PERFORMANCE OPTIMIZATION OF SOFTWARE DISTRIBUTED SHARED MEMORY SYSTEMS

WEISONG SHI

Department of Computer Science
Wayne State University
Detroit, USA

Higher Education Press, China

To My Wife, Wei, and My Parents

Abstract

Software Distributed Shared Memory Systems (DSMs, or Shared Virtual Memory) are advocated to be an ideal vehicle for parallel programming because of its combination of programmability of shared memory and scalability of distributed memory systems. The challenge in building a software DSM system is to achieve good performance over a wide range of parallel programs without requiring programmers to restructure their shared memory parallel programs or less modification to their sequential programs. The overhead of maintaining consistency in software and the high latency of sending messages make this target difficult to achieve.

In this dissertation, we try to improve the performance of software DSM system by examining cache coherence protocol, memory organization scheme, system overhead, load balancing, and communication optimization respectively.

By analyzing the disadvantages of snoopy and directory-based cache coherence protocol, we propose a lock-based cache coherence protocol for scope consistency. The unique characteristic of this protocol is applying the “home” concept to not only data information but also coherence information. Each coherence information has a static home according to the corresponding synchronization object. As such, the lock-based protocol has least coherence related overheads for ordinary read or write misses. Moreover, the lock-based protocol is free from the overhead of maintaining the directory. Based on this protocol, we designed a simple but efficient software DSM system named JIAJIA. JIAJIA employes home-based but with a novel memory organization scheme through which the overhead of address translation is eliminated and a large shared address space combined by the physical memories of multiple nodes is supported.

Based on the detailed analysis about the system overhead of software DSM systems, we propose several optimal techniques, such as read notice, hierarchical barrier implementation, cache only write detection scheme, etc., to reduce system overheads in home-based software DSMs. Performance evaluation results show that the performance can be improved greatly by these techniques.

The dissertation proposes an affinity-based self scheduling (ABS) method for loop scheduling. ABS achieves the best performance compared with other schemes proposed in the past in meta-computing environment because of the reduction of synchronization overhead and waiting time resulting from load imbalance, and is comparable with the best scheduling schemes in dedicated environment. For iterative scientific applications, we argue that a task should be a combination of computation subtask and it’s corresponding data subtask firstly. Then we propose a task migration scheme which integrates computation migration and data migration together for achieving better resource utilization in meta-computing environment. The relationship between computation subtask and it’s corresponding data is mined by our run-time system, and data migration is completed by a novel home migration scheme, which is an important characteristic of JIAJIA. To our knowledge, this is the first implemention in home-based software DSMs.

Finally, the dissertation designs and implements a user-level communication optimization scheme (JMCL1.0) which is specific to home-based software DSM system on Myrinet. Consequently, the frequency of memory copy of one communication is reduced from 7 to 2. Furthermore, the interface between communication substrate and software DSMs become simpler than that of traditional UDP/IP network protocol.

Acknowledgements

There is a proverb that says a journey of thousands miles begins with a single step, which is why I am dedicating this dissertation to my parents. My desire to seek higher education could not have been possible without the infinite love and support of them. The silent sacrifice of their life made my life shine.

I would like to thank my advisors Prof. Peisu Xia and Prof. Zhimin Tang for their guidance on my dissertation. Together with Prof. Weiwu Hu, they have provided me with advice, encouragement, support, and the opportunity of working in a first-class research environment throughout my graduate career. I was very fortunate to have Prof. Peisu Xia as my advisor. I cannot express how deep my gratitude is for her thoughtful guidance and serious research paradigm. She taught me not only the way to be a researcher, but also the way to be a judicious people. I wish to thank Prof. Zhimin for his guidance and friendship. Without his help, I would not finish my dissertation research so successfully. He taught me the style of research, from which I have learned and benefited greatly. I wish I could have learned half of what Zhimin had taught me about how to think and especially, how to write. I will not forget those midnights, together with Weiwu, to debug our JIAJIA system. Prof. Weiwu's insight, creativity, and diligent spirit gives me encouragement to conquer any problems in the study.

My time at Institute of Computing Technology was made special by several wonderful colleagues whom I shared so many days and, especially, so many nights. Zhiyu helped me to port several applications from SPLASH2 to JIAJIA system. It is really exciting to talk with her, her humor and many constructive comments about the draft of my papers give me great help in the past 3 years. Xianghui, who like an elder brother, make our work space fulfilled with happiness.

I would like to thank Yuquan of Institute of Software, who makes my graduate life more interesting and pleasant. I am deeply benefit from his many constructive comments about life and study.

I would like to thank Prof. Henri E. Bal of Vrije Universteit. Without his advice, the English version of this dissertation has no chance to appear. I am greatly indebted to him for his careful review of my dissertation. Thanks also to M. Rasit Eskicioglu of University of Alberta, his great help make our JIAJIA system propagate so fastly. His many good suggestions to the draft of my papers also give me great help. It is his help make my dissertation more easy to read. Thanks also gives to Prof. Liviu Iftode of Rutgers University, who give me detailed comments about our papers and review my dissertation carefully. I appreciate Dr. Evan Speight of Rice University, Prof. John Carter of Utah University, Prof. Zvi M. Kedem of New York University and Prof. Christine Morin of IRSIA/INRIA for their careful review of this dissertation. Discussion with Rauol Bhoedjang, Asek Plaat, and Thilo Kielmann of Vrije Universteit give me many inspiration and extend my viewpoint significantly.

Thanks also give to many staffs of Center of High Performance Computing, and Institute of Computing Technology.

Finally, I wish to acknowledge the National Center of Intelligent Computing System (NCIC) of China for allocation of computer time, which provide excellent hardware environment for our test.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past decade, software DSM systems, or shared virtual memory (SVM) systems, have been extensively studied to provide a good compromise between programmability of shared memory multiprocessors and scalability of message passing multicomputers. Many software DSM systems, such as Ivy[78], Midway[11], Munin[19], TreadMarks[71], Quarks[122], CVM[68], and Brazos [116] have been implemented on the top of message passing hardware or on networks of workstations.

Instead of implementing the convenient shared memory programming model with hardware, software DSMs support this abstraction on the top of message passing hardware which is more scalable and less expensive to build. However, software DSMs suffer from the high system overhead of protocol invocation and processing, large granularity of communication and coherence which causes problems of false sharing and extra communication, and high communication latencies which is critical to performance in NOW (Network of Workstations) based software DSMs. Fortunately, software DSM systems can use sophisticated protocols to improve performance. Many techniques, such as multiple writer protocol[19], lazy release consistency[70], probowner directory[78], dynamic prefetching[67], compiler support[138, 29], and hardware support[14], have been proposed to alleviate false sharing, reduce remote communications, hide communication latency, and reduce protocol processing overheads.

This dissertation proposes a new cache coherence protocol and a novel memory organization scheme and implements the JIAJIA software DSM system which is simpler, and more efficient and scalable than conventional approaches. It also introduces an affinity-based self scheduling scheme and a dynamic task migration scheme for load balancing in meta-computing environment. Finally, the dissertation takes the challenge to design a home-based software DSMs specific communication substrate.

This chapter presents a brief survey of software DSM systems. It analyzes software DSMs in the aspects of implementation method, memory organization, memory consistency model, cache coherence protocol, and application programming interface. Characteristics of over 20 representative software DSM systems are reviewed. Recent progress in software DSMs and some open questions are also discussed. Finally, the contributions and organization of this dissertation are listed.

1.1 Basic Idea of Software DSM

A *software DSM system* is a single address space shared by a number of processors (also named shared virtual memory space), as shown in Figure 1.1. Any processor can access any memory location in the whole address space directly[79]. Each processor has a *software DSM layer*. Software DSM layers implement the mapping between local memories and the shared virtual memory address space. Other than mapping, their chief responsibility is to keep the address space *coherent* at all times. That is, the value returned by a read operation is always the same as the value written by the most recent write operation to the same address. Application programs can use the shared virtual memory just as they do on a traditional virtual memory except that processes can run on different processors in parallel.

Figure 1.1: Basic idea of software DSM.

In a classical software DSM system which involves sequential consistency memory model, , a shared virtual memory address space is partitioned into *pages*, which named *page-based* software DSM system. Pages can be replicated and migrated between processors on demand, just as cache line in hardware DSM system. In order to keep the multiple copies coherent, the software DSM system must supply a mechanism to maintain the coherency between them, which are named *cache coherence protocol*. Generally, the state space of each page is *read-only, read-write, invalid*. Pages that marked *read-only* can have copies residing in the physical memories of many processors at the same time, but a page marked *read-write* can reside in only one processor's main memory. The software DSM layer views its local memory as a large cache of the shared virtual memory address space for its associated processor, and manages it with full associative mode at page granularity. Like the traditional *virtual memory*, the shared memory itself exists only *virtually*. A memory reference causes a page fault when the page is not in a processor's current physical memory. When this happens, the software DSM layer retrieves the page from the memory of another processor. If the page of the faulting memory reference has copies on other processors, then the corresponding software DSM layer must cooperate to keep the memory coherent.

A very simple form of shared virtual memory coherence is illustrated in Figure 1.2, following an invalidation protocol. At the beginning, no node has a copy of the gray shared virtual page with which we are concerned. Events occur in the order 1, 2, 3, 4. Read 1 (R(1)) incurs a page fault during address translation and fetches a copy of the page to P0. Read 2 (R(2)) incurs a page fault and fetches a read-only copy to P1 (from P0), but at two different physical addresses in the two memories. Write 3 (W(3)) incurs a page fault (write to a read-only page), and page

Figure 1.2: Illustration of simple software DSM system.

fault handler determines that P1 has a copy and causes it to be invalidated. P0 now obtains read-write right to the page, which is like the modified or dirty state. When Read 4 (R(4)) by P1 tries to read a location on the invalid page, it incurs a page fault and fetches a new copy from P0. A few aspects are worthy of attention. First, the physical address of the page in processor P1's local memory may be completely different from that of the copy in processor P0's local memory, but the pages have the same shared virtual address. Second, a page fault handler must know or determine from where to obtain the up-to-date copy of a page that it has missed on and what pages to invalidate if needed, before it can set the page's access rights as appropriate and return control to the application process. Every page may have a home, determined by its virtual address, and a sharing list similar to a directory entry maintained at the home, or more elaborate mechanisms may be used, as we will see in the following section. In fact, software DSM protocols tend to be more and more complex.

The problems with page-based shared virtual memory are: (i) the high overhead of communication, (ii) the high overheads of protocol invocation and processing, and (iii) the large granularity of coherence and communication.

The first issue is implicit in software DSM systems since many of them are implemented on networks of workstations (NOWs) or multicomputers, where the default communication protocol is TCP/IP. This protocol will result in large software overhead essentially. The second issue is expensive because most of the work is completed by software on a general-purpose uniprocessor. Page faults take time to cause an interrupt and to switch into the interrupt processing program, the protocol processing itself is done in software. On a representative software DSM system in 1997, the round-trip cost of satisfying a remote page fault ranges from a hundred microseconds with aggressive system software support to over ten milliseconds. In addition, since protocol processing is typically done on the main processor(due to the goal of using commodity uniprocessor nodes with no additional hardware support), even incoming requests interrupt the processor, pollute the cache and slow down the currently running process.

The large granularity of communication and coherence is problematic for two reasons: first, if the spatial locality of the application is not very good, it will cause a lot of fragmentation in communication (i.e., only a word is needed, but a whole page is fetched). Second, it can easily lead to false sharing (i.e., sharing of blocks without true sharing of data), which causes these expensive protocol operations to be invoked frequently. For example, in a sequential consistency model, invalidations are propagated and performed as soon as the corresponding write is detected, so pages may be frequently ping-ponged back and forth among processors due to either true or false sharing. This is extremely undesirable in software DSM systems due to the high cost of the communication.

Therefore, in order to implement an effective software DSM system, there are many important issues must be resolved. In the following sections, five important issues will be depicted respectively.

1.2 Memory Consistency Model

The main difficulty in building a software DSM system is solving the *memory coherence problem*, namely, the problem of how to make a newly written value to be visible to demanding processors on time. *Memory consistency model* and *cache coherence protocol* are two important aspects of solving this problem, where the memory consistency model determines *when* the modified data should be visible to other processors and cache coherence protocol determines *what* values should be visible to other processors. In this and the following section, these two issues will be depicted clearly.

The Memory consistency model is an interface between the programmer and the system. The memory consistency model of a shared memory system formally specifies how the memory system will appear to the programmer. Normally, a memory consistency model defines constraints on the order in which memory accesses can be performed in shared memory systems. It influences both programming complexity and performance. The stricter the memory consistency model, the easier for programmer to program, and the less opportunity for optimization. A strict consistency model like *sequential consistency* [73] is intuitive to the programmer. However, with the large granularity of coherence unit (a page) in shared virtual memory systems, the false sharing problem will be so serious that the performance of software DSM systems under sequential consistency is very poor. For example, the performance of the first software DSM system IVY is so poor that the main contribution of it is the original idea about software DSM, while the practical system is useless.

To improve performance, software DSM systems normally adopt relaxed memory consistency models which separate synchronization operations from ordinary *load* and *store* operations and allow ordinary operations to be executed out of order. The propagation and application of coherence operations are postponed until synchronization points. In the past decade, almost 20 different memory consistency models have been proposed for hardware-coherent systems. Among those relaxed consistency models, *release consistency (RC)* [41] which separates *acquire* from release synchronization inspire a major breakthrough in the performance of software DSM systems[19].

Although the memory consistency model specifies when coherence operations and data need to become visible, it can actually be implemented with various degrees of “laziness” in the propagation and application of both coherence and data. Greater laziness implies greater complexity and protocol and state, but fewer communication and protocol operations. For example, hardware-coherent systems that use release consistency tend to propagate coherence and apply them immediately, thus simplifying the data structures that need to memorize the state of the cache line.

In software DSMs, it is very important to reduce the number of messages exchanged, because sending a message in a software DSM is more expensive than that in a hardware DSM. Therefore, in Munin’s implementation of release consistency (so-called eager release consistency)[19], writes to shared memory are buffered until a release, at which all writes to the same destination are merged into a single message. By merging writes instead of pipelining them, messages exchanged are greatly reduced.

The TreadMarks' lazy implementation of release consistency goes further[70]. It does not propagate the modifications in a critical section at the time of release. Instead, modifications are buffered and are propagated merely to the processor that acquires the released lock until the time of acquire. In this way, lazy release consistency reduces both the number of messages and the amount of data exchanged. In LRC, before a processor can pass an acquire operation, all modifications that have been visible to the releasing processor must also be visible to the acquiring processor.

1.3 Cache Coherence Protocol

The presence of multiple cached copies of a page requires a mechanism to notify other sharers of a modified memory location. A new value is propagated by either *invalidating* or *updating* each copy, which correspond to *write-invalidate* and *write-update* coherence protocol respectively. Generally, the coherence protocol consists of the data structures and algorithms used to implement a given memory consistency model. Stenstrom provides a taxonomy of existing cache coherence protocol in [118].

A cache coherence protocol is in fact a mechanism of propagating newly written values so that all processors have a coherent view of the shared memory. It is normally designed under the requirement of a memory consistency model which specifies the coherence requirement on a cache coherence protocol, i.e., what "coherent view" of shared memory a cache coherence protocol should provide. Many new ideas accompany with relaxed memory consistency models have been proposed to efficiently propagate newly written values. Examples of these new ideas include *lazy release consistency*[70], and *delay consistency*[28], etc.

Traditionally, there are two main methods to maintain coherence: *snoopy protocol* and *directory-based protocol*. The snoopy coherence protocol requires the support of hardware so it is not widely used in software DSM systems. Up to now, almost all software DSM systems adopt the directory scheme or that similar to directory scheme. However, the scalability of the directory-based scheme is limited by the organization mode of the directory. Therefore, we propose a new scheme to maintain the coherence between multiple cache copies. More details can be found in Chapter 2 .

1.4 Application Programming Interface

As we know, whether a new software system is competitive depends greatly on the friendship of it's application programming interface(API). In parallel computing field, *shared memory* and *message passing* are two mainstream programming model which are widely used in the world. Shared memory programming model is a natural extend of traditional sequential programming model so that it is easy to be accepted by the application programmers. In this model, the application programmer does not need to consider the data partition, migration and underlying communication mechanism. On the contrary, the message passing programming model requires the application programmer to take the data partition into account, and use system supplied communication functions explicitly, which burden the application programmer much.

As a result, the shared memory model is widely adopted and advocated by the researchers, application programmers and corporations, which fuels the success of shared virtual memory system.

Generally speaking, the programming model of software DSM system is similar to traditional shared memory systems, such as *Single Program and Multiple Data* (SPMD) and Multithreading, which are familiar with many programmers. However, the API of a software DSM system is closely related to the memory consistency model implemented in the system and the implementation level of the system. For example, in a software DSM system with sequential consistency, the programmer can write the program just like a general SPMD programming mode. They can allocate a shared variable at any time and use them easily. However, if the *Entry Consistency* [11] is used, the application programmer must mark the affinity between shared data and synchronization object explicitly in the program.

The implement level of software DSM system will affect the programming interface too. If the system is implemented on the language level, such as Linda [3] and Orca [8], the programmer must learn those new characteristics in the languages related to shared memory address space. If the system is implement on the complier or operating system level, all the work are done by the compiler and all the changes are transparent to application users, as in Shasta[100]. If the software DSM system is implemented by runtime library, what the application programmer should do is adding some function calls in the source program, and linking the related library when compiling.

1.5 Memory Organization

The memory organization of a software DSM system determines the way shared virtual memory is organized on top of the isolated memory address space. It has great influence on the cache coherence protocol and consequently on system overhead.

Normally, the shared virtual memory address space of a software DSM system is organized in a COMA-like way, where the local memory of a host is treated as a large cache, and pages can be *replicated* and *migrated* on demand. Migrating the owner of data according to the sharing pattern reduces the probability of access miss for some applications characterized with single writer reference pattern, but requires a mechanism to locate the data when local access is failed. In systems that adopt the COMA shared virtual memory organization scheme, each page has an owner, and mechanisms such as probable owner or approximate copyset are employed to find where the owner of the faulting page is when the page fault occurs. Examples of this kind of system includes IVY, Munin, TreadMarks, CVM, and Quarks.

The shared virtual address space of a software DSM can also be organized like traditional CC-NUMA. In this kind of system, each shared page has a fixed home, when the page faulting occurs, the faulting processor can fetch the up-to-date page from the home directly, only one round-trip communication is needed. This requires that the coherence protocol to propagate modified information to the home node on time. The main advantage of the home-based protocol is its very low memory overhead compared to homeless counterpart. Besides the simplicity of servicing ordinary page fault, another advantage of the home-based software DSM system is that no *diffs* generation are required for writes to home pages. Evaluation results from Zhou et.al [139] demonstrate that home-based system have comparable or better performance than its COMA counterparts, though platform dependent evaluation make the evaluation results not so convincible.

Some other memory organization schemes which stem from COMA and NUMA have been implemented and studied in DSM systems. The I-ACOMA[130] and simple-COMA are examples of these varieties.

1.6 Implementation Method

1.6.1 Implementation Levels

Although the first software DSM system is implemented by modifying the OS kernel, it is not the unique method to implement a software DSM system. A software DSM system can be implemented in a variety of levels ranging from the language and complier level, to the user level library and OS level. Some recent software DSMs even summon hardware support to reduce system overhead.

The first software DSM system, Ivy[78], supports the shared virtual memory in OS level. It modifies the MMU (memory Management Unit) of OS to map between the shared virtual memory address space and local memory and to keep the shared pages coherent. A memory reference will cause a page fault when the page containing the memory location is not in the processor's current physical memory. The MMU then retrieves the fault page either from the disk or from the memory of another processor. The major drawback of this method is the requirement to be familiar with the source code of operating system, which is unavailable to most software DSM designer. Therefore, this scheme is not widely used now. Other software systems which involves OS modification include Mirage[39], Munin[19], SoftFLASH[34] etc.

Instead of modifying the OS, the *user level runtime library* method of designing software DSM relies on OS interface to detect and service page faults. Normally, `mmap()` and `mprotect()` system calls are employed to set the map and protect state of a page. References to a page with incorrect protect state will cause the delivery of a SIGSEGV signal. The SIGSEGV handler then retrieves a valid copy of the fault page from the memory of another node. (In some home-based systems, the faulting processor can fetch it from it's home directly, which requires only one round-trip communication, while in COMA-like system, the owner of the faulting page is searched by some kinds of directory mechanism such as probable owners[19], and approximate copyset[71], which requires more communications.) Since this scheme is implemented in the user level completely and no OS modification is required, it is widely used in software DSMs. Some recent software DSMs, such as TreadMarks, CVM, Quarks, and Brazos, are implemented in the user library level.

With the development of compiler technology and parallel language, some researchers extend or reimplement the programming language to support shared virtual memory address space. Linda[3], and Orca[8] are examples of software DSMs implemented at the language level. In Linda, the software DSM is organized as a "tuple space"— a common pool of user-defined tuples that are addressed by logical names. Linda provides several special language operators for dealing with such distributed data structures, like inserting, removing, and reading tuples. It avoids the consistency problem: a tuple must be removed from the tuple space before an update, and the modified version is reinserted. Whether or not this alternative is feasible is not yet clear.

Yet many systems adopt a combination of multiple implementation levels to get better performance or to ease the programming. For example, the Midway[11] software DSM system which provides a user level library also adopts compiler support to implement the protocol.

The NCP2[14] system employs hardware support to reduce system overhead. The Munin[19] system which modifies the OS also provides a user level library with the support of compiler.

The implementation level of a software DSM system determines many factors that have significant influence on the system performance. Among them the granularity is the dominant factor.

1.6.2 Granularity of the System

Granularity refers to the size of the unit of sharing: word, page, or some complex data structure. Most software DSMs utilize the hardware page size as the unit of sharing, while hardware implementation typically utilize cache line as a coherence unit.

Though the large moving unit of data may amortize some overhead of communication, the probability of data thrashing increases with the size of sharing unit as well. Thrashing occurs when two processors are actively accessing the same variables, which is named *true sharing*, or (and most frequently) when two or more unrelated variables are located in the same page, which is known as the notorious *false sharing* problem. False sharing is a particular serious problem in software DSM systems, not only because interprocessor communication is very expensive in software DSMs, but also because the large granularity of sharing in a software DSM system greatly increase the probability of false sharing. Though false sharing can be substantially alleviated by using relaxed memory consistency models and multipler-writer protocols[19], and works quite well for coarse-grained applications, for applications which exhibit fine-grain sharing these optimizations are not enough to achieve acceptable performance.

Another factor affecting the choice of page size is the need to keep directory information about pages in the system. The smaller the page size, the larger the directory needed to keep track of them.

To reduce the impact of false sharing, some systems use a hybrid approach that supports the page as the unit of data movement and a small block size as the unit of coherence. This normally requires support from complier, such as in Midway[11], or hardware, such as in Simple-COMA [97] and Tempest[92]. The advantage of fine-grained approach is simplicity, since it can use a sequential consistency model without suffering much from false sharing and communication fragmentation. The programming mode of fine-grained software DSM system is exactly the same as in more mainstream hardware DSM systems. Recent much progress on fine-grained software DSM make it a possible direction for next generation software DSM system.

1.7 Some Representative software DSMs

Table 1.1 lists 20 representative software DSM systems in the aspects of implementation method, granularity, memory consistency model, and cache coherence protocol.

1.8 Recent Progress on Software DSM and Open Questions

Since the first software DSM system was developed in 1986, there has been a lot of excellent work in developing protocols and systems for shared virtual memory in the last ten years. The development has led to the maturity of protocols and memory consistency models for

Table 1.1: Some Representative Software DSM Systems

System(year)	Developer	Implementation Method	Granularity	MC Model	CC Protocol
IVY (1986)	Yale	User-level library + OS modification, Apollo domain workstation, Aegis OS, 12Mbps token ring	Page (1KB)	SC	WI (Write Invalidate)
Munin (1991)	Rice	Runtime system library + Preprocessor + OS modification, Sun 3 workstation, System V, Ethernet	Variable size objects	Eager RC	Type-specific (WU/WI)
TreadMarks (1992)	Rice	User-level library, DECstation-5000/240, 100Mbps ATM or 10Mbps Ethernet.	Page (4KB)	LRC	WI
CVM (1997)	Maryland	User-level library, Solaris 2.5 on Sparcs, AIX4.1 on DEC Alpha	Page	LRC-MW, LRC-SW, SC	MW (Multiple Writer), WI
Midway (1991)	CMU	Runtime system + compiler, DECstation 5000, Mach OS 3.0, Ethernet LAN	Variable size objects	EC, PC, RC	WU (Write Update)
NCP2 (1997)	UFRJ,Brail	TreadMarks-like, PCI-based Hardware support, Cluster of PowerPC 604s, Myrinet Switch, LINUX	Page (4KB)	Affinity EC, RC	WU/WI
Quarks (1995)	Utah	User-level library, SunOS 4.1, HP-UX	Region, Page	RC, SC	WU/WI, MW
softFLASH (1996)	Stanford	OS modification, two level design, SGI Power Challenge, 100Mbyte/sec HIPPI,Irix 6.2.	Page(16k)	Epoch-based RC, DIRC	FLASH-like
Cashmere-2L (1997)	Rochester	Runtime library, Two level design, 8-node, DEC AlphaServer, Memory Channel	Page(8k)	Moderately HLRC	WU
Brazos (1997)	Rice	Runtime library +multithread, Windows NT. Cluster of Compaq Proliant 1500, 100Mbps Ethernet	Page	ScC	Early update, WU
Shasta (1996)	DEC WRL	Compiler support, Alpha Workstation, Memory Channel or ATM	variable granularities	SC	WI
Mermaid (1990)	Toronto	User-level library + OS modifications, SunOS	Page (1KB,8KB)	SC	WI
Dsoftware DSM6K (1993)	IBM Research	OS modification, IBM RISC System/6000, AIX v3,	Page (4KB)	SC	WI
Mirage (1989)	UCLA	OS Kernel, VAX 11/750, Ethernet	512 bytes	SC	WI
Plus (1990)	CMU	PLUS kernel, Motorola 8800, Caltech mesh routing	Page (4KB)	PC	WU
Simple-COMA (1995)	SICS(Sweden) and SUN	OS Modification and Hardware Support, Simulated 16-node DEC Alpha Multiprocessor,	Page alloc., fine-grain comm. & coh.	SC	WI
Blizzard-S (1994)	Wisconsin	User-level library + OS kernel modification	Cache line	SC	WI
Shrimp (1996)	Princeton	OS modification and hardware support, 16 Pentium PC nodes, Intel Paragon	Page	AURC, SC	WU/WI
Linda (1986)	Yale	Language, Ethernet based MicroVax and Intel iPSC hypercube	Variable size	SC	Impl. dependent
Orca (1988)	Vrije Univ., Netherlands	Language	Object size	Syn. dependent	WU

software DSM systems and led to a few key protocols and implementation mechanisms that are most promising for the future software DSMs. The time is now ripe to focus research attention primarily on bridging the gap between the performance of software DSM systems and hardware counterparter, such as Origin 2000, and it's message passing counterparter, such as MPI and PVM. Some of the key research directions in software DSM system include: (1) software DSM-oriented application research, (2) Fine grain versus Coarse grain software DSM system, (3) Hardware supported software DSM system, (4) More relaxed memory consistency model, and (5) SMP-based Hierarchical software DSM system.

1.8.1 Software DSM-oriented Application Research

Research in applications for software DSM systems plays an important role in the lifetime cycle of software DSM systems. Although some "standard" benchmarks have been used to evaluate software DSM tradeoffs, most of these applications were written for hardware coherent DSM, such as Stanford SPLASH[114] and SPLASH2[134]. Restructuring the applications or data structures appropriate for software DSM might change the performance dramatically[64]. Simultaneous research in applications and systems will help to understand whether software DSM can indeed work well for a wide range of applications, and how applications might be structured to perform portably well across different shared memory systems, and what types of further protocol optimizations and architectural support might be useful.

Studying application-protocol interactions can help both in improving protocols to match the applications and in restructuring applications to fit the protocols. In the former area, early work done in application-specific[37] protocols for fine-grained software shared memory showed fairly dramatic performance gains. In the context of software DSM, research has been done in having the compiler to detect mismatches between application sharing patterns and protocols and adjust the protocol accordingly[29].

For restructuring applications, a study was performed to examine application restructuring for software DSM systems and the performance portability of applications across software DSM and hardware-coherent systems [64, 63]. The study showed that in most cases the applications can indeed be restructured to deliver good performance on moderate-scale software DSM systems, and the improvements in performance is quite dramatic. Another important area is the development of tools for understanding performance bottlenecks in applications, because many problems arise due to contention which is very difficult to diagnose. Recently, Meira et. al. developed a performance measurement and analysis tool named *carnival* in University of Rochester[86]. This performance visualization tool can be used to understand an application performance and tune the implementation. This will be an important direction for future research in DSM.

Besides, to evaluate tradeoffs of protocols, the standard of performance evaluation now needs to be stepped up. High-level comparison among protocols are clearly quite dependent on lower-level implementation mechanisms used as well as on architectural support and system parameters, and evaluation studies should now take these into account. On the other hand, a significant limitation of most of the studies performed in this area has been with applications. "Standard" applications that are used for hardware-coherent have only recently begun to be used, such as Stanford SPLASH and SPLASH2, and most programs are too simple to be used to evaluate software shared virtual memory systems. Performance evaluation for software DSM should use a wider range of applications in different classes with regard to sharing patterns at page-grain[59].

1.8.2 Fine-grain versus Coarse-grain Software DSM Systems

The large, page-grained coherence granularity is a major performance obstacle in software DSM systems. Though techniques such as relaxed consistency models, multiple writer protocol, and lazy propagation of modifications can substantially alleviate the performance penalty of large granularity, and work quite well for coarse-grain applications, for applications which exhibit fine-grain sharing these optimizations are not enough to achieve acceptable performance. Besides, the fact that complex protocol activity is performed at synchronization points makes frequent synchronization a critical bottleneck. These facts fuel for building software DSM systems with coherence performed at fine (or variable) grain. Normally fine coherence granularity of software DSMs is achieved through providing access control and coherence instrumenting memory operations in the code rather than through the virtual memory mechanism[103].

The advantage of the fine-grained approach is simplicity, since it can use a sequential consistency model without suffering much from false sharing and communication fragmentation, and consequently can keep synchronization protocol free and much less expensive. Furthermore, the programming model under fine-grained granularity is exactly the same as that of the mainstream hardware DSM. In addition, tools for code instrumentation operate on executables rather than on source code, which represents a tremendous advantage over software DSM system. However, code instrumentation is not always portable and adds overhead on each load and store. Since communication may be more frequent, this approach relies on low-latency message passing.

While Blizzard-S is the first all-software fine-grain DSM system, the recent Shasta system breathes new life into the approach by demonstrate quite good performance on a high performance configuration (a cluster of 275MHz DEC Alpha systems interconnected with Memory Channel)[100]. This was possible due to a number of subtle optimizations in instrumentation. Among protocol-level optimizations, support for multiple coherence granularities on a per data structure basis was found to be particularly useful. It helps to optimize the prefetching effect of large granularity communication without inducing false sharing, though it does rely on some support from the programmer. Recently, Schoinas et. al.'s work on *Sirocco*[102] takes all advantages of SMP nodes to implement fine grain distributed shared memory (FGDSM). Their results show that SMP is especially beneficial for FGDSMs with high overhead coherence operations.

Fine grain and coarse grain are two very different approaches to software shared memory. A group of researchers compared the two approaches on a fairly large and varied set of applications using the Typhoon-0 platform, which allows protocols to be run in software but uses a uniform hardware access control mechanism for both approaches[139]. The results show that for almost all applications, the page-grain home-based LRC protocol performs similarly to or better than a fine-grain SC protocol (even though there is no instrumentation overhead), except in one case (the original Barnes application before restructuring) where there is an overwhelming amount of synchronization. However, this result is quite platform-related and does not use full software access control. More research is needed to understand the tradeoffs between these two approaches, and to determine whether one is clearly superior to the other given future trends in communication performance. Recently, Itzkovitz et. al proposed the concept of multiview and implemented Millipage system [61] to support fine-grain sharing in page-based system.

1.8.3 Hardware Support for Software DSM System

Although relaxed memory consistency models can help to reduce the frequency of communication, the communication overhead in software DSM is so large that the performance is unacceptable yet. To reduce system overhead further, some research groups try to build software DSM system with a little hardware support[14, 139, 56, 72, 120, 94, 55]. Their results show that the little hardware is worthwhile, which corroborates with the promise of the earlier simulation results of Cox et.al [24].

Another important direction is using hardware support for protocol processing. The network interface can be used not only to avoid interrupting the compute processor but also to perform protocol precessing, including diff creation and application and the management of timestamps and write notices. This approach is taken in [14, 139]. However, the performance benefit in this case was found to be small. In the context of SMP, a compute processor in an SMP node can be reserved for protocol processing. This approach is examined in [38], where it was found that the benefit is small due to the poor utilization of that processor, compared to a system that uses all processors both for computing and protocol processing.

Using network interface to support transparent remote data and synchronization handling is the most recent research topics. In this case the remote compute processor is not involved in handling message requests. Some previous work in this area relies on specialized network interface, such as automatic update in SHRIMP[56], remote memory write mechanism of memory channel in Cashemere-2L[120]. Recently, Bilas et.al propose using remote deposit and remote fetch functions provided by VMMC[17] to avoid asynchronous message handling overhead [16]. With this novel character of underlying communication substrate, the cache coherence protocol will become more simple.

Although a little hardware support has the potential to provide similar performance to pure hardware coherent DSM, the commodity-oriented strategy made by corporations make them don't want to support it since this will result in serious incompatible. Fortunately, remotre write/read operations have been accepted by VIA standard[22] recently.

1.8.4 More Relaxed Memory Consistency Model

Though early studies of consistency models more relaxed than release consistency did not indicate significant performance benefits, recent results are somewhat encouraging.

Entry consistency [11] proposed the idea of binding data to synchronization objects, and at a synchronization point only the data bound to that synchronization object are kept coherent. However, the programmer had to provide the binding affinity explicitly, which was a major burden. Scope consistency (ScC) [59] shows that the affinity between synchronization object and shared data can be achieved dynamically when a write memory access fault occurs inside a scope, without including the binding requirement in the definition. This reduces the programming complexity dramatically. Besides, ScC is able to reduce false sharing problem while still preserving the prefetching potential of pages and keeping the state overhead quite low.

ScC was initially evaluated through simulation with a home-based protocol using automatic update support. The first all software home-based DSM system which implement ScC is JIAJIA[51], but with different cache coherence semantics, more details can be found in following chapters. The model has also been implemented for a homeless multiple writer scheme similar to that used in TreadMarks, in the Brazos[116]. The reported results corroborate the promise of the earlier simulation results.

Recently, Amorim et. al. propose *affinity entry consistency (AEC)* [104], which is claimed as a tradeoff between EC and ScC. AEC hides the overhead of generating diffs behind three types of synchronization delays, which overlap computation and communication significantly. AEC has the performance potential of EC, while does not burden the application to define the relationship between synchronization variable and shared data. Sandhu et. al. introduce *multiple-writer entry consistency (MEC)* which combines the efficient communication mechanisms of LRC with the flexible data management of the Shared Regions[65] and EC model[95]. This is achieved in MEC by decoupling synchronization from coherence (in contrast to the tight coupling of synchronization and coherence in EC) while retaining the familiar synchronization structure in RC programs.

While the programming burden still keeps these models (or protocols) outside the mainstream, EC may fit well with object-oriented languages like Java which inherently provide for object-to-lock association, and ScC's small programming burden may be alleviated by tools to modify the properly labeled program for RC to satisfies ScC.

1.8.5 SMP-based Hierarchical Software DSM System

Another recent development in software DSM is motivated by the increase popularity of small-scale hardware-coherent multiprocessors (SMPs), and the development of clusters that use these multiprocessors as their nodes. There are two strong arguments for using a software DSM layer across the nodes in such a cluster. First, it provides a uniform (coherent shared memory) programming interface rather than a hybrid message-passing/shared memory interface. Second, because the local coherence (within the multiprocessor) is performed by hardware, the overall performance of the system is expected to be better than the performance of the uniprocessor based software DSM systems.

