compare the performance of PNN between HHEAC and a message passing system. The chosen message passing system is LAM/MPI (Burns, Daoud and Vaigl, 1994). Message Passing Interface (MPI) is now the de facto standard of message passing programming. Therefore it is in our interest to see whether the performance over our HHEAC DSM system can be comparable with that over the MPI system. We chose PNN since it can represent many characteristics
of scientific parallel applications such as iterative and repetitive operations. The performance comparisons between DSM and PVM for other applications can be found elsewhere (Lu, 1995; Werstein et al., 2003).

Figure 7.31 shows the performance of PNN over HHEAC and the LAM/MPI. It shows that HHEAC has shorter execution times up to 8 nodes. However, from 16 nodes MPI outperforms HHEAC. Since the data communication in the message passing implementation of PNN is more efficient without the burden of maintaining memory coherence in HHEAC, the performance of PNN over MPI is more scalable as the number of nodes increases. For example over 32 nodes we measured the total data traffic over MPI to be 56 MB compared with 190 MB in HHEAC. Nonetheless, the figure shows that HHEAC has comparable scalability with MPI.
Chapter 8

Conclusions

8.1 Conclusions

Parallel programming with shared memory over a cluster-based system is a natural extension of multithreaded programming in a uniprocessor system in order to exploit the cluster computing power. A software DSM system is the answer to providing a scalable shared memory platform over a cluster-based system. The advantages of a software DSM system are its cost-effectiveness compared to a hardware DSM system, and that it provides a more familiar programming model compared to the message passing programming model.

However, the challenges for developing an efficient and scalable software DSM system over a cluster-based system are difficult to overcome since it is hard to optimize data communication traffic due to the need to provide a coherent view of memory over the physically separated local memories. Also, providing consistent memory in a software DSM system is more costly due to the high memory access latency and low bandwidth in the internetwork used. Furthermore, the size of a memory unit is relatively large, which causes more false sharing, thereby producing unnecessary page faults and data communication.

To overcome these inherent problems in a software DSM system, many relaxed memory consistency models and latency hiding techniques have been proposed as described in Chapter 2. Recently, the homeless and home-based protocols have been used in DSM systems. The two protocols are fundamentally different in terms of how to provide memory coherency. In this thesis, the home-based protocol has demonstrated more scalable performance as shown in Chapter 4. The reasons for the scalability in the home-based protocol are due to fewer hot spots, no garbage collection being required
and no diff accumulation.

However, the home-based protocol is not always the winner in all the applications we tested. TSP, which has fine-grained memory access patterns inside a critical section, shows that diff-based memory updating in the homeless protocol is more efficient. Also, without correct home assignment to shared pages, the home-based protocol shows poor performance, as shown in SOR and Gauss.

We concluded that even though the home-based protocol is more scalable, the homeless protocol can be more efficient when applications have frequent lock synchronisations and fine-grained memory access patterns inside a critical section. Also, without dynamic home assignment that can be adapted to a memory access pattern in an application, the home-based protocol would suffer from unnecessary page faults, differencing, twinning and data communication. A similar effect to the lazy differencing in the homeless protocol, which is very effective in applications showing a regular memory access pattern without migration, can be obtained in the home-based protocol as well if the home-based protocol has a dynamically adapted home assignment scheme. Thus, the home-based protocol can fully exploit the positive home effect.

Therefore, we designed a DSM system which combines the advantages of the homeless and home-based protocols. To maximize the advantages and make the implementation simpler, we have chosen ScC as the base memory consistency model and created a new memory consistency model. We call the protocol used in our system hybrid home-based EAC (HHEAC). Compared to the previous home-based ScC DSM implementations, HHEAC does not update home nodes between two consecutive barriers. Updating home nodes occurs only at the next barrier time in which all nodes find out the optimum home nodes for the dirty pages. The lazy home update not only reduces data traffic and the number of messages but also interrupts the computation in home nodes less often. HHEAC supports the fine-grained diff-based memory update used in the homeless protocol inside a critical section by eagerly sending the required diffs to the next lock owner.