Building an efficient DSM on a multiprocessor cluster is not as simple as porting uniprocessor protocol. Issues like *lazy propagation*, *multiple writers* and *protocol handling* must be reinvestigated[54].

- With respect to lazy protocols, the question is how quickly the coherence information from outside propagate within the multiprocessors. For example, when one processor has to invalidate a page, should the page also be invalidated for other processors in that node[120]? There is a contradiction between the need for some software involvement even within a node to provide some extent laziness, and the need to reduce software involvement to exploit hardware sharing within a node.

- For multiple writers protocol, the question is how to prevent other processors in the same node from writing into the page while one processor fetches the page as a result of an invalidation. One of the first implementation indicates that shooting down the TLBs inside the multiprocessor can be particularly expensive if it occurs frequently[34]. An alternative called *incoming diffs* was suggested in [120], but found do not help much. Unfortunately, they found contrary conclusion about TLB shoot down within one multiprocessor. This question is not clear yet.

- For protocol handling, there are questions about whether to use a dedicated protocol processor within the node, and if not, how to distribute protocol processing in response to incoming messages across processors. Systems have also been developed to implement software DSM across hardware DSM system rather than bus-based multiprocessor[136].

The performance gains in switching from uniprocessor to multiprocessor clusters are not yet clearly understood, and may be affected by the performance of the communication support within and across nodes, in addition to the degree of laziness implemented in the protocol. In the four published papers described real software DSM systems built across SMP clusters [34, 98, 120, 102], the comparison is made between the real DSM across multiprocessor clusters and a DSM for uniprocessor nodes emulated on the same hardware. At the same time, simulation studies indicate that while there is usually a benefit to using multiprocessor rather than uniprocessor nodes, the benefit is usually small when node size is small relative to overall system size. In fact, there is no clear answer to this question. Recently, Samanta et. al. study the performance of software DSM on clusters of SMP, and find a most promising tradeoff scheme by quantitative analysis, more details can be found in [94].

Up to now, almost all the proposed software DSM systems are based on Unix OS system. Accompany with the popularization of Windows NT, Windows NT based software DSM system[116] and heterogeneous software DSM system based on Java [137] may be a research direction for the next generation software DSM system.

1.9 Summary of Dissertation Contributions

This dissertation has eight main contributions in the area of software distributed shared memory which were published in the *Journal of Computer Science & Technology*[49][105][110], *Journal of Scheduling*[111], *Proceedings of the 7th High Performance Computing and Networking Europe (HPCN'99)* [51][107], *Proceedings of the 2nd Merge Symposium of Parallel Processing and Parallel and Distributed Processing (IPPS'99)*[52], *Proceedings of the ACM Workshop on Scheduling Algorithms on Parallel and Distributed Computing*[112], *Proceedings of the 1st ACM Workshop on Software DSM* [50], *Proceedings of 8th IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*[106], *Proceedings of the ACM/IEEE 6th International Conference on High Performance Computing (HiPC'99*[109], and *Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'99)*[108].

Several techniques have been proposed to improving the performance from cache coherence protocol, memory organization scheme, system overhead, load balancing, and communication optimization etc. five aspects. The main contributions of this dissertation are listed as follows.

1. **Lock-based Cache Coherence Protocol** By analyzing the disadvantages of snoopy and directory based cache coherence protocols and several different release consistency protocols, we propose a novel lock-based coherence protocol for scope consistency. The unique characteristic of this new protocol is the using of "home" concept not only for data information, but also for coherence information, such as *write notices*. Generally, home-based software DSM systems propagate data information and apply them at homes eagerly, but this information is fetched by others lazily. On the other hand, these systems propagate coherence information and apply them either eagerly or lazily[57]. In our lock-based cache coherence protocol, coherence information is processed in a similar way to data information, and each coherence information has a static home according to the corresponding synchronization object (e.g., a lock or a barrier manager). So, coherence information is propagated to the corresponding home at release time, and is lazily fetched by the next acquirer of the same synchronization object. Compared with directory-based

protocols, all coherence related actions in the protocol are applied at synchronization points. In this way, the lock-based protocol has least coherence related overheads for ordinary read or write misses. Moreover, the lock-based protocol is free from the overhead of maintaining the directory.

2. **JIAJIA Software DSM System** Based on this new protocol, we design a software DSM system named JIAJIA, which is simple but very efficient. Unlike other software DSMs that adopt the COMA-like memory architecture, JIAJIA organizes the shared memory in a NUMA-like way. Consequently, references to local part of shared memory always hit locally, while references to remote shared pages cause these pages to be fetched from its home and cached locally. When cached, the remote page is kept at the same user space address as that in its home node. In this way, shared address of a page is identical in all processors, no address translation is required on a remote access. With home-based memory organization scheme, JIAJIA is the first software DSM system that has the capability to combine physical memories of multiple computers to form a larger shared space.

3. **Reducing System Overhead** In order to find the main performance obstacles which prevent software DSMs from becoming the acceptable parallel programming environment, we perform a detailed analysis about the system overhead. We observe that remote data penalty and synchronization overheads are two culprits of large system overhead, therefore, we propose several techniques to reduce these system overheads efficiently, such as read notice, hierarchical barrier implementation, cache only write detection (CO-WD), etc.. The evaluation results show that the performance is improved greatly with our optimization schemes.

4. **Home Migration Scheme** Since the data distribution plays an important role in home-based software system, we propose and design a novel home migration mechanism to reduce the remote data communication. The basic idea behind our home migration is migrating the home of a page to a new host on a barrier only if the new home host is the single writer of the page during the last barrier interval. In cooperation with JIAJIA's novel memory organization scheme, no page copy is required at home migration phase. Evaluation shows home migration can reduce diffs dramatically and improve performance significantly for some applications.

5. **Affinity-based Self Scheduling** As the prevalence of metacomputing environment, loop scheduling and balancing are critical issues in order to achieve high performance. We propose a new affinity-based self scheduling (ABS) scheme for load balancing in home-based software DSM systems. ABS takes the static data allocation into account during loop scheduling. All processes share a global queue but with different segments to keep processor affinity. The main difference between ABS algorithm and traditional central queue based scheduling schemes lies in the organization and management of the central queue. In our scheme, the queue is partitioned among these processors, and each processor first accesses it's own part. During the local scheduling phase, each processor obtains task from it's initially allocated segment. When load imbalance occurs, the idle processor obtains the task from the segment which belongs to the most heavily loaded processor. Compared with Markatos et.al 's affinity scheduling[84], the synchronization overhead is reduced significantly.

6. **Dynamic Task Migration** For iterative scientific applications, we introduce a novel dynamic task migration scheme, which can balance the load efficiently. Firstly, a new definition about task is given. In distributed computing environment, the data distribution plays great important role in final performance. Therefore, a task is the sum of *computation subtask* and *data subtask* Based on this observation, we propose a task migration scheme which executes computation migration and data migration sequentially. Evaluation results show task migration performs better than traditional computation migration greatly.

7. **Communication Substrate Specific to Home-based Software DSM** As the introduction of Virtual Interface Architecture (VIA), the technology of User Level Networking (ULN) should become more and more mature. So it is time to study application-level issues of ULN, such as integrating the high level application and protocol buffer management, optimizing the communication path. The exposing of all underlying memory management to high level software give us the chance to design a highly efficient, application-specific communication substrate. Meanwhile, it has been widely accepted that communication overhead is the main obstacle that prevents the software DSMs becoming the desirable vehicle for parallel computing. Recently, based on the observation of communication requirements of software DSMs, we propose and design a Myrinet-based communication library named JMCL specific to J IAJIA. JMCL provides reliable, protected, in order message delivery for high level JIAJIA system. With JMCL, the communication module of JIAJIA system become more simple and efficient than that in traditional UDP/IP protocol. Our design of JMCL is closely related to the VIA standard.

8. **Confidence Interval Based Summarizing Method and Real World Application** Traditionally, to demonstrate the advantages of the new idea, several favorite small benchmarks are used in the evaluation, such as those taken from SPLASH[114] and SPLASH2[134], NAS Parallel Benchmark[7]. Therefore, the confidence of their results is suspicious. We propose a confidence interval based evaluation method to analyze the experimental results. Furthermore, I have ported a real world application which compute resonant frequency of electromagnetic simulation to JIAJIA system successfully and have achieved desirable results. This application is a productive program of the Institute of Electronics of Chinese Academy of Sciences.

1.10 Organization of the Dissertation

My dissertation begins with a brief overview of the software DSM system research, according to five important issues to the design of software DSM system.

In Chapter 2, I introduce the lock-based cache coherence protocol and compare it with traditional snoopy and directory-based protocols. The correctness is also proofed in this Chapter.

In Chapter 3, the design and implementation of JIAJIA are described. Performance evaluation and analysis of JIAJIA, including comparison to CVM, is also presented.

In Chapter 4, I present the detail analysis of system overhead firstly, then several techniques are proposed to reduce the system overhead.

In Chapter 5, I propose and design a novel affinity-based self scheduling algorithm. The evaluation focuses on two cases: dedicated environment and metacomputing environment.

In Chapter 6, with respect to iterative scientific applications, I give a new definition of *task* in distributed memory system firstly, then I propose a dynamic task migration scheme.

In Chapter 7, I argue that specific communication substrate for software DSMs should be an important research direction for both software DSMs and user level networking area. According to the specific communication requirements of software DSMs, I propose and implement a communication substrate specific to JIAJIA.

My dissertation ends with conclusions and future directions.

Chapter 2

Lock-based Cache Coherence Protocol

2.1 Cache Coherence Protocol

Caching techniques play a determining role in a shared-memory system. Caching of data in shared-memory systems can not only hide the large latency caused by the spatial distribution of processors and memory modules, but also reduces the contention in accessing shared resources.

However, caching in a shared-memory system will cause the serious problem of cache coherence: namely, the problem of how to maintain data coherence among the various copies of data which can reside in multiple caches and main memory. Many cache coherence protocols have been proposed to solve this problem. A cache coherence protocol is in fact a mechanism of propagating newly written values so that all processors have a coherent view of the shared memory.

Many methods have been proposed to efficiently propagate newly written values to other processors. They constitute various cache coherence protocols. These methods characterize themselves in the following aspects:

- How to propagate newly written values: *write-invalidate* or *write-update*.
- Who can write a new value to a block: *multiple writer protocol* or *single writer protocol*.
- When to propagate newly written value: *early propagation* or *delayed propagation*.
- To whom the newly written value is propagated: *snoopy protocol* or *directory protocol*.

In the above aspects of design a coherence protocol, compromises should be made between the complexity and performance of the protocol.

In this section, we briefly review tradeoffs between write-update and write-invalidate, single writer and multiple writer, early propagation and delayed propagation. The snoopy protocol and directory-based protocol are examined in a little detail in the following sections.

2.1.1 Write-Invalidate vs. Write-Update

Based on the write propagation policy adopted, cache coherence protocols can be divided into two categories: write-invalidate and write-update. In write-invalidate protocols, if a block is modified by a processor, all copies of this block in other processors are invalidated. When one of other processors wants to reference this block afterwards, an up-to-date copy of the block is reloaded into its cache. In write-update protocols, if a block is modified by a processor, the newly written value of this block is propagated directly to all processors that have a copy of this block.

Compared to that of write-update protocol, the performance of write-invalidate protocol suffers more from the effects of false sharing, i.e., any time a data item in a block is modified by one processor, all copies of the block are invalidated, whether or not the modified data item is shared by other processors. On the other hand, the performance of write-update protocols suffers from the fact that updates to a particular data item are propagated to all its replicas, including those that are no longer being used. Hence, write-invalidate protocols are good for applications characterized with a sequential pattern of sharing, while write-update protocols work best in applications with tight sharing. As a result, invalidate and update policies can be combined to meet the sharing pattern of different applications. The protocol can start with write-update and turn into write-invalidate when a long sequence of local writes is encountered or predicted, or start with write-invalidate and then turn into write-update when a memory line is frequently invalidated among different processors.

2.1.2 Multiple Writer Protocol

The multiple writer protocol is proposed specifically to mitigate the effect of false sharing[20].

False sharing is the sharing of cache block without true sharing of data. It occurs when two or more processors concurrently access different shared data items that stay in the same block, and at least one operation is write.

Conventionally, false sharing results in non-optimum protocol. In the case of a write-invalidate protocol, modification of a word in a block causes invalidation of all cached copies of the block, whether or not the modified word is shared by other processors. Thus, more invalidations are sent than needed by the parallel application and its data sharing requirements. This false sharing can cause a page to “ping-pong” back and forth between different processors. Similarly, in a write-update protocol, modification of a word in a block causes this modified word to be sent to all processors which has a copy of the block, even if the processor will never use the modified word.

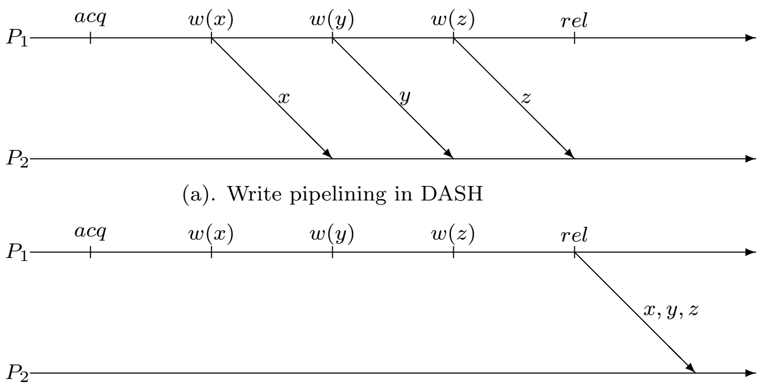
False sharing is a particular serious problem in software DSM system based on the network of workstations, not only because interprocessor communication is very expensive in network of workstations, but also because the large granularity of sharing in a software DSM system greatly increases the probability of false sharing.

Multiple writer protocol alleviates the effect of false sharing by allowing concurrent writers of the same block. It is the responsibility of the programmer to guarantee that different processors read from and write to independent portions of the block. The concurrent writers of the same block buffer modifications until synchronization requires their propagation.

The multiple writer protocol has been implemented in some recent software DSM systems such as Munin, TreadMarks, CVM, and Quarks.

2.1.3 Delayed Propagation Protocol

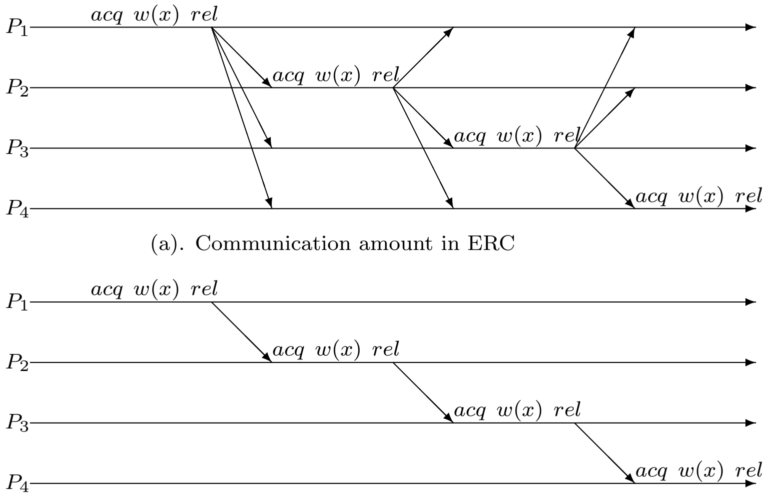
In software DSMs, it is important to reduce the number of messages exchanged, because sending a message in a software DSM is more expensive than in a hardware DSM. Therefore, in Munin’s implementation of release consistency (so-called eager release consistency, ERC), writes to shared memory are buffered until a release, at which point all writes to the same destination are merged into a single message[19]. By merging writes instead of pipelining them, messages



(a). Write pipelining in DASH

(b). Write merging in Munin

Figure 2.1: Write merging in eager release consistency.



(a). Communication amount in ERC

(b). Communication amount in LRC

Figure 2.2: Comparison of communication amount in eager and lazy RC.

exchanged are greatly reduced. Figure 2.1 shows the merging of multiple writes in eager release consistency.

The TreadMarks' lazy implementation of release consistency goes even far [70]. It does not propagate the modification of writes in a critical section at the time of a release. Instead, modifications are buffered and are propagated merely to the processor that acquires the released lock until the time of acquire. In this way, lazy release consistency reduces both the number of messages and the amount of data exchanged. Suppose variables x, y, and z are shared by processors P_1, P_2, P_3, and P_4, Figure 2.2 shows messages exchanged in eager and lazy implementation of release consistency.

Delay consistency (DC) [28] is a tradeoff between ERC and LRC. In this case, incoming invalidations are not applied right away but are queued at the destinations until they perform their next acquire operation (whether causally related to that release or not). Acknowledgements are sent back as soon as invalidations are received so that the releaser can proceed. By queuing incoming invalidations, DC protocol avoids the false sharing misses which might otherwise have occurred due to accesses between the time the invalidations are received and the next acquire operation is performed. Currently, Cashmere system adopts DC[72].

2.2 Snoopy Protocols

According to the most widely accepted classification which is based on how state information about data blocks is organized (centralized in memory or distributed in caches) and what is responsible for preserving coherence in the system (global or local controller), cache coherence protocols can be divided into two large groups: snoopy and directory-based protocols.

Snoopy protocols do not use a centralized controller or global state information. Instead, all actions for a currently shared block are broadcasted to all caches in the system. Data coherence are maintained by requiring each cache to snoopy the broadcast and take appropriate actions for addresses that are present in that cache. With the facility of the broadcast, only the actions of local cache controller and distributed local state information maintain coherence. Snoopy protocols are ideally suited for multiprocessors that use a shared bus as a global interconnection since the shared bus provides an inexpensive and speedy broadcast mechanism. However, the bus will become a bottleneck when too many processors are connected to it.

Based on the write policy adopted, snoopy protocols can be further divided into two categories: write-invalidate and write-update. Write-invalidate snoopy protocols allow multiple readers but only one writer at a time. Each time a processor wants to update a block, it broadcasts this action to all caches. All caches snoopy the broadcast and those that have a shared copy of the block invalidate the block. Once the block is made exclusive, inexpensive local writes can proceed until some other processor requires the same block. Examples of this type of protocols are Write-Once protocol proposed by Goodman[44], which is the first write-back snoopy protocol, the Berkeley protocol which is implemented in the SPUR multiprocessor and which takes advantage of the ownership principle, and the Illinois protocol which can recognize the sharing status on a block read[89].

Write-update snoopy protocols, on the other hand, allows multiple writers to the same block to exist. Each time a processor wants to update a block, it broadcasts this new value to all caches. All caches snoopy the broadcast and those that containing that block update it. Examples of write-update snoopy protocols are the Firefly protocol which is implemented in the DEC Firefly multiprocessor workstation[126], and the Dragon protocol which is developed for the Xerox Dragon multiprocessor workstation[6].

Analysis and performance studies show that write-invalidate protocols are good for applications characterized with a sequential pattern of sharing, while write-update protocols work best in applications with tight sharing. As a result, some adaptive protocols[66] which combine invalidate and update policies have been proposed. They start with write-updates, but when a long sequence of local writes is encountered or predicted, the protocol turns into write-invalidate.

Being simple and effective, the bus-based snoopy protocol provides poor scalability because i) the bus is a mutual exclusive resource and can be used by only one processor at a time, ii) all processors must participate in an arbitration phase before accessing the bus, iii) the bus clock cycle must be long enough so that signals can propagate throughout the entire extension of the bus, iv) as the number of processors increases, the propagation speed is reduced.

2.3 Directory-Based Protocols

As multiprocessors are scaled beyond single bus system, directory-based protocol offer an attractive alternative by providing cache coherence without requiring broadcast and consuming only a small fraction of the system bandwidth. Directory-based protocols maintain a directory in the main memory to keep track of all processors caching a memory line. For each correctly cached memory line, the directory should identify the set of caches that contain this line (called the sharing set), the state of a memory line (e.g., Dirty or Clean), and the state of the cache blocks (e.g., Owned, Shared, or Invalid). These information are then used to selectively send invalidate/update signals when a memory line is written. Directory-based protocols are primarily suitable for multiprocessors with general interconnection networks.

The directory of a directory-based cache coherence protocol can be organized as either fixed or dynamic. Fixed directories include the full bit vector directory and limit pointer directory, while linked list directory and probowner directory are examples of dynamic directories.

2.3.1 Full Bit Vector Directory

The full map directory-based protocol maintains a directory entry with one present bit per processor and one dirty bit for each memory line. Each present bit in the directory entry represents whether or not the memory line is presented in the corresponding processor's cache. If the dirty bit is set, then only one present bit can be set and the corresponding processor has permission to write into the block. The Stanford DASH prototype[76] and its successor, Origin 2000 of SGI[75], adopt the full bit vector directory scheme.

2.3.2 Limited Pointer Directory

The full bit vector scheme is impracticable for machines with a very large number of processors, because the directory memory needed by this scheme grows as the square of the number of processors. With the knowledge that for most kinds of data object the corresponding memory locations are cached by only a small number of processors at any given time, we can restrict the number of simultaneously cached copies of any particular memory line and associate with each memory line a limited number of pointers, each pointing to a processor caching that memory line. This limits the growth of the directory to a constant factor. The limited pointer protocols are similar to the full bit vector protocol when every memory line is cached by less than n caches, where n is the number of pointers in each directory entry. But when a memory line is cached by more than n caches, there must exist some mechanism to handle this pointer overflow situation. Several alternatives, such as limited pointers with broadcast (Dir_iB)[2], limited pointers with pointer replacement (Dir_iNB)[2], limited pointers with coarse vector overflow (Dir_iCV_r)[40], limited pointers with software support (Dir_iSW) [21], and limited pointers with dynamic pointer overflow (Dir_iDP) [113], have been proposed to deal with memory lines that are cached by more processors than the number of pointers in the directory entry. Depending on the alternative ways of handling pointer overflow, the coherence and data traffic generated may vary greatly. The Alewife machine built at MIT adopts software support to deal with pointer overflow situation[1].

2.3.3 Linked List Directory

Directories of both the full bit vector scheme and the limited pointer scheme are fixed. Another way of directory organization is to dynamically link cache blocks caching the same data. These dynamic link schemes address the directory scalability problem by dynamically distributing the information about which caches have copies of memory line over all cache blocks caching the data. There may be many ways to link cache blocks caching the same data, linked list directory and probowner directory are typical examples of them.

In the linked list scheme, each line in the memory and the cache has one (for singly linked list) or two (for doubly linked list) cache pointers associated with it. Caches that share a memory block are linked together through a singly or doubly linked list whose head is pointed by a cache pointer kept at the corresponding memory line. In the linked list schemes, the directory services a read or write miss request by changing the pointer in the directory entry to point to the requesting cache. Variations of the linked list protocols have been proposed, including the SDD (Singly-linked Distributed Directory protocol)[127] and the IEEE SCI standard protocol[43].

2.3.4 Probowner Directory

In the above memory directory schemes, each directory has a fixed home. Anytime when a processor accesses a directory entry whose home is not local, it has to conduct expensive remote communication.

To avoid expensive remote directory entry access, the directory can be organized in the probowner mode[79]. In the probowner organization of the directory, each memory page has an owner which contains the up-to-date copy of the page. For all pages, the processor maintains a probowner field which probably points to the owner of the line. The field is called "probowner" because its content is the "best guess" of the owner of the memory line, and may not be precise. The pointer may point directly to the owner of the line, or may start a sequence of processors through which the true owner can be found. The probowner field is updated whenever a processor receives an invalidation request, relinquishes ownership of the cached block, or forwards a page fault request.

The probowner organization of directory reduces some look-up and update of remote directory entry when maintains coherence and hence is especially suitable for shared virtual memory systems in which remote communications are expensive. Software DSM systems such as Ivy and Munin adopt the probowner directory.

2.4 Lock-Based Cache Coherence Protocol

In directory-based protocols, additional memory is required to store the directory and the amount of directory memory increases much faster than the number of nodes. This limits the scalability of the protocol. Besides, the complexity of keeping the directory entries consistent with the cached copies further limits the scalability. In this section, we propose a cached coherence protocol which does not rely on the directory information.

2.4.1 Design Consideration

Based on the observation that the benefit of complex design of a software DSM system may be offset by the system overhead caused by the complexity, the cache coherence protocol of our new software DSM system JIAJIA is designed as simple as possible. Instead of supporting multiple consistency models and adaptive write propagation strategies as some previous DSM systems did, we want to use a fixed memory consistency model (scope consistency) and fixed write propagation (write invalidate) strategy. Multiple writer technique is employed to alleviate false sharing.

In our protocol, a newly written value (data information) is propagated later than the eager release protocol, but earlier than the lazy release protocol. Eager release protocol propagates a newly written value to its replicas on a release, while the lazy release protocol buffers the newly written value and propagates merely to the next acquiring processor until the time of acquire. With the new protocol, the newly written value is propagated to its home on a release, and to the next acquiring processor on the time of page fault.

It has been mentioned that a cache coherence protocol is essentially a mechanism to propagate newly written values. In traditional directory-based cache coherence protocols, the writer is responsible for sending the modification to processors which have a copy of the modified block. To avoid broadcast, a directory item is maintained for each memory line to keep track of sharers of that line. This directory information is then used to selectively propagate (invalidate or update) modifications when a memory line is written.

Another way of propagating a newly written value is to require the following user of the associated block to actively fetch the modification on accessing the modified block. This method eliminates propagation of superfluous updates or invalidates but requires a method to tell the following consumer of the newly written value when and where to fetch the modification. The synchronization mechanism of relaxed memory consistency models provides such a method. In relaxed memory models such as release consistency (RC), the programmer bears the responsibility of separating any conflicting accesses of different processors with a release-acquire pair. A processor can fetch modifications made by the releasing processor on a following acquire. The LRC of TreadMarks adopts this idea.

In TreadMarks and other recent software DSM systems which adopt the COMA-like memory organization scheme, a mechanism is required to locate the last modifier(s) of the associated page on a page fault. Complex data structures such as probowner, approximate copyset, write notice records, and interval records are employed to address this problem. In JIAJIA which maintains a fixed home for each shared page, a faulting processor can get the fault page directly from its home. Coherence information made by a releasing processor is processed with the similar way as data information. Namely, this information can be propagated to the acquiring processor through the lock manager. This is the central idea of JIAJIA's lock-based cache coherence protocol. Compared to the TreadMark and HLRC's protocols, the lock-based protocol is simpler and consequently more efficient and scalable.

2.4.2 Supporting Scope Consistency

A cache coherence protocol is normally designed under the requirement of a memory consistency model. A memory consistency model specifies the coherence requirement on a cache coherence protocol, i.e., what "coherent view" of shared memory a cache coherence protocol should provide. For example, in RC, a processor may not observe the new value of a location until the processor who writes that new value reaches a release operation, while in sequential consistency(SC), a newly written value is propagated to all other processors as soon as possible. Hence, the "coherent view" of shared memory required by SC is different from that required by RC. As a result, a cache coherence protocol for SC is totally different from that for RC.

Lock-based cache coherence protocol implements the ScC which is even lazier than LRC. In LRC, when processor P acquires lock l from processor Q, it has to pick up all modifications that have been visible to Q. In ScC, only those made in critical sections protected by l are delivered to P. The memory event ordering constraints of ScC is as follows:

- Before a lock acquire is allowed to perform at processor P, all memory operations performed with respect to that lock must also be performed with respect to P.[1]
- A memory access is allowed to perform only after all previous acquire events have been performed.

A memory operation is said to be performed with respect to a lock when the lock is released.

The lock-based cache coherence protocol implements ScC as follows.

- To ensure that all memory operations performed with respect to a lock is also performed with respect to the processor acquiring the lock, the lock releaser sends identifications of all pages modified in the associated critical section to the lock. On an acquire, the acquiring processor knows from the lock which pages have been modified by other processors and invalidates these pages in its cache.
- To ensure that a memory access is allowed to perform only after all previous acquire events have been performed, the acquiring processor is stalled until all previous acquire operations have been performed.

Adopting the ScC greatly simplifies the lock-based cache coherence protocol of JIAJIA. In TreadMarks which implements LRC, complex data structures such as *intervals* and *vector timestamp* are employed to record the "happen-before-1" relation of memory accesses among different processors. In the context of lock-based coherence protocol, ScC does not require to implement the complete "happen-before-1" relation. Rather, it only requires the "last interval" of the releaser to be visible to the acquiring processor on an acquire.

2.4.3 The Basic Protocol

With the above concepts, the lock-based cache coherence protocol for ScC can be described as follows.

In the basic protocol, each page has a fixed home and can be cached by a non-home processor in one of three states: Invalid (INV), Read-Only (RO), and Read-Write (RW). Since the multiple-writer technique is assumed, a page may be cached by different processors in different states concurrently.

As a special kind of shared object, each lock also has a home node. Besides ensuring exclusive access to certain shared pages, the lock also records the modified page identifications in corresponding critical sections.

On a release, the releaser performs a comparison of all cached pages written in this critical section with their twins to get *diffs* related to this critical section. These *diffs* are then sent to homes of associated memory lines. At the same time, a release message is sent to the home

[1] The original condition in [59] is: "before a lock acquire is allowed to perform at processor P, all *writes* performed with respect to that lock must also be performed with respect to P." We believe it is also necessary for associated *reads* to be finished before an acquire is allowed to perform.

of the associated lock to release the lock. Besides, the releaser piggybacks write-notices of the critical section on the release message notifying the modifications in the critical section.

On an acquire, the acquiring processor locates the home of the lock and sends an acquiring request to it. The requesting processor is then stalled until it is granted the lock. The home processor of any lock queues up all requests for the lock. On receiving an acquiring request, the acquiring request is added to the tail of the requesting queue. When the lock is released, the node in the front of the queue is granted the lock. Again, write-notices associated with this lock are piggybacked on the granting message. After the acquiring processor receives this granting message, it invalidates all cached pages that are notified as obsolete by the associated write-notices. *Diffs* of pages in RW state are written back to the home before they are invalidated.

A barrier can be viewed as a combination of a lock and an unlock. Arriving at a barrier ends an old "critical section", while leaving a barrier begins a new one. In this way, two barriers enclose a critical section. On a barrier, all write notices of all locks are cleared.

On a read miss, the associated memory page is fetched from the home in RO state in the local memory.

On a write miss, if the written page is not presented or is in INV state in the local memory, it is fetched from the home in RW state. If the written page is in RO state in the local memory, the state is turned into RW. A write-notice is recorded about this page and a twin of this page is created before written.

Write-notices associated with a lock are cleared when there are no space left for write notices in the lock. On a clear, these write-notices are sent to all processors to invalidate the associated page.

It it to be noted here that in order to ensure that the diffs computed at release arrive at home before the lock arrive at the next acquirer we must impose message ordering by using acknowledgements. In Cashmere, it relies on the global ordering guaranteed by the switch, while in Shrimp there is no such guarantee and no use acknowledgement, therefore they need to use timestamps to control the order.

2.4.4 Correctness of The Protocol

Basic Notations

Table 2.1 lists some basic notations used in this Chapter.

Correctness Proof

The conditions of scope consistency described in Section 2.4.2 can be formulated as follows:

$$\left.\begin{array}{rl} u \overset{PO}{\to} acq \overset{PO}{\to} v & \Rightarrow u^c < v^j \\ acq(l) \to u \overset{PO}{\to} rel(l) \overset{SYN}{\to} \cdots \overset{SYN}{\to} acq(l) \overset{PO}{\to} v & \Rightarrow u^c < v^j \end{array}\right\} \tag{2.1}$$

where u and v may be write or read operation, $j = 1, 2, \cdots, N$, c is the number of processor that issues v, "$\cdots$" represents a series of $acq(l) \overset{PO}{\to} rel(l)$ pairs. Note that by the requirement of scope consistency, all synchronization operations access the same lock.

Here, we have following theorem.

Theorem 2.1 *(2.1) is a correct implementation of scope consistency.*

Table 2.1: Some Notations

Notation	Definition
$w_i(x)$	Processor P_i writes location x
$r_i(x)$	Processor P_i reads location x
$u_i(x), v_i(x)$	Processor P_i accesses(either writes or reads) location x
$acq_i(l)$	Processor P_i acquires lock l
$rel_i(l)$	Processor P_i releases lock l
u^i	Memory access u is "performed with respect to"[41] processor P_i
$a \overset{R}{\rightarrow} b$	$(a, b) \in R$, where R is an order relation
$A \Rightarrow B$	If A then B
$a < b$	Event a physically happens after event b
$u \overset{PO}{\rightarrow}\mathrm{v}$	Event u happens before event v in program order
$u \overset{SYN}{\rightarrow}\mathrm{v}$	Event u happens before event v in synchronization order

The proof of this theorem can be found in [47]. Therefore, it can be seen that an implementation of scope consistency is correct if the shared memory access event ordering of the protocol meets the requirement of (2.1).

It is easy to see that if u and v are operations of the same process such that $u \overset{PO}{\rightarrow} acq \overset{PO}{\rightarrow} v$, then $u^c < v^j$ is certainly met, where $j = 1, 2, \cdots, N$, and c is the identification number of the processor that issues v.

If $acq(l) \overset{PO}{\rightarrow} u \overset{PO}{\rightarrow} rel(l) \overset{SYN}{\rightarrow} \cdots \overset{SYN}{\rightarrow} acq(l) \overset{PO}{\rightarrow} v$, there may be two cases:

- If u is a read, since $u \overset{PO}{\rightarrow} rel(l)$, u is issued before $rel(l)$ and consequently before the following $acq(l)$ which gets the lock released by $rel(l)$. Hence, any write to the location read by u after the $acq(l)$ has no influence on the value returned by u. By the definition of "performed with respect to", $u^c < v^j$ where $j = 1, 2, \cdots, N$, and c is the identification number of the processor that issues v.

- If u is a write, since the processor which performs $rel(l)$ sends the write notice about u to the lock and modification made by u to the home of the modified page, and the processor which performs the following $acq(l)$ invalidates its local copy of the page written by u according to the write notice it gets from the lock, the value written by u is visible to the acquiring processor (or, u is performed with respect to the acquiring processor) before the processor issues any following operations, i.e., $u^c < v^j$ where $j = 1, 2, \cdots, N$, and c is the identification number of the processor that issues v.

From the above analysis, we can see that the lock-based cache coherence protocol meets the requirement of (2.1).

2.4.5 Advantages and Disadvantages

Figure 2.3 shows the state transition graph of the lock-based cache coherence protocol. The transition from RW to RO on a release or an acquire in Figure 2.3 is for recognizing write notices in an interval. If there are other write trapping methods (e.g. hardware support or software dirty bit[138]) to recognize whether an RW page is updated in an interval, then the state of an RW page is unnecessary to transit to RO on a release or an acquire. The CO-WD technique

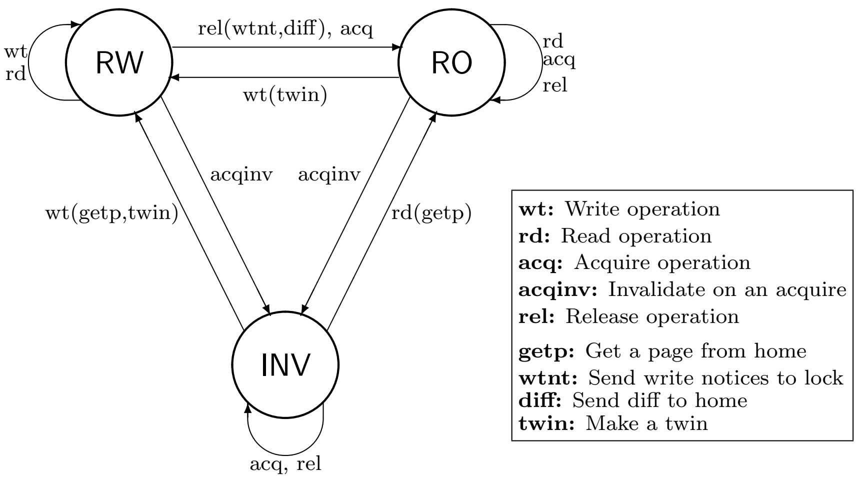


Figure 2.3: State transition digram of the lock-based cache protocol.

proposed in Section 4.4.2 eliminates this extra overhead efficiently for some applications with specific data distribution.

We can see from Figure 2.3 that, compared with directory-based protocols, all coherence related actions in our protocol are taken in synchronization points. In this way, the lock-based protocol has least coherence related overheads for ordinary read or write operations. Table 2.2 shows the number of messages sent on an ordinary access miss, a lock, an unlock, or a barrier in the lazy release protocol and the lock-based protocol. The zero message count in access miss of the lock-based protocol represents the write miss on an RO page situation. It can be seen from Table 2.2 that, compared to the lazy release protocol, our protocol has less message cost on both ordinary accesses or lock, but requires to write diffs back to home of associated pages on a release or a barrier. Besides, the lock-based protocol is free from the overhead of maintaining the directory on ordinary read or write operations.

Table 2.2: Message Costs of Shared Memory Operations

Protocols	**Access Miss**	**Lock**	**Unlock**	**Barrier**
LRC	2m	3	0	2(n-1)
lock-based	0 or 2	2	f+1	2(n-1)+F

m = # concurrent last modifiers for the missing page
f = # messages to send diffs
n = # processors in the system
F = $\sum_{i=1}^{n}$(# messages to send diffs by processor i)

Though a memory consistency model specifies when coherence operations and data must become visible, it can actually be implemented with various degrees of laziness in the propagation and application of both coherence information and data information. From viewpoint of laziness, Table 2.3 compares four representative cache coherence protocols. *Lock* represents the lock-based protocol.

Table 2.3: Comparison of Different Coherence Protocols

Protocol	Coherence Information		Data Information	
	Propagation	Application	Propagation	Application
ERC	Eager	Eager	Eager	Eager
DC	Eager	Lazy	Eager	Lazy
Lock	Eager(home)	Intermediate	Eager(home)	Intermediate
LRC	Lazy	Lazy	Lazy	Lazy

It can be found that the unique characteristic of this new protocol is the use of "home" concept not only for data information, but also for coherence information, such as *write notices*. Generally, home-based software DSM systems propagate data information and apply them at homes eagerly, but this information is fetched by others lazily[2]. On the other hand, these systems propagate coherence information and apply them either eagerly or lazily. In our lock-based cache coherence protocol, coherence information is processed in a similar way to data information, and each coherence information has a static home according to the corresponding synchronization object (e.g., a lock or a barrier manager). So, coherence information is propagated to the corresponding home at release time, and is lazily fetched by the next acquirer of the same synchronization object. Compared with directory-based protocols, all coherence related actions in the protocol are applied at synchronization points. In this way, the lock-based protocol has least coherence related overheads for ordinary read or write misses. Moreover, the lock-based protocol is free from the overhead of maintaining the directory.

2.5 Summary

By analyzing the disadvantages of snoopy and directory based cache coherence protocols and several release consistency protocols, a novel lock-based coherence protocol for scope consistency is proposed in this Chapter. This new protocol has least coherence related overheads for ordinary read or write misses. Moreover, the lock-based protocol is free from the overhead of maintaining the directory. The correctness of lock-based protocol is proof too.

[2] Liviu uses intermediate to represent this data propagation and application mechanism[54].

Chapter 3

JIAJIA Software DSM System

3.1 Introduction

This chapter introduces our design and evaluation of a software DSM system called JIAJIA. JIAJIA is characterized by its lock-based cache coherence protocol for scope consistency (ScC) and home-based memory organization scheme.

Another distinguishing feature of JIAJIA is that it combines physical memories of multiple computers to form a larger shared space. In other recent software DSMs such as TreadMarks, CVM, and Quarks, the shared address space is limited by the local memory size of one machine. In JIAJIA, the shared space is distributed across processors and the local part of shared memory in each processor is the home of that part of shared memory. With this memory organization, the size of shared space can be as large as the sum of each machine's local memories. Since each page has a fixed home and adopting of global ordering with acknowledgements, JIAJIA totally eliminates the complexity of local diffs keeping, garbage collection, local address to global address translation, and timestamp maintenance.

Performance measurements with some widely accepted DSM benchmarks such as SPLASH2 program suite and NAS Parallel Benchmarks indicate that, compared to recent software DSM systems such as CVM, higher speedup is reached by JIAJIA. Besides, JIAJIA can solve large problems that cannot be solved by other software DSMs due to memory size limitation.

The rest of this chapter is organized as follows: Section 2 introduces the memory organization of JIAJIA. Section 3 briefly describes the lock-based cache coherence protocol implemented in JIAJIA. Section 4 gives JIAJIA's application programming interface. Section 5 presents the details of implementation. Evaluation results and analysis, especially, evaluation of a real world application, Em3d, are presented in Section 6. Finally, the scalibility are also studied.

3.2 Memory Organization

Normally, memory organization of a DSM system takes either CC-NUMA or COMA architecture (or home-based and homeless[23], symmetry and asymmetry[69]). In CC-NUMA, the shared memory is distributed across all nodes statically (and most often evenly). Each location of the shared memory has a fixed physical address. A node is the *home node* of its part of shared addresses. In this way, each location has a fixed *home node*, which is determined by the physical address of the location. When a cache miss occurs, the request for data is directed, according to its physical address, to the home node of the location.

Figure 3.1: Memory organization of CC-NUMA.

In COMA, even though the shared memory is also distributed across all nodes, the location of a data item is no longer fixed to a certain physical address. Instead, any data item can be automatically migrated or replicated in the memory of different processors depending on the memory reference pattern. Migrating data at the main memory level reduces the probability of remote access on a cache miss, but requires a mechanism to locate the data when local access is failed. Normally, an *owner* is maintained for each data item to identify its last modifier in COMA. A processor visits the *owner* of the associate data item on a cache miss. Unlike *home*, the *owner* of a data item moves across processors dynamically according to the memory reference pattern.

The memory organization of CC-NUMA and COMA are sketchy shown in Figure 3.1 and Figure 3.2 respectively.

Most recent software DSM systems adopt the COMA-like memory architecture. In those systems such as Munin, TreadMarks, CVM, and Quarks, the shared memory is normally initially allocated in host 0 (or server). When referenced by other processors, the page moves and replicates across processors dynamically. Complex directory data structures such as probowner, approximate copyset, write notice records, and interval records are employed to locate the last modifier(s) of a page on a page fault. In these systems, a shared page stays at different user space virtual addresses at different processors, and is identified by a global address which is identical in all processors. Each processor maintains a table to map between the global and local addresses of all pages. When a remote access occurs, the processor looks up on the table to find the global address. After the miss request arrives at the destination, the serving node turns the global address into its local address to locate the page. In these software DSM systems, the shared space is limited by the physical memory (including main memory and swap space) of one machine due to the following facts:

- All shared space is initially allocated in one node and hence the total shared space allocated is limited by the physical memory of one machine.

- To keep coherence, each processor maintains a local page table to keep directory information, twins, diffs, protect states, and local and global addresses of all pages. The

Figure 3.2: Memory organization of COMA.

size of this page table scales linearly with the number of shared pages while the local memory of each processor does not scale as well. For example, suppose the page size is 4KB and each item of the page table takes 1KB on average, then for an application which requires 256MB shared space, each processor has to maintain a page table with the size of 64MB which exceeds main memory capacity of many workstations, let alone other memory requirements of the system and application.

- The last copy problem[119] makes it is very difficult to implement replacement algorithm in COMA-like systems.

Figure 3.3 shows JIAJIA's organization of the shared memory. Unlike other SVM systems, JIAJIA organizes the shared memory in a NUMA-like way. In JIAJIA, each shared page has a fixed home node and homes of shared pages are distributed across all nodes. References to local part of shared memory always hit locally. References to remote shared pages cause these pages to be fetched from its home and cached locally. When cached, the remote page is kept at the same user space address as that in its home node, i.e., address of a page is identical in all processors, no matter that page is a home page or a cached page in the processor. In this way, no address translation is required on a remote access.

With the above memory organization, JIAJIA is able to support shared memory that is much larger than physical memory of one machine. In JIAJIA, homes of shared pages are distributed across all processors and hence total shared pages allocated are not limited by the physical memory of a machine. Instead of keeping information of all shared pages, the page table of each processor contains only information about its "cached" pages. Besides, for each cached page, the page table only keeps its address, protect state, and a twin for read-write pages. Diffs of a page are written back to its home in proper time to avoid local diffs keeping, and the directory information is unnecessary in the lock-based cache coherence protocol of JIAJIA.

In JIAJIA, shared pages are allocated with the `mmap()` system call. Each shared page has a fixed global address which determines the home of the page. Initially, a page is mapped to its global address only by its home processor. Reference to a non-home page will cause the

Figure 3.3: Memory organization of JIAJIA.

delivery of a SIGSEGV signal. The SIGSEGV handler then maps the fault page to the global address of the page in the local address space. Since the total shared memory allocated may be much larger than the physical memory of one host, mapping too many remote pages will break the system down. To avoid this, each host maintains a "cache" data structure to record all locally kept non-home pages. Any locally kept remote page must find a position in the local cache. If the number of locally kept remote pages is larger than the maximum number allowed, some aged cache pages must be replaced to make room for the new page, i.e., the aged page is unmapped.

Another interesting feature of JIAJIA's memory organization scheme is that the home size can vary from processor to processor and can be indicated by the programmer. Since homes of shared memory are allocated sequentially from the first host to the last node, varying the home size of hosts and allocation order allows the programmer to control the distribution of shared data.

For example, consider a configuration of four nodes each with 16MB physical memory. Suppose an application uses a 12MB-size shared variable `a` and an 8MB-size shared variable `b`.

- If the programmer indicates that each node has a 5MB home, and allocates `a` before `b`, the distribution of `a` and `b` across the four nodes is shown in Figure 3.4(a).

- If the programmer indicates that the first two nodes have 6MB home each, and the third node has a home of size 8MB, the distribution of `a` and `b` across the four nodes is shown in Figure 3.4(b).

- If the programmer decides to allocate `a` evenly across the first three processors and `b` solely in the fourth node, then home sizes of the four processors are 4MB, 4MB, 4MB, and 8MB respectively. Figure 3.4(c) shows data distribution of this case.

- If the programmer decides to allocate `b` in the first host and `a` evenly between the third and the fourth node, then home sizes of the four processors are indicated as 8MB, 0MB, 6MB, and 6MB respectively, and `b` is allocated first. Figure 3.4(d) shows the data distribution.

(a) (b)

(c) (d)

Figure 3.4: Memory Allocation Example.

Compared with the homeless memory organization scheme, the home-based scheme has following advantages: (1) Only one round trip to the corresponding home of the fault page is required for each access fault; (2) Writes to home pages do not produce twins and diffs; (3) Diffs are only applied at home by one time, the effects of cache pollution is neglectable; (4) Diffs are sent to home eagerly at the next synchronization point, hence consumes much less memory, eliminates garbage collection; (5) No diff accumulation problem; (6) Easy to implement cache replacement; and (7) Simple coherence state. On the other hand, the potential disadvantages of the home-based protocol scheme are (1) the whole page is transferred on a fault; (2) that the performance depends greatly on the home distribution.

3.3 Lock-Based Cache Coherence Protocol

The basic description of the lock-based cache coherence protocol is described in the last Chapter. It can be found that the unique characteristic of this new protocol is the use of "home" concept not only for data information, but also for coherence information, such as *write notices*.

In the basic protocol, a cached page is invalidated on an acquire (or barrier) if there is a write notice in the lock indicating that this page has been modified. However, if the modification is made by the acquiring processor itself, and the page has not been modified by any other processors, then the invalidation is unnecessary since the modification has already been visible to the acquiring processor. With this optimization, a processor can retain the access right to pages modified by itself on passing an acquire (or barrier), to avoid the unnecessary state transition from `RW` to `RO`.

3.4 Programming Interface

The API of JIAJIA is similar to that of the Treadmarks software DSM systems. It provides six basic calls: `jia_init(argc,argv)`, `jia_alloc(size)`, `jia_lock(lockid)`, `jia_unlock(lockid)`, `jia_barrier()`, and `jia_exit()` to the programmer. Besides, JIAJIA offers some subsidiary calls: `jia_setcv(condvar)`, `jia_resetcv(condvar)`, and `jia_waitcv(condvar)` to provide the conditional variable synchronization method, `jia_clock()` to return the elapsed time since the start of application in seconds in `float` type, and `jia_error(char *str)` to print out the error string `str` and shut down all processes.

JIAJIA provides two variables `jiapid` and `jiahosts` to the programmer. They specify the host identification number and the total number of hosts of a parallel program.

JIAJIA looks for a configuration file called `.jiahosts` in the directory where the application runs. This file contains a list of hosts to run the applications, one per line. Each line contains 3 entries: *machine name, user account*, and *password*. The first line of `.jiahosts` should be the master on which the program is started.

A distinguishing feature of JIAJIA's API is that it allows the programmer to flexibly control the initial distribution of homes of shared locations. The basic shared memory allocation function in JIAJIA is `jia_alloc3(size, blocksize, starthost)` which allocates `size` bytes cyclically across all hosts, each time `blocksize` bytes. The `starthost` parameter specifies the host from which the allocation starts. JIAJIA also defines two simpler shared memory allocation calls: `jia_alloc2(size, blocksize)` which equals to `jia_alloc3(size, blocksize, 0)` and `jia_alloc(size)` which equals to `jia_alloc3(size, Pagesize, 0)`.

Another interesting character of JIAJIA's API is that it also provides some MPI-like message passing calls: `jia_send(buf, len, topid, tag)`, `jia_recv(buf, len, frompid, tag)`, `jia_bcast(buf, len, root)`, and `jia_reduce(sndbuf,rcvbuf, count, operation, root)`. These message passing calls allows the programmer to write message passing program for some modules (or just port existing message passing modules) and write shared memory program for other modules. Besides, using some message passing primitives in shared memory programs helps to improve performance in many cases.

3.5 Implementation

Our implementation[1] of JIAJIA mainly includes the initialization module (`init.c`), the memory management module (`mem.c`), the synchronization module (`syn.c`), and the communication module (`comm.c`). The initialization module is responsible for initializing internal data structure and starts multiple processes (`jia_init()`). The memory management module provides a shared memory allocation routine `jia_alloc`, processes SIGSEGV signal, and services GETP (get page) and DIFF (write back *diff*s) requests to the home of associated page. The synchronization module implements `jia_lock`, `jia_unlock`, `jia_barrier`, and `jia_wait` calls and serves associated ACQ (acquire), REL (release), BARR (barrier), and WAIT (wait) requests at the home of the lock or barrier. The communication module provides reliable, in-order, deadlock-free asynchronous interprocesses communication to the memory management and synchronization module. Besides, there is a tool module (`tools.c`) which provides some tools

[1] Here, we describe the original implementation of JIAJIA (version 1.0). Several new functions are provided in the new version of JIAJIA with different optimization techniques in the rest of this dissertation.

and utility functions and an exit module (`exit.c`) which provides the `jia_exit()` call and collects statistics of all hosts.

3.5.1 Starting Multiple Processes

Figure 3.5 shows the flow charts of creating multiple processes[2]. Multiple processes of the application program are created in `jia_init(argc, argv)` which must be called at the beginning of the program.

Since JIAJIA adopts the SPMD programming model, the same program is executed by both the master and the slaves. To ensure only the master can creat remote processes, it is important to distinguish the master from slaves at the very beginning. This is done by requiring the master to be specified as the first host in the `.jiahosts` configuration file which specifies the machine name, user name and password of all hosts, one per line. After the program is started, it first decides its *hostid* through comparing its own host name and user name (get through system call) with those of hosts specified in `.jiahosts`. The *hostid* of the host specified in the ith line (excluding blank and/or comment lines) of `.jiahosts` is $i-1$. The host with host id of 0 identifiers itself as the master.

The master then starts remote processes through calling `rexec()`. On a success call of `rexec()`, a file descriptor is returned as the standard I/O of the started process. This file descriptor is used to synchronize the master and slaves later.

3.5.2 Shared Memory Management

Shared memory management includes shared memory allocation routine, `SIGSEGV` handler, and GETP and DIFF requests server at the home of a page.

Shared Memory Allocation

The function `jia_alloc(size)` allocates `size` bytes of shared memory. As has been indicated, JIAJIA allows allocation of shared memory larger than the real memory of a machine. The amount of shared memory allocated is only limited by two factors: the total amount of main memory in all machines of the system and the address capability(4GB in 32-bit OS system) of the system. Since the shared space is larger than the main memory of a machine, each machine only physically kept part of the total shared memory. This physically kept part of a shared page is called the *home* of the page. Homes of all shared pages are distributed among all nodes. When a node accesses a shared page that is not physically kept in its local home, it fetches the accessed page from the home node of page and keeps the page in its “cache” which is a part of its physical memory. Replacement of “cache lines” occurs when too many remote pages are cached. Hence, in our implementation, a fixed amount of the physical memory in each node is allocated as the home and another fixed amount of the physical memory is used as the cache.

Figure 3.6 shows the flow chart of the function `jia_alloc`. It first checks if there are enough shared memory to be allocated. Then the argument `size` is aligned to the boundary of page. The following loop allocates `size` bytes shared memory. Shared pages are allocated with the UNIX `mmap()` system call. They are mapped to the fixed address with the `mmap()` parameter `MAP_FIXED|MAP_PRIVATE`. A shared page is only mapped in its home node. It is initially mapped as `PROC_READ` only to detect writes by the home host. Optimization is done to mapped shared

[2] In NFS environment, the remote copy is not required.

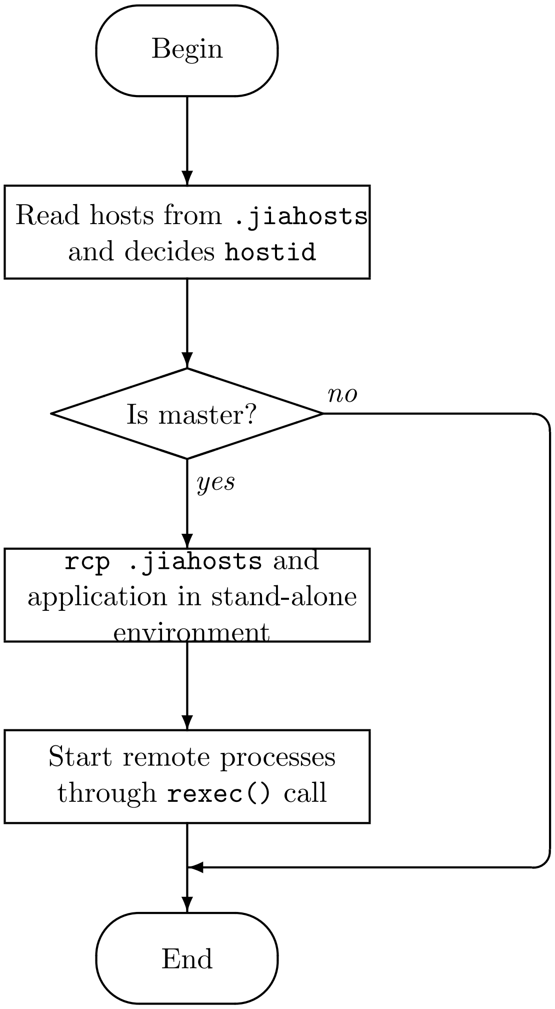

Begin
Read hosts from .jiahosts and decides hostid
Is master?
no
yes
rcp .jiahosts and application in stand-alone environment
Start remote processes through rexec() call
End

Figure 3.5: Flow chart of threads creating procedure `jiacreat()`.

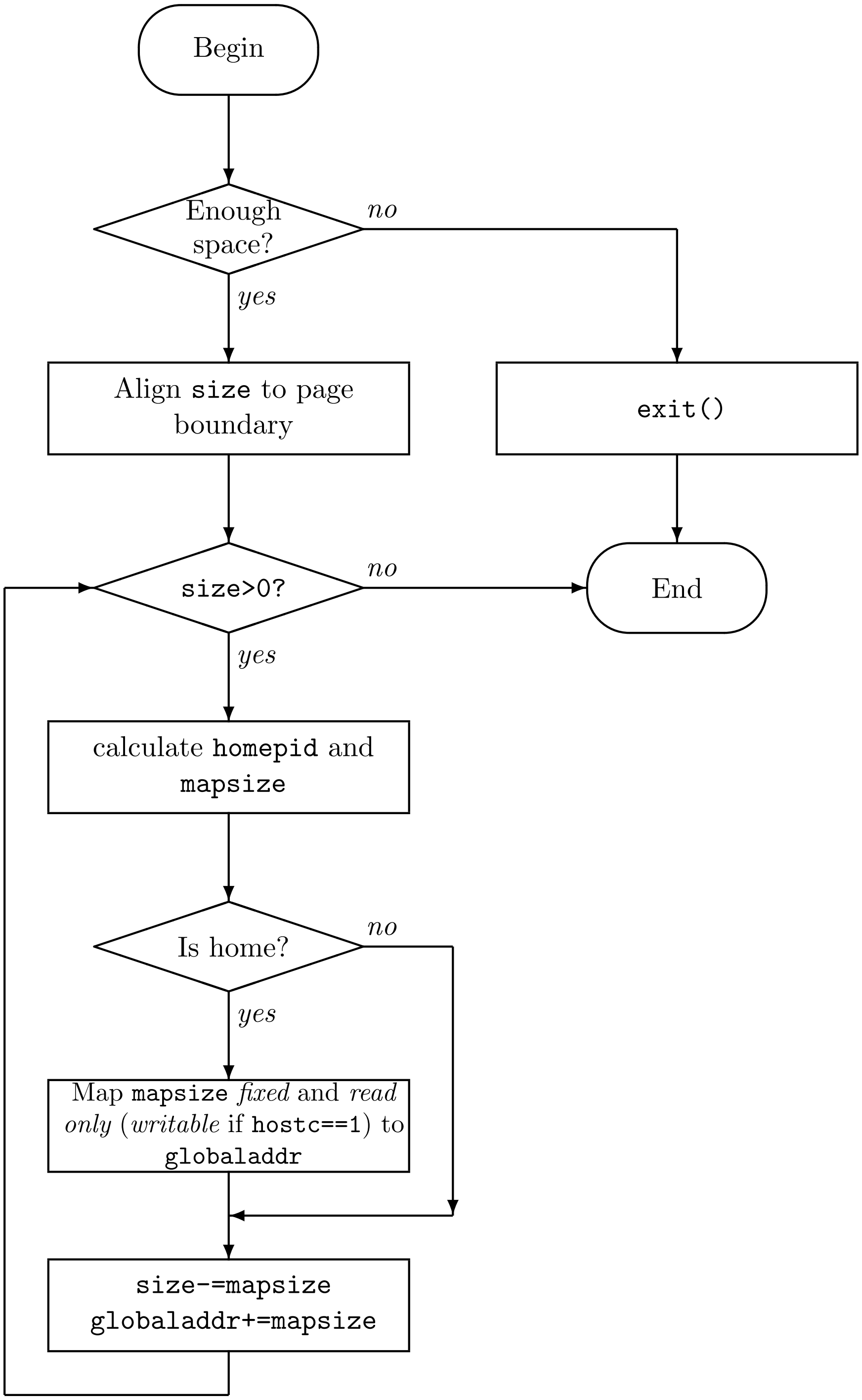

Begin
Enough space?
no
yes
Align size to page boundary
exit()
size>0?
no
End
yes
calculate homepid and mapsize
Is home?
no
yes
Map mapsize fixed and read only (writable if hostc==1) to globaladdr
size-=mapsize
globaladdr+=mapsize

Figure 3.6: Flow chart of memory allocation `jia_alloc(size)`.

pages as PROC_READ|PROC_WRITE if the number of hosts is judged to be one. The home of shared pages is sequentially allocated from the first node to the last node. The variable globaladdr, which records the shared bytes allocated, decides the home(denoted by homepid) for a shared page. At a jia_alloc(size) call, the size shared space is allocated in a round-robin way by default, with the page size as block size.

SIGSEGV Handler

Accesses to locations which are not in the correct protection mode will result in the delivery of a SIGSEGV signal.

On catching of a SIGSEGV signal, the SIGSEGV handler first gets the fault address and fault state from the system. If the fault address indicates that the access fault happens at the home of the page, then it must be a write fault (since all home pages are mapped as PROT_READ or set to PROT_READ at the beginning of an interval), the handler simply changes the protect mode of the page into PROT_WRITE|PROT_READ and records the fact that the page has been modified.

If the host is not the home of the fault page, then a location in the cache is found for the page.

As has been stated, shared pages are initially mapped at their home only. When a host accesses a shared page that is not physically mapped in its local home, it fetches the page from the home of the page and maps it locally into the same address as that in the home node. Since the total shared memory space allocated may be much larger than the physical memory space of a host, allocating too many physical space for remote pages will break the system down. Hence, the number of physically kept non-home pages in a host should be limited. To do this, each host maintains a "cache" data structure to record all locally kept non-home pages. The cache keeps global address and protect state (unmapped, invalidated, read-only, or read-write) for any physically kept pages whose homes are in remote hosts. Twins of writable non-home pages are also kept at the cache data structure. Any locally kept remote pages must find a position in the local cache. If the number of locally kept remote pages is larger than the maximum number allowed, some aged cache pages must be replaced to make room for the new page, i.e., the aged page is unmapped.

A location for a fault page is found in the following order. First, a cached page with the same address as the fault page is looked for (in case of a read fault on an INV page or a write fault on an INV or RO page). The cache is then searched for an unmapped page if the page with the same address as the fault page does not exists. An invalidated page is then looked for to be replaced if there is no unmapped page in the cache. If all the above methods fail to find a cache location for the fault page, a read-only or read-write page is replaced randomly or in a round robin way to make room for it. When a writable page is replaced, the related *diff* and write notice are recorded to be delivered to the home of the page and of associated lock respectively when a release or barrier is encountered later.

After a location is found in the cache for the fault page, the state of the fault page is set to PROT_READ|PROT_WRITE if the page has already been mapped locally (in case of a read fault on an INV page or a write fault on an INV or RO page), or the fault page is mapped as PROT_READ|PROT_WRITE into the fault address otherwise.

On a write fault, a get page request GETP is sent to the home of the page if it is not a write fault on a RO page. After the get page acknowledgement GETPGRANT is received, the fault page is copied to the associated address of the page. The cache then records the page address and page access state (read-write). A twin of the page is created and kept in the cache.

On a read fault, a get page request GETP is sent to the home of the page. After the get page acknowledgement GETPGRANT is received, the fault page is copied to the associated address of the page and the protect mode of the page is set to PROT_READ. The page address and access state (read-only) are then recorded in the cache.

Servers

The memory management module also services get page request GETP and write diff request DIFF. Both servers are called by the SIGIO handler which is activated by arriving of a message.

On receiving a GETP request, the server takes the page address from the request message, reads the page from memory, and sends the page back to the requesting host on the acknowledgement message GETPGRANT.

On receiving a DIFF request, the server takes the page address and diff from the request message, decodes the *diff* and apples it to the memory. An acknowledgement message DIFFGRANT is then sent back to the requesting processor. Note that, the home page of the *diff* should be set to writable before the *diff* is written and reset to its original state after that.

Both GETP and DIFF requests are serviced only at the home of the associated page.

3.5.3 Synchronization

In the lock-based cache coherence protocol, all coherence related actions are taken at the synchronization point. JIAJIA implements two kinds of synchronizations: lock and barrier. In JIAJIA, `jia_lock(lockid)` and `jia_unlock(lockid)` acquire and release a lock respectively, while `jia_barrier()` provides a global synchronization mechanism.

Requirement of Scope Consistency

By the semantics of scope consistency, a lock acquirer can observe any modifications made by the last releaser of the same lock, or any host who has just left a barrier can observe any modifications made by any hosts before arriving the barrier. In our lock-based cache coherence protocol for scope consistency, this is done by requiring the releaser of a lock (or the host who arrives at a barrier) to send all write notices produced in the related critical section to the home of the lock, and the acquirer of a lock (or the host who leaves a barrier) to invalidate locally cached pages according to the write notices related to the lock (or barrier).

The above protocol guarantees the semantics of scope consistency to be obeyed. However, there is room for improvement. If a page has been modified by only one processor since last barrier, then this processor need not invalidate its cached copy of that page on a barrier or an acquire of the associated lock. This does not violate the semantics of scope consistency because the real purpose of invalidation on an acquire or a barrier in scope consistency is to make the modification of the invalidated page be visible to the acquiring or barrier-leaving processor, and the processor who uniquely modified the page has already observed this modification.

The above minor improvement of invalidation on an acquire or a leave of barrier often benefits the performance a lot. In many applications, a page is normally modified by only one processor at a time (even if the multiple-write protocol does not forbid a page to be modified by multiple processors simultaneously) and it is possible that the processor will reference the page again.

Lock Structure

As has been stated, in our locked-based cache coherence protocol for scope consistency, all coherence related actions are taken through writing and reading write notices in the lock. To avoid the lock becoming the bottleneck of the system, the locks are evenly distributed among all hosts.

JIAJIA maintains an array of `Maxlocks` locks, each element of the array is a structure to record write notices related to this lock. Write notices are recorded in a lock on a release of the lock. The lock is cleared and all recorded write notices are removed on a leave of barrier.

To make the above optimization, a write notice of JIAJIA not only includes the address of the modified page, but also includes the processor who makes the modification. If a page is modified by more than two processors, then the page is recorded as has been modified by `Maxhosts` processors, where `Maxhosts` is the maximum number of processors allowed in JIAJIA, and is not the real processor id of any processor.

Lock Stack

To identify modified pages of a critical section, all pages are set to non-writable when entering that critical section. Any written to a non-writable page will lead to the delivery of a SIGSEGV signal. The written page state is then turned into RW in SIGSEGV handler. On leaving of a critical section, *diff*s and write notices are produced for all RW pages.

Cares should be taken in identifying modified pages of a critical section when there are nested critical sections. In program segments of Figure 3.7(a), `x0` is modified only in critical section of lock 0, while in program segments of Figure 3.7(b), `x0` is modified in critical sections of both lock 0 and lock 1. For program segments of Figure 3.7, the following measures should be taken to precisely identify the *diff*s and write notices of critical sections.

- To decide whether `x0` is modified in the inner critical section, the state of `x0` should be initialized to RO before entering the inner critical section.

- To record that `x0` has been modified in the outer critical section before entering the inner critical section, *diff*s and write notices of `x0` should be kept somewhere before `x0` is set to RO state.

- To identify that `x0` is modified in the outer critical section again after leaving the inner critical section, the state of `x0` should be set to RO when leaving the inner critical section.

We can conclude that, to precisely decide *diff*s and write notices of nested critical sections, states of all RW pages should be set to RO whenever entering or outing a critical section. *Diff*s and write notices of RW pages should be calculated and kept whenever a read-write page becomes read-only.

In JIAJIA, a lock stack is employed to record *diff*s and write notices of nested critical sections. The stack records lock id and write notices and *diff*s produced in the critical section protected by the lock. The stack is pushed when entering a critical section, and is popped when leaving. When entering or outing a critical section, *diff*s and write notices are calculated and recorded in stack top before the stack is pushed or popped. In this way, the stack top always records *diff*s and write notices of the present critical section. Besides, since modifications of an inner critical section are also modifications of an outer critical section, all stack top write

```
jia_lock(0);
  x0=1;
  jia_lock(1);
    x1=1;
  jia_unlock(1);
  x0=2;
jia_unlock(0);
```

(a). Program Segment 1

```
jia_lock(0);
  x0=1;
  jia_lock(1);
    x0=1;
  jia_unlock(1);
  x0=2;
jia_unlock(0);
```

(b). Program Segment 2

Figure 3.7: Examples of nested critical sections.

notices are copied to the next layer before the stack top is popped on a leaving of critical section.

A barrier can be viewed as a combination of a lock and an unlock. Arriving at a barrier ends an old "critical section", while leaving a barrier begins a new one. In this way, two barriers enclose a critical section. In JIAJIA, the critical section between two barriers (the begin and end of a program can also be viewed as a hide barrier) must be the most outside critical section, i.e., the `jia_barrier()` is not allowed to be called between a paired `jia_lock(lockid)` and `jia_unlock(lockid)`. A hide lock is defined to records write notices related with barriers. The hide lock is pushed into the lock stack at the initialization stage, i.e., the lowest level of lock stack always keeps write notices and *diff*s related with the barrier.

Implementation of `jia_lock()`

Based on the above analysis, the actions taken on a `jia_lock(lockid)` are:

1. Calculate *diff*s and write notices for all RW pages and put them on the stack top.
2. Set states of all RW pages to RO.
3. Send acquire request ACQ to home of associated lock and wait until lock grant message ACQGRANT is received.
4. On receiving of ACQGRANT message, cached pages are invalidated according to write notices sent together with the ACQGRANT message.
5. Push stack(Initialize stack top).

On receiving of the ACQ request, the lock takes the following actions:

1. Add the requesting processor to the tail of the wait queue.
2. If there is only one requester on the wait queue, then send acquire grant message to the requester, write notices of the lock are sent together.

Implementation of `jia_unlock()`

On a `jia_unlock(lockid)`, the following actions are taken:

1. Calculate *diff*s and write notices for all RW pages and put them on the stack top.
2. Set states of all RW pages to RO.
3. Send *diff*s to home of associated pages and wait until write *diff* acknowledgement message DIFFGRANT is received.
4. Send release request REL to home of the lock, write notices are sent together.
5. Copy write notices on the stack to next level of the stack.
6. Pop stack (Clear *diff*s and write notices of the stack top).

Note that, to implement correct event ordering required by scope consistency, *diff*s should have been written back to their home before the lock is released. Hence, before sending release request REL to the lock, all DIFF requests should have been acknowledged. The processor stalls between write *diff* request DIFF have been sent out and DIFF acknowledgement message DIFFGRANT is received.

On receiving of the REL request, the lock takes the following actions:

1. Remove the requesting host in the head of wait queue.
2. If wait queue is not empty, then send acquire grant message to the requester in the head of the queue, write notices of the lock are sent together.

Implementation of `jia_barrier()`

The actions taken on a barrier are:

1. Calculate *diff*s and write notices for all RW pages and put them on top of the stack.
2. Set states of all RW pages to RO.
3. Send *diff*s to home of associated pages and wait until write *diff* acknowledgement message DIFFGRANT is received.
4. Send barrier request BARR to the hide lock, write notices are sent together. Wait until barrier grant message BARRGRANT is received.
5. Invalidate cached pages according to the write notices piggybacked by the BARRGRANT message.
6. Clear write notices of all local locks.
7. Pop stack (Clear *diff*s and write notices of the stack top).
8. Push stack(Initialize stack top).

On receiving of the BARR request, the home host of the hide lock takes the following actions:

1. Increase the barrier counter by one.

2. If the barrier counter equals the number of processors, send barrier grant message BARRGRANT to all hosts, write notices related to the hide lock are sent together.

3. After the BARRGRANT have been sent to all processors, the barrier counter is set to zero and all write notices recorded in the hide lock are cleared.

Implementation of `jia_wait()`

Besides the barrier, JIAJIA also provides a simple global synchronization mechanism to the user. The `jia_wait()` call is similar to `jia_barrier()` in that both of them require the arrival of all hosts before any one can proceed. They are different in that `jia_wait()` does not enforce any coherence across hosts. Therefore, `jia_wait()` is a simple synchronization mechanism that requires all processors to wait together before going ahead. Since no coherence related actions are taken at `jia_wait()`, it can be called inside a critical subsection.