The aggregated diff-based memory update inside a critical section in HHEAC is more efficient than the page-based memory update in conventional home-based DSM systems using the write-invalidate protocol, since only the data that are likely to be accessed inside a critical section are transferred as a single message. Furthermore, our diff integration technique provides more efficient fine-grained memory update inside a critical section, which reduces data communication. Since integrated CS diffs that were previously made inside the same critical section are piggybacked on a message
granting lock ownership and are applied to the next lock owner before entering the CS, the next lock owner experiences fewer page faults inside the critical section.

HHEAC also removes the shortcoming of the original home-based protocol — the home assignment problem. HHEAC finds the optimum home node for each dirty page and dynamically reassigns the home node in order to produce the minimum home update data traffic.

The performance evaluation of HHEAC with diff integration and dynamic home assignment has shown that HHEAC combines the strengths of the homeless and home-based protocols but mitigates the shortages of the home-based protocol. However, the added overheads due to the dynamic home assignment offset the advantages in some applications tested, such as IS-B which has a migratory memory access pattern and Gauss which has frequent barrier synchronisation during its computation. These data suggest that the dynamic home assignment is not very effective in migratory applications. Our initial dynamic home migration technique reduces the overheads of the dynamic home migration, so further improves Barnes-Hut and Gauss.

Below, we summarise the characteristics of HHEAC which are different from the previous software DSM systems:

1. Hybrid home-based implementation of EAC.
2. Novel diff integration technique used.
3. Novel dynamic home migration technique used.
4. Efficient and simpler implementation of the write-update protocol during lock synchronisation.

### 8.2 Future Research

As described in Section 7.8, the current implementation and the performance evaluation of HHEAC have several limitations. First, in order to be truly scalable, the HHEAC implementation should be able to not only allocate more shared memory but also add more nodes than the current implementation can. This will enable the use of larger problem sizes and a larger number of nodes than 32. This is a limitation of TreadMarks, on which the current HHEAC implementation is based. Future research directions include reimplementing HHEAC so that its performance can be evaluated with more nodes and larger problem sizes in order to better determine its scalability.
Second, an important area for future research is to seek the solution to the nested lock problem. The HHEAC programming model recommends not using a nested lock as illustrated in Figure 5.8. In particular, a nested lock should be carefully used that a shared variable is not updated under the protection of more than one lock between two consecutive barriers, which is the third constraint of the HHEAC programming model as specified in Section 5.4.

However, in case there is no other solution other than using a nested lock, the application may not be executed correctly under the current HHEAC implementation. The nested lock problem under the current HHEAC implementation is mainly due to the large granularity of the memory unit in the page-based DSM system. If HHEAC were implemented in a DSM system where each shared variable is the memory unit, the nested lock problem could be solved much more easily. As briefly mentioned in Section 5.4, a solution under the page-based DSM system would add other fields, such as diff_version_number and nest_flag, to the diff data structure, and slightly complicate diff application in a home node at a barrier time.

Related to the HHEAC programming model, our system also needs the addition of a conditional synchronisation primitive in order to implement efficient producer-consumer type communication, as required in the example program in Figure 5.8.

The design and implementation techniques used in HHEAC are not only applicable to any software page-based DSM system but also can be employed to any scalable shared memory systems which need an efficient memory coherence protocol. It would be interesting to see whether an object-based software DSM system, a hardware DSM system or a multithreaded DSM system could have benefits from HHEAC. Similarly, our contributions can be applied to give an efficient implementation of the OpenMP Application Program Interface (API) (OpenMP, 2005).

Finally, the applications tested in this thesis reflect only the use of DSM in scientific areas. However there are other areas in which DSM can be used. Such systems can be used not only for high performance computing as presented in this thesis but also for efficient distributed computing. The latter is another area that HHEAC can be used.