The actions taken on a `jia_wait()` are:

1. Send wait request WAIT to wait manager.

2. Wait until wait grant message WAITGRANT is received.

On receiving of the WAIT request, wait manager takes the following actions:

1. Increase the wait counter by one.

2. If the wait counter equals the number of processors, send wait grant message WAITGRANT to all hosts.

3. After the WAITGRANT have been sent to all processors, the wait counter is set to zero.

3.5.4 Communication

The communization module provides a reliable and in-order message delivery on top of UDP/IP protocol. The asynchronous send function `asendmsg(msg_t *msg)` sends a message to a host whose host id is indicated in the `*msg`. On arrival at its destination, the message causes the delivery of a SIGIO signal to the target process. The sigio handler of the target process then calls the corresponding message server according to the message type.

Message Structure

A message in JIAJIA includes the following 6 parts:

- **Message type**. The `op` field of a message points out what kind of message it is. There are 14 types of messages in JIAJIA.

- **Message source**. The `frompid` field of a message records the sender of the message.

- **Message destination**. The `topid` field of a message indicates the host id of the target host.

- **Message serial number**. The `seqno` field of a message is used to ensure the in order transferring of a message.

- **Message size**. The `size` field of a message indicates the size of data part of the message.

- **Message data**. The `data` field of a message records the transferred data of the message. The format of this field is vary in different message type.

Besides, pad are used to allocated page aligned messages.

The 14 types of message are[3]

- **DIFF** The DIFF message sends *diff*s produced in a critical section to the home of associated pages. The `data` field records the encoded *diff*s. A DIFF message may include *diff*s of multiple pages with the home.

- **DIFFGRANT** A DIFFGRANT message is sent to the DIFF requesting processor after the *diff*s has been applied to its home. This message is necessary because the DIFF requesting processor must be sure that the *diff*s of a critical section have been written back to their home before it can release a lock.

- **GETP** The GETP message is sent when there is an access fault and the fault host does not have a valid copy of the page. The `data` field records the fault address. Only one page is get on a GETP request in our present implementation. In the future, multiple pages may be prefetched on a reference fault.

- **GETPGRANT** The GETPGRANT message sends the required page of GETP to the requesting processor. The `data` field of a GETPGRANT message includes the fault address and the fault page.

- **ACQ** The ACQ message is sent on an acquiring of a lock. The lock id is recorded in the `data` field of the message.

- **ACQGRANT** An ACQGRANT message is sent to the requesting processor if the lock is free when receiving an ACQ request, or to the first processor of lock waiting queue on receiving an REL request of the lock. Write notices of the lock are sent together.

- **BARR** A BARR message is sent to the hide lock on a barrier. Write notices produced before arriving the barrier are sent together.

- **BARRGRANT** A BARRGRANT message is sent to all processors from the hide lock after the last BARR message is received. Write notices of the hide lock are sent together.

- **WAIT** The WAIT message is sent to wait manager on arriving of a `jia_wait()`.

- **WAITGRANT** On receiving of a WAIT request, if the wait counter indicates that all hosts have arrived at the `jia_wait()`, A WAITGRANT message is sent to all hosts.

- **REL** The REL message is sent to the released lock on a `jia_unlock()`. Write notices of the associated critical subsection are sent together.

[3] In the new version 2.1, two extra messages are added, such as **LOAD, LOADGRANT**, which is used for load balancing.

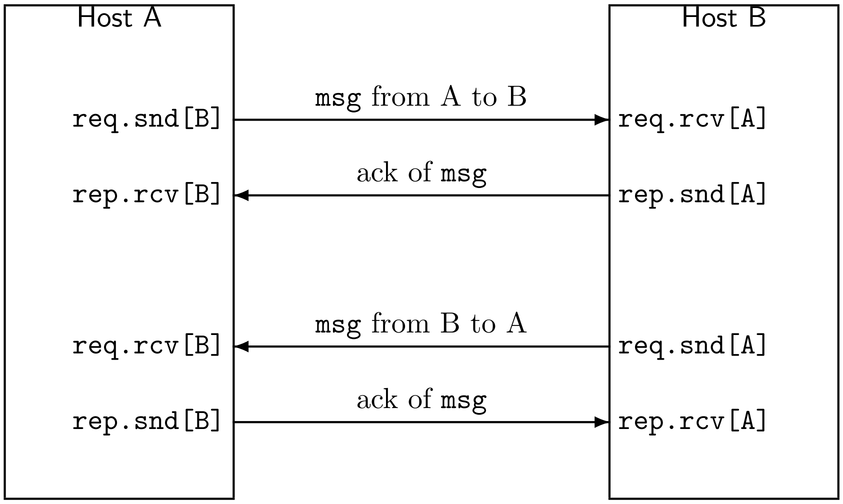


Figure 3.8: Communication between two processors.

- **INVLD** When there are too many write notices of a lock to be carried by a ACQGRANT or BARRGRANT message, these write notices can be enclosed into one or more INVLD messages which are sent to the requesting processor before ACQGRANT or BARRGRANT.
- **WTNT** Like INVLD, the WTNT message is only used when there are too many write notices of a critical subsection to be sent by one REL or BARR message, i.e., a release or barrier message may be composed of a series of WTNTs followed by a REL or BARR.
- **JIAEXIT** This message is used to inform other hosts to exit when an error has been detected by a host.

More details of the format of these messages can be found in our technical report[48]. Besides, there are two messages STAT and STATGRANT which are used for collecting statistics of all host to host 0 in `jia_exit()`.

Interprocessor Communication

To provide reliable, in-order message delivery on top of UDP/IP, four sockets are created for each pair of processors. Figure 3.8 shows the interaction between host A and host B in sending and receiving messages.

When processor A wants to send message to processor B, it sends the message through socket `req.snd[B]` and waits for acknowledgement in socket `req.rcv[B]`. On receiving of this message, a SIGIO signal is delivered to processor B. In the SIGIO handler, processor B receives the message from A through socket `req.rcv[A]` and send the sequential number of the received message to A through socket `rep.snd[A]`. This acknowledgement message is then received by A through socket `req.rcv[B]`.

If, for some reasons, host A cannot receive the acknowledgement message from B in a `TIMEOUT` period of time, it resends the message to B and waits for an acknowledgement again. Host A gives up and exits if it cannot receive any acknowledgement after `MAX_RETRIES` retries.

If host B receives a message whose sequential number is less than or equal to that of the message it has received before from A, then the received message is determined to be a resend message. Host B then throws away this message.

Sending Message In Order

The `asendmsg(jia_msg_t *msg)` routine sends the message pointed by `msg` to host `msg->topid`. However, there may be case that the sending process is interrupted by arriving of a message from another host. As has been mentioned, the associated message server should be called on receiving of a message. If the message server needs to send a message out (e.g., GETPGRANT message is sent in the GETP server), then the second message produced in message server is sent before the first message whose sending process is interrupted. This causes out of order sending of messages., i.e., the message will small sequential number is sent after the message with big one.

One way to avoid out of order message sending is to disable SIGIO signal when sending message. Anyone who wants to send message to the host whose SIGIO signal is disabled has to wait until the signal is enabled. However, since the message sending process is quite long, the time other hosts wait may be quite long. This, of course, is detrimental to performance.

Our way to solve this problem is to maintain a sending queue for each host. When a host needs to send a message, it puts the message in the queue. If there is only one message in the queue, then the message is sent to the target host as is required. Otherwise, messages in the sending queue are sent in the order they are put in. In this way, the SIGIO signal is disabled only when accessing the queue.

Serving Message In Order

On catching of a SIGIO signal, the SIGIO handler is called to receive and serve message. In a multiprocessor environment, it is usual that another message arrives while a message is being served. Serving messages out of the coming order may cause serious problems. For example, after the home host of hide lock receives the last BARR request it needs, the barrier server is called to send BARRGRANT to all hosts. If a host who receives the BARRGRANT arrives the next barrier immediately, another BARR request is sent to the hide lock. The hide lock, which is still busy sending BARRGRANT to other hosts, is then interrupted to receive and serve the new received BARR request. The barrier counter is increased before it is cleared. In this way, the next barrier can never be passed because the arrival of one BARR request is not taken into account.

To enhance performance, it is not suitable to disable the SIGIO signal while a message is being served. In order to serve messages in the order they are received, a receiving queue is maintained. On receiving of a message, the message is put into the receiving queue. If there are more than one message in the receiving queue, i.e., the SIGIO signal is received while another queued message is being served, then the SIGIO handler is returned. Otherwise, the corresponding message server is called for the received message. Again, the SIGIO signal is disabled only when the receiving queue is being accessed, which is much easier to be implemented.

3.5.5 Deadlock Free of Communcation Scheme

we have following theorem for the communication scheme used in JIAJIA.

Theorem 3.1 *Communication scheme used in JIAJIA is deadlock-free.*

Proof: Firstly, we assume the underlying communication substrate is deadlock-free. This assumption is reasonable. (1) From above description, it can be seen that two independent logical networks are used for requesting and acknowledgement messages respectively, which is similar to the interconnection network used in DASH [77]. Therefore, the requesting messages and acknowledgement messages have no chance to form a cycle. (2) Most importantly, no forwarding message is required makes there is no nonpreemptive resources in this communication scheme. As such, the second sufficient condition of deadlock is violated [123]. Therefore, communication scheme used in JIAJIA is deadlock-free. Box

3.6 Performance Evaluation and Analysis

The evaluation is done in the Dawning-1000A parallel machine developed by the National Center of Intelligent Computing Systems (NCIC). The machine has eight nodes each with a PowerPC 604 processor and 256MB memory. These nodes are connected through a 100Mbps switched Ethernet. In the test, all libraries and applications are compiled by `gcc` with the `-O2` optimization option. To compare the performance of JIAJIA with other software DSMs, the same applications are also run under CVM which is another software DSM system developed in Maryland University.

3.6.1 Applications

We port some widely accepted DSM benchmarks to evaluate the performance of JIAJIA. This chapter shows results of seven applications, including Water, Barnes, and LU from SPLASH and SPLASH2, EP and IS from NAS Parallel Benchmarks[7], and SOR and TSP from Rice University[82]. Memory reference characteristics of these applications are described as follows.

Water is a program for molecular dynamics simulation. Its main data structure is a one-dimensional array of records, each represents a molecule. Water statically divides this array into equal contiguous chunks, each assigned to a processor. The most time- and message-cost work happens in the inter-molecular force computation phase, in which each host computes and updates the intermolecular forces between each of its molecules and each of $n/2$ molecules following it in the array in warp-around fashion. Each processor is assigned a lock to protect updating forces of molecules belonging to it. Computation of iterations are separated by barriers.

Barnes simulates the evolution of bodies under the influence of gravitational forces. The hierarchical Barnes-Hut method is used to reduce the complexity of the algorithm. Its major data structure is the Barnes-Hut tree which is represented with an array of bodies that are leaves of the tree and an array of internal cells in the tree. Only the array of bodies is shared. Each iteration of the program constitutes four phases: constructing the Barnes-Hut tree, partitioning the bodies among processors, computing forces, and updating the position and velocities of bodies. Most computation happens in the force computation phase. Barriers are the main synchronization mechanism.

LU factors a dense matrix into the product of a lower triangular and an upper triangular matrix with the block factorization algorithm. We select the contiguous block allocation LU which allows blocks to be allocated contiguously and entirely in the local memory (home in JIAJIA) of processors that "own" them. The algorithm factors the matrix in steps. Each step

Table 3.1: Characteristics of Benchmarks and Execution Results

Appl.	**Size**	**Shared**	**Barr.**	**Lock**	**Seq.**	**8-proc. Time**		**Speedup**	
		Memory	**#**	**#**	**Time**	**JIAJIA**	**CVM**	**JIAJIA**	**CVM**
Water	1728 mole.	484KB	35	65	178.00	26.47	39.79	6.72	4.47
Barnes	16384	1636KB	28	8	413.24	64.75	66.02	6.38	6.26
LU	2048×2048	32MB	128	0	84.86	25.04	35.21	3.39	2.41
LU	8192×8192	512MB	512	0	5464.80*	909.39	—	6.01	—
EP	2^{24}	4KB	1	1	49.69	6.30	6.30	7.89	7.89
IS	2^{24}	4KB	30	10	30.10	4.84	4.59	6.22	6.56
SOR	2048×2048	16MB	200	0	68.44	11.45	15.25	5.98	4.49
SOR	8192×8192	256MB	200	0	1235.76**	166.20	—	7.44	—
TSP	`-f20 -r15`	788KB	0	140	175.36	33.25	76.20	5.27	2.30

$*$: Estimated as eight times of 4096×4096 LU sequential time (683.10 seconds), sequential time of 8192×8192 LU is not available due to memory size limitation.

$**$: Estimated as four times of 4096×4096 SOR sequential time (308.94 seconds), sequential time of 8192×8192 SOR is not available due to memory size limitation.

first factors the diagonal block, then the following blocks in the same column is divided by the diagonal block, and the trailing submatrix is updated at last. Barriers are used to separate the three phases in each factorization step. Two sizes of LU, 2048×2048, and 8192×8192, are run in our evaluation. To achieve good reference locality, the block size is set to 32 and the page size is set to 8192 bytes (the default page size is 4096 bytes) in the evaluation.

EP generates pairs of Gaussian random deviates with a scheme that is well-suited for parallel computation and tabulates the number of pairs successively. The only communication and synchronization of EP is summing up a ten-integer list in a critical section at the end of program.

IS from NAS Benchmarks ranks an unsorted sequence of keys using bucket sort. It divides up the keys among processors. There is a shared bucket for all processors and each processor has a private bucket. First, each processor counts its keys in the private array of buckets. These values in private buckets are then summed up into the shared bucket in a critical section which is protected by a lock. Finally, each processor reads the sum and ranks their keys.

SOR solves partial differential equations with red-block successive over-relaxation method. In the program, the red and black array are allocated in shared memory and divided into roughly equal size bands of rows. Each processor computes a red and a black band, and synchronizes with other processors with barriers. Communication occurs across the boundary rows on a barrier. Two barriers are used for each iteration. We run two sizes of SOR, 2048×2048 and 8192×8192, for 100 iterations in the evaluation. The page size is set to 16384 in the 8192×8192 case to achieve good reference locality.

TSP solves the traveling salesman problem using a branch and bound algorithm. The major shared data structures of TSP include a pool of partially evaluated tours, a priority queue containing pointers to tours in the pool, a stack of pointers to unused tour elements in the pool, and the current shortest path. Processors evaluate the partial paths successively and alternatively until the shortest path is found. Locks are used to ensure exclusive accesses to shared objects.

The left part of Table 3.1 shows characteristics of the above benchmarks.

Table 3.2: Eight-Processor Execution Statistics

Appl.	**Messages**		**Msg. amt.(KB)**		**SIGSEGVs**		**Remote accesses**	
	JIAJIA	**CVM**	**JIAJIA**	**CVM**	**JIAJIA**	**CVM**	**JIAJIA**	**CVM**
Water	10828	95513	16850	72408	4892	30899	2847	19416
Barnes	37018	59520	80569	64960	34144	37370	17844	18193
LU2048	25992	24733	99950	203604	32663	23998	12072	11874
LU8192	386664	—	1569946	—	1108875	—	189720	—
EP	77	245	60	65	22	23	14	14
IS	1050	735	896	2710	230	240	140	150
SOR2048	8412	5692	11763	12662	2800	3341	2800	2786
SOR8192	8413	—	46135	—	2800	—	2800	—
TSP	20290	9046	24265	4948	8312	8773	1690	6308

3.6.2 Performance of JIAJIA and CVM

The right part of Table 3.1 shows sequential and eight-processor parallel run time of the benchmarks. As indicated in the table, the 8192×8192 LU and SOR cannot be run on single machine due to memory size limitation and the corresponding sequential run times are estimated values. Table 3.2 shows some runtime statistics of JIAJIA and CVM, include message counts, message amounts, SIGSEGV signal counts, and remote access counts.

It can be seen from Table 3.1 that, for most tested applications, JIAJIA achieves satisfied performance and speedup.

Both Water and Barnes are N-body problems and are characterized with tight sharing. As has been stated, the lock-based protocol of JIAJIA takes all coherence related actions in synchronization points and has least message overheads in ordinary write and read misses. In Water and Barnes, the number of shared pages is small, and this small number of pages are referenced frequently by multiple processors. Hence, the overhead at synchronization points is not high, and page faults caused by ordinary write and read operations introduce least messages in JIAJIA. As a result, compared to CVM, JIAJIA achieves a higher speedup in Water and Barnes. Statistics in Table 3.2 show that CVM introduces much more remote misses and consequently communication than JIAJIA in Water, while JIAJIA and CVM cause similar number of remote misses in Barnes. Table 3.2 also shows that, with the same number of remote misses, CVM causes more messages but less message amounts than JIAJIA. This is because compared to JIAJIA, CVM requires more messages to locate the owner and collect diffs of a page, but transfers only diffs instead of the whole page (as in JIAJIA) on a remote miss.

Computation-to-communication ratios of both LU and SOR are $O(N)$ where N is the problem size. As a result, speedups of both LU and SOR are acceptable and scale with the problem size. Frequent inter-processor synchronization contributes the main reason for the moderate speedup of LU and SOR in the 2048×2048 cases, because computation steps of LU and SOR are separated by barriers. Besides, in LU, only the updating of the trailing submatrix computation phase of each step is fully parallelized. In JIAJIA, matrices are initially distributed across processors in the way such that each processor keeps in its home the data it processes (writes). As a result, no diffs are produced and propagated in JIAJIA since all writes are made directly to home pages. This is an important reason for the superior performance of JIAJIA to CVM. Another reason for the performance difference between JIAJIA and CVM is that CVM requires a cold startup time to distribute the matrix across processors (though the `-m2` command line option is used in SOR). Statistics in Table 3.2 show that, CVM causes more message amounts than JIAJIA in LU and SOR though remote misses are similar in JIAJIA and CVM.

Both EP and IS have separate computation and communication phases, i.e., computation of EP and IS happen locally and communication happens only at the end of computation. EP achieves a speedup of 8 in both JIAJIA and CVM because the communication and computation ratio of EP is low. In IS, the most time-consuming computation is for each processor to count its local part of keys, while summing the counting results in the local buckets up into the shared bucket constitutes the communication work. We keep the number of buckets at 1024 which makes the communication amount relatively small compared to the computation work of counting 2^{24} keys. As a result, an acceptable speedup is achieved in both JIAJIA and CVM. CVM slightly outperforms JIAJIA in IS because JIAJIA has more overhead than CVM on a release. In JIAJIA, diffs have to be sent to its home before the release message is sent to the lock, while CVM keeps diffs locally. As a result, when summing up values of private buckets into the shared bucket, JIAJIA takes more time for each processor to enter and leave the critical section sequentially.

TSP specializes in that all inter-processor synchronizations are taken through locks. In TSP, each processor frequently reads from and writes to the pool of tours and the priority queue, causing tight sharing of pages (a page can store 27 paths in TSP[83]). The result of Table 3.1 and Table 3.2 seems contradicting: JIAJIA causes much more communications than CVM but performs better in TSP. Further experiments show that the single processor execution time of CVM is 478.07 seconds, much larger than the sequential time. When compared to its own single processor time, CVM achieves a better speedup (6.29). The reason for the more communication of JIAJIA than CVM lies on the different memory organization and coherence protocol in CVM and in JIAJIA. First, the COMA architecture of CVM allows the owner of a page migrates across processors and consequently reduces page faults to some extent. Second, on a page fault, CVM only transfers diffs which is rather small in TSP, while JIAJIA fetches the whole page from its home. Table 3.2 shows that JIAJIA has one time more messages but four times more message amounts than CVM.

It can also be seen from Table 3.1 that LU-8192 and SOR-8192 which cannot run on CVM due to memory size limitation can be run in JIAJIA with multiple processors.

3.6.3 Confidence-interval Based Summarizing Technique

Noteworthy, although JIAJIA outperforms CVM for all seven applications but IS, it is difficult to draw a definite statement that JIAJIA is better than CVM easily, but a probabilistic statement about the comparison can be made with the help of confidence interval. Therefore, we propose a confidence-interval based summarizing technique[110].

Concept of Confidence Interval

In order to get a clear understanding about confidence interval, we must have a background about *sample mean* and *population mean*. Suppose there are several billion random numbers with a given property, for instance, population mean μ and standard deviation σ. We now put these numbers in an urn and draw a sample of n observations. Suppose the sample $x_1, x_2, ..., x_n$ has a sample mean $\overline{x}$, which is likely to be different from population mean μ. Each sample mean is an estimate of the population mean. Given k samples, we have k different estimates.

The general problem is to get a single estimate of the population mean from these k estimates. That is similar to conclude which alternative is the best by comparing some benchmarks.

In fact, it is impossible to get a perfect estimate of the population mean from any finite number of finite size samples. The best we can do is to get probabilistic bounds. Thus, we may be able to get two bounds, for instance, c_1, c_2, such that there is a high probability, $1-\alpha$, that the population mean is in the interval(c_1, c_2):

$$\text{Probability}\{c_1 \leq \mu \leq c_2\} = 1-\alpha.$$

The interval (c_1, c_2) is called the **confidence interval** for the population mean, α is called the **significance level**, and $100(1-\alpha)$ is called the **confidence level**. There are two ways to determine the given confidence interval: *two-sided* or *single-sided.*

From above definition, in order to estimate the population mean, we need to gather k samples, which will entail large efforts. Fortunately, it is not necessary to gat too many samples because of the **central limit theorem**, which allows us to determine the distribution of the sample mean. This theorem tells us the sum of a large number of independent observations from any distribution tends to have a normal distribution.

With the central limit theorem, a two-sided 100(1-α)% confidence interval for the population mean is given by

$$(\overline{x} - z_{1-\alpha/2} s\sqrt{n}, \quad \overline{x} + z_{1-\alpha/2} s\sqrt{n}),$$

we $\overline{x}$ is the sample mean, s is the sample standard deviation, n is the sample size, and $z_{1-\alpha/2}$ is the $(1-\alpha/2)$-quantile of a unit normal variate. This formula applies only for the cases we the number of samples is larger than 30. When the number of samples is less than 30, confidence intervals can be constructed only if the observations come from a normally distributed population. For such samples, the $100(1-\alpha\ )$% confidence interval is given by :

$$(\overline{x} - t_{[1-\alpha/2;n-1]} s/\sqrt{n}, \quad \overline{x} + t_{[1-\alpha/2;n-1]} s/\sqrt{n}),$$

we $t_{[1-\alpha/2;n-1]}$ is the $(1-\alpha/2)$-quantile of a t-variate with $n-1$ degrees of freedom. The interval is based on the fact that for samples from a normal population $N(\mu, \sigma^2)$, $(\overline{x} - \mu/(\sigma/\sqrt{n})$ has a $N(0,1)$ distribution and $(n-1)s^2/\sigma^2$ has a chi-square distribution with $n-1$ degrees of freedom, and therefore, $(\overline{x} - \mu)/\sqrt{s^2/n}$ has a t distribution with $n-1$ degrees of freedom. Figure 3.9 shows a sample t density function: the probability of the random variable being less than $-t_{[1-\alpha/2;n-1]}$ is $\alpha/2$. Similarly, the probability of the random variable being more than $t_{[1-\alpha/2;n-1]}$ is $\alpha/2$. The probability that the variable lies between $-t_{[1-\alpha/2;n-1]}$ and $t_{[1-\alpha/2;n-1]}$ is $1-\alpha$.

New Data Summerizing Technique

We assume that n benchmarks are used for performance evaluation. The performance metric is the computation time[4]. The results of system A are represented by $a_1, \cdots, a_n$, the results of system B are represented by $b_1, \cdots, b_n$, and the ratio between these two systems are represented by $r_1, \cdots, r_n$[5]. Now we assume these results have already been measured . The next problem is how to summarize these results?

We propose two methods, *paired* and *unpaired*, to summarize these results here.

[4] In some other cases, for example, when memory hierarchy performance is evaluated, cache hit ratio, and other performance metrics will be used.

[5] In the following discussion, we will compare two systems only, however, those methods can be extended to for more systems easily.

Figure 3.9: $(\overline{x}-\mu)/\sqrt{(s^2/n)}$ follows a t(n-1) distribution

3.6.4 Paired Confidence Interval Method

In above discussion, if all n experiments were conducted on two systems such that there is a one-to-one correspondence between the i-th test on system A and the i-th test on system B, thus the observations are called **paired**. If there is no correspondence between two samples, the observations are called **unpaired**. The later case will be considered in the next subsection. This subsection consider paired case only.

When some absolute deviations between two systems are greater than zero, and others are less than zero, it is difficult to summarize them by ratio, which is similar to the case when the ratios between two systems are fluctuate around one. In this case, we may draw the conclusion that one system is better than other system with 100(1-α)% confidence level. For example, six similar workloads were used on two systems. The observations are {(5.4, 19.1), (16.6, 3.5), (0.6, 3.4), (1.4, 2.5), (1.4, 2.5), (0.6, 3.6), (7.3, 1.7)}. From these data, we get the absolute performance differences constitute a sample of six observations, {-13.7, 13.1, -2.8, -1.1, -3.0, 5.6}. For this sample, sample mean=-0.32, some researchers will conclude that system B is better than system A. However, in 90% confidence interval, there is no difference between these two systems. The deducing procedure are as follows.

Sample mean=-0.32
Sample variance =81.62
Sample standard deviation =9.03
Confidence interval for mean=$-0.32 \pm t\sqrt{(81.62/6)} = -0.32 \pm t3.69$

The 0.95-quantile of a t-variate with five degrees of freedom is 2.015: 90% confidence interval =(-7.75, 7,11). Since the confidence interval includes zero, these two systems are not different significantly. If we apply confidence interval method on ratios and find that one drops corresponding confidence interval, we will conclude that there is no difference between these two systems too, which will be shown in the example three of next section.

In fact, if the confidence interval locates below zero, we say the former system is better than the later, if the confidence interval locates above zero, we say the later system is better than the former, when the zero is included in the confidence interval, we say these two systems have no difference at this confidence level.

Unpaired Confidence Interval Method

Sometimes, we need to compare two systems using *unpaired* benchmarks. For example, when we want to compare the difference between PVM and TreadMarks, the applications used as benchmarks for these two systems may be different [83], , which will result in unpaired observations. How to make comparison in this case is a bit more complicated than that of the paired observations.

Suppose we have two samples of size n_a and n_b for alternatives A and B, respectively, The observations are unpaired in the sense that there is no one-to-one correspondence between the observations in these two systems. Then the steps to determine the confidence interval for the difference in mean performance requires making an estimate of the variance and effective number of degrees of freedom. The procedure is shown as follows:
1. Compute the sample means:

$$\overline{x_a} = \frac{1}{n_a}\sum_{i=1}^{n_a} x_{ia}, \quad \overline{x_b} = \frac{1}{n_b}\sum_{i=1}^{n_b} x_{ib},$$

we x_{ia} and x_{ib} are the i-th observations in system A and B respectively.
2: Compute the sample standard deviations:

$$s_a = \sqrt{\left(\frac{(\sum_{i-1}^{n_a} {x_{ia}}^2) - n_a\overline{x_a}^2}{n_a - 1} \right)},$$

$$s_b = \sqrt{\left(\frac{(\sum_{i-1}^{n_b} {x_{ib}}^2) - n_b\overline{x_b}^2}{n_b - 1} \right)}.$$

3: Compute the mean difference: $\overline{x_a} - \overline{x_b}$.
4: Compute the standard deviation of the mean difference:

$$s = \sqrt{\frac{{s_a}^2}{n_a} + \frac{{s_b}^2}{n_b}}.$$

5: Compute the effective number of degrees of freedom:

$$\nu = \frac{{(\frac{{s_a}^2}{n_a} + \frac{{s_b}^2}{n_a})}^2}{\frac{1}{n_a+1}{(\frac{{s_a}^2}{n_a})}^2 + \frac{1}{n_b+1}{(\frac{{s_b}^2}{n_b})}^2} - 2.$$

6: Compute the confidence interval for the mean difference:

$$(\overline{x_a} - \overline{x_b}) \pm t_{[1-\alpha/2;\nu]}s,$$

we, $t_{[1-\alpha/2;\nu]}$ is the $(1-\alpha/2)$-quantile of a t-variate with ν degrees of freedom.
7. If the confidence interval includes zero, the difference is not significant at $100(1-\alpha)\%$ confidence level. If the confidence interval does not include zero, then the sign of the mean difference indicates which system is better. This procedure is known as a ***t* test**.

With above paired summarizing method, we find that the conclusion that JIAJIA is better than CVM has 80% confidence level. More details about this technique can be found in [110].

Table 3.3: Execution Time, Fixed Speedup(S_f) and Memory Requirement for Different Scales

Scale	Seq. time(s)	4 way /speedup	8 way /speedup	Memory(MB)
60 × 30 × 208	6963.08	3410.09/2.09	3600.00/1.93	20
120 × 60 × 416	44710.66	19200.00/2.33	10920.01/4.09	160
240 × 60 × 832	——	614760.00/1.00	45744.02/13.38	660

3.6.5 Real World Application: Em3d

In addition the representative benchmarks, we port one important real world computational electromagnetics application (Em3d) to evaluate JIAJIA system. Em3d is the parallel implementation of FDTD (Finite Difference Time Domain) algorithm to compute the resonant frequency of a waveguide loaded cavity[107]. This program is a productive program of Institute of Electronics, Chinese Academy of Sciences. The three electronic field components, three magnetic field components, and eight coefficient components are allocated in shared space. The electronic and magnetic field components are updated alternatively in each iteration. Barriers are used for synchronization. 6000 iteration steps are required to reach the stable state. The principle of FDTD is out of the scope of this thesis. Here, we describe the numerical results.

Firstly the fixed speedup S_f is used to evaluate our implementation, which is a measure of the speedup of a fixed size problem as the number of processors is increased. Then the scaled speedup S_s, which is the measure of the speedup of a fixed load per processor, is used for further evaluation of our implementation. The definition of these two speedups are $S_f = T_1/T_n, S_S = n \times T_1/T_n$ respectively, where T_1 is the time of single processor running, T_n is the time of n processors.

For simplicity, we compute the same problem in three different scales in terms of gridding space: small (60 × 30 × 208), middle (120 × 60 × 416) and large (240 × 60 × 832). Table 3.3 shows the running time (for 6000 iteration steps), fixed speedup and memory requirement for three scales respectively.

Keep the computation load in each processor be fixed (120 × 60 × 208), we measure the scaled speedup. The results are shown in Table 3.4 together with the running time (also for 6000 iteration steps).

From Table 3.3 and Table 3.4, it can be concluded that:

1. For small-scale problem, 2.09 of fixed speedup is achieved in 4-way parallelism. The degradation of fixed speedup with 8-way parallelism is due to the low computation to communication ratio, which effects the parallel efficiency greatly. When the scale of the problem is increased, the speedup is increased too. For the middle scale problem, 2.33 speedup on 4-processors is achieved, which is greater than reported in [93], and slightly less than reported in [17]. Since the speedup is closely related to the problem space, we conclude that our speedup is comparable to that obtained in PVM. We ascribe the main reason to the flexible memory allocation scheme supplied by JIAJIA system, which exploits the potential of memory locality greatly.

2. As we stated in Section 3, the JIAJIA system can supply large memory space, which is helpful for solving large scale problems. In Table 3.3, for the largest scale 240 × 60 × 832 problem, 660 MB memory space is required which prevents the successful running of sequential code on single processor. Therefore, we measure the results on 4-way and 8-way parallelism only. The figure in the table shows that the speedup from 4 to 8

Table 3.4: Execution Time, Scaled Speedup(S_s) for Problem Scale 120 × 60 × 208

Processors	1	2	4	8
Time(Sec.)	21661.80	23754.60	24384.00	28650.03
Scaled speedup	1.00	1.82	3.55	6.05

processors is superlinear (13.38 compared to ideal value 2). We ascribe the main reason to the large memory space required by this scale problem. When 4 processors are used, nearly 220 MB (86% of total main memory) memory space are allocated on each node, causing data "ping-pong" frequently between main memory and hard disk, which will affect its performance greatly. However, only 43% of main memory space is needed when 8 processors are used.

3. Comparison between Table 3.3 and Table 3.4 shows that scaled speedup is more desirable for evaluating the parallel performance. Though the parallelizing effort is reduced greatly compared with other work, our results are comparable to those obtained in [133], which is implemented in the PVM environment.

3.6.6 Scalability of JIAJIA

In this subsection, the scalability of the JIAJIA system is evaluated and analyzed. Figure 3.10 lists the speedup of the 8 applications under 2, 4, 8, 16 processors respectively [6]. It can be seen that the performance of JIAJIA system is scalable for 5 of 8 applications, such as Water, TSP, Barnes, Mat, and Em3d. For other three applications, i.e., LU, SOR, and IS, we ascribe the decrease of speedups to two culprits: (1). Small problem size. Though the communication traffic of these three applications has been reduced to minimum, communication dominates the large part of the execution time compared with the small computation time. (2). Long synchronization overhead. From Figure 3.10(b), we find that the synchronization overhead dominates the whole system overhead for these three applications. Thus, the more processors are used, the larger the synchronization overhead is. Based on the assumption that the large system should be used to solve large scale problem, i.e., Gustafson's law, we believe that the JIAJIA system has the desirable scalability when larger scale problems are used.

3.7 Summary

In this Chapter, the design and implementation of JIAJIA are described. Performance evaluation and analysis of JIAJIA, in comparison with CVM, is also presented. With its NUMA-like memory organization scheme, JIAJIA can combine memories of multiple processor to form a large shared memory. JIAJIA achieves better performance than CVM for most tested applications, which benefits greatly from the simplicity of the lock-based cache coherence protocol that entails little system overheads on ordinary access miss. Other factors, such as the home-based memory organization which requires no diffs generating for home pages, the uniformed local

[6] All these results are tested on Dawning-2000I parallel cluster system, which is similar to Dawning 1000A system but has faster CPU and more system software. As such, the sequential time required by Em3d increases from 62.36 seconds to 484.03 seconds on Dawning 2000I due to the frequent ping pong between main memory and disk. Therefore, the speedup of Em3d is computed based on the two processor execution time.

(a) (b)

Figure 3.10: Speedups of 8 applications under 2, 4, 8, 16 processors.

and global addresses mapping of JIAJIA, and the flexible API which allow the programmer to control initial distribution of shared data, also help to improve performance. The lazy release protocol of CVM has less overhead than the lock-based coherence protocol of JIAJIA on release operation. Besides, JIAJIA needs to take a full page from home on a page fault, while CVM only takes diffs of the fault page and helps to reduces communication amount.

Chapter 4

System Overhead Analysis and Reducing

4.1 Introduction

In general, *communication overhead* and *coherence-induced overhead* are two main culprits for performance loss in software DSM systems. The communication cost in networks of workstation environment is obviously high, and large unit of coherence in software DSM systems results in the false sharing problem, encoding and decoding diffs in multiple writer protocol are time-intensive, and virtual page protection and segment violation interrupt introduce much system overhead. Many new techniques have been proposed to reduce the above two overheads in the past, such as relaxed memory consistency models [70, 11, 58] to reduce the frequency of communication, the multiple writer protocol to reduce the effect of false sharing[19], and the help of hardware [14], etc. The performance of recent software DSM systems made great progress in comparison with early systems[78].

However, according to some recent performance evaluation results[83], there remains a long way to the availability of software DSM systems. Where should we put our strength on in the future? This question is so important that is should be answered in time. However, up to now, there is no clear answer. In this chapter, we try to take the challenge to answer this question clearly. First, we analyze the different components of whole execution time within a software DSM system in detail. We measure and analyze the performance of several widely used benchmarks, and draw five important conclusions. Finally, we propose several techniques to optimize the system overhead, evaluation results show good achievements by these optimizations.

The rest of this chapter is organized as follows. We begin in Section 2 by describing a detailed analysis of software DSM system overhead. Next, in Section 3, we analyze the system overhead of the target system JIAJIA. In Section 4, based on the performance measurement results, we analyze and draw some important conclusions. Several optimization techniques are proposed and evaluated in Section 5.

4.2 Analysis of software DSM System Overhead

In order to depict the system overhead of software DSM system clearly, we first give a general prototype of software DSM system as shown in Fig 4.1(a). The single address space supported by software DSM system is shared by a number of processors. From the viewpoint of programmers, any processor can access any memory location in the address space directly. Each

(a) (b)

Figure 4.1: (a). General prototype of software DSM system, (b). Basic communication framework of JIAJIA.

processor has a *DSM layer*, which is responsible for the mapping between local memory and shared virtual memory address space, and keeping the address space *coherent* at all time. Application programmers can use the distributed shared memory just as they do on a traditional shared memory machine .

Generally, three states, RO, RW,and INV, are used to maintain the coherence among multiple copies of a memory location residing on different machines, and coherence is maintained at the page level. When an access miss locally, the operating system will deliver a segment violation (SIGSEGV) signal, and the DSM layer then takes over and processes this exception appropriately(such as issuing getpage request, memory protect system call, etc.) according to the original state of the faulting page and the type of fault(read fault or write fault). In comparison with sequential programs, all these operations(including interrupt, getpage request, system call, etc.) will add overheads to ordinary shared memory references. We use T_{data} to depict the time spent in the segment violation handler and T_{segv} to represent the cost of entering/leaving the segment violation handler respectively. The sum of T_{data} and T_{segv} comprises the data miss penalty. Furthermore, a remote interrupt for getting pages will occur in the segment violation handler, which can be divided into two kinds according to their occurrence time. When the remote interrupt appears when the CPU is waiting for something to complete, we use $T_{overlapint}$ to represent it, otherwise $T_{segvint}$ is used. The relationship between these time variables is shown in Figure 4.2(a).

Currently, *lock, barrier, and conditional variable* are the three most commonly used synchronization mechanisms in software DSM systems. As the development of relaxed memory consistency models and the lazy implementation of cache coherence protocol, the overhead associated with the synchronization operation increased greatly. For example, at the acquire operation in LRC, the requesting processor records the write notices generated in the last interval first, then sends them to the lock manager and waits until the last releaser grant the lock. Of course, the operations related to the synchronization operation depend greatly on which cache coherence protocol is used. In the following discussion, we use T_{syn} to represent all the synchronization time, including acquire time, release time, and barrier time. However, the time of entering and leaving synchronization function calls is neglected in our model. Again, remote interrupts may occur during the time of synchronization. We use T_{synint} and $T_{overlapint}$ to represent the interrupt time occurred in non-waiting and waiting time respectively. Figure 4.2(b) shows the relationship among these time variables.

In addition to data miss penalty and synchronization time, interrupt service time T_{server} is

(a) (b)

Figure 4.2: Time partition of SIGSEGV handler and synchronization operation.

neglectable in software DSM systems. However, as we know, the interrupt service time may overlap with segment violation handler time and synchronization time. In order to count this time accurately, we divide it into three parts: $T_{userint}$, T_{synint} and $T_{segvint}$, where T_{synint} and $T_{segvint}$ are defined as above. $T_{userint}$ represents the interrupt server time in the computation phase. Although $T_{overlapint}$ belongs to interrupt server time too, we do not include it in server time since it is overlapped with waiting time thoroughly and has been counted in other variables.

Moreover, the twinning and diffing mechanism are used to detect the write operation. We count the time spent on diff-related operations as T_{diff} to get more insight in the overhead of the whole system. Based on the LogP model proposed by David Culler et. al. [25], we measure the software overhead(O) and hardware latency(L) of communication overhead in detail. In most interrupt-based software DSM systems[1], communication software overhead includes two aspects: sender overhead (O_s) and receiver overhead(O_r). Sender overhead includes the overhead of sending messages and the overhead of receiving acknowledgements. Receiver overhead includes the overhead of I/O interrupt, the overhead of receiving data, and the overhead of sending acknowledgements, as shown in Figure 4.1(b). Assume that $O_s = \alpha_s + \beta_s \times l, O_r = \alpha_r + \beta_r \times l, L = \alpha + \beta \times l$, where l is the length(in number of bytes) of the message, the communication software overhead $T_{commsoft}$ and communication hardware overhead $T_{commhard}$ are computed as follows:

$$T_{commsoft} = N_s \times \alpha_s + \beta_s \times L_s + N_r \times \alpha_r + \beta_r \times L_r, \tag{4.1}$$

$$T_{commhard} = N_s \times \alpha + \beta \times L_s, \tag{4.2}$$

where N_s and N_r represent the total number of messages sent and received by one processor respectively, L_s and L_r represent the total length(in number of bytes) of messages sent and received by one processor respectively. These two time variables will help us to gain insight about the communication overhead, and to find the bottleneck of communication.

Finally, there are some other overheads that we can not measure directly, such as those resulting from TLB miss and first-level cache miss, cache pollution time, and function invocation time. We represent this time with T_{other} in our analysis.

[1] There are some software DSM systems adopt polling mechanism to process messages, which requires special hardware support, such as memory channel[120, 99]. However, interrupt remains the most common way adopted in software DSM systems to receive messages. We restrict our attention to interrupt-based systems only in this chapter.

Table 4.1: Description of Time Statistical Variables

Time Items	Descriptions
T_{busy}	useful computation time
T_{syn}	synchronization time
T_{data}	data miss penalty time excluding T_{segv}
T_{segv}	the time spend on entering and leaving the segment violation handler
T_{server}	interrupt server time
T_{other}	other miscellaneous time
$T_{userint}$	interrupt time spent in computation stage
T_{synint}	interrupt time spent in synchronization stage
$T_{segvint}$	interrupt time spent in segment violation handler
$T_{overlapint}$	interrupt time overlapped with CPU wait time
T_{diff}	encode and decode diff time
$T_{commsoft}$	software overhead of communication
$T_{commhard}$	hardware latency of communication

Theoretically, the relationship between total execution time T_{total} and the above time components should satisfy the following equation,

$$T_{total} = T_{busy} + T_{data} + T_{segv} + T_{syn} + T_{server} + T_{other}, \tag{4.3}$$

where T_{busy} represents the useful computation time, which can be briefly computed by dividing the sequential time by the number of processors. A general definition of these time variables is listed in Table 4.1.

4.3 Performance Measurement and Analysis

4.3.1 Experiment Platform

The evaluation is mainly done on the Dawning-1000A parallel system developed by the National Center for Intelligent Computing Systems. The system we used has 8 nodes each with a 200 MHZ PowerPC 604 processor and 256MB local memory. These nodes are interconnected through a 100Mbps switched Ethernet. In the testing, all libraries and applications (except a FORTRAN application) are compiled by `gcc` with the `-O2` optimization option.The basic performance characteristics of JIAJIA in this environment are as follows. The round trip latency for a 1-byte message using the UDP/IP protocol is 279.68 microseconds on this platform. The time to acquire a lock is 2224.34 microseconds. The time for an eight processor barrier is 6780.32 microseconds. The time to obtain a page from the home node is 1002.76 microseconds.

In order to get the parameters described in Section 2, we add some statistics in our run time library. For $\alpha, \alpha_s, \alpha_r, \beta, \beta_s, \beta_r$, we adopt the measure method similar to that of [25], and obtain the linear equation by regression with 95% confidence. According to the definition in Section 4.2, the communication parameters are as follows: sender software overhead O_s is $37.11 + 0.02 \times l$, receiver software overhead O_r is $124.90 + 0.02 \times l$, hardware latency L is $184.63 + 0.12 \times l$, where l is the length of the message.

Table 4.2: Characteristics of Applications

Apps.	**Size**	**Memory**	**Seq. Time(secs.)**	**Syn.**
SOR	2048×2048, 10 iter.	16MB	6.99	barr.
LU	2048×2048	32MB	71.11	barr.
Water	1728 mol, 5 iter.	384KB	180.63	lock, barr.
TSP	19 cities	788KB	260.70	lock
Barnes	16384 bodies	1638KB	372.58	lock, barr.
Mat	1024×1024	12MB	562.20	barr.
Em3d	120x40x416, 10 iter.	160MB	62.36	barr.
IS	2^{24}	4KB	31.29	lock, barr.

(a) (b)

Figure 4.3: (a). Speedups of applications on 8 processors, (b). Time statistics of applications.

4.3.2 Overview of Applications

Our test suite includes 8 applications, covering a broad range of problem domains with varying behaviours. We show the characteristics of eight applications as listed in Table 4.2. *Memory* item in Table 4.2 represents the memory space required for the specified problem size. *Seq. Time* is the execution time of the sequential program, and *Syn.* represents the synchronization mechanism used in these applications. More details about these applications can be found in last chapter.

4.3.3 Analysis

For eight applications used in our experiment, JIAJIA achieves moderate to high speedup as shown in Figure 4.3(a), ranging from 2.91(LU) to 6.96(TSP). According to the parallel efficiency, the 8 applications are classified into three categories. The first group includes Water TSP, IS, and Barnes, which obtain about 80% parallel efficiency. We ascribe the main reason to the lock-based cache coherence protocol used in JIAJIA. In fact, JIAJIA performs better than TreadMarks for these kind of applications on other hardware platforms, such as IBM SP2 [35]. Due to the moderate computation to communication ratio of the small scale of SOR and MAT, these two applications achieve about 70% parallel efficiency. When the problem size is increased, the speedup of the second group of applications will improve too. Finally, LU and Em3d belong to the third category, which get less than 50% efficiency. The main reason is the inherent load

Table 4.3: Breakdown of Execution Time These Applications

Apps.	T_{total}	T_{data}	T_{segv}	T_{syn}	T_{server}	$T_{overlap}$	$T_{commsoft}$	$T_{commhard}$	T_{diff}
SOR	1.20	0.15	0.00	0.17	0.07	0.07	0.03	0.05	0.00
LU	24.43	6.22	0.07	6.00	3.33	1.29	1.02	2.10	0.00
Water	26.54	1.08	0.02	2.59	0.56	0.41	0.28	0.48	0.15
TSP	37.33	0.89	0.09	2.96	0.89	0.27	0.27	0.49	0.12
Barnes	59.43	6.14	0.10	5.20	1.39	2.10	0.80	1.66	0.72
Mat	96.59	23.51	0.09	0.07	2.54	2.24	1.10	2.18	0.00
Em3d	17.60	2.16	0.03	4.91	3.08	0.63	0.36	0.74	0.10
IS	4.95	0.13	0.00	0.90	0.04	0.06	0.03	0.05	0.01

imbalance of these applications which results in the dilation of synchronization operations. Figure 4.3(b) presents a detailed view of the run time breakdown of these applications under the JIAJIA system on 8 processors. The bars in the figure show normalized execution time of T_{server}, T_{data}, T_{segv}, T_{syn}, and T_{busy}[2]. Overlapped time, communication and diff-related overhead are also listed in Table 4.3. Based on the experimental results shown in Figure 4.3(b) and Table 4.3, we draw conclusions as follows.

1. The system overhead possesses 12.94%(TSP) to 63.96% (LU) total time for these applications. Although there are about 29.97% system overhead on the average, the JIAJIA system is better than TreadMarks, which has about 33% system overhead[14].

2. Interrupt service time overlaps with CPU waiting time significantly. For all eight applications, interrupt service time takes 2.02%(IS) to 21.08%(Em3d) of the total execution time, however, almost 40.94% interrupt service time is overlapped with CPU waiting time on the average. For example, interrupt service time takes 18.91% and 11.67% time in LU and SOR, 27.92% and 50.00% time are overlapped. Therefore, using special hardware support to process messages is not worthwhile with current CPU and network technologies since the main CPU is idle when the message arrives. However, finding an efficient communication mechanism to replace the interrupt or polling is worth studying, and this has been a direction for future research on software DSM[15, 57].

3. Data miss penalty contributes 11.75% of the whole execution time on average. In other words, 39.17% system overhead is spent on data miss penalty. So finding an efficient way to reduce the data miss penalty is the most important thing. Although remote page fetching costs much time, we find the time spent on entering/leaving SIGSEGV handlers, i.e., T_{segv}, is unneglectable too. For example, 9.46% time of SOR is spent on this operation when the virtual memory detection scheme is used. With the cache only write detection scheme, T_{segv} is reduced to 0%.

4. Encoding and decoding diffs do not take so much time as we may imagine. For all eight applications, T_{diff} is lower than 1% of total time except Barnes(1.2%). Home-based memory organization contributes much to this, since the write operation in the home node does not incur any diff operation[54]. On the other hand, the shared data is uniformly distributed among the processors, which favors some applications greatly. For example, in SOR and LU, there are no diffs to be generated at all. In order to have

[2] In our test, all the T_{other}'s are so small that they are ignored here.

a clear understanding about the diff-related time, we allocate all shared data on host 0 intentionally. Though the number of diffs increases greatly, the synchronization time increases too, so does the total execution time, which results in that T_{diff} remains less than 1% of total time. Therefore, using hardware support to encode and decode diff is not a good idea.

5. Synchronization takes considerable time in software DSM system. In our eight applications, on average T_{syn} takes about 13.65% total execution time. Especially, for LU, Em3d and IS, it costs 24.56%, 27.90%, and 18.18% respectively. We ascribe this overhead to the following two reasons. One is the relaxed memory consistency model which is widely adopted in software DSM system to solve the false sharing problem, since almost all the coherence related operations are completed at synchronization points in relaxed memory consistency models. The other is load imbalancing in some applications which leads to dilation of some synchronization operations. Therefore, improving the implementation of synchronization in software DSM system is most important. It includes three aspects:(1) finding fast synchronization algorithm, (2) reducing the operations associated with synchronization points, (3) optimizing the implementation of the synchronization operations.

6. The time spent on communication is unneglectable either, ranging from 1.62% (IS) to 12.77% (LU). On average, 4.63% time is spent on communication. If we intent to distribute the shared data non-uniformly, the communication time will increase greatly. T_{comm} increases to 34% of total time when all shared data are allocated in host 0 in the LU applications. Therefore, much energy should be focused on hiding or reducing communication overhead. Prefetching and multithreading are two common techniques to hide(or tolerate) communication latency. However, published results[87] [128] show that the effectiveness of these two methods depends greatly on the memory reference patterns of applications. Therefore, reducing the communication overhead radically plays an important role in future design. Generally, three methods can be employed:(1) reducing the frequency of communication, (2) reducing the communication traffic, and (3) bypassing the operating system and using user-level communication to support software DSM system. Many relaxed memory consistency models and coherence protocols are proposed to improve the former two aspects[70, 11, 58], while the user-level communication without hardware support and operating system modification and specific for software DSM system is neglected, although there are many research groups endeavouring to design user-level communication for general distributed system, such as U-Net[32], FM[88]. SHRIMP in Princeton [17] is claimed to support software DSM system, however, it requires modifying both the network interface and operating system.

   However, we find an important fact in our test. Communication software overhead ($T_{commsoft}$) does not occupy significant part of the the whole communication time as presented in the literature. For all of the eight applications, the ratio between $T_{commsoft}$ and $T_{commhard}$ is about 1 : 2, i.e., the software overhead of communication occupies only about 1/3 of whole communication overhead. Therefore, we conclude that in the context of our testbed, communication bandwidth, not communication latency, is the main bottleneck of the current test configuration.

4.3.4 The CPU Effect

Many overheads discussed in the last subsection are affected by the CPU speed except those related to communication hardware latency. Naturally, a simple method to reduce this part of overhead is using an aggressive CPU. In order to evaluate the effect of CPU performance to overhead of software DSM systems, we retest all the applications with a slower CPU, and compare the results with last subsection. Here, we obtain the 1/2 speed CPU by adding an artificial load to every node of the system. Although most overheads will be lengthed in our test as expected, some overheads which require extremely short time interval will not be affected since this overhead lie in one piece of time slice of the OS. Fortunately, the time slice of OS is so tiny that this negative effect can be neglected when the problem size is large enough.

Normally, the execution time of parallel applications are divided into two parts [45]: T_{comp} and T_{comm} in distributed memory systems, and the corresponding speedup can be calculated as follows:

$$Speedup = \frac{T_{seq}}{T_{comp} + T_{comm}}, \tag{4.4}$$

Based on the definition of Equation 4.4, the speedup of a parallel system will be decreased as the CPU speed increases since the T_{seq} and T_{comp} will be reduced, while T_{comm} remains unchanged. However, a contrary conclusion is obtained in our test, as shown in Figure 4.4(a), the speedup of all of 8 applications increases ranging from the lowest 9.78% for TSP to the largest 54.79% for LU, and on average improve about 21.51%. We find that the main reason is that the system overhead has been reduced furthermore. Thus the definition of speedup in Equation 4.4 does not reflect the system behaviour accurately between fast CPU and slow CPU. So we deduce a new formula to compute the speedup more accurately in software DSM systems

$$Speedup = \frac{T_{seq}}{T_{comp} + T_{overhead} + T_{commhard}}, \tag{4.5}$$

where $T_{overhead}$ includes the software overhead spent on coherence processing, sending and receiving messages, and waiting. The definition of $T_{commhard}$ is as aforementioned. In our test, when the CPU speed increases two times, $T_{overhead}$ reduces about 75% on average, i.e., the system overhead reduces about 4 folds. This is demonstrated in Figure 4.4(b). We ascribe the great reduction of system overhead to the reduction of contention and waiting time. When the CPU speed increases, the waiting time for each critical resource will be reduced (lock acquire and release time, barrier time, etc.). Here, Mat is a local simple matrix multiplication program with inner product algorithm. All arrays are allocated in shared memory space. In order to have a good data locality, arrays are distributed uniformly among all processors in the parallel version. In this program, only 3 barriers are used to synchronize each processor.

However, this conclusion is kept only when the network speed is higher with respect to CPU speed. Once the CPU speed increases to some extent, i.e., the system overhead is dominated by the network speed, then improving the CPU speed will do little to improve system performance.

Unlike with CPU speed, increasing the network bandwidth always helps to the performance of the system, which can be deduced from Equation 4.5. However, with respect to a fixed CPU speed, there is a fixed tradeoff point of network speed matching with it. The system performance will not be affected by the network speed after that tradeoff point. Currently, increasing the network bandwidth will do great help to JIAJIA system performance since $T_{commhard}$ occupies 66.67% of the whole communication time.

(a) (b)

Figure 4.4: (a). Comparison of speedups of fast CPU and slow CPU, (b). Effects of CPU speed to system overhead.

4.4 Reducing System Overhead

4.4.1 Reducing False Sharing

Though ScC reduces false sharing compared to LRC[59], ScC does not eliminate this problem thoroughly. In the lock-based cache coherence protocol of JIAJIA, coherence is maintained through requiring the lock releasing (or barrier arriving) processor to send write-notices generated in the associated critical section to the lock (or barrier) manager and the lock acquiring (or barrier leaving) processor to invalidate its locally cached data copies according to the write notices of the lock (or barrier) manager. To reduce false sharing, write notices sent to the lock acquiring (or barrier leaving) processor should be minimized. Write notices that will cause superfluous invalidation should not be included in the message from the lock releasing (or barrier arriving) processor to the lock (or barrier) manager, or from lock (or barrier) manager to the lock acquiring (or barrier leaving) processor. We take the following measures to reduce write notices sent to acquiring processor on a lock or barrier.

- The first improvement to the basic protocol is called the *read notice* technique. In the basic protocol, write notices produced in a critical section are sent to the lock on a release or barrier. However, if a page is modified only by its home node, and no other processors have read the page since the last barrier, then it is unnecessary to send the associated write notice to the lock on the arriving of the barrier. To do this, a *read notice* is recorded in the home of a page any time a remote get page request is received and served. For applications with good data distribution, different processors work on different parts of the data, this optimization can greatly reduce the amount of messages exchanged.

- The *incarnation number* method implemented in Midway and ScC is adopted to eliminate unnecessary invalidation on locks. With this method, each lock is associated with an incarnation number. Every time a lock is transferred, its incarnation number is incremented. Besides, each processor maintains a local incarnation number for each lock. When a write notice is recorded in the lock, the current incarnation number of the lock is recorded as well. On a lock acquire, the acquiring processor includes its incarnation number of the lock in the acquiring message. On a lock grant, the current incarnation number of the lock and those write notices which have an incarnation number larger than the incarnation number in the request is sent to the acquiring processor. The acquiring processor then sets its local incarnation number of the lock to the one received from the lock and invalidates cached pages according to write notices received.

4.4.2 Reducing Write Detection Overhead

The multiple writer protocol needs to detect writes to shared memory so that the protocol can be activated to correctly propagate the writes. Two write detection schemes are implemented in JIAJIA. One is the traditional virtual memory page fault write detection (VM-WD) scheme which identifies writes to shared pages through page faults. The other is a cache only write detection (CO-WD) scheme which does not write-protect home pages but invalidates all cached pages at the beginning of an interval.

With the VM-WD scheme, both home and cached shared pages are initially write-protected at the beginning of an interval, as is shown in Figure 2.3. A SIGSEGV signal is delivered when a processor first writes to a shared page in the interval.

- For a write fault on a cached page, a *twin* of the page is created and a write notice is recorded for this page in the SIGSEGV handler. Write protection on the shared page is then removed so that further writes to this page can occur without page faults. At the ending of the interval, a word-by-word comparison is performed between the written page and its twin to produce *diff* about this page. Write notices and *diff*s are then sent to the associated lock and page home respectively.

- If the SIGSEGV signal is caused by a write fault on a home page, then a write notice is recorded for this page and write protection on the shared page is removed. At the ending of the interval, write notices about home pages are sent to the associated lock.

The above VM-WD scheme detects writes through page faults and entails additional runtime overhead on the protocol. Our previous experiments show that, write-protecting home pages causes significant overheads for applications with large shared data set and good data distribution so that most writes hit in the home. The CO-WD scheme reduces the overhead of home pages write detection at the cost of some extra cache misses. In the CO-WD scheme, all cached shared pages are conservatively assumed to be obsolete and are invalidated at the beginning of an interval. No write detection about home pages is required in CO-WD because the purpose of detecting write notices of home pages is to maintain coherence through invalidating associated cached pages, and the CO-WD scheme has already invalidated all cached pages when starting an interval. Only writes to cached pages are detected to generate *diff*s which are sent to their home at the end of an interval. *Diff*s are generated through comparing the dirty page with its twin as in the VM-WD scheme.

The CO-WD scheme is expected to work well for applications with good data distribution(i.e., most writes hit in the home) and little cached pages.

4.4.3 Tree Structured Propagation of Barrier Messages

The barrier provides a convenient yet expensive way of inter-processor synchronization. Normally, there is a barrier manager to handle barrier requests. Each processor which arrives at a barrier sends a barrier request to the barrier manager. After receiving barrier requests from all processors, the barrier manager then sends a barrier acknowledgement to each processor. In the lock-based protocol of JIAJIA, the barrier manager is also responsible for combining write notices received from all processors and sending these write notices to each processor one-by-one. Processing barrier messages in the above sequential way makes the barrier manager a potential bottleneck of the system when the number of processors is large.

To alleviate the bottleneck problem of the barrier manager, JIAJIA provides a tree structured propagation of barrier messages. It maps all hosts into a binary tree, with the barrier manager in the tree root. When a processor arrives at a barrier, it does not send a barrier request directly to the barrier manager. Instead, it sends the barrier request to its parent host in the tree. After receiving barrier requests from both of its children, the parent host combines write notices received from its children and then delivers the request to its parent. All processors arrive at a barrier when the root host receives barrier requests from both of its children. Similarly, barrier acknowledgements are propagated from the root to each node of the tree in an inverse way.

The above tree structured propagation of barrier messages reduces barrier time from $O(N)$ to $O(log_2N)$ and is expected to be helpful in reducing barrier overheads in systems with large number of nodes connected by a switched network.

4.4.4 Performance Evaluation and Analysis

Both the hardware and software environment are the same as those in Section 4.3.

Applications

Here, we show the results of nine applications, include Water, Barnes, and LU from SPLASH and SPLASH2, EP and IS from NAS Parallel Benchmarks, SOR, TSP, and ILINK from Rice University, and Em3d. The brief introduction of all these applications except ILINK can be found in Chapter 3.6.1.

ILINK is the parallel implementation of a widely used linkage analysis program that locates specific disease genes on chromosomes. We use the ILINK of FASTLINK which is designed by Rice University [74, 30, 101]. The main data structure in ILINK is a pools of gene bank which contains the probability of each genotype for an individual. The program traverses family trees and visits each nuclear family. The computation either updates a parent's gene bank conditioned on the spouse and all children, or updates one child conditioned on both parents and all the other sibling. Barriers are used for synchronization.

For comparison, Table 4.4[3] shows characteristics and the sequential run time of the benchmarks. As indicated in the table, the 8192 × 8192 LU and SOR cannot be run on single machine due to memory size limitation and the corresponding sequential run times are estimated values.

Overall Performance of JIAJIA

Table 4.5 shows eight processor execution results of the original JIAJIA (JIA$_{base}$), JIAJIA with the write notice minimization improvement (JIA$_w$), JIA$_{w+c}$ which uses the cache-only write detection scheme on the basis of (JIA$_w$), and JIA$_{w+c+b}$ which is the tree structured barrier message propagation version of JIA$_{w+c}$. Eight-processor execution time, speedup, message

[3] The sequential time listed here is different from that of Table 4.2 because different problem scales are used in these tests.

Table 4.4: Characteristics of the Benchmarks

Appl.	**Size**	**Shared Memory**	**Barrier #**	**Lock #**	**Seq. Time**
Water	1728 mole.	484KB	35	65	178.00
Barnes	16384	1636KB	28	8	413.24
LU	2048×2048	32MB	128	0	84.86
LU	8192×8192	512MB	512	0	5464.80*
EP	2^{24}	4KB	1	1	49.69
IS	2^{24}	4KB	30	10	30.10
SOR	2048×2048	16MB	200	0	68.44
SOR	8192×8192	256MB	200	0	1235.76**
TSP	`-f20 -r15`	788KB	0	140	175.36
ILINK	`LGMD-1-2-3`	12MB	740	0	650.94
Em3d	$120 \times 60 \times 416$	160MB	20	0	65.20

$*$: Estimated as eight times of 4096×4096 LU sequential time (683.10 seconds), sequential time of 8192×8192 LU is not available due to memory size limitation.
$**$: Estimated as four times of 4096×4096 SOR sequential time (308.94 seconds), sequential time of 8192×8192 SOR is not available due to memory size limitation.

counts, message amounts, SIGSEGV signal counts, and remote get page request counts are shown in Table 4.5.

Figure 4.5 shows execution time breakdown of JIA, JIA$_w$, and JIA$_{w+c}$ for tested benchmarks except EP and IS. Computation time, server time, SIGSEGV time, synchronization time, and other overhead are present.

It can be seen from Table 4.5 that, for most tested applications, JIAJIA achieves satisfactory performance and speedup.

Both Water and Barnes are N-body problems are characterized with tight sharing. As has been stated, the lock-based cache coherence protocol of JIAJIA takes all coherence related actions in synchronization points and has the least message overhead in ordinary write and read operations. In Water and Barnes, the number of shared pages is small, and this small number of pages are referenced frequently by multiple processors. Hence, the overhead at synchronization point is not too high, and page faults caused by ordinary write and read operations introduce least messages in JIAJIA. As a result, compared to other software DSM systems, JIAJIA achieves an acceptable speedup in Water and Barnes.

In LU and SOR, matrices are initially distributed across processors in such a way such that each processor keeps in its home the data it processes. As a result, computation-to-communication ratios of both LU and SOR are $O(N)$ where N is the problem size. Hence, speedups of both LU and SOR are acceptable and scale with the problem size. Another reason for the acceptable speedups of LU ans SOR is that no *diff* production is required for home pages in home-based software DSMs. Frequent inter-processor synchronization contributes the main reason for the moderate speedup of LU and SOR in the 2048×2048 cases, because computation steps of LU and SOR are separated by barriers. Besides, in LU, only the updating trailing submatrix computation phase of each step is fully parallelized.

Both EP and IS have separate computation and communication phases, i.e., computation of EP and IS happen locally and involves no communication, communication happens only at the end of computation. EP achieves a speedup of 8 in eight processors because the communication and computation ratio of EP is low. In IS, the most time-consuming computation is for each

Table 4.5: Eight-way Parallel Execution Results

Appl.	**Eight-proc. Time**				**Eight-proc. Speedup**			
	JIA$_{base}$	JIA$_{w}$	JIA$_{w+c}$	JIA$_{w+c+b}$	JIA$_{base}$	JIA$_{w}$	JIA$_{w+c}$	JIA$_{w+c+b}$
Water	27.75	26.47	30.62	—	6.41	6.72	5.81	—
Barnes	71.89	68.53	70.90	—	5.75	6.03	5.83	—
LU2048	26.56	25.04	24.64	24.27	3.20	3.39	3.44	3.50
LU8192	964.32	909.39	878.95	876.23	5.67	6.01	6.22	6.24
EP	6.30	6.30	6.31	—	7.89	7.89	7.80	—
IS	4.87	4.84	4.85	—	6.18	6.22	6.21	—
SOR2048	33.52	21.97	11.45	11.16	2.04	3.12	5.98	6.13
SOR8192	486.21	265.64	166.20	163.18	2.54	4.65	7.44	7.58
TSP	45.99	33.25	47.65	—	3.81	5.27	3.69	—
ILINK	230.24	188.63	294.15	293.60	2.83	3.45	2.21	2.22
Em3d	49.84	25.00	18.74	—	1.31	2.61	3.48	—
	Message #				**Message Amt.(KB)**			
	JIA$_{base}$	JIA$_{w}$	JIA$_{w+c}$	JIA$_{w+c+b}$	JIA$_{base}$	JIA$_{w}$	JIA$_{w+c}$	JIA$_{w+c+b}$
Water	11918	10828	22000	—	19108	16850	40125	—
Barnes	37752	37018	54307	—	82099	80569	116461	—
LU2048	25992	25992	25992	25992	100406	99950	99830	99830
LU8192	386666	386664	386616	386616	1593850	1569946	1567908	1567908
EP	77	77	77	—	60	60	60	—
IS	1050	1050	1050	—	896	896	895	—
SOR2048	8413	8411	8412	8414	23225	11837	11763	11772
SOR8192	8413	8413	8413	8413	91997	46146	46135	46162
TSP	37062	20290	39375	—	59382	24265	64025	—
ILINK	177392	140164	249228	249228	632606	418714	928756	928756
Em3d	7602	4479	16499	—	19809	8282	33250	—
	SIGSEGV #				**Get Page #**			
	JIA$_{base}$	JIA$_{w}$	JIA$_{w+c}$	JIA$_{w+c+b}$	JIA$_{base}$	JIA$_{w}$	JIA$_{w+c}$	JIA$_{w+c+b}$
Water	5325	4892	10075	—	3385	2847	8430	—
Barnes	34508	34144	39442	—	18208	17844	26487	—
LU2048	32663	32663	12072	12072	12072	12072	12072	12072
LU8192	1108876	1108875	189696	189696	189721	189720	189696	189696
EP	22	22	21	—	14	14	14	—
IS	230	230	210	—	140	140	140	—
SOR2048	412000	412000	2800	2800	2800	2800	2800	2800
SOR8192	164080	164080	2800	2800	2800	2800	2800	2800
TSP	16530	8312	17194	—	14082	5682	15231	—
ILINK	95870	86758	117709	117709	73191	54577	109109	109109
Em3d	186071	184510	9830	—	3512	1690	7970	—

Figure 4.5: Breakdown of Execution Time

processor to count its local part of the keys, while summing the counting results in the local buckets up into the shared bucket constitutes the communication work. We keep the number of buckets at 1024 which makes the communication amount relatively small compared to the computation work of counting 2^{24} keys. As a result, a speedup of 6.2 is achieved. Because all processors have to sequentially enter the critical section to sum their counting results up, the speedup of IS is not linear even though the communication amount is little.

TSP is special in that all inter-processor synchronizations are taken through locks. In TSP, each processor frequently reads from and writes to the pool of tours and the priority queue, causing tight sharing of pages (a page can store 27 paths in TSP). As a result, when entering a critical section, a cached page is normally invalidated because it has been written by other processors. Again, the speedup of TSP is acceptable because the lock-based cache coherence protocol of JIAJIA has least overhead on ordinary read and write miss.

The speedup of ILINK is not so high compared to other software DSM systems such as TreadMarks. We ascribe the main reason to the relatively little problem scale in our evaluation (LGMD-1-2-3 vs. LGMD-2-10-7) and slow communication (100Mbps Ethernet) to computation (0.4GFLOPS single node performance) ratio in our environment. As is shown in Table 4.5, the communication amount of ILINK is tremendous. Besides, the difference of memory organization between JIAJIA and TreadMarks may also contribute to the results.

It can also be seen from Table 4.5 that LU-8192 and SOR-8192 which cannot run on single processor due to memory size limitation can be run with multiple processors because JIAJIA can combine memories of multiple processors to form a large shared memory.

Effect of Reducing False Sharing

Table 4.5 shows that, JIA_w outperforms JIA_{base} significantly ($18\% \sim 50\%$) in SOR, TSP, ILINK, and Em3d, slightly ($5\% \sim 6\%$) in Water, Barnes, and LU, while the difference between JIA_{base} and JIA_w is trivial in EP and IS.

In LU and SOR where each processor only writes to the part of data kept in its home, the *read notice* optimization technique is the main reason for the better performance of JIA_w than JIA_{base}. It can be seen from Table 4.5 that, in LU and SOR, JIA_w and JIA_{base} have the same number of messages, SIGSEGVs, and remote get page requests. However, JIA_w transfers less message amounts than JIA$_{base}$ because with the *read notice* technique, JIA$_w$ only sends write notices of boundary pages to the barrier, while JIA$_{base}$ sends write notices of all home pages to the barrier. Statistics in Figure 4.5 confirms the above analysis. As indicated in Figure 4.5, the synchronization overhead of JIA$_w$ is much less than that of JIA$_{base}$ in LU and SOR.

The incarnation number technique makes JIA$_w$ significantly outperforms JIA$_{base}$ in TSP. In JIAJIA, write notices in all locks are cleared only on a barrier which is not the synchronization method in TSP. As a result, the tight sharing pattern of TSP causes write notices of a lock to accumulate as more releases are performed, and some cached pages of a processor are unnecessarily invalidated on an acquire of the lock in JIA$_{base}$. In JIA$_w$, on the other hand, the incarnation number technique of JIA$_w$ helps to reduce unnecessary invalidations because only write notices with an incarnation number larger than the acquiring processor's local incarnation number of the lock are sent to the acquiring processor. It can be seen from Table 4.5 that, JIA$_w$ has much less remote get page requests than JIA$_{base}$ in TSP. As a result, JIA$_w$ has less SIGSEGV overheads than JIA$_{base}$, as indicated in Figure 4.5.

Statistics in Table 4.5 shows that remote get page requests and message amounts of JIA$_w$ are less than those of JIA$_{base}$ in ILINK and Em3d in which barrier is the mere synchronization method. The less number of remote accesses of JIA$_w$ than JIA$_{base}$ is caused mainly by the optimization which keeps a cached single-writer page valid on the writing processor on a leaving of barrier. Besides, with the *read notice* technology, JIA$_w$ sends much less write notices than JIA$_{base}$ on barriers, causing the synchronization time of JIA$_w$ to be much less than that of JIA$_{base}$ in ILINK and Em3d. Figure 4.5 shows that both SIGSEGV time and synchronization time of JIA$_w$ is much less than that of JIA$_{base}$ in ILINK and Em3d.

Comparison of Different Write Detection Schemes

It can been seen from Table 4.5 that, CO-WD outperform VM-WD in LU, SOR, and Em3d, while VM-WD outperforms CO-WD in Water, Barnes, TSP, and ILINK.

In LU and SOR, matrices are distributed across processors in a way that each processor only writes to its home part of the matrices in the computing. Since the computation of an iteration is synchronized with barriers and passing a barrier causes all shared pages to be write-protected in VM-WD, a page fault occurs for writing each home page in an iteration. The CO-WD scheme, on the other hand, does not write protect shared pages on a barrier, and writing to home pages of a processor can process smoothly without any intervention. Table 4.5 shows that the CO-WD scheme causes much less SIGSEGVs than the VM-WD scheme, while remote get page request numbers are the same for VM-WD and CO-WD in LU and SOR. Consequently, CO-WD has much less system overhead than VM-WD. It can be derived from Table 4.5 that JIA$_{w+c}$ requires 37% $\sim$ 48% less time than JIA$_w$ in SOR, and 2% $\sim$ 3% less time than JIA$_w$ in LU. Figure 4.5 shows that JIA$_{w+c}$ has much less SIGSEGV overhead than JIA$_w$ in LU and SOR.