Possible application areas of the latter include the following:

- Reducing implementation complexity of distributed applications: The abstraction of single shared memory in distributed computing helps a programmer to develop an application more conveniently. The idea of HHEAC can be applied to provide an efficient implementation of this abstraction. InterWeave (Chen, Tang, Chen, Dwarkadas and Scott, 2001; Chen, Tang, Chen, Dwarkadas and...
Scott, 2002) is a good example of using software DSM for this purpose in a wide-area distributed system. InterWeave provides a single logical “shared state” of program variables which are cached over many nodes. InterWeave simplifies distributed computing by replacing remote invocation for memory coherence with access to a single logical shared state. The consistency model and coherence protocol used in InterWeave are similar to those of HHEAC in the way that 1) memory segments are protected by reader-writer locks. 2) an InterWeave server acts as a home node in HHEAC. However, in InterWeave, write updates created within a critical section are transferred via the server between releasing and acquiring nodes using the write-invalidate protocol whereas in HHEAC, these updates are directly transferred from the last lock owner to the current lock owner using the write-update protocol. In this thesis, we argued that the write-update protocol is more efficient for memory coherence inside a critical section. We believe that distributed applications such as InterWeave would benefit from using the HHEAC protocol.

- Cluster-based database management systems (DBMSs): The cache coherence problem and its solution are similar between software DSM systems and cluster-based DBMS (Franklin, Carey and Livny, 1997; Wu, fei Chuang and Lilja, 2004). HHEAC can be used to implement a cache consistency protocol in parallel DBMSs in the same way that LRC has been used in DSMIO (Osthoff, Bentes, Ariosto, Mattoso and de Amorim, 2002).

- Distributed software games: The techniques used in HHEAC can help to implement consistent memory among distributed game players. A gaming framework on top of a DSM system has been developed before (Schöttner, Wende, Göckelmann, Bindhammer, Schmid and Schulthess, 2003). However, that framework was implemented on a specialized OS (Wende, Schöttner, Göckelmann, Bindhammer and Schulthess, 2002). This approach of implementing DSM in the OS is very different from HHEAC. It would be worth investigating how the techniques used in HHEAC can be exploited in implementing distributed game software.
References


Appendix A

Source Code for two Home-based DSM implementations

Due to the copyright of TreadMarks, we only present home_update.c used in HLRC and HHEAC respectively to avoid possible disclosure of the source code of TreadMarks.

A.1 Home-based LRC DSM Implementation

A.1.1 home_update.c

```c
#include "Tmk.h"

struct req_typ req_home_update = {0, 0, REQ_HOME_UPDATE};
struct iovec req_home_update_iov[MSG_MAXIOVLEN] =
    {{&req_home_update, sizeof(req_home_update)}};
struct msghdr req_home_update_hdr = {0, 0, req_home_update_iov, 0, 0, 0};

void update_home_with_diffs() {
    int i;
    req_home_update.from = Tmk_proc_id;
    for(i=0; i < Tmk_nprocs; i++) {
        if(i == Tmk_proc_id) continue;
        send_diffs(i);
    }
    Tmk_diff_repo();
}

void send_diffs(int i) {
    int fd = req_fd_[i];
    int j = 0, k = 1, size = sizeof(req_home_update),n;
    sigset_t mask;
    unsigned short page_no;
    int rep_seqno;
    page_t page;
    write_notice_t write_notice;
    struct hostent * hp;
    struct sockaddr_in addr;
```
if (diff_update_pages_[i][0] == Tmk_npages) {
    return;
}

do {
    page_no = diff_update_pages_[i][j];
    j++;
    page = &page_array_[page_no];
    write_notice = page->write_notice_[Tmk_proc_id];
    if ((size += write_notice->diff_size) > MTU) {
        req_home_update_hdr.msg_iovlen = k;
        req_home_update.seqno = req_seqno += SEQNO_INCR;
        rexmit1:
            while (0 > sendmsg(fd, &req_home_update_hdr, 0))
                Tmk_errno_check("send_diffs<sendmsg1>");
            k = 1;
            size = sizeof(req_home_update) + write_notice->diff_size;
            Tmk_tout_flag = 0;
            setitimer(ITIMER_REAL, &Tmk_tout, NULL);
            sigio_mutex(SIG_UNBLOCK, &ALRM_and_IO_mask, &mask);
        retry1:
            if ((n = recv(fd, (char *)&rep_seqno,sizeof(rep_seqno),0)) < 0)
                if(Tmk_tout_flag) {
                    if(Tmk_debug)
                        Tmk_err("timeout: senddiffs first send\n");
                    sigio_mutex(SIG_SETMASK, &mask, NULL);
                    goto rexmit1;
                }
                else if(errno == EINTR)
                    goto retry1;
                else
                    Tmk_perrexit("<recv1>Tmk_send_diffs");
            if(rep_seqno != req_home_update.seqno) {
                if(Tmk_debug)
                    Tmk_err("bad seqno:%d>Tmk_send_diffs(first recv):seqno == %d(received:%d)\n",i, req_home_update.seqno, rep_seqno);
                goto retry1;
            }
            Tmk_stat.messages++;
            Tmk_stat.bytes += n;
            sigio_mutex(SIG_SETMASK, &mask, NULL);
        }
        if ((k > 2) && (write_notice->diff ==
            ((caddr_t) req_home_update_iov[k-1].iov_base +
            req_home_update_iov[k-1].iov_len)))
            req_home_update_iov[k-1].iov_len += write_notice->diff_size;
        else {
            req_home_update_iov[k].iov_base = write_notice->diff;
            req_home_update_iov[k].iov_len = write_notice->diff_size;
            if ((k += 1) == MSG_MAXIOVLEN) {
                req_home_update_hdr.msg_iovlen = k;
                req_home_update.seqno = req_seqno += SEQNO_INCR;
                rexmit2:
                    while (0 > sendmsg(fd, &req_home_update_hdr, 0))
                    Tmk_errno_check("send_diffs<sendmsg2>");
            }
        }
    }
}

156
Tmk_errno_check("send_diff<sendmsg2>");
k = 1;
size = sizeof(req_home_update);
Tmk_tout_flag = 0;
setitimer(ITIMER_REAL, &Tmk_tout, NULL);
sigio_mutex(SIG_UNBLOCK, &ALRM_and_IO_mask, &mask);
retry2:
if((n = recv(fd, (char *)&rep_seqno,sizeof(rep_seqno),0)) < 0)
   if(Tmk_tout_flag){
      if(Tmk_debug)
         Tmk_err("timeout: senddiffs second send\n");
      sigio_mutex(SIG_SETMASK, &mask, NULL);
      goto rexmit2;
   } else if(errno == EINTR)
      goto retry2;
   else
      Tmk_perrexit("<recv2>send_diffs");
if(rep_seqno != req_home_update.seqno) {
   if(Tmk_debug)
      Tmk_err("bad seqno:%d>send_diffs(first recv):seqno ==
      %d(received:%d)\n", i, req_home_update.seqno, rep_seqno);
   goto retry2;
}
Tmk_stat.messages++;
Tmk_stat.bytes += n;
sigio_mutex(SIG_SETMASK, &mask, NULL);
}
page_no = diff_update_pages_[i][j];
} while (page_no != Tmk_npages);
if (k > 1) {
   req_home_update_hdr.msg_iovlen = k;
   req_home_update.seqno = req_seqno += SEQNO_INCR;
rexmit3:
   while (0 > sendmsg(fd, &req_home_update_hdr, 0))
      Tmk_errno_check("send_diffs<sendmsg3>");
   Tmk_tout_flag = 0;
   setitimer(ITIMER_REAL, &Tmk_tout, NULL);
   sigio_mutex(SIG_UNBLOCK, &ALRM_and_IO_mask, &mask);
   retry3:
if( 0 > recv(req_fd_[i], (char *)&rep_seqno,sizeof(rep_seqno),0))
   if(Tmk_tout_flag){
      if(Tmk_debug)
         Tmk_err("timeout: senddiffs last send i:%d\n",i);
      sigio_mutex(SIG_SETMASK, &mask, NULL);
      goto rexmit3;
   } else if(errno == EINTR)
      goto retry3;
   else
      Tmk_perrexit("<recv3>send_diffs");
if(rep_seqno != req_home_update.seqno) {
   if(Tmk_debug)
Tmk_err("bad seqno:%d>send_diffs(first recv):seqno ==
%d(received:%d)\n", i, req_home_update.seqno, rep_seqno);
goto retry3;
}
Tmk_stat.messages++;
Tmk_stat.bytes += n;

diff_update_pages_[i][0] = Tmk_npages;
sigio_mutex(SIG_SETMASK, &mask, NULL);
}

void update_home_sigio_handler(const struct req_typ *update_req, int size){
caddr_t diff = (void *)&update_req[1];
caddr_t diff_end = diff + size;
unsigned short page_no;
page_t page; int times=0;
struct sockaddr_in addr;
struct hostent *hp;