In Water, Barnes, TSP and ILINK, the number of shared pages is not large and hence the advantage of CO-WD over VM-WD in keeping home pages writable on a synchronization point is not significant. On the other hand, since CO-WD invalidates all cached pages on an acquire, it causes much more remote page faults and consequently messages than VM-WD for applications with irregular memory reference pattern. As a result, both the SIGSEGV time and server time of JIA$_{w+c}$ is large than JIA$_w$ in these applications, as indicated in Figure 4.5.

In Em3d, CO-WD also introduces much more remote page faults than VM-WD. However, since Em3d requires 160MB shared memory, VM-WD causes much more SIGSEGVs than CO-

WD because it write-protects all home pages at the beginning of an iteration. Figure 4.5 shows that the overhead of additional virtual memory page faults in VM-WD is similar to that of the additional getting remote pages in CO-WD. CO-WD performs better than VM-WD because it has less other overheads in Em3d.

Effect of Tree Structured Barrier Message Propagation

Since the improvement of JIA$_{w+c+b}$ over JIA$_{w+c}$ is the tree structured propagation of barrier messages, JIA$_{w+c+b}$ is tested only for applications with more than 100 barriers.

As can be seen from Table 4.5, JIA$_{w+c+b}$ outperforms JIA$_{w+c}$ slightly (around 1% $\sim$ 3%) for all applications we tested. This slight improvement is acceptable because the advantage of JIA$_{w+c+b}$ over JIA$_{w+c}$ is not significant for eight processors. In our test of passing 1000 barriers, JIA$_{w+c+b}$ finishes in about 3 seconds while it takes JIA$_{w+c}$ about 6 seconds to finish. It is expected that the difference between JIA$_{w+c+b}$ and JIA$_{w+c}$ will be more significant when more processors are involved.

4.5 Summary

With JIAJIA as an example, this Chapter investigated the system overhead of software DSMs, and proposed several techniques to reduce system overhead. According to our evaluation, synchronization overhead and data miss penalty comprise the most significant system overhead and should be reduced in the next generation software DSMs. We also observed that interrupt servicing and diff encoding/decoding do not cost so much time as imagined, so using hardware support for interrupt handling and diff encoding is not worthwhile, which is contrary to the conclusion of [14].

Since the testing environment we used is similar to a common network of workstations environment, i.e., modern workstations interconnected with megabit switched ethernet, most of our conclusions apply in these situations. If the interconnection is upgraded to much faster media, e.g. Myrinet, the conclusion that communication hardware overhead is greater than communication software overhead will not hold, and improving communication software layers will continue be important. More details can be found in Chapter 7.

The effects of CPU speed on the performance, speedup, and system overhead are interesting. This Chapter extended the common speedup formula from distributed memory systems to software DSM systems, and pointed out that since a fast CPU reduces system overhead, fast CPUs can increase the speedup of applications even if the network speed is constant.

For the proposed optimization techniques, we have following conclusions:

- The methods of reducing false sharing through minimizing the number of write notices propagated between lock manager and lock releaser (acquirer) is effective for different kinds of applications.

- CO-WD works well for applications with good data distribution and most writes hit in home pages, such as LU and SOR. However, it is sensitive to data distribution and performs terribly bad when most shared memory references happen in the cache, as in TSP.

- Tree structured barrier message propagation performs slightly better than propagating barrier messages sequentially for all applications tested. It is expected to bring more advantages when more processors are involved.

Chapter 5

Affinity-based Self Scheduling

5.1 Background

In the previous chapters, we discussed the methods to improve the performance of software DSM systems from the viewpoint of coherence protocol and memory management. and all the performance evaluation were performed in dedicated environemnt. However, as the development of high speed networking technology and high performance microprocessors, network of workstation (NOW) becomes an important cost effective hardware platform for high performance computing. In the context of NOW, all the workstions might be shared with other users, i.e., meta-computing environment, therefore dynamic scheduling is required to achieved high performance and better resource utilization in this environement. In this and following chapters, we will discuss dynamic scheduling techniques.

Loop scheduling plays an important role in parallel and distributed systems in achieving good performance. This issue is even more critical in a metacomputing environment where the machines might be shared among many other users simultaneously. In order to maximize performance with given resources, the parallel system must optimally distribute the work according to not only the inherent computation and communication demands of the applications, but also dynamically available computing resources. Here, we discuss dynamic load balancing scheme, i.e., the run time system needs an appropriate way to change the amount of work assigned to each processor in order to efficiently utilize the computing resources. In many scientific applications, a set of independent tasks generally exist in a parallel loop called DOALL loop. Therefore, changing the number of the loop iterations performed by each processor can balance the load. We only restrict our attention to this kind of applications in this paper.

The performance of a loop scheduling scheme is mainly affected by three overhead resources: *loop allocation overhead*, *load imbalance*, and *remote data communication*. It is difficult to find an optimal algorithm to minimize three overheads simultaneously because of the conflicts among them. Load balance prefers the "fine grain" allocation of loop iterations to minimize the effects of uneven allocation. However, the "fine grain" allocation tends to increase the loop allocation overhead (including corresponding synchronization overhead). Exploiting processor affinity[135] can efficiently reduce the remote data communication overhead, which tends to cause load imbalance. In different applications, each of these three overheads affects the performance in different ways. Hence, an efficient loop scheduling algorithm should take loop allocation overhead, load imbalance, and remote data communication overhead into account integratedly. Moreover, the execution time of each iteration (parallel task) is unpredictable at loop scheduling.

Loop scheduling has been extensively studied in past years[124, 90, 31, 53, 131, 84, 81, 135].

However, all these previous work focused on either shared memory multiprocessors or dedicated hardware distributed shared memory systems, i.e., they assumed that the available computing power of each processor (or node) was equal, and neglected the practical computing power of underlying hardware. Therefore, results are not applicable under metacomputing environment. Take matrix multiplication as an example, the static scheduling scheme is demonstrated as the best among all the schemes by previous research. However, we will show that static scheduling is the worst scheme in metacomputing environment. Up to now , only one paper has addressed this issue in metacomputing environment for software DSM systems[60], which required compiler support to extract access pattern from the source code.

This chapter has two contributions: (1) As far as we concerned, this is the first time that these several well known different scheduling schemes are evaluated and compared in a software DSM system under dedicated and metacomputing environment respectively. We find that many previous theoretical analyses do not work any more. (2) We present and evaluate a new affinity-based self scheduling algorithm (ABS) for home-based software DSM systems. The unique characteristic of ABS scheme, the static affinity between processor and initial data distribution, is considered when scheduling.

Our target software DSM system is JIAJIA, which is a novel software DSM system based on a new lock-based cache coherence protocol. JIAJIA is the unique home-based software DSM system that implements home migration. We present that static processor affinity plays more important role in home-based software DSM systems than in COMA-like software DSM systems.

According to the evaluation results, we show that our ABS scheme outperforms static scheduling and other eight well known self scheduling algorithms in software DSM systems.

This rest of the chapter is organized as follows. Related work is listed in Section 2. In Section 3, the design and implementation of new affinity-based self scheduling algorithm are described. Based on five representative kernels, performance of different scheduling schemes are compared and evaluated in dedicated and metacomputing environment respectively in Section 4.

5.2 Related Work

Since loops are the richest source of parallelism in most applications, various loop scheduling schemes were proposed to exploit parallelism. Generally, scheduling schemes can be classified into *static scheduling* and *dynamic scheduling*. Static scheduling scheme assigns iterations to processors at compiler time while dynamic scheduling scheme assigns iterations at run time. Static scheduling scheme has the advantage of no loop allocation overhead. However, this advantage makes it unable to respond to an imbalanced workload or imbalanced computing power among the processors. Dynamic scheduling schemes are designed to alleviate this problem.

In the presence of variable length iteration execution time or variable computing power of processors, dynamic scheduling approaches are superior to static schemes. However, this comes at the cost of additional loop allocation overhead. The common approach of dynamic scheduling is the self scheduling technique which maintains a global queue of tasks. An idle processor fetchs some tasks from the list and executes it. In other words, the processors "self schedule" themselves at the run time. Well known dynamic scheduling algorithms include Self Scheduling(SS) [124], Block Self Scheduling (BSS), Guided Self Scheduling (GSS)[90], Factoring Scheduling (FS)[53], Trapezoid Self Scheduling(TSS)[131], Affinity Scheduling (AFS)[84], Safe

Self Scheduling (SSS)[81], Adaptive Affinity Scheduling (AAFS)[135], etc.. We show how to schedule N iterations of a parallel L on a p-processor cluster using these well known self scheduling schemes.

5.2.1 Static Scheduling (Static)

The static scheduling assigns each processor with $\lceil N/p \rceil$ consecutive iterations statically. Except when both the iteration execution times and the computing power of each processor are roughly the same, such an assignment of iterations to processors may result in load imbalance. Alternatively, the iterations can be assigned in round-robin fashion, i.e., iteration i is assigned to processor $i \bmod p$. This approach may produce a more balanced schedule only for some applications, such as LU factorization, where the execution time decreases with the loop index i. However, this approach will suffer a low cache hit ratio due to the way data is accessed.

5.2.2 Self Scheduling(SS)

The basic principle of self scheduling is that a processor fetches one iteration at a time during runtime when it becomes idle. SS always achieves a well balanced workload whether in dedicated or metacomputing environment. However, this well balanced load does not always yield a good performance because the amount of scheduling overhead is proportional to the number of iterations N. For fine grain applications, this amount of overhead could be large compared to the cost of computation. In addition, the high frequency of mutually exclusive access to central queue, may cause contention and seriously degrade performance. SS may be appropriate for loops with relatively few iterations but long length execution times compared to the loop allocation overhead.

5.2.3 Block Self Scheduling(BSS)

Block self scheduling reduces scheduling overhead by have each processor take K iterations instead of one each time. The algorithm amortizes the cost of each loop allocation overhead over the execution time of K iterations. When $K = 1$, BSS evolutes to SS. When $k = \lceil N/P \rceil$, it seems that this scheme is similar to static scheduling, however, there is great difference between them as we will point out later. The main drawback of BSS is its performance depends on the chunk size and the characteristics of each loop. Too large a chunk size may cause a load imbalance while too small a chunk size may increase the overhead and memory contention. Worse yet, it is difficult to find an appropriate chunk size for a application since the execution time does not monotonically increase or decrease with the chunk size.

5.2.4 Guided Self Scheduling(GSS)

Polychronopoulos and Kuck show that there cannot be an optimal value of K in BSS for even the simplest cases and propose guided self scheduling. In GSS, $1/p$ of unscheduled iterations is assigned to an idle processor. As such, large chunks of iteration are allocated at the beginning of a loop so as to reduce loop allocation overhead, while allocating small chunks towards the end of the loop to balance the workload. Assuming all loop iterations take the same amount of time to complete, GSS ensures that all processors finish within one iteration of each other and use the minimal number of synchronization operations[90]. However, we find that this conclusion does not be keep in undedicated environment. Since processors take only a small number of iterations from the work queue near the end of the scheduling process, GSS will suffer from excessive contention for the work queue and loop allocation overhead. If each iteration takes a short time to complete, then processors spend most of their time on competing to take iterations from the work queue, rather than executing iterations. Adaptive guided self-scheduling[31] addresses this problem by using a back off method to reduce the number of processors competing for iterations during periods of contention.

5.2.5 Factoring Scheduling (FS)

In some cases, GSS might assign too many work to the first few processors, so that the remaining iterations are not sufficiently to balance the workload. This situation arises when the initial iterations of a loop are much more time consuming than later iterations or the computing power of later allocated processors are better than first few allocated processors. The factoring scheduling proposed by Hummel et.al. addresses this problem. In FS, fixed size chunks of iterations are allocated to processors in batches (a set of p consecutive chunks forms a batch). With FS, $c_i = \lceil R_i/2p \rceil$ iterations are allocated to each of the p processors at each scheduling step i, where R_i is the number of iterations remaining at the beginning of step i. R_i is decremented at each scheduling step as $R_{i+1} = R_i - p \times c_i$. The chunk size of each batch is halved from that of the previous batch. The main focus of FS is to achieve a balanced workload with a little increase of loop allocation overhead. However, this target can not be achieved in metacomputing environment since the iterations $(p \times c_i)$in each batch will not be distributed evenly among the processors as expected. FS can be viewed as a generalization of GSS and BSS: GSS is factoring where each batch contains a single chunk, and BSS is factoring where there is a single batch.

5.2.6 Trapezoid Self Scheduling(TSS)

Tzen et. al. proposed the trapezoid self scheduling algorithm to improve GSS furthermore and eliminate the portion of chunk scheduling code that must be executed sequentially in a critical section. TSS(N_s, N_f) assigns the first N_s iterations to the processor starting the loop and the last N_f to the processor performing the last fetch, the number of iterations assigned to the processor between the first and the last fetch is then between N_s and N_f and linearly decreases based on some step d. Tzen proposed the TSS$(N/2p, 1)$ as a general selection of N_s and N_f, with $d = N/8p^2$. The difference in the size of successive chunks is always a constant in TSS, whereas it is a decreasing function in GSS and FS.

5.2.7 Affinity Scheduling(AFS)

Different with previous work, affinity scheduling introduced by Markatos et. al. demonstrates the benefit of processor affinity[1]. AFS uses a deterministic assignment policy to assign repeated executions of a loop iteration to the same processor, thereby ensuring most data accesses will be to the local memory. In contrast to most previous algorithms, AFS employs per-processor local work queues, which minimize the need for synchronization across processors. AFS includes three phases: (1) Initial phase, the N iterations of a parallel loop are divided into p chunks

[1] *Processor Affinity* means the relationship between processor and the data set it process. For example, if unit of work can be processed more efficiently on processor A than on processor B, due to facts such as the presence of required data in a local cache, the unit of work is said to have an “affinity” with processors A.

with $\lceil N/p \rceil$ iterations each, the ith such chunk is placed on the local work queue of processor i. (2) Local scheduling phase. An idle processor removes $1/k$, where k is suggested to be equal to p, of the iterations from its local work queue and execute them. (3) Remote scheduling phase. When a processor's work queue becomes empty, it finds "the most loaded processor", and removes $\lceil 1/p \rceil$ from that processor's work queue. In comparison with other central queue based algorithms, Markatos et. al. [84] show that AFS achieves the best performance in all tested cases. We ascribe the main reason to the low overhead of finding "the most loaded processor" in shared memory multiprocessors. However, in distributed systems, this operation requires at least $2p + 2$ messages and $p + 1$ synchronization operations, which is very time consuming and affect the final execution time greatly. Detailed performance evaluation will validate our viewpoint.

5.2.8 Safe Self Scheduling(SSS)

Take the variable length iteration execution time into account, SSS tends to assign each processor the largest number m of consecutive iterations having a cumulative execution time just exceeding the average processor workload E/p, i.e., $\sum_{i=s}^{s+m-1} e(i) < E/p \leq \sum_{i=s}^{s+m} e(i)$, where $E = \sum_{i=1}^{N} e(i)$ and s is some starting iteration number of the chore, $e(i)$ is the workload of iteration i, i.e., the execution time needed by iteration i. Here, s is the *smallest critical chore size* because adding any more iterations to this chore further will imbalance the schedule. In the implementation of SSS, the size of chores in the first batch is $\alpha \times N/p$, which is assigned statically. α is called the allocation factor, and determines the fraction of the unscheduled iterations allocated during each batch, and it is computed by $e(i)$. After a processor finishes the iterations assigned to it, the ith fetching processor is then assigned a chore of $(1-\alpha)^{\lceil i/p \rceil} \times N/p \times \alpha$. Since the value of α is related to the execution time of each iteration, it is difficult to predict. The suggested value for α is 0.90625 in [81]. In fact, the implementation of SSS is similar to that of FS greatly except that the value of α is different, as shown in Table 5.1.

5.2.9 Adaptive Affinity Scheduling(AAFS)

In the local scheduling phase of AFS, the processor fetch $1/p$ of the rest iterations in it's local work queue without any consideration of other processors. This scheme may not be efficient. For example, if the initial loop partition is not balanced, those lightly loaded processors should finish execution of the iterations in their local queue as soon as possible so that they can immediately turn to help heavily loaded processors. Hence, processors should be able to dynamically increase or decrease their local allocation granularity based on runtime information. Based on above observation, Yong et. al. proposed adaptive affinity scheduling algorithm (AAFS). In each local queue visiting, the average number of iterations executed by all processors, i.e., $\sum_{i=1}^{p} s_i/P$, is computed as the pivot to partition the workload states of processors into three types: *heavily loaded*, *lightly loaded*, and *normally loaded*. According the workload state, AAFS determines the chunk size of next fetch. [135] gave 5 different schemes to change the chunk size. Though the initial motivation of AAFS is reducing the synchronization and loop allocation overhead, however, the collection of other processors' state at each loop allocation will result in great synchronization overhead, especially in software DSM systems. Therefore, their conclusion need to be revisited.

Assume $N = 400$ and $p = 5$, Table 5.1 shows the chunk size for these scheduling schemes.

Table 5.1: Chunk Sizes of Different Scheduling Schemes

Scheme	Change of chunk size with time									
Static	80	-	-	-	-	-	-	-	-	-
SS	1	1	1	1	1	1	1	1	1	1
BSS(k=20)	20	20	20	20	20	20	20	20	20	20
GSS	80	64	51	41	33	26	21	17	13	11
FS	40	40	40	40	40	20	20	20	20	20
TSS	40	38	36	34	32	30	28	26	24	22
AFS	16	13	11	8	7	5	4	4	3	2
SSS	72	72	72	72	72	7	7	7	7	7
AAFS	16				variable					

All those dynamic scheduling algorithms discussed above fall into two distinct classes: *central queue based* and *distributed queue based*. In central queue based algorithms, such as SS, BSS, GSS, FS, TSS, SSS, iterations of a parallel loop are all stored in a shared central queue and each processor exclusively grabs some iterations from the central queue to execution. The major advantage of using a central queue is the possibility of optimally balancing the load. While keeping a good load balance, these central queue based algorithms differ in the way of reducing loop allocation overhead. However, three limitations are associated with the use of these traditional central queue method[2]:

- An iteration in the central queue is likely to be dynamically allocated to execute on any processor since the order of each processor visiting this central queue is random, which does not facilitate the use of processor affinity In distributed memory systems processor affinity is more important than load balancing.

- During allocation, all the processors but one should remotely access the central work queue, and thereby generate heavy network traffic;

- Because all the processors contend for the central queue, the central queue tends to be a performance bottleneck, which results in a longer synchronization delay.

The limitation 2 and 3 can be solved by using distributed queue methods as AFS and AAFS. Furthermore, since the local work queue of AFS and AAFS assign the same iterations statically for repeated execution loops, the data required by each processor will be cached or moved[3] to related processor so that the processor affinity can be used. However, the extra synchronization overhead associated with AFS and AAFS when load imbalance occurs are neglected in previous work. In distributed systems, this operation requires at least $2P + 2$ messages and $P + 1$ synchronization operations, where P is the number of processors. This operation requires much more time and will affect the final execution time greatly. On the other hand, the allocation overhead associated with each local allocation does not be reduced at all in AFS and AAFS because the local queue must be accessed exclusively too.

Moreover, the processor affinity exploited in AFS and AAFS is limited to applications only with following characteristics.

[2] In order to distinguish those central queue methods from our new ABS scheme, we use traditional central queue method to represent them.

[3] Only systems supporting data migration can achieve this.

1. The same parallel loop will be nested to a sequential loop, so the same data has to be repeatedly used by successive executive of an iteration (iteration based applications satisfy this condition), and

2. The cache (in software DSM this means software cache) is large enough to hold the data until it can be reused.

For many applications without these characteristics, the advantages of AFS and AAFS can not be exploited any more.

Markatos and Le Blanc analyzed the importance of load balancing and locality management in shared memory multiprocessors[85]. They concluded that locality management was more important than load balancing in thread assignment. They introduced a policy called *memory conscious scheduling* which assigned thread to processor whose local memory hold most of the data to be accessed by this thread. Their results showed that the looser the interconnection work, the more important the locality management. Our algorithm depends greatly on this rationale.

In fact, the most important difference between our work and previous work lies in evaluation method. All previous work was on shared memory multiprocessors or dedicated hardware distributed shared memory systems, while our work is implemented in a real home-based software DSM system on dedicated and metacomputing environment respectively.

5.3 Design and Implementation of ABS

5.3.1 Traget System

In order to support different scheduling schemes, JIAJIA provides an auxiliary call `jia_gettask(&begin, &end)`, which returns the value of `begin` and `end` to the caller according to the corresponding scheduling schemes. The basic framework of the application is shown in Figure 5.1.

```
void worker()
{
    while(1) {
        jia_gettask(&begin, &end);
        if (there is left work) {
            /* execute the work */
        } else {
            break;
        }
    }
}
```

Figure 5.1: Basic framework of application.

5.3.2 Affinity-based Self Scheduling Algorithm

From above, we know that we must take the processor affinity as the most important factor when designing a scheduling algorithm for distributed memory systems. Reducing the necessary synchronization overhead must be kept in mind too.

Fortunately, the concept of processor affinity can be extended much more in home-based software DSM systems. The data must be predistributed in a home-based software DSM system, so that the affinity between each iteration and it's corresponding data can be determined statically, i.e., the initial data distribution should be taken into account in the scheduling algorithm.

The basic idea of our affinity-based self scheduling (ABS) is to distribute the loop iterations to the processor that already has corresponding data while minimizing the loop allocation overhead (i.e., synchronization overhead to access exclusive shared variables). However, in traditional central queue based scheduling algorithms, the initial data distribution is separated from the scheduling algorithm so that the affinity between the initial data distribution and the following computing can not be exploited since the order of each processor visiting the central queue is not deterministic. For example, assume the first and second chunks of matrix A are allocated on processor 0 and 1 respectively. Processor 1 obtains the first chunk, while processor 0 gets the second chunk in the computing phase. So both of them need fetch the data remotely.

ABS includes three steps: *static allocation*, *local dynamic scheduling*, and *remote dynamic scheduling*. We implement a globally shared central queue, which is partitioned into $\lceil p \rceil$ segments and the ith is assigned to processor i statically according to the initial data distribution. At local dynamic scheduling phase, BSS or GSS scheduling algorithms can be employed. When load imbalance occurs, the idle processor obtains the task from the most heavily loaded processor. It seems that there is no much difference between ABS and AFS at the later scheduling phases. However, the use of central shared queue, the lazy memory consistency model adopted by software DSM system, and the large granularity inherent in the software DSM system contributes greatly to our ABS scheduling algorithm. The large grain (generally one page is larger or equal to 4K bytes) causes the whole shared queue to be allocated in one page. Therefore there is no difference between local scheduling and remote scheduling, both of them require only one synchronization operation and one message. Table 5.2 compares the number of required messages and synchronization operations[4] of ABS, AFS and AAFS. From the table, we conclude that when load imbalance occurs, more messages and synchronizations are required in AFS and AAFS to gather other processors' load information. We can also find that the synchronization overhead associated with the loop allocation of AAFS is worse than that of AFS, which will be demonstrated by the performance evaluation results in the next section.

Table 5.2: # of Messages and Synchronization Operations Associated with Loop Allocation

Scheme	Local scheduling		Remote scheduling	
	# of message	sync.	# of message	sync.
AFS	0	1	2p+2	p+1
AAFS	2p	p+1	2p+2	p+1
ABS	0 or 2	1	4	2

The main difference between ABS algorithm and traditional central queue based scheduling

[4] Here, the entering and departure of one critical section are treated as one synchronization operation.

Table 5.3: Description of the Symbols

Symbol	Description	Note
p	the number of processors in the system	
N	the total number of iterations in a parallel loop	
c_i	the computation of the ith iteration	
p_i	the practical available computing power of processor i	
r	the ratio of the execution time on local processor and remote processor	
S	the loop allocation overhead	
K	the chunk size	used in BSS, ABS
k	the control variable	used in GSS,AFS,AAFS
R_i	the rest iterations in central queue	used in GSS
r_o	the largest rest iterations in other's queue	used in AFS,AAFS
r_i	the rest iterations in local queue	used in AFS, AAFS,ABS
N_s	the initial chunk size	used in TSS
δ	the decrement value	used in TSS
α	allocation factor	used in FS, SSS

schemes lies in the organization and management of the central queue. In our scheme, the queue is partitioned among these processors, and each processor first accesses it's own part. During the local scheduling phase, each processor obtains task from it's initially allocated segment. When load imbalance occurs, the idle processor obtains the task from the segment which belongs to the most heavily loaded processor.

With ABS, the processor affinity of the above example can be utilized as follows. Processor 0 gets the first chunk even if it visits the global queue later than processor 1.

5.4 Analytic Evaluation

Before evaluating these scheduling algorithms on experimental platform, a simple but efficient analytic model is given in this section.

As we described in Section 5.1, loop allocation overhead, load imbalance, and remote data communication must be considered in a scheduling scheme. The hardware environment considered in this paper is homogeneous uniprocessor or SMP-based cluster. Assume there is p processors in the whole system. Though the computing power of each processor is equal in dedicated case, the practical computing power of each processor varies according the practical loads on that processor in metacomputing environment. Thus, the practical available computing power of the ith processor is represented by p_i. Synchronization operation must be employed in loop allocation phase. We use S to represent this overhead. From the viewpoint of the application, there is inherent load imbalance among iterations. The computation of the ith iteration is denoted by c_i. The ratio of the execution time of a iteration on local processor and remote processor is assumed to be r. The value of r in distributed memory systems is greater than that of shared memory multiprocessors, and is neglected in previous analysis [131]. The basic definitions of the symbols used in our analysis are listed in Table 5.3.

Because there are two factors affect the load balancing, one from the hardware computing environment, the other from application itself, it is difficult to predict the total execution time of different scheduling schemes. Hence, we list the time for the ith loop allocation and corresponding execution time of loop allocation on processor m only in the following.

$$T_{Static} = \frac{\sum_{j=1}^{N/P} (\frac{1}{P}c_j + \frac{P-1}{P} r \times c_j)}{p_m} \tag{5.1}$$

$$T_{SS} = S + \frac{(\frac{1}{P}c_i + \frac{P-1}{P} r \times c_i)}{p_m} \tag{5.2}$$

$$T_{BSS} = S + \frac{\sum_{j=1}^{K} (\frac{1}{P}c_j + \frac{P-1}{P} r \times c_j)}{p_m} \tag{5.3}$$

$$T_{GSS} = S + \frac{\sum_{j=1}^{\frac{R_i}{P}} (\frac{1}{P}c_j + \frac{P-1}{P} r \times c_j)}{p_m} \tag{5.4}$$

$$T_{TSS} = S + \frac{\sum_{j=1}^{N_s-(i-1)\times\delta} (\frac{1}{P}c_j + \frac{P-1}{P} r \times c_j)}{p_m} \tag{5.5}$$

$$T_{FS} = S + \frac{\sum_{j=1}^{(0.5)^{\lceil i/P \rceil}\times\frac{N}{P}\times 0.5} (\frac{1}{P}c_j + \frac{P-1}{P} r \times c_j)}{p_m} \tag{5.6}$$

$$T_{SSS} = S + \frac{\sum_{j=1}^{(1-\alpha)^{\lceil i/P \rceil}\times\frac{N}{P}\times \alpha} (\frac{1}{P}c_j + \frac{P-1}{P} r \times c_j)}{p_m} \tag{5.7}$$

$$T_{AFS} = \begin{cases} S + \frac{\sum_{j=1}^{\frac{r_i}{k}} (\frac{1}{P}c_j + \frac{P-1}{P} r \times c_j)}{p_m}, & \text{if } r_i > 0, \\ (P+1)S + \frac{\sum_{j=1}^{\frac{r_o}{P}} (\frac{1}{P}c_j + \frac{P-1}{P} r \times c_j)}{p_m}, & \text{if } r_i = 0. \end{cases} \tag{5.8}$$

$$T_{AAFS} = \begin{cases} (P+1)S + \frac{\sum_{j=1}^{\frac{r_i}{k}} (\frac{1}{P}c_j + \frac{P-1}{P} r \times c_j)}{p_m}, & \text{if } r_i > 0, \\ (P+1)S + \frac{\sum_{j=1}^{\frac{r_o}{P}} (\frac{1}{P}c_j + \frac{P-1}{P} r \times c_j)}{p_m}, & \text{if } r_i = 0. \end{cases} \tag{5.9}$$

$$T_{ABS} = \begin{cases} S + \frac{\sum_{j=1}^{K \text{or} \frac{r_i}{k}} c_j}{p_m}, & \text{if } r_i > 0, \\ S + \frac{\sum_{j=1}^{K \text{or} \frac{r_o}{P}} r \times c_j}{p_m}, & \text{if } r_i = 0. \end{cases} \tag{5.10}$$

In Equation 5.8 and 5.9, the processor affinity does not reflected since it can be exploited in some parallel loops only. While in Equation 5.10, the static processor affinity is reflected. Based on Equation 5.1∼ 5.10, we have following qualitative remarks.

- When both the workload and hardware computing environment are balanced, i.e., $\forall i, j, p_i = p_j, c_i = c_j$, the ideal minimal execution time for N loops is $\frac{\sum_{i=1}^{N} c_i}{P \times p_i}$. In comparison with the ideal case, the time needed by ABS is $\frac{\sum_{i=1}^{N} c_i}{P \times p_i} + \frac{N}{P \times K} S$, which has a small increment. This result manifests that ABS is superior to SS, BSS when BSS is used in local scheduling phase of ABS, and GSS when GSS is adopted as local scheduling scheme in ABS. Compared with GSS, TSS, FS, SSS, and Static, the practical computing time of ABS is the lowerest. However, the synchronization time is difficult to analyze.

- With respect to that of the AFS[84], $O(k \log(\frac{N}{Pk}) + P \log(\frac{N}{P^2}) \times (P + 1)$ [5], the number of synchronization overhead of ABS is $O(k \log(\frac{N}{Pk}) + P \log(\frac{N}{P^2}))$ when GSS is used in ABS. Therefore, ABS will benefit greatly from this reducing of loop allocation overhead, and outperform AFS at any time.

- When BSS is used in ABS scheduling scheme, the synchronization overhead is fixed to $\frac{N}{P \times K} S$. Since the parameter K plays significant role here, the final performance is sensitive to the choose of K. Therefore, it is valuable to evaluate the effects of K on ABS scheme. Furthermore, whether fixed chunk size performs better than varied chunk size or not need to be studied.

In order to validate our analysis results, detailed performance evaluation is conducted in following section.

5.5 Experiment Platform and Performance Evaluation

5.5.1 Experiment Platform

The evaluation is done in the Dawning-1000A parallel machine developed by the National Center of Intelligent Computing Systems. The machine has eight nodes each with a 200MHz PowerPC 604 processor (the peak performance of each processor is 0.4Gflops) and 256MB memory. These nodes are connected through 100Mbps switched Ethernet. In the test, all libraries and applications are compiled using `gcc` with the `-O2` optimization option. The basic performance characteristics of JIAJIA are as follows. The round trip latency for a 1-byte message using the UDP/IP protocol is 239.68 microseconds on this platform. The time to acquire a lock is 2610.34 microseconds. The time for an eight processor barrier is 7180 microseconds. The time to obtain a page from home node is 1503.76 microseconds.

5.5.2 Application Description

In order to conduct a fair comparison among different scheduling schemes, we choose five applications which cover the range of all applications used in the literature. The selected applications include Successive Over-Relaxation (SOR), Jacobi Iteration(JI), Transitive Closure (TC), Matrix Multiplication (MAT), and Adjoint Convolution (AC).

SOR All the iterations of the SOR parallel loop take about the same time to execute and each iteration always accesses the same set of data. Exploiting processor affinity may improve

[5] This formula is a little bit different from the result in [84], the synchronization operations associated with collecting other processors' load state is neglected in [84] .

performance better than balancing the workload. The pseudo code for SOR is shown as follows. The computational granularity of each parallel iterations is $O(N)$.

```
  DO PARALLEL 2 J = 1, N
    DO SEQUENTIAL 3 K= 1, N
      A(J, K) = 0.25*(A(J-1,K)+A(J+1,K)+A(J,K+1)+A(J,K-1))
3   CONTINUE
2 CONTINUE
```

JI In the JI program, the top 20% of rows of in matrix A are nozero elements, which are generated by a random number generator. The iterations of parallel loop have a different workload which is determined by the distribution of nonzero elements in A. However, the data accessed by each iteration is not changed when it is executed repeatedly. The j iteration of the parallel loop always accesses the jth row of the matrices A, B[j], and X0[j]. When it is fixed to be executed repeatedly on a processor, it only needs to reload $X0[j]$ because $X0[j]$ is updated after each execution of the parallel loop. Hence, there is less communication between neighboring iterations.

```
   DO SEQUENTIAL I=1, L /* L controls  iteration precision */
     DO PARALLEL 2 J = 1, N
       X1[J] =0
       DO SEQUENTIAL 3 K=1, N
         IF (A[J][K] .NE.0).AND.(J .NE. K)
           X1[J] = X1[J]+A[J][K]*X0[K]
3      CONTINUE
       X1[J] = (B[J]-X1[J])/A[J][J]
2   CONTINUE
    DO SEQUENTIAL 4 L = 1,N
      X0[L] = X1[L]
4   CONTINUE
1 CONTINUE
```

TC From the pseudo code of TC shown following, we find the TC program may exhibit more serious load imbalance than JI, where each iteration of the parallel loop may have computational granularity of $O(1)$ or $O(n)$, depending on the input Matrix A.

```
   DO SEQUENTIAL 1 I= 1, N
     DO PARALLEL 2 J = 1, N
       IF (A[J][I] .EQ. TRUE) THEN
          DO SEQUENTIAL 3 K = 1, N
            IF (A[I][K] .EQ. TRUE)
               A[J][K] = TRUE
3         CONTINURE
2    CONTINUE
1  CONTINUE
```

MAT In [135], the authors point out that there is no affinity in MAT program. However, there exists affinity in MAT if we take the initial data distribution into account. This affinity can only be exploited in ABS scheduling scheme, which will be validated in Section 5.5.3. For other scheduling schemes, reducing loop allocation overhead is the only way to improve performance. This application is used to investigate which scheme has the lowest scheduling overhead.

```
   DO PARALLEL 1 I=1, N
     DO SEQUENTIAL 2 J=1, N
        DO SEQUENTIAL 3 K  1, N
          C[I][J] = C[I][J]+A[I][K]*B[K][J]
3       CONTINUE
2    CONTINUE
1 CONTINUE
```

AC Similar to MAT, the parallel loop in the AC kernel only executes once, hence it does not exhibit processor affinity in Yong et. al. viewpoint. However, as we described above, there is static affinity in AC. The computational grain of the ith parallel iteration is $O(n^2 - i)$, changing as a specific function of the control variable i, and leads to a significant imbalanced load distribution (a triangular pattern). This kernel is used to examine how efficiently of the different scheduling schemes to applications with inherent load imbalance.

```
   DO PARALLEL 1 I=1, N*N
     DO SEQUENTIAL 2 K=I, N*N
       A[I] = A[I]+X*B[K]*C[I-K]
2    CONTINUE
1  CONTINUE
```

5.5.3 Performance Evaluation and Analysis

Locality and Load Balancing

In order to have a clear understanding about the importance of locality and load imbalance[6] in distributed memory system, we run five applications under different data distribution schemes and compare it with the performance in load imbalance environment. The schedule scheme used in this comparison is static scheduling. Table 5.4 shows the test results in 8-way parallelism. The *Seq.* and *Size* columns list the problem sizes used and the sequential execution time under our hardware platform respectively. The run time with evenly data distribution and unevenly data distribution are listed in columns *Even* and *Uneven*. Test results shown in column *Ueven+HM* depict the effect of the home migration scheme supported by JIAJIA. Finally, the run time under load imbalancing condition (metacomputing environment) are presented in last column *Load imbalance*. Here, two artificial loads are added to half of the 8 nodes.