for(;;) {
  if(times == 0) {
    page_no = *(unsigned short *)diff;
    times++;
  }
  else {
    page_no = *((unsigned short *)diff - 4);
    diff -= 8;
  }

diff=(unsigned short*)diff+1;
page = &page_array_[page_no];
if(page->state == read_only){
  if(0 > mprotect(page->vadr, Tmk_page_size, PROT_READ|PROT_WRITE))
    Tmk_perrexit("<mprotect>update_home_sigio_handler");
  diff = diff_apply(page->vadr,diff);
  if(0 > mprotect(page->vadr, Tmk_page_size, PROT_READ))
    Tmk_perrexit("<mprotect>update_home_sigio_handler");
}
else if(page->state == read_write) {
  diff = diff_apply(page->vadr,diff);
}
else Tmk_perrexit("home pages shouldn’t have invalid pages");
if(diff == diff_end) {
  goto send_ack;
}
}

send_ack:
if(send(rep_fd_[update_req->from],(char *)&update_req->seqno,
  sizeof(update_req->seqno),0)<0)
  Tmk_err("Can’t send acknowledement");
}
A.2 Hybrid Home-based EAC DSM implementation

A.2.1 home_update.c

#include "Tmk.h"

struct req_typ req_home_update = {0, 0, REQ_HOME_UPDATE};
struct iovec req_home_update_iov[MSG_MAXIOVLEN] =
    {{&req_home_update, sizeof(req_home_update)}};
struct msghdr req_home_update_hdr = {0, 0, req_home_update_iov, 0, 0, 0};

diff_info_t diff_update_pages_[NPROCS][NPAGES];//initialized -1
unsigned short home_index_[NPROCS];//number of total diffs for each node
unsigned short page_number_[NPROCS][NPAGES];
unsigned short diff_number_[NPROCS][NPAGES];
unsigned char last_again_no_inserted[NLOCKS];//for ordering cs diffs by happen-before
unsigned char done[NLOCKS];
unsigned char first_found_lock[NLOCKS];
//flag whether a diff is the top or not in diff_info data structure
unsigned short lock_home_index_[NLOCKS];//index for each lock id
unsigned short diff_page_index_[NPROCS];//index of next element of number of diff and page number
unsigned char done_lock_[NLOCKS];

void update_homes() {
    page_t end = &page_dirty;
    page_t page = end->prev, page1;
    diff_info_t diff;
    int i,j=0,k=0,num_ncsdiff=0, last_again;
    unsigned short node,page_no; char page_no_inserted = 0;
    int ncs_diff_number = 0; //number of ncs diffs
    int ncs_diff_index; //next home_index[node] for a ncs diff
    int num_last_lock_owned,page_index;
    for(i=0;i<NLOCKS;i++)
        if(last_lock_owner[i]==1) {
            last_lock_id[j++]=i;
            last_lock_owner[i]=0;//no need to use so reset now
        }
    num_last_lock_owned = j;
    if(num_last_lock_owned) {
        for(i=0;i<num_last_lock_owned;i++) {
            for(page_index = 0; page_index < lock_page_next[last_lock_id[i]]; page_index++) {
                page_no = lock_page_list[last_lock_id[i]][page_index];
                page1 = &page_array_[page_no];
                if(done_page[page_no]) continue;
                if((node = page1->home) != Tmk_proc_id) {
                    done_page[page_no] = 1;
                    page_number_[node][diff_page_index_[node]] = page_no;
                    for(diff = page1->diff_info_[0]; diff; diff=diff->next) {
                        
                        
                        159
if (k == num_last_lock_owned) break;
if (done_lock_[diff->lock_id] == 1) continue;
for (j = 0; j < num_last_lock_owned; j++) {
    if (diff->lock_id == last_lock_id[j]) {
        if (first_found_lock[diff->lock_id]) {
            diff_update_pages_[node][lock_home_index_[diff->lock_id]] = diff;
            lock_home_index_[diff->lock_id]--;
            if (diff->again_no == 1) {
                k++;
                done_lock_[diff->lock_id] = 1;
            }
        }
        else { // first top diff of each lock id will be here first
            first_found_lock[diff->lock_id] = 1;
            diff_number_[node][diff_page_index_[node]] += diff->again_no;
            home_index_[node] += diff->again_no;
            lock_home_index_[diff->lock_id] = home_index_[node] - 1;
            diff_update_pages_[node][lock_home_index_[diff->lock_id]] = diff;
            lock_home_index_[diff->lock_id]--;
            if (diff->again_no == 1) {
                k++;
                done_lock_[diff->lock_id] = 1;
            }
        }
    }
}
}
else { // first top diff of each lock id will be here first
    first_found_lock[diff->lock_id] = 1;
    diff_number_[node][diff_page_index_[node]] += diff->again_no;
    home_index_[node] += diff->again_no;
    lock_home_index_[diff->lock_id] = home_index_[node] - 1;
    diff_update_pages_[node][lock_home_index_[diff->lock_id]] = diff;
    lock_home_index_[diff->lock_id]--;
    if (diff->again_no == 1) {
        k++;
        done_lock_[diff->lock_id] = 1;
    }
}
break;
}
}
k = 0;
diff_page_index_[node]++; // one page finished and for next page
for (j = 0; j < num_last_lock_owned; j++) {
    first_found_lock[last_lock_id[j]] = 0;
    done_lock_[last_lock_id[j]] = 0;
}
}
}
}
}
for (i = 0; i < num_last_lock_owned; i++) {
    for (page_index = 0; page_index < lock_page_next[last_lock_id[i]]; page_index++)
        done_page[lock_page_list[last_lock_id[i]][page_index]] = 0;
    lock_page_next[last_lock_id[i]] = 0;
}
if (page != end) {
    end->prev = end->next = end;
    do {
        page_t partner = page->partner;
        partner[1].dirty_toggle = page->dirty_toggle = 0;
        do {
            if ((node = partner->home) != Tmk_proc_id) {
                k = 0;
                for (diff = partner->diff_info_[1]; diff; diff = diff->next) {
                    ncs_diff_number ++;
                }
            }
        }
    }
} else if (page == end) {
    done_page[lock_page_list[last_lock_id[i]]] = 0;
    lock_page_next[last_lock_id[i]] = 0;
}
ncs_diff_index = ncs_diff_number + home_index_[node];
home_index_[node] = ncs_diff_index;
for(diff = partner->diff_info_[1]; diff; diff=diff->next) {
    if(!page_no_inserted) {
        page_number_[node][diff_page_index_[node]] = partner - page_array_;  
        diff_number_[node][diff_page_index_[node]] = ncs_diff_number;
        diff_page_index_[node]++;
        page_no_inserted = 1;
    }
    diff_update_pages_[node][ncs_diff_index-1] = diff;
    ncs_diff_index--;
}
page_no_inserted = 0;
ncs_diff_number = 0;
}
partner->diff_info_[1] = 0;
}
while ((partner -= 1) >= page);
}
while ((page = page->prev) != end);
}

for(i=0; i < Tmk_nprocs; i++) {
    if(i == Tmk_proc_id || home_index_[i] == 0) continue;
    send_diffs(i);
}

for(i=0; i < Tmk_npages; i++)
    page_array_[i].diff_info_[0] = 0;
}

void send_diffs(int i) {
    int fd = req_fd_[i];
    diff_info_t diff;
    int current_diff_index = 0,k=1,size=sizeof(req_home_update),j;
    char page_inserted = 0,first_diff_inserted = 0;
    unsigned short page_no, previous_page_no, diffs_remained;
    int diff_page_index = 0;
    char before_page_no;
    sigset_t mask; int n, rep_seqno; int m; caddr_t diff_print;
    req_home_update.from = Tmk_proc_id;
    do {
        do {
            diff = diff_update_pages_[i][current_diff_index];
            if(!page_inserted) { //this if statement just once executed for each page
                size += 4;
                page_inserted = 1;
                previous_page_no = page_number_[i][diff_page_index];
                diffs_remained = diff_number_[i][diff_page_index];
                before_page_no = 1;
                first_diff_inserted = 1;
            }
        } else {
            size += diff->diff_size;
            before_page_no = 0;
        }
        }
    }