Table 5.4: The Effects of Locality and Load Imbalance (unit: second)

Application	Size	Seq.	Even	Ueven	Ueven+HM	Load imbalance
SOR	4096, 10 iter.	26.86	3.85	54.85	21.46	7.70
JI	5000, 10 iter.	34.74	5.36	22.66	6.43	10.72
TC	2048 × 2048	36.01	12.01	54.34	18.73	24.02
MM	2048 × 2048	500.80	79.52	198.07	198.07	164.04
AC	256^2	180.56	42.53	43.45	43.45	43.34

[6] Here, load imbalance is from the viewpoint of the practical available computing power.

The results in the Table 5.4 show that all the execution time with unevenly data distribution are longer than those in load imbalance cases except AC kernel, therefore, we conclude that locality is more important than load balancing intuitively. Though we can not draw this conclusion quantitatively because in few extremely case, load balancing maybe more important than locality. In fact, the execution time of AC reflects the locality is more important than load balancing too. In AC kernel, the affinity of 2/3 data can not be exploited at all, therefore, whether the data is distributed evenly or not does not affect the performance any more. The execution time does not change in load imbalance case manifests that the effect of load balancing can be neglected with respect to locality violation.

Theoretically, the performance will decrease to $1/n$ in load imbalancing environment, where n is the maximum number of load on one processor. While the performance of application will decrease to $1/m$ when locality can not be exploited, where m is the ratio between remote memory access time and local memory access time. Generally, m is larger than n in NOW environment in most cases. This conclusion conforms to that of Markatos and LeBlanc[85]. From Table 5.4 we find that our home migration improve the performance significantly by reducing the execution time about 61%, 75%, 65% for SOR, JI, and AC respectively. However, home migration does not affect the performance of MAT and AC at all. The main reason is that the home migration scheme is occurred in barrier operation only in current implementation. However, there is no barrier operation in MAT and AC two kernels at all. Hence the performance of these two kernels will not be affected.

Dedicated Environment

We first compare the efficiency of different scheduling schemes in dedicated environment, where there is no any other user processes in the system. We set value α of SSS to 0.90625, which is recommended in [81], and the block size of BSS to $\frac{N}{4P}$. The parameter α used in AAFS is set to the same as [135].

(a) (b) (c)

(d) (e)

Figure 5.2: Execution time of different scheduling schemes in dedicated environment: (a)SOR, (b) JI, (c) TC, (d) MAT, and (e) AC.

Table 5.5: The Number of Synchronization Operations of Different Scheduling Algorithms in Dedicated Environment

Apps.	Static	SS	BSS	GSS	FS	TSS	SSS	AFS	AAFS	ABS
SOR	0	41040	250	610	274	270	220	3832	13201	170
JI	0	50080	260	620	310	270	238	5147	13766	170
TC	0	205600	2500	5500	2318	2700	2033	35385	117337	1700
MAT	0	2056	25	55	43	27	33	408	1188	257
AC	0	65544	4105	81	31	27	41	1241	2641	465

Table 5.6: The Number of Getpages of Different Scheduling Algorithms in Dedicated Environment

Apps.	Static	SS	BSS	GSS	FS	TSS	SSS	AFS	AAFS	ABS
SOR	280	255762	79620	77515	77989	77592	80872	1094	4690	420
JI	317	178553	77046	62522	73843	70042	51161	2460	3038	453
TC	700	320150	43234	41738	73169	57772	103378	9106	25947	2106
MAT	13547	33153	21306	26491	24977	21605	22144	15487	15541	15509
AC	100	58735	4185	237	211	213	175	774	826	476

Figure 5.2 presents the execution time of different schemes under dedicated environment. In order to compare the corresponding loop allocation overhead, we list the number of synchronization operations resulting from loop allocation in Table 5.5. The number of remote getpages, which reflects the effects of remote data communication, is depicted in Table 5.6. More intuitive effects of remote data communication, load imbalance and loop allocation, are presented in Table 5.7 and Table 5.8, where the *Data*, *Syn*, and *Server* columns represent the time for data fetching, synchronization, and servicing the remote SIGIO requests respectively.

From Figure 5.2(a), (b), (c), and (d), we find that, because of the inherent characteristic of four kernels, SOR, JI, TC, and MAT, i.e. the processor affinity can be used statically and load balance exists in these kernels, static scheduling scheme works better. However, static scheme does not perform well for applications which have variable execution time length for load imbalance, which is demonstrated in AC as shown in Figure 5.2(e) and Table 5.7.

Among all the dynamic scheduling schemes, SS is the worst one for the five applications. The data in Table 5.5 and Table 5.6 gives two culprits. One is the large number of synchronization overhead resulting from the loop allocation, the other is the violation of processor affinity which is inherent in traditional central queue based scheduling algorithms, as we stated in Section 5.2.9. The effects of these two culprits are reflected in Table 5.7 obviously.

Compared with SS, the performance of BSS improves significantly, ranging from the lowest of 30% for MAT to the largest of 92.3% for TC. However, due to the large extra overhead entailed on loop allocation, and corresponding violation of processor affinity associated with BSS, GSS, FS, TSS, and SSS scheduling schemes, as listed in Table 5.5 and Table 5.6, the performance of these 5 schemes remains unacceptable with respect to Static scheduling scheme in dedicated environment for all but AC kernels, which is illustrated in Figure 5.2. These results show that the previous conclusions are no longer applicable in software DSMs on distributed memory systems.

Compared with the former 6 dynamic scheduling schemes, AFS makes a significant improvement, as shown in Figure 5.2, because the processor affinity is exploited in AFS. Therefore, the number of remote getpages reduces greatly, e.g., from 80872 of SSS to 1094 of AFS for

Table 5.7: System Overhead of Different Scheduling Algorithms in Dedicated Environment(I)(second)

Apps.	Static			SS			BSS			GSS			FS		
	Data	Syn	Serv	Data	Syn	Serv	Data	Syn	Serv	Data	Syn	Serv	Data	Syn	Serv
SOR	0.15	0.23	0.07	210.58	337.33	95.03	49.35	14.36	17.21	60.41	9.72	16.29	4 8.95	158.40	22.77
JI	0.28	0.62	0.09	127.38	307.79	75.35	55.37	21.52	14.17	54.16	4.18	12.09	48.77	119.09	19.76
TC	3.05	2.29	0.33	147.82	1188.34	197.81	29.64	67.41	14.05	145.38	45.85	14,41	48.02	222.48	24.93
MAT	9.11	1.5	4.69	35.80	3.10	9.51	15.31	3.15	6.95	18.47	4.53	8.36	17.80	24.08	7.9 7
AC	0.14	19.6	0.01	25.73	224.1	47.35	1.90	8.58	3.21	0.16	20.37	0.10	0.14	0.91	0.0 7

Table 5.8: System Overhead of Different Scheduling Algorithms in Dedicated Environment(II)

Apps.	TSS			SSS			AFS			AAFS			ABS		
	Data	Syn	Serv	Data	Syn	Serv	Data	Syn	Serv	Data	Syn	Serv	Data	Syn	Serv
SOR	49.76	11.74	16.22	51.90	26.74	11.88	0.52	1.74	0.41	2.71	9.70	3.33	0.21	0.84	0.2
JI	51.37	19.64	13.89	35.89	43.16	11.28	1.58	4.21	1.43	1.64	11.67	3.15	0.24	1.14	0.23
TC	42.06	58.72	17.19	59.58	195.72	33.03	7.14	22.46	4.11	14.19	91.38	23.60	3. 05	13.52	1.62
MAT	16.23	16.60	6.66	15.93	1.93	7.00	19.61	0.31	3.79	19.24	0.68	4.24	18.93	0.34	5 .01
AC	0.14	0.63	0.06	0.12	17.68	0.06	0.4	0.43	0.36	0.4	0.92	0.66	0.24	0.61	0.32

SOR, from 62522 of GSS to 2460 of AFS for JI, etc.. And so do the data time, as depicted in Table 5.8, e.g., the data time reduces from 51.90 of SSS to 0.52 of AFS for SOR kernel.

The evaluation results presented in [135] demonstrated that AAFS performed better than AFS for all five applications in their two hardware platforms. However, our results show the contrary conclusions. We find that AAFS is a little worse than AFS for all five kernels because of the large number of synchronization operations associated with AAFS as shown in Table 5.5. Therefore, the corresponding synchronization overhead increases greatly. For example, the synchronization time increases from 22.46 to 91.38 for TC with these two scheduling schemes, other results can be found in Table 5.8.

Among all of the dynamic scheduling schemes, ABS achieves the best performance. Compared with AFS, the synchronization overhead resulting from loop allocation in ABS reduces by one order of magnitude, as shown in Table 5.5. Furthermore, the number of getpages reduces significantly due to the reduction of synchronization operations. In lock-based cache coherence protocol of JIAJIA system, all the write notices are accompanied with lock, therefore, the larger the number of lock operation, the more possible is the invalidation and the getpage operation. In fact, most of the getpage operations here are about the task queue itself, not shared data of applications. The synchronization overhead reduces from 22.46 of AFS to 13.52 of ABS for SOR, and from 4.21 of AFS to 1.14 of ABS for JI. ABS has a little extra overhead compared with Static scheduling scheme for SOR, JI, MAT. While for TC, the difference between Static and ABS is mainly affected by the extra synchronization overhead of ABS, increasing from 2.29 to 13.52. However, our ABS scheme performs much better than Static for AC, as shown in Figure 5.2(e), where the synchronization waiting time reduces greatly, from 19.1 to 0.61 seconds. This reflects the dynamic load balancing ability of the ABS scheduling scheme.

In a word, Static scheduling is appropriate for well balanced applications in dedicated environment. All the dynamic scheduling schemes based on traditional central queue , such as SS, GSS, FS, TSS, and SSS, can not work in software DSM system. ABS scheme is the best scheduling scheme in dedicated environment.

(a) (b) (c)

(d) (e)

Figure 5.3: Execution time of different scheduling schemes in metacomputing environment:(a)SOR, (b) JI, (c) TC, (d) MAT, and (e) AC.

Metacomputing Environment

As the prevalence of NOWs, metacomputing environment is becoming the mainstream hardware platform for parallel computing. Therefore, evaluating the efficiency of these scheduling schemes in metacomputing environment is important. Although the practical computing power of each node is variable and nondeterministic, we assume a pseudo metacomputing environment by adding some artificial workloads to some nodes. In this test, two nodes have one added workload; one node has two artificial workloads; and one node has three other workloads. The rest nodes are free. The parameters used in some scheduling schemes are set to the same values as those in the last subsection.

Figure 5.3 illustrates the execution time of different schemes under metacomputing environment. Similar to the analysis in dedicated environment, the loop allocation overhead is listed as the number of synchronization operations in Table 5.9. The number of remote getpages, which reflects the effect of remote data communication, is shown in Table 5.10. More intuitive effects of load imbalance, loop allocation, and remote data communication are shown as execution time in Table 5.11 and Table 5.12. Here, the *Syn.* overhead includes two parts. One is the synchronization overhead associated with loop allocation operation, the other is the waiting time at synchronization point because of load imbalance.

As analyzed in Section 5.5.3, load imbalance is less important than locality in software DSMs. Therefore, for three kernels with fine computation granularity, SOR, JI, and TC, Static scheduling scheme remains well since the processor affinity is remained in Static scheme with respect to some dynamic schemes, such as SS, GSS etc. However, for coarse grain applications and applications which have limited locality, the performance of Static scheme becomes unacceptable because of the existence of load imbalance, such as MAT and AC. Figure 5.3 demonstrates our analyses. The *Syn* overhead listed in Table 5.11 and Table 5.12 shows the effects of load imbalance.

Similar to the dedicated environment, though SS promises the perfect load balance among

Table 5.9: The Number of Synchronization Operations of Different Scheduling Algorithms in Metacomputing Environment

Apps.	Static	SS	BSS	GSS	FS	TSS	SSS	AFS	AAFS	ABS
SOR	0	41040	410	610	310	270	270	7003	14414	970
JI	0	50080	420	620	315	270	288	7056	7251	3130
TC	0	205600	2500	5500	3154	2700	2761	56924	123529	5788
MAT	0	2056	41	55	31	27	26	1040	1565	227
AC	0	65544	4105	81	31	27	41	1411	2595	465

Table 5.10: The Number of Getpages of Different Scheduling Algorithms in Metacomputing Environment

Apps.	Static	SS	BSS	GSS	FS	TSS	SSS	AFS	AAFS	ABS
SOR	8456	260838	67418	82892	77177	75212	81928	12091	10478	6215
JI	13820	182889	75719	70413	77531	75621	57852	9045	7251	4058
TC	2392	335924	54418	55878	82701	54153	22773	41511	49191	19129
MAT	13547	33910	23843	26391	22965	21846	21935	17998	18198	17550
AC	100	58032	838	818	205	209	175	804	775	465

all the nodes, SS remains the worst scheme for all applications except MAT because of the large loop allocation overhead, as shown in Table 5.9. Furthermore, Table 5.10 shows the inherent drawback of traditional central queue based scheduling algorithm, i.e., potential to violate processor affinity, and results in large count of remote getpage operations. Two columns , *Data* and *Syn*, listed in Table 5.11, reflect the effects of these two aspects obviously. SS performs better than Static for MAT because the synchronization waiting time dominates the whole synchronization overhead here. The corresponding *Syn.* value reduces from 157.13 in Static to 8.07 in SS in MAT kernel.

BSS, GSS, FS, TSS, and SSS make some progress compared with SS, especially for MAT kernel. The performance of these 5 scheduling schemes is acceptable. However, as discussed in the last subsection, due to the large extra overhead resulting from loop allocation and corresponding potential of violating processor affinity associated with BSS, GSS, FS, TSS, and SSS scheduling schemes, as listed in Table 5.9 and Table 5.10, the performance of these 5 schemes remains unacceptable with respect to Static scheduling scheme in metacomputing environment for three fine grain kernels, which is illustrated in Figure 5.3. Among these five scheduling schemes, SSS is the worst due to the large chunk size allocated in the first phase, which results in large amount of waiting time at synchronization points. For example, the *Syn.* value of SSS is $(SOR, 122.50), (JI, 138.71)$, while the corresponding values in FS are $(SOR, 32.84), (JI, 65.00)$[7]. The rest are presented in Table 5.11 and Table 5.12 respectively. Therefore, the conclusions proposed in [81] are no longer available. However, from the Figure 5.3, we find the FS is prior to GSS in metacomputing environment as claimed in [53].

For all five applications but TC, the performance of AFS is improved significantly compared with the former 6 dynamic scheduling schemes, as shown in Figure 5.3. Though the number of synchronization operations of AFS increases about one order of magnitude, as shown in Table 5.9, the number of remote getpage reduces about one order of magnitude, as listed in

[7] Since the algorithm of FS and SSS is very similar except the value of chunk size in allocation phase, we compare it with FS here.

Table 5.11: System Overhead of Different Scheduling Algorithms in Metacomputing Environment(I)

Apps.	Static			SS			BSS			GSS			FS		
	Data	Syn	Serv	Data	Syn	Serv	Data	Syn	Serv	Data	Syn	Serv	Data	Syn	Serv
SOR	24.29	65.41	5.34	954.48	834.48	259.49	192.69	48.98	36.54	181.77	89.29	37.06	1 54.58	32 .84	32.90
JI	29.80	25.95	4.28	601.00	518.90	169.71	214.44	52.16	34.08	172.98	85.65	34.2 2	175.90	65.00	36.21
TC	21.76	116.20	4.89	345.1	2884.96	305.29	93.19	159.34	42.64	141.59	102.14	39.60	204.59	1 44.44	57.28
MAT	22.90	157.13	13.34	79.35	8.07	20.22	56.70	18.25	16.58	47.84	65.35	19.79	42.14	54.38	16. 38
AC	0.33	39.14	0.06	35.57	777.31	88.98	0.49	0.48	0.16	0.5	45.57	0.2	0.43	27.18	0.17

Table 5.12: System Overhead of Different Scheduling Algorithms in Metacomputing Environment(II)

Apps.	TSS			SSS			AFS			AAFS			ABS		
	Data	Syn	Serv	Data	Syn	Serv	Data	Syn	Serv	Data	Syn	Serv	Data	Syn	Serv
SOR	148.30	42.55	28.88	157.80	122.50	38.66	70.02	29.64	15.91	44.01	51.90	16. 37	30.27	23.72	10.23
JI	165.53	86.77	34.12	126.80	138.71	30.97	40.59	28.62	11.66	15.25	61.16	11.67	3.95	32.01	5 .40
TC	98.13	146.61	41.16	53.08	93.19	19.37	64.13	212.96	40.98	72.23	541.22	80.9 8	38.89	99.78	1 9.32
MAT	39.81	59.23	15.88	40.78	137.54	19.14	55.23	2.18	11.1	49.81	3.35	14.5	50.90	1. 12	11.99
AC	0.5	28.66	0.14	0.36	46.10	0.14	0.81	2.04	0.71	0.75	4.48	1.33	0.5	1.55	0.58

Table 5.10, which amortizes the effects of loop allocation overhead. Surprisingly, the number of remote getpage operations of AFS in TC leads to the contrary conclusion, i.e., it does not reduce at all with respect to former scheduling schemes. The main culprit here is the relaxed memory consistency model used in state-of-the-art software DSM system, where all the data associated with one synchronization object will be invalidated when requesting this synchronization object. Therefore, the large number of synchronization operations of AFS will lead to large amount of invalidation, which in turn leads to large number of remote fetchs. So synchronization overhead becomes unacceptable in AFS for TC, as shown in Table 5.12.

As described before, AAFS increases the number of synchronization overhead in local scheduling phase in order to collect the states of other nodes. Therefore, the corresponding synchronization overhead becomes larger than that of AFS. For example, in TC, the number of synchronizations increases from 56924 of AFS to 123529 of AAFS, the corresponding synchronization time increases from 212.96 seconds in AFS to 541.22 seconds in AAFS. Other results are presented in Figure 5.3, Table 5.9 and Table 5.10, and Table 5.12. Our results give the contrary conclusion to that presented in [135] where AAFS was better than AFS for all five applications in their two hardware platforms.

As in dedicated environment, the ABS scheduling scheme achieves the best performance among all of the dynamic scheduling schemes, as shown in Figure 5.3, and is superior to Static for all 5 kernels except TC. From Table 5.9 and Table 5.10, we can find that the great reduction of the number of getpages, an obviously result of the exploitation of processor affinity, amorizes the negative effects of the increasing of synchronization overhead. Table 5.11 and Table 5.12 show this case. For TC kernel, the advantage of ABS does not manifest with respect to the best Static scheduling scheme. There is only 8% performance gap between them. Compared with AFS, the synchronization overhead resulting from loop allocation in ABS reduces about one order of magnitude or more, as shown in Table 5.9. Furthermore, the number of getpages reduces significantly because of the reduction of synchronization operations and the lock-based cache coherence protocol adopted in JIAJIA. In this protocol, all the write notices are accompanied with lock. Therefore, the larger the number of lock operations, the more possible is the invalidation and the getpage operation. So the corresponding values of *Data*, *Syn.* and *Server* all reduce greatly from AFS to ABS as listed in Table 5.12.

In summary, SS remains an inappropriate scheduling scheme for metacomputing environment. For fine grain applications and applications with predictable processor affinity, Static scheduling works well. While for coarse grain applications or applications which have unpredictable locality characteristic, BSS, GSS, FS, and TSS perform better than Static and SS schemes, which conforms to the previous conclusions to some extent. However, the evaluation resulting of SSS are contrary to the assertion of Liu et.al. [81]. AFS, AAFS, and ABS achieve good performance because of the use of processor affinity. However, similar to the conclusion in dedicated environment, the advantage of AAFS with respect to AFS proposed by Yong et al. [135] does not exist at all. On the contrary, AFS is better than AAFS in metacomputing environment. Because of the reduction of synchronization overhead and the great improvement of waiting time resulting from load imbalance, ABS achieves the best performance among all dynamic scheduling schemes.

Determining of Important Parameters

(a) (b) (c)

(d) (e)

Figure 5.4: Execution time with different chunk size under ABS scheduling scheme in metacomputing environment:(a)SOR, (b) JI, (c) TC, (d) MAT, and (e) AC.

In previous section, we use R_i/p as the chunk size at each local allocation operation. In fact, there is many different schemes for the chunk size selection. Here, we test several values of chunk size in order to find an optimal value. We compare the performance of ABS scheme with following 6 chunk size allocation scheme: N/P, $N/2P$, $N/4P$, $N/8P$, $N/16P$, R_i/P. Figure 5.4 show the execution time of five kernels respectively. In order to have a clear understanding about the difference between these variations of ABS, we list the number of synchronization operations associated with loop allocation and the number of remote getpages in Table 5.13 and Table 5.14 respectively. Furthermore, the corresponding system overhead with these variations are presented in Table 5.15.

Based on these figure and tables, we have following conclusions.

Table 5.13: The Number of Synchronizations with Different Chunk Sizes in Metacomputing Environment

Apps.	N/P	N/2P	N/4P	N/8P	N/16P	R_i/P
SOR	170	620	741	1046	1618	970
JI	170	607	797	1096	1695	3130
TC	1700	4473	5788	8752	14837	25700
MAT	17	89	113	131	175	257
AC	17	146	199	169	250	465

Table 5.14: The Number of Getpages with Different Chunk Sizes in Metacomputing Environment

Apps.	N/P	N/2P	N/4P	N/8P	N/16P	R_i/P
SOR	419	10991	8677	8635	8780	6215
JI	455	10481	8875	9013	8535	4058
TC	2095	22885	19129	17891	19879	24222
MAT	13560	16516	17237	17494	17957	17550
AC	105	315	374	337	415	465

1. Generally, as the size of chunk decreases, the number of synchronization overhead will increases, and so does the the number of getpages since the more possible of violation of processor affinity, as shown in Table 5.13 and Table 5.14. However, since locality is more important than load balance for fine grain applications, as we analyzed in Section 5.5.3, the performance of SOR, JI, TC, will decrease accompany with the chunk size.

2. As we described before, the synchronization time *Syn* includes two aspects: one is related to loop allocation operation, which favor large chunk size, the other is related to load imbalance, which favor small chunk size. Therefore, there is a tradeoff between them. Therefore, there is a tradeoff between the chunk size and the performance for general case, which is reflected in Figure 5.4 and Table 5.15.

3. For coarse grain applications, such as MAT and AC, the synchronization waiting time dominates the main performance effects. Therefore, the smaller the chunk size is, the more possible of load balance, and so the better the performance is. For example, though the value of *Data* increases from 23.78 (N/P) to 54.55 ($N/16P$), the synchronization overhead *Syn* reduces from 155.63 (N/P) to 4.15 ($N/16P$), which will amortize the effect of large amount of getpages. However, when the chunk size reduces furthermore, the effect of loop allocation and remote data communication will increase too. So there is a tradeoff between these factors too.

In summary, it is difficult to find an optimal value for any applications within ABS scheduling scheme. However, we find that R_i/P is an acceptable chunk size in ABS scheme.

5.6 Summary

In this Chapter, we evaluate and compare several well known different scheduling schemes in a software DSM system under both dedicated and metacomputing environment. Experimental results manifest that many conclusions for hardware shared memory systems are not applicable

Table 5.15: System Overhead of Different Chunk Sizes in Metacomputing Environment

Apps.	N/P			N/2P			N/4P			N/8P			N/16P			R_i/P		
	Data	Syn	Serv	Data	Syn	Serv	Data	Syn	Serv	Data	Syn	Serv	Data	Syn	Serv	Data	Syn	Serv
SOR	1.60	15.69	1.38	41.34	25.94	12.28	41.30	14.05	10.39	54.25	17.15	12.57	56.29	19.15	13.12	30.27	23.72	10.23
JI	0.61	12.26	0.55	39.75	12.27	9.59	37.91	9.46	8.97	51.88	9.30	11.20	32.44	15.61	9.79	3.95	32.01	5.40
TC	4.92	38.58	3.63	68.90	111.51	27.73	38.89	99.78	19.32	32.04	114.62	19.82	23.88	157.08	23.45	17.85	236.75	30.67
MAT	23.78	155.63	13.22	29.93	54.68	13.45	52.58	33.22	14.41	50.93	10.50	11.84	54.55	4.15	12.66	50.90	1.12	11.99
AC	0.21	39.06	0.10	0.38	12.67	0.35	0.41	7.51	0.40	0.43	4.14	0.36	0.45	3.69	0.42	0.50	1.55	0.58

in software shared memory systems. Meanwhile, we propose and implement a new affinity-based (ABS) dynamic self scheduling algorithm for home-based software DSM systems. Due to taking the static processor affinity into account, ABS performs better not only in dedicated environment but also in metacomputing environment. Therefore, ABS is the best alternative scheduling scheme to be used in software DSM systems.

In order to find an optimal value for local scheduling of ABS, we compare several different chunk size selection schemes. We find that R_i/P is an acceptable chunk size scheme.

Chapter 6

Dynamic Task Migration Scheme

6.1 Introduction

The four substantial characteristics of meta-computing environments, namely *heterogeneity, variable resource capacity*, *distribution*, and *non-dedicated nature* generally reduce the application performance, thus making such environments not as effective as multiprocessors. In order to maximize the application performance based on available resources, a parallel system must dynamically distribute the work not only according to the inherent computation and communication demands of the application, but also according to the available computation resources. Dynamic task migration is an effective strategy to achieve this goal, and it has been widely studied in the literature[124, 90, 121, 46, 80].

According to the granularity of a task, previous work can be classified into three categories: *loop-level*, *thread-level*, and *process-level*. Loop-level and thread-level task migration schemes are widely used in shared memory environments, such as shared memory multiprocessors and software DSM systems. Whereas process level task migration is generally adopted in message passing environments, such as distributed systems and MPPs. The application domain assumed in this chapter is that of highly iterative scientific code running on home-based software DSM systems.

In both loop- and thread-level task migration schemes proposed in the past, however, a "task" means the corresponding code of computation, i.e., the data related to this computation is totally neglected. As such, when one task is migrated from processor A to another processor B, the data required by this task remains on processor A. Thus, the processor B has to perform remote communication when it executes this task, eliminating the advantage of task migration, or even making the performance worse. Hence, the definition of traditional "task" should be revisited. We define a task as follows:

$$\textbf{Task} = \textbf{Computation subtask} + \textbf{Data subtask}$$

Computation subtask is the program code to be executed, while the data subtask is the operations to access the related data located in memory. In fact, in distributed memory systems, as the speed gap between processors and memory becomes larger and larger, the importance of data subtask becomes more obvious than before. Therefore, we argue that both subtasks should be migrated to a new processor during task migration.

Based on this observation, we propose a new dynamic loop-level task migration scheme for iterative scientific applications. This scheme is implemented within the context of the JIAJIA software DSM system[51]. The evaluation results show that the new task migration scheme

```
for (i=0; i<N; i++) {
        execute task;
        synchronization;
        if (i % $\alpha$ == 0) {
                collect computing power of each processor;
                computation subtask migration;
        }
        if (i % $\alpha$ == 1) {
                data subtask migration;
        }
}
```

Figure 6.1: Basic framework of dynamic task migration scheme.

improved the performance of our benchmark applications by 36% to 50%, compared with static task allocation schemes. As a result, our new task migration scheme performs better than other computation-only migration schemes for an average of 30%.

The rest of the chapter is organized as follows. The rationale and implementation details of the dynamic task migration scheme are depicted in the following two Sections, respectively. In Section 4, experimental results and performance analysis are given. Related other work are compared with our new scheme in Section 5.

6.2 Rationale of Dynamic Task Migration

For simplicity, we define one loop iteration and it's corresponding data as a task here, and we assume that the tasks are allocated statically with good processor affinity, i.e., both the computation and the data subtask are allocated as close as possible.

Since the execution time of each task is nondeterministic and the practical capacity of each processor can not predetermined in meta-computing environments, static task allocation schemes do not perform well. On the other hand, in iterative applications, the available computing power which is effected by the application and processor capacities can be obtained at run time. Accordingly, tasks can be reallocated to balance the remaining iterations, thus improving the resource utilization.

In dynamic task migration schemes, however, it is difficult to determine statically the relationship between computation and data subtasks. In general, only computation tasks are migrated in such scheme, while the data remain at the original processor, resulting in remote data access latency. For example, the dynamic scheme of Subramaniam and Eager[121] belongs to this category. We use *CompMig* to represent this scheme. Although the cache of remote processor will utilize data reuse, due to the limited cache size, the contents of the cache are replaced randomly, thus, yielding inevitably many remote data misses after computation migration. In our new migration scheme, not only the computation subtasks, but also the data subtasks are migrated to appropriate processors. This scheme is represented as $TaskMig$. In order to find the relationship between computation and data subtasks, the migration of data is delayed one iteration for every α iterations. The pseudo code of our new scheme is shown in Figure 6.1. In this framework, N is the total number of iterations and α is the migration step, which determines the frequency of task migration. Currently, the value of α is set to 1/10 of N.

Table 6.1: Definition of the Symbols

Symbol	definition
P	the number of processors
W	total number of tasks
W_i	the workload of processor i
T_i	the computation time of last interval* of processor i
P_i	the available computing power of processor i
$W_i{}'$	the new workload of processor i after task migration
$E(t)$	expected value of execution time T_i
$\sigma(t)$	variance of execution time T_i
α	migration step
β	significance parameter in migration decision

*Here, the interval is delimited by two consecutive task migration calls.

In iterative applications, a global synchronization (i.e., barrier) is required at the end of each iteration to maintain coherence among all the processors. Though the synchronization operation and task migration are shown separately in Figure 6.1, we embed all the migration work in the synchronization operation in our implementation, so that the overhead of task migration is minimized to be negligible. Details about the implementation is presented in the following section.

6.3 Implementation

6.3.1 Computation Migration

We augmented the global synchronization operation of the JIAJIA system to count T_i, W_i[1] dynamically. Based on these two variables, P_i is defined as $\frac{W_i}{T_i}$. Ideally, the task can be reallocated according to P_i. However, experimental results show that a small change in the load might cause the system to oscillate and can be harmful for the overall application performance. So, we borrowed the idea of the *variance*, which is used to measure the degree of the deviation of random variable to *average expected value*, to decide whether the workload should be reallocated. The expected value and the variance of execution time T_i is defined as follows:

$$E(t) = \frac{1}{P}\sum_{i=1}^{P} T_i \quad \text{and} \quad \sigma(t) = \sqrt{E(t^2) - E(t)^2} \tag{6.1}$$

Only when the ratio between $\sigma(t)$ and $E(t)$ is larger than a threshold β, we perform task migration. The parameter β varies for different applications and different environments. In our tests, the value of β was set to 0.2. After task migration, the workload $W_j{}'$ of each processor j falls into the following range:

$$\lceil \frac{\sum_{i=1}^{j-1} P_i}{\sum_{i=1}^{P} P_i} W \rceil \sim \lceil \frac{\sum_{i=1}^{j} P_i}{\sum_{i=1}^{P} P_i} W \rceil$$

[1] Table 6.1 defines all the symbols used in this paper.

This algorithm is implemented entirely in `jia_lbarrier(&begin, &end)` which is a variation of the original `jia_barrier()` function. The parameters `begin` and `end` represent the upper and lower bounds of the task allocated to the local (caller) processor, respectively. In order to reduce extra system overhead associated with the computation migration, the computation migration is called at every α steps. If α is too small, then the overhead becomes non-negligible. Otherwise, the dynamic environment will not be reflected accurately, and the performance will degrade. Therefore, α is another key parameter in this migration scheme.

6.3.2 Data Migration

As shown in Figure 6.1, data migration is called after computation migration is iterated once. During this iteration, the relationship between migrated tasks and their corresponding data is detected by run time system. According to this relationship, the related data are sent to the appropriate processors transparently.

The heuristic idea about relationship between data and computation is based on the observation that the computation task always processes (including read and write) the data exclusively and appears as a single writer in cache coherence protocols[4]. In the context of a single writer, the data can be migrated close to computation task so that all the future accesses will hit locally. This can be implemented in underlying cache coherence protocol automatically. However, we basically assumed that all the data and computation are allocated in home statically. Therefore, the migration of data necessitates the support of home migration, which has not been implemented in any previous home-based software DSM systems. As a matter of fact, we recently proposed a simple but efficient home migration scheme in the JIAJIA system as described in the following. With this migration scheme, all the data with single writer characteristic will migrate to the new home automatically. The data migration is implemented by a new function called `jia_config(HMIG, ON)`. Here, the first parameter indicates the home migration scheme, and the second parameter controls the beginning and the end of our home migration scheme.

In the current implementation, both migration schemes are embedded in the barrier operation at the end of each iteration, to reduce the system overhead.

6.4 Home Migration

As has been stated, the performance of home-based software DSMs is sensitive to the distribution of home pages. A good distribution of home pages should keep the home of each shared page at the processor that most frequently writes it so that the processor can write directly to the home and least diff is produced and applied.

A problem that arises in home migration is that there must be some mechanism for all hosts to locate the new home of a migrated page. The most straightforward solving method is to broadcast the new home of migrated pages to all hosts. Broadcast, however, is very expensive. To reduce message overhead, the decision of migrating the home of a page is made at the barrier manager and the page migration information is piggybacked on the barrier grant message in JIAJIA.

To further reduce migration overheads, JIAJIA migrates home of a page to a new host on a barrier only if the new home host is the single writer of the page during last barrier interval. In the basic protocol, a cached page is invalidated on a barrier if this page has been modified in last barrier interval. However, if the page is modified by only one processor during last barrier interval and the modifying processor is not the home of the page, then it is unnecessary for the modifying processor to invalidate the page from its cache because the page is already a coherent one, non-home copies of this page in other processors are invalidated as normal. Hence, if the home of this page is migrated to the single writing processor, no page propagation is required because the new home already has the most recently modified page. JIAJIA recognizes the single writer of a page in the barrier manager and informs all hosts to migrate the home of the page to the single writing processor on the barrier grant message.

Since no page propagation happens in home migration of a page, the barrier manager must ensure that the cache of new home host (or single writer) must have a copy of the migrated page. This may not be true in JIAJIA because the page may have already been replaced from the cache of the single writing processor to make room for other pages. JIAJIA solves this problem with a tag in each write notice sent to the barrier manager from the barrier arriving processor. This tag indicates whether the cache of the barrier arriving processor has a valid copy of the associated page.

With the above analysis, the home migration algorithm of JIAJIA can be described as follows.

1. On arriving at a barrier, the barrier arriving host sends write notices of the associated barrier interval to the barrier manager. Each write notice is the page-aligned address of a page modified since last barrier. The least significant bit of the write notice is tagged as "1" if the associated page is valid in the cache.

2. On receiving a barrier request, the barrier manager records write notices carried on the barrier request. Each write notice is associated with a `modifier` field which indicates the identifier of the host which modifies the page. If a page is modified by more than two hosts, then the `modifier` field is tagged as MULTIPLE.

3. After all barrier requests are received, the barrier manager broadcasts all write notices (together with the `modifier` field for each write notice) to all hosts.

4. On receiving a barrier grant message, the processor checks each received write notice and associated `modifier` field. If the write notice and the associated `modifier` field indicate that the page is modified during last barrier interval by multiple hosts or by a single host other than itself, then the associated page is invalidated if it is cached. If the write notice and the associated `modifier` field indicates that the page is modified by only one host and the cache of the host has a valid copy of the page (the least significant bit of the write notice is "1"), then home of the page is migrated to the modifying host. (1). If the host is the new home of the migrated page, then an item is allocated in the home array for the page, the item in the cache array is released, and the home host field of the associated global page table item is set to point to the new home. (2). If the host is the old home of the migrated page, then the associated item in the home array is released, the page is invalidated if there is no free item in the cache array, or is kept as valid if a free item is found in the cache array for the page, and the home host field of the associated global page table is set to point to the new home. (3). If the host is neither the new home nor the old home of the migrated page, then only the home host field of the associated global page table item is set to point to the new home.