}
if (size > MTU) {
    mtu:
    req_home_update_hdr.msg_iovlen = k;
    req_home_update.seqno = req_seqno += SEQNO_INCR;
    rexmit1:
    while (0 > sendmsg(fd, &req_home_update_hdr, 0))
        Tmk_errno_check("send_diffs<sendmsg1>");
    k = 1;
    Tmk_tout_flag = 0;
    setitimer(ITIMER_REAL, &Tmk_tout, NULL);
    sigio_mutex(SIG_UNBLOCK, &ALRM_and_IO_mask, &mask);
    retry1:
    if((n = recv(fd, (char *)&rep_seqno,sizeof(rep_seqno),0)) < 0)
        if(Tmk_tout_flag) {
            if(Tmk_debug)
                Tmk_err("timeout: senddiffs first send\n");
            sigio_mutex(SIG_SETMASK, &mask, NULL);
            goto rexmit1;
        } else if(errno == EINTR)
            goto retry1;
        else
            Tmk_perrexit("<recv1>Tmk_send_diffs");
    if(rep_seqno != req_home_update.seqno) {
        if(Tmk_debug)
            Tmk_err("bad seqno:%d>Tmk_send_diffs(first recv):seqno == %d(received:%d)\n",i, req_home_update.seqno, rep_seqno);
        goto retry1;
    }
    Tmk_stat.messages++;
    Tmk_stat.bytes += n;
    sigio_mutex(SIG_SETMASK, &mask, NULL);
    if(before_page_no) //previous packet sent just after page no and diff no
        size = sizeof(req_home_update) + 4;
    else
        size = sizeof(req_home_update)+ diff->diff_size + 4;
    req_home_update_iov[k].iov_base = &page_number_[i][diff_page_index];
    req_home_update_iov[k].iov_len = 2;
    k++;
    diff_number_[i][diff_page_index] = diffs_remained;
    //necessary to have remaining number of diffs
    req_home_update_iov[k].iov_base = &diff_number_[i][diff_page_index];
    req_home_update_iov[k].iov_len = 2;
    k++;
    page_inserted = 0;
    first_diff_inserted = 0;
    diff = diff_update_pages_[i][current_diff_index];
}
else {
    size = sizeof(req_home_update)+ diff->diff_size + 4;
    req_home_update_iov[k].iov_base = &page_number_[i][diff_page_index];
    req_home_update_iov[k].iov_len = 2;
    k++;
    diff_number_[i][diff_page_index] = diffs_remained;
    req_home_update_iov[k].iov_base = &diff_number_[i][diff_page_index];
req_home_update_iov[k].iov_len = 2;
k++;
req_home_update_iov[k].iov_base = diff->diff;
req_home_update_iov[k].iov_len = diff->diff_size;
k++;
diff = diff_update_pages_[i][current_diff_index++];
if(!diffs_remained--) break;
page_inserted = 0;
first_diff_inserted = 0;
diff = diff_update_pages_[i][current_diff_index];
}
} //if(size > MTU)
if(page_inserted && first_diff_inserted) {
//now actual page and diff number inserted into iov
req_home_update_iov[k].iov_base = &page_number_[i][diff_page_index];
req_home_update_iov[k].iov_len = 2;
k++;
if(k == MSG_MAXIOVLEN) {
    goto maxiov;
}
req_home_update_iov[k].iov_base = &diff_number_[i][diff_page_index];
req_home_update_iov[k].iov_len = 2;
k++;
if(k == MSG_MAXIOVLEN) {
    k--;
    goto maxiov;
}
first_diff_inserted = 0;
}
req_home_update_iov[k].iov_base = diff->diff;
current_diff_index++;
if((size += req_home_update_iov[k].iov_len = diff->diff_size) > MTU) {
    current_diff_index--;
    goto mtu;
}
diffs_remained--;
if ((k += 1) == MSG_MAXIOVLEN) {
maxiov:
    req_home_update_hdr.msg_iovlen = k;
    req_home_update_hdr.seqno = req_seqno += SEQNO_INCR;
    rexmit2:
    while (0 > sendmsg(fd, &req_home_update_hdr, 0))
        Tmk_errno_check("send_diff<sendmsg2>");
    Tmk_tout_flag = 0;
    setitimer(ITIMER_REAL, &Tmk_tout, NULL);
    sigio_mutex(SIG_UNBLOCK, &ALRM_and_IO_mask, &mask);
    retry2:
    if((n = recv(fd, (char *)&rep_seqno,sizeof(rep_seqno),0)) < 0)
        if(Tmk_tout_flag){
            if(Tmk_debug)
                Tmk_err("timeout: senddiffs second send\n");
            sigio_mutex(SIG_SETMASK, &mask, NULL);
            goto rexmit2;
        }
else if (errno == EINTR)
    goto retry2;
else
    Tmk_perrexit("<recv2>send_diffs");
if (rep_seqno != req_home_update.seqno) {
    if (Tmk_debug)
        Tmk_err("bad seqno:%d>send_diffs(first recv):seqno ==
        %d(received:%d)\n", i, req_home_update.seqno, rep_seqno);
    goto retry2;
}
Tmk_stat.messages++;
Tmk_stat.bytes += n;
sigio_mutex(SIG_SETMASK, &mask, NULL);
k = 1;
size = sizeof(req_home_update) + 4;
req_home_update_iov[k].iov_base = &page_number_[i][diff_page_index];
req_home_update_iov[k].iov_len = 2;
k++;
diff_number_[i][diff_page_index] = diffs_remained;
req_home_update_iov[k].iov_base = &diff_number_[i][diff_page_index];
req_home_update_iov[k].iov_len = 2;
k++;
}
} while (diffs_remained > 0);
page_inserted = 0;
diff_page_index++;
} while (home_index_[i] > current_diff_index);
assert(diff_page_index == diff_page_index_[i]);
if (k > 1) {
    req_home_update_hdr.msg_iovlen = k;
    req_home_update.seqno = req_seqno += SEQNO_INCR;
rexmit3:
    while (0 > sendmsg(req_fd_[i], &req_home_update_hdr, 0))
        Tmk_errno_check("send_diffs<sendmsg3>");
    Tmk_tout_flag = 0;
    setitimer(ITIMER_REAL, &Tmk_tout, NULL);
    sigio_mutex(SIG_UNBLOCK, &ALRM_and_IO_mask, &mask);
    retry3:
        if (0 > (n = recv(req_fd_[i], (char *)rep_seqno, sizeof(rep_seqno), 0)))
            if (Tmk_tout_flag)
                if (Tmk_debug)
                    Tmk_err("timeout: send_diffs last send i:%d\n", i);
                sigio_mutex(SIG_BLOCK, &ALRM_and_IO_mask, NULL);
                goto rexmit3;
        else if (errno == EINTR)
            goto retry3;
        else
            Tmk_perrexit("<recv3>send_diffs");
    if (rep_seqno != req_home_update.seqno) {
        if (Tmk_debug)
            Tmk_err("bad seqno:%d>send_diffs(first recv):seqno ==
            %d(received:%d)\n", i, req_home_update.seqno, rep_seqno);
        goto retry3;
Tmk_stat.messages++;
Tmk_stat.bytes += n;
for(j=0; j < diff_page_index_[i]; j++) {
    page_number_[i][j]=0;
    diff_number_[i][j]=0;
}
home_index_[i] = 0;
diff_page_index_[i] = 0;
sigio_mutex(SIG_SETMASK, &mask, NULL);
} //if (k > 1)

void update_home_sigio_handler(const struct req_typ *update_req, int size) {
    caddr_t diff = (void *)&update_req[1];
    caddr_t diff_end = diff + size - sizeof(struct req_typ);
    diff_info_t diff_info;
    page_t page;
    unsigned char from = update_req->from;
    unsigned short num_of_diffs;caddr_t diff_print;
    for(;;) {
        page = &page_array_[*(unsigned short *)diff];
        diff += 2;
        num_of_diffs = *(unsigned short *)diff;
        diff += 2;
        do {
            diff = diff_apply(page->vadr, diff);
            num_of_diffs--;
        } while (num_of_diffs > 0);
        if(diff+4 >= diff_end) break;
    }

    if(send(rep_fd_[from], (char *)&update_req->seqno,
            sizeof(update_req->seqno),0) < 0)
    Tmk_err("Can't send update home acknowledgement");
}