It can easy be seen that, the above algorithm does not require any extra message for home migration, neither messages to inform the migration nor migrated page propagation is required. In fact, home migration can improve the locality of the system even in dedicated environment. As a result, the creation of twins and diffs are avoided significantly because writes to home pages do not produce twin and diff in home-based system. More detais can be found at [50].

6.5 Experimental Results and Analysis

6.5.1 Experiment Platform

We conducted our evaluation on the Dawning-1000A parallel machine developed by the National Center of Intelligent Computing Systems of China. This machine has eight nodes, each with a PowerPC 604 processor and 256MB memory. The nodes are connected through a 100Mbps switched Ethernet. During the experiments, all libraries and applications (except a FORTRAN application) are compiled by `gcc` with the `-O2` optimization option. The meta-computing environment is assumed by adding artificial loads to some processors in order to have a fair comparison. For the experiments, we added 2, 1, 3, 1 loads to the second, fourth, sixth, and eighth processors, respectively.

6.5.2 Applications

Three iterative applications were used to evaluate our task migration scheme: A locally developed inner-product of matrix multiplication (MAT) code, with matrix size of 1024×1024 and repeated 100 times. SOR (Successive Over-Relaxation), with a matrix size of 4096×4096 floats, and 100 iterations. SOR is an iterative method for solving partial differential equations, with nearest neighbor 5 point difference as the main computation. Finally, Em3d from Institute of Electronics, Chinese Academy of Sciences[107] is used. Barriers are used for synchronization. The grids used in this test is 60×30×832, and 100 iterations are run in our test, while the real run requires 6000 iterations.

6.5.3 Performance Evaluation and Analysis

Figure 6.2(a) shows the relative performance, with the largest values listed on the top of the bar. For comparison, we listed the execution time in dedicated *Dedi* and meta-computing *Meta* environment without task migration on the left. The execution time with computation migration *CompMig* and task migration *TaskMig* schemes are listed on the right of each bar groups. The breakdown of system overhead under different schemes are shown in Figure 6.2(b). System overhead is divided into three parts: *Segv*, *Syn*, and *Serv*. *Segv* represents the data miss penalty, including local and remote misses. *Syn* and *Serv* represent the time spent on synchronization and servicing remote requests, respectively. Detailed analysis about the breakdown can be found in [105]. The leftmost bar reflects the system overhead of *Meta*, while the overhead of *CompMig* and *TaskMig* are shown as the middle and the right bars, respectively.

Figure 6.2(a) shows that *TaskMig* improves the performance significantly compared with *Meta*, ranging from 36% for MAT to 50% for SOR and Em3d. We ascribe this to the great reduction of waiting time at synchronization points, as shown in Figure 6.2(b).

(a) (b)

Figure 6.2: Performance comparison: (a) execution time, (b) system overhead.

Table 6.2: System Overheads in Unbalanced Environment

Overhead	1	2	3	4	5	6	7	8
Segv.	0.43	15.74	12.94	13.51	12.88	25.22	25.24	26.17
Syn.	288.82	123.94	**294.80**	211.81	291.04	**23.57**	284.35	187.90
Serv.	19.01	9.64	5.29	5.89	10.00	10.96	5.25	1.70

The difference between *CompMig* and *Meta* manifests that the effectiveness of *CompMig* is limited and that it relies greatly on the granularity of the application. For coarse-grain applications, such as MAT, the induced remote data miss penalty of *CompMig* is by the reduction of the long computation time. Hence, *CompMig* performs well. However, for relatively fine grain applications, such as SOR and Em3d, the remote data miss can not be compensated by computation, and thus, the performance of *CompMig* is not satisfactory. The results shown in Figure 6.2(b) confirm our analysis.

In comparison with *CompMig*, our new task migration scheme takes the data migration into account. As such, not only the synchronization overhead, but also the remote data miss penalty *Segv* are reduced significantly, as shown in Figure 6.2(b). So the total execution time of *CompMig* are improved about 12.5%, 37.5% and 37.5% for MAT, SOR, and Em3d, respectively.

In order to clarify the effectiveness of our task migration scheme, we now argue about the system overhead of all of the 8 processors using Em3d results. Table 6.2 lists the results in meta-computing environment without load balancing, while Table 6.3 shows the case with new task migration scheme.

It can be seen from Table 6.2 that the main difference is induced by synchronization. For example, *Syn* overhead of the slowest processor (#6) is 23.57 seconds, while in the fastest processor (#3) is 294.80 seconds. In other words, the faster processor is idle for 76% due to the computing power imbalance. However, this problem is solved significantly with the new task migration scheme. As shown in Table 6.3, the gap between the maximal and minimal values of *Syn* is 51.11, i.e., only 19% with respect to the counterpart in meta-computing environment with static task allocation scheme. From the tables we find that the overhead incurred by *Segv* and *Serv* are slightly increased with new task migration method. This is because the data migration is performed one iteration later than the computation migration, and there were some remote data misses in that iteration. However, compared with the decrease in *Syn*

Table 6.3: System Overheads in Unbalanced Environment with Task Migration

Overhead	1	2	3	4	5	6	7	8
Segv.	20.25	15.90	21.27	20.14	30.43	19.40	16.70	5.59
Syn.	36.59	**25.21**	58.84	39.94	50.13	26.01	**76.32**	62.94
Serv.	7.32	19.57	12.49	15.42	15.90	22.61	14.67	6.96

overhead, this minor increase is negligible.

Furthermore, in order to evaluate the resource utilization, we define the efficiency E of the new task migration scheme as the ratio between the ideal execution time (T_{Ideal}) and that of $TaskMig$ scheme ($T_{TaskMig}$). Since the total workload is fixed, we have the following equation:

$$T_{Ideal} \times Power' = T_{Static} \times Power \tag{6.2}$$

where $Power$ represents the total computing power of 8 processors, $Power'$ represents the available computing capacity of these processors. In our test environment, $Power' = (1 + 1 + 1 + 1 + 0.5 + 0.5 + 0.33 + 0.25)/8 = 0.6975 \times Power$. As such, the efficiency E is defined as follows:

$$E = \frac{T_{Ideal}}{T_{TaskMig}} = \frac{1.43 \times T_{Static}}{T_{TaskMig}} \tag{6.3}$$

Given the above, the efficiencies of $TaskMig$ for three benchmarks are 53%, 71% and 74%, respectively. Therefore, we conclude that our new task migration scheme can achieve a satisfactory resource utilization as well.

6.6 Related Work

Self scheduling[124] and guided self scheduling[90] fit for non-iterative applications in shared memory multiprocessors, and they do not consider the relation between computation and corresponding data at all. Although Markatos and Le Blanc[84] took processor affinity into account, their scheme relies on the cache to store the remote data, which is limited by the cache size and random replacement of cache lines.

Subramaniam and Eager[121] proposed dynamic and wrapped schemes for task migration, however, their scheme is computation migration, not task migration. Ioannidis and Dwarkadas[60] addressed the load balancing in software DSM systems with the similar computation migration scheme as ours. However, their scheme requires compiler support to extract access patterns from the source code to find the relationship between computation subtask and data subtask. Hsieh et al.[46] proposed computation migration to enhance locality for distributed memory parallel systems with the goal to improve locality. Their scheme requires the programmer to add some annotations in the program to direct computation migration, which we believe is error prone.

Thread migration is widely used to balance the load and improve the locality in software DSM,systems such as Millipede[62] and D-CVM[129]. Compared with thread migration, our scheme has two different characteristics: (1) Lower overhead. Migrating an entire thread is expensive, since there may be a large amount of state to move. Moreover, thread migration would lead to inconsistencies in the shared memory state[129]. While in our task migration, the computation migration is embedded in the synchronization operation with minimal additional overhead. (2) Ease of use. Task migration proposed in this paper is performed automatically without requiring any extra information about application characteristics. While in D-CVM's thread migration scheme, the static relation among different threads and the cost of each thread are required in order to make an appropriate decision. Though Millipede uses access histories in choosing threads for automatic migration too, it lacks the global view since it only considers the interactive between any two nodes and neglects the communication used to keep the shared data coherent between the processors.

The dynamic task scheduling technique proposed by Liang et al.[80] is similar to our work. Their scheduling strategy also includes two steps: *Migration* and *Exchange*. However, there are three substantial differences: (1) In their migration stage, the decision of migration is according to computation time only, and the processor power is assumed to be fixed, which does not reflect the real world of scenarios. (2) The exchange stage is used according to access patterns, i.e., in their scheme, the pair of the tasks with the most number of shared pages will be reallocated onto the same processor in the exchange stage. In our scheme, the data, not the computation task, is migrated according to access patterns. (3) Their scheme is thread-level, while ours is loop-level.

6.7 Summary

In this Chapter, we identified the disadvantages of traditional task migration methods and presented a new definition of a task, which takes into account both computation and data. Based on this observation, we proposed a new dynamic task migration and evaluated it within the context of the JIAJIA software DSM system. A detailed comparison with other related work shows the advantages of our scheme. The evaluation results show that new task migration scheme improves the performance ranging from 36% to 50% compared with a static task allocation scheme in a meta-computing environment. Our task migration scheme (with the data subtask component) performs better than traditional task (computation-only) migration approach about 12.5% for MAT, and 37.5% for SOR and Em3d. Higher resource utilization, 53% for MAT, 71% for SOR, 74% for Em3d, is also archieved by the proposed task migration scheme.

Chapter 7

Communication Optimization for Home-based Software DSMs

With the introduction of the Virtual Interface Architecture (VIA), which is sponsored by Compaq, Intel, and Microsoft, the technology of User Level Networking (ULN) should become more and more mature. So it is time to study application-level issues of ULN, such as integrating high-level application and protocol buffer management and optimizing the communication path. The exposure of all underlying memory management to high level softwares gives us a chance to design a highly efficient, application-specific communication substrate. In this chapter, we present our ongoing work on designing a communication substrate specific to home-based software DSM systems. Based on the observation of communication requirements of software DSMs, we propose and design a Myrinet-based communication library named JMCL specific to JIAJIA.

7.1 Introduction

Recent progress in user-level networking (ULN)[13] and software distributed shared memory (DSM) system motivate us to do the work presented in this chapter. Firstly, in user level networking area, with the introduction of the Virtual Interface Architecture (VIA) [22] sponsored by Compaq, Intel, and Microsoft, the technology of user-level networking should become more and more mature. It is therefore time to study application-level issues of ULN, such as integrating high-level application and protocol buffer management and optimizing the communication path[132], such as Evan et.al work on MPI library using VIA[115]. The exposure of all underlying memory management to high-level softwares gives us the chance to design a highly efficient, application-specific communication substrate.

Meanwhile, it is widely accepted that communication overhead is the main obstacle that prevents the software DSMs becoming the desirable vehicle for parallel computing [105]. Several techniques were proposed to reduce the communication overhead in the past, such as using relaxed memory consistency models to reduce the frequency of communication, using multiple writer protocol to reduce communication traffic, etc. [20, 70]. However, as considering protocols have matured, there is little space for reducing the frequency and traffic of communication urther. Therefore, we must find other ways to reduce communication overhead, for example by minimizing the latency of communication.

Nowadays, almost all available software DSMs use the UDP/IP network protocol as their communication substrate, which is time consuming and unreliable. Thus, the high-level software DSMs must manage flow control, packet order, and error control by themselves, which wastes much unnecessary time. In fact, these extra system software processing overhead and communication latency associated with each communication can be avoided by using user-space communication[5], which substantially reduces the layers of user-level and kernel-level software that is required to get to and from the network. Many user-space communication libraries have been proposed recently, such as AM[33], AM-II, FM[88], FM/MC[9], U-Net[32], PM[125], VMMC[27], VMMC-2[26], BIP[91], and LFC[12], etc.[96], of which some support reliable, ordered, low latency (zero or one copy) communication. Unfortunately, these user-space communication libraries were designed for standard message passing, remote memory write, handler carrying message communication models only [13, 96], and many of them even did not support interrupt-based notification. In other words, many communication characteristics of software DSMs are not taken into consideration by these systems designing. Currently, our group is developing a Myrinet-based user level communication substrate specific for JIAJIA, a state-of-the-art home-based software DSM system.

The rest of this chapter is organized as follows. Eight key issues of ULN are surveyed in the following section. According to these key issues, the communication requirements of software DSMs are analyzed in Section 3. Then the design of JMCL is proposed in Section 4. The current state of JMCL and future work are described in Section 5. Section 6 concludes this chapter.

7.2 Key Issues of ULN

Though getting the OS kernel out of the critical path of message passing is the basic idea of all ULN implementation, there are several important issues related to the performance and functionality of ULNs: communication model supported by ULN; whether data transfers are implemented via DMA or PIO; how multiplexing and protection are implemented; to implement true zero copy; where the addresses are translated; whether the NI implements message pipelining; how the destination is notified of a message arrival; whether communication is reliable; and whether multicast is supported[13]. In the next few subsections, we discuss these issues.

7.2.1 Communication Model

The communication model defines how the higher level software must interface with ULN. The most common communication modes are:*standard message passing*, *remote memory write*, and *handler-carrying message*.

Standard Message Passing In this communication model, a message send operation involves specifying the destination and a pointer to the data to be transferred. The receiver is responsible for determining where in memory the data should be stored via rendezvous-style, queue of buffers, and redirection. Under a rendezvous-style implementation, a data transfer only takes place when the message receive operation is executed, forcing sender and receiver to synchronize. This style is widely used in message passing programming environments, such as MPI. Under a queue of buffers implementation, all received messages are stored in a preallocated receive queue firstly, and the receiver may check the arrival of message by means of polling or interrupt mechanism. Software DSM systems always adopt this communication model; furthermore, they require the receiver to reply to the sender as soon as possible. Therefore, we name this communication model the *request-reply* model. The redirection mechanism sits between the former two approaches. Under this mechanism, the receiver transfers the data directly to its final destination, if the message receive operation has been executed when the data arrives. Otherwise, the data is transferred to a temporary buffer and later moved to its final destination when the message receive operation is executed, like VMMC-2, etc..

Remore Memory Write In this model, a communication setup phase creates a mapping between virtual memory addresses at the sender and receiver. A message send operation specifies the virtual address where to store the data at the destination. DEC's memory channel[42] and Princeton's VMMC adopts this model.

Handler-Carrying Message In this model, the message send operation specifies the address of a handler that should be executed at the receiver side upon message arrival. Both AM and FM adopt this model.

7.2.2 Data Transfer

There are two strategies to transfer messages from host memory to the NI and vice versa, i.e., via a DMA device or via programmed I/O (PIO) by the host itself. DMA operations are usually more efficient than PIO and free the host processor from spending time on data movement. For short messages, however, the DMA setup time may be as long as the data transfer time itself, making PIO more appropriate. In addition, since the DMA device only deals with physical addresses while user processes deal with virtual addresses, the user can not easily use DMA to transfer data straight to the NI. The usual approach is then to copy data to a buffer inside the OS kernel or to a pinned-down buffer before initiating the DMA operation. The memory copy generates extra overhead.

7.2.3 Protection

Since ULN give users direct access to the NI, protection is most important when multiple users must multiplex the NI. A straightforward solution to this problem is to use the virtual memory system to give each user access to a different part of NI memory, named endpoint entities. Each endpoint comprises all the queues and buffer areas used by a process. Protection is guaranteed by using the operating system to create two endpoints and to establish a connection between them. Since memory on NI is typically small, only a limited number of processes can be use this memory. To solve this problem, some systems use a caching technique, where inactive endpoints are stored in host memory.

7.2.4 Address Translation

As described in last subsection, though using PIO model can implement true zero-copy, the extra cost and keeping the host processor involved in the network operation for too long times make this is an unrealistic solution for long messages. To avoid this and yet achieve zero-copy transfer, several ULN proposed to dynamically pin (and unpin) user pages such that DMA transfer can be performed directly to those pages. The main implementation problem is that the NI needs to know the current virtual-to-physical mappings of individual pages. Several techniques were proposed to solve this problem. For example, BIP require users to translate the addresses, VMMC-2 and U-Net/MM let the kernel and NI cooperate such that the NI can keep track of valid address translations. More details can be found in related papers[91, 26, 10].

7.2.5 Message Pipelining

There can be up to four DMA operations involved during a message transfer: from host memory to NI memory, from NI memory to the network, from the network to NI memory, and from NI memory to host memory. One way of increasing throughput is to overlap multiple message transfers by pipelining DMA operations. We refer to this technique as inter-message pipelining.

Another technique extends this idea by overlapping different DMA operations belonging to a single message transfer. This strategy not only increases throughput but also reduces per message latency in the same way as wormhole routing does. This technique is called intra-message pipelining, such as Trapeze[].

7.2.6 Arrival Notification

Generally, the host processor may be notified by message arrival by polling flags or by receiving interrupts triggered by the NI. The tradeoff between these two mechanisms is message latency and overhead. Using polling, the host processor needs to check for message arrival by polling a status flag maintained by the NI. The question is at which frequency the host processor should poll. If it polls too frequently, it may waste time on unnecessary checks. If it polls too infrequently, it may result in increasing the message latency may increase greatly. Using interrupts, the host processor does not need to poll at all, however, servicing an interrupt is usually expensive. Several ULNs support both polling and interrupt mechanisms simultaneously.

7.2.7 Reliability

Higher-level software often requires reliable message delivery. Message transfers may be unreliable, however, due to unreliable network hardware and/or flow control (buffer overflow) problems. In the presence of unreliable hardware, the usual approach is to use timeouts, acknowledgement messages, and retransmission. When the hardware is reliable, as in the case of the Myrinet, hardware failures can be treated as fatal events. Therefore, the ULN may focus on preventing or avoiding buffer overflow problems by using window-based, credit-based, or back pressure flow control techniques. These techniques obviate the need for timeouts, but still require ACK/NACK messages and occasional retransmissions. The number of messages required by different communication substrate is shown in Figure 7.1. It can be seen that the number of messages will be reduced halfly in the context of reliable communication support with respect to unreliable counterpart.

7.2.8 Multicast

The last but not the least important issue of ULN is supporting of multicast. Multicast is an efficient mechanism in parallel computing, such as scatter/gather operation. The simple way to implement a multicast is to let the sender send a point-to-point message to each multicast destination. This scheme is inefficient because the startup cost is incurred for every multicast destination. To solve this problem, NI-supported multicasting approach has been proposed recently, where differences among different schemes lie in flow control strategy. For example, FM/MC uses a central buffer manager to keep track of the available buffer space at all receivers[9]. LFC uses an NI-level flow control scheme for multicast traffic. Barzos is the first system that expliots underlying multicast support [117].

Figure 7.1: Comparison of different communication substrate:(a). unreliable, (b). reliable

7.3 Communication Requirements of Software DSMs

Communication requirements are intimately related to coherence protocols. For example, in homeless system, such as CVM[68] and TreadMarks[71], scatter and gather operations are of great importance to propagate and apply diffs, while in home-based system, such as JIAJIA[51] and HLRC[54], this requirement is unnecessary. Fortunately, homeless and home-based software DSMs have very similar communication requirements except scatter/gather operations. In the rest of this chapter we therefore take the home-based JIAJIA system as our target software DSM system.

Most software DSM systems use the *request-reply* communication model[96]. They exhibit this behaviour for shared data, through getpage request and reply messages (communicating with home in JIAJIA) and sending diff request and diff apply completion messages (at release operation), as well as synchronization, through lock acquiring and lock granting messages and barrier arrival and departure messages. This communication model results in the following specific communication requirements.

1. **Sender** On the requesting side, the process sends a request message and immediately starts waiting for the corresponding reply. Therefore, a polling mechanism should be provided for message arrival notification. Moreover, the sender must respond to requests from other processes (act as a receiver) while waiting, or else deadlock can occur.

2. **Receiver** On the receiving side, when the process is notified about the arrival of a request, the appropriate handler extracts the request from the network, processes it and replies to requesting process. Although *interrupt* normally generates high system overheads, it is very difficult to implement a polling strategy that fit for asynchronous notification in software DSMs appropriately. This makes the interrupt is the best option for message notification at the receiver side.

3. **Flow control** Generally, high level softwares based on UDP/IP network protocol should provide flow control by themselves, since UDP/IP provides only unreliable, out-of-order, connectless communication. However, by analyzing of the request-reply model, we find that flow control is unnecessary in software DSMs. The reason for this is that there is

only one pending message for any node at any time[1], thus out-of-order transmission is impossible. Furthermore, the receiver buffer will not be overflowed if the receive buffer is larger than $(N + 1)$ elments, where N is the number of nodes.

4. **Reliablity** Since many software DSMs are implemented on top of UDP/IP protocol, which is unreliable, software DSMs must provide reliability by themselves. For example, the requesting node must wait for an acknowledge message from the receiver firstly, then start polling the normal replying message, as the dashed line shown in Figure 7.4. In fact, this acknowledging procedure can be avoided if the underlying communication system can guarantee reliable communication. As such, the communication overhead will be reduced greatly.

5. **Buffer space** As aforementioned, to provide reliable communication, timeout retransmission mechanism is used, which results in the requirement of a *sender queue* to reserve a message and one additional memory copy. In fact, both of these are unnecessary if the underlying communication system is reliable. The receive queue is inevitable. The size of this queue will be discussed later. In the context of UDP/IP network protocol, the message is transferred from NI (network interface) memory to physical host memory in kernel, and finally to the virtual memory of user space as shown in Figure 7.4. It can be seen that the kernel copy is unnecessary and time consuming. Therefore, the new communication substrate should support direct message transfers from NI to user space.

6. **Message type** Based on the run time statistics of message length[105], we observe that traffic pattern of software DSMs is bimodal with two main categories: *long* and *short*. Long messages include page reply, large diff, and synchronization reply, while short messages include synchronization request, page request, and diff grant message. The size of short message is less than 20 bytes, while that of large message is application-dependent. However, the hierarchy concept of network protocol encapsulates the content of the low level message, and treats both of them in the same way, which is inefficient. So the communication substrate should process them respectively. For short messages, the data will be sent to user process directly by NI, while storing at receiver buffer space is required for long messages. Some ULNs distinguish large and small messages at receiver side too, however, they only distinct by the PIO or DMA data transfer methods.

7.4 Design of JMCL

According to the communication requirements of software DSMs, we design a new communication substrate called JMCL (JIAJIA specific Myrinet Communication Library). Myrinet's high speed and programming interfaces at multiple levels inspire us to use Myrinet as our experimental platform [13, 18]. The role of JMCL is providing reliable, protected, in order message delivery for high level JIAJIA system, as shown in Figure 7.2. The JIAJIA communication module that uses JMCL is simpler and more efficient than the one based on UDP/IP. Our design of JMCL is closely related to VIA standard, as shown later.

[1] The reason for this is that message forwarding is not required in JIAJIA. In software DSMs that require forwarding of messages, the maximum number of pending messages is equal to the number of nodes.

Figure 7.2: Interface description of JMCL

7.4.1 JMCL API

JMCL's application programming interface (API) provided by is simple. JMCL provides 7 calls as listed in Table 7.1.

7.4.2 Message Flow of JMCL

Similar to VIA, the communication procedure of JMCL includes two phases: *communication initiation* and *data transfer*. In the communication initiation stage, which is accomplished by `JMCL_initialize()`, both the send and receive queue are preallocated by mapping a portion of physical memory. The memory space occupied by these queues will not be released until the end of the program by calling `JMCL_finalize()`. The sizes of the queues are determined by the arguments of `JMCL_initialize()`. Those queues can be accessed from both user space and kernel space, which results in the elimination of one memory copy as explained later.

The basic message flow diagram of JMCL at data transfer stage is shown in Figure 7.3; the message flow chart of UDP/IP is depicted in Figure 7.4 for comparison. With JMCL, the procedure of a message propagation is described as follows. Firstly, the high level application acquires a free position in the send queue, and fills the message. Secondly, the application calls `JMCL_isend()` to notify the underlying communication hardware to transmit this message, this operation is similar to the doorbell operation in VIA. Thirdly, this message will be copied from the send queue to a temporary buffer in the network interface card (NIC). Fourthly, at the receive side, the message will be buffered in the NIC too. After doing some checks, the message will be transmitted to the appropriate queue according to it's size. Finally, the corresponding process will be interrupted to process this message by appropriate message handler.

Considering the request-reply communication model, the sending process must wait for a reply message from other process many times. Therefore, JMCL provides a new polling mechanism to wait for this reply message. Upon the arrival of the reply message, JMCL copies this message to a customized reply buffer and sets a flag which is polled by JMCL_replypoll(). Since at any time, there is at most one pending message for a process, so the reply buffer size is set to the maximum message size. With this optimization, a user process interrupt is avoided.

JMCL exploits the software DSMs specific communication requirements greatly. At the

Table 7.1: Descriptions of JMCL Applications Programming Interface

API	Function description
`JMCL_initialize(JMCL_InitInfo*)`	Communication initialization function, which has the responsibility to create a port, register memory for the send and receive queue from kernel space, return corresponding memory handle, physical node number, process id, and port number to the caller. All the input and output information are stored in JMCL_InitInfo pointer.
`JMCL_finalize()`	Communication exit function. This function releases all communication related resources, such as communication port, registered kernel memory space, etc..
`JMCL_isend(int send_seqno, int recv_seqno, int len, int flag, int type, char* buffer)`	Non-blocking send function. JIAJIA uses this function to post send request, which is similar to the doorbell mechanism of VIA. Whether or not the message can be sent out directly, this function will return immediately. A message identifier is returned by this function. To save the overhead of address translation, physical node number and port number are used to appoint the destination directly. The motivation to leave the address translation to high level software is that this overhead in underlying software will dilate the occupancy of critical resources, which results in low communication bandwidth and long latency.
`JMCL_msgdone(int msgid)`	Checks the state of sending message. *msgid* is the message identifier to be checked. If the message has been sent successfully, 0 is returned, otherwise it will return -1.
`JMCL_isend_instant(int send_seqno, int recv_seqno, int len, int flag, int type, int data)`	Direct-data send function. This function is designed for small messages specifically.
`JMCL_replypoll(JMCL_replyrecvinfo *replyrecvinfo)`	Polls for the reply of pending request. Exits when the reply message is received. This call is provided specifically for the request-reply model. This function should be called after the send functions at the sender side. As such, a signal handler can be avoided.
`JMCL_recv(JMCL_recvinfo *recvinfo)`	Collects the information about the receiving message. This call should be called in the SIGIO handler at the receiving side. All the information about the message, such as the sequence number of sender, message size, message flag, message type, message content pointer, etc. will be stored in *recvinfo*.

Figure 7.3: Message transfer flow in JMCL

Figure 7.4: Message transfer flow in UDP/IP

sender side, the *Outqueue* is removed because the reliability is provided by underlying JMCL, so that the message copy from user space to outqueue becomes unnecessary. Here the *sendqueue* is located in the middle of user space and kernel space, and can be accessed by the Myrinet driver directly. The sendqueue is used to assemble message, and has the same role as *malloc* in the Figure 7.4. `JMCL_isend` directly transfers the message from the sendqueue to the memory on network interface card directly. As such, the two extra memory copy operations in UDP/IP. At the receiver side, the inqueue is divided into two queues: *Lqueue* for large message, and *Squeue* for small messages. Incoming messages are inserted into *Lqueue* or *Squeue* according to their message type by the underlying MCP (Myrinet Control Program) on the Myrinet NI card. As shown in Figure 7.3 and Figure 7.4, JMCL stores the inqueue in a different location than UDP/Ip. To reduce the number of message copies, JMCL pins the inqueue into physical memory which can be used by both driver and high level software, while UDP/IP stores inqueue in user space. When the receiver is notified that a message was arrived, it calls the corresponding message server directly, since it knows the message has been put into the head of Lqueue or Squeue. Lqueue and Squeue have corresponding message servers. *Smsgserver* bears responsibility for small messages; *lmsgserver* handles large messages. The distinction between small and large messages reduces the DMA time from the network interface to kernel space. As such, the *recv* call in the UDP/IP protocol is removed from the critical path too. Consequently, the numbers of memory copies will be reduced from 7 to 2 in our new JMCL. Furthermore, the interface between communication substrate and software DSMs becomes simple, as shown in Figure 7.3.

To support multiple users, JMCL provides an efficient way for protection, which is very similar to the VIA standard. Memory is registered on a per process basis. The memory space registered to one process will be associated with a unique memory handles. With the protection of memory handler, one process cannot use another process's memory handle to access the same memory region. An attempt to do so will result in a memory protection error.

In summary, JMCL has following six characteristics: (1) It is a software DSM specfic user level communication library; (2) Without the requirement of flow control and time-out retransmission makes the JMCL iteself thiner than other ULNs; (3) It is very easy to implement the address translation between virtual address and physical address because of the fixed size of the send and receive queue; (4) Distinction between direct instant and long message; (5) Avoiding the extra signal at request send side by polling the network interface directly; (6) Keep close to VIA standard.

7.5 Current State and Future Work

Our experimental hardware platform contains 8 nodes each with a 300MHz PowerPC 604 processor, 256MB DRAM memory, and running the AIX 4.2 operating system. These nodes are connected through a Myrinet switch and Fast Ethernet. Since Myricom's products do not support AIX, we design the MCP and driver by ourselves. Currently, we have constructed the basic framework of JMCL; the final library will be completed in the autumn of 1999. We will evaluate the raw performance of JMCL with several widely used microbenchmarks. In particular, the improvement of the performance of applications built on JIAJIA will be evaluated, because there is few results about the effects of user-space communication to high level applications in the past. The expected raw performance of JMCL is 30 microseconds for one way latency (including the interrupt overhead of remote process, about 10 microseconds), and the round-trip time is 50 microseconds. The first version (JMCL 1.0) considers the basic communication requirements only, the coupling between the coherence protocol and the communication substrate will be taken into account in the next version, such as supporting Remote DMA (RDMA) write/read to send diff, and fetching a page from a remote home etc. In fact, RDMA read/write operations are required by the VIA standard too.

On the other hand, we have proposed a new multicast scheme which is easy to implement on the network interface card recently[140]. We plan to implement this scheme in the future version of JMCL.

7.6 Conclusion

We argue that specific communication substrate for software DSMs should be an important research direction for both software DSMs and user level networking area. According to the specific communication requirements of software DSMs, we take the challenge to design a low level communication library (JMCL) here. The basic idea and detailed application programming interface of JMCL are presented in this paper.

Chapter 8

Conclusions and Future Directions

In this dissertation, we investigated how to improve the performance of software DSM system. Several techniques have been proposed to improving the performance from six aspects, such as cache coherence protocol, memory organization scheme, system overhead, loop scheduling and task migration, and communication optimization etc.. The main conclusion of this research is that the performance of software DSM system can be significantly improved by these solutions. As such, software DSM has becoming the ideal vehicle for scientific applications with regular character, and the ideal target for parallelizing compilers for distributed memory environment. However, it becomes obvious that there is still a performance gap between hardware shared memory and software shared memory approaches, between software DSM and message passing counterpart, such as MPI, PVM. More research is needed, especially in hardware support and application design, before the coming of shine sky for software DSM system.

By analyzing the disadvantages of snoopy and directory based cache coherence protocols and several release consistency protocols, we proposed a novel lock-based coherence protocol for scope consistency. The unique characteristic of this new protocol is the use of "home" concept not only for data information, but also for coherence information, such as *write notices*. Generally, home-based software DSM systems propagate data information and apply them at homes eagerly, but this information is fetched by others lazily. However, these systems propagate coherence information and apply them either eagerly or lazily. In our lock-based cache coherence protocol, coherence information is processed in a similar way to data information, and each coherence information has a static home according to the corresponding synchronization object (e.g., a lock or a barrier manager). So, coherence information is propagated to the corresponding home at release time, and is lazily fetched by the next acquirer of the same synchronization object. Compared with directory-based protocols, all coherence related actions in the protocol are applied at synchronization points. In this way, the lock-based protocol has least coherence related overheads for ordinary read or write misses. Moreover, the lock-based protocol is free from the overhead of maintaining the directory.

Based on this new protocol, we designed a software DSM system named JIAJIA, which is simple but very efficient. Unlike other software DSMs that adopt the COMA-like memory architecture, JIAJIA organizes the shared memory in a NUMA-like way. In JIAJIA, each shared page has a home node and homes of shared pages are distributed across all nodes. References to local part of shared memory always hit locally. References to remote shared pages cause these pages to be fetched from its home and cached locally. When cached, the remote page is kept at the same user space address as that in its home node. In this way, shared address of a page is identical in all processors, no address translation is required on a remote access.

In order to find the main obstacles which prevent software DSMs from becoming the acceptable parallel programming environment, we performed a detailed analysis about the system overhead. We observed that remote data penalty and synchronization overheads are two culprits of large system overhead, therefore, we proposed several techniques to reduce system overhead efficiently, such as read notice, hierarchical barrier implementation, cache only write detection (CO-WD), etc.. The evaluation results show that the performance is improved greatly with our optimization schemes.

Since the data distribution plays great important role in home-based software system, we proposed and designed a novel home migration mechanism to reduce the remote data communication. The basic idea behind our home migration is migrating the home of a page to a new host on a barrier only if the new home host is the single writer of the page during the last barrier interval. In cooperation with JIAJIA's novel memory organization scheme, no page copy is required at home migration phase. Evaluation shows home migration can reduce diffs dramatically and improve performance significantly for some applications.

As the prevalence of metacomputing environment, loop scheduling and load balancing are critical issues in order to achieve high performance. We proposed a new affinity-based self scheduling (ABS) scheme for load balancing in home-based software DSM systems. ABS takes the static data allocation into account during loo scheduling. All processes share a global queue but with different segments to keep processor affinity. The main difference between ABS algorithm and traditional central queue based scheduling schemes lies in the organization and management of the central queue. In our scheme, the queue is partitioned among these processors, and each processor first accesses it's own part. During the local scheduling phase, each processor obtains task from it's initially allocated segment. When load imbalance occurs, the idle processor obtains the task from the segment which belongs to the most heavily loaded processor. Compared with affinity scheduling, the synchronization overhead is reduced significantly.

For iterative scientific applications, we designed a novel dynamic task migration scheme, which can balance the load efficiently. Firstly, a new definition about task is given. In distributed computing environment, the data distribution plays great important role in final performance. Therefore, a task is the sum of *computation subtask* and *data subtask* Based on this observation, we propose a task migration scheme which executes computation migration and data migration sequentially. Evaluation results show Task migration performs better than traditionally computation migration greatly.

Based on the observation that almost all available software DSMs use UDP/IP network protocol as communication substrate, which is time consuming and unreliable. Thus, the high level software DSMs must manage the flow control, order of packet, and error control by themselves, which wasting more unnecessary time. In fact, these extra communication latencies associated with each communication can be avoided by using user-space communication. Though many user-space communication libraries are proposed and claimed to support reliable, ordered, low latency (zero or one copy) communication, unfortunately, these user-space communication libraries were designed for standard message passing, remote memory write, handler carrying message communication models only, many of them even did not support interrupt-based notification. In other words, many communication characteristics of software DSMs are not taken into consideration when designing. Currently, we are developing a Myrinet-based user level communication substrate called JMCL specific for JIAJIA. JMCL provides reliable, protected, in order message delivery for high level JIAJIA system. With JMCL, the communication module of JIAJIA system become more simple and efficient than that in traditional UDP/IP protocol. Our design of JMCL is closely related to VIA standard.

Traditionally, to demonstrate the advantages of the new idea, several favorite small benchmarks are used in the evaluation, such as SPLASH or SPLASH2, NAS Parallel Benchmark. Therefore, the confidence of their results is suspicious. We propose a confidence interval based evaluation method to analyze the experimental results. Furthermore, I have ported one real world applications, i.e., Electromagnetic Simulation, to JIAJIA system successfully.

8.1 Future of Software DSM

The research and development of software DSM system have past 14 years, there are thousands of research papers and dozens of theses. However, the real world impact of software DSMs is not as image as researchers. Why? Up to now, as the maturity of coherence protocol and memory consistency model, and the performance gap between software DSMs and its message passing counterpart is becoming more and more close, in my opinion, the portability of the application programming interface is the vital factor of the future of software DSMs. So support OpenMP[36] is the necessity for future systems.

On the other hand, since the idea behind software DSMs is encapsulating underlying communication from high level applications. Therefore, the burden of exploiting the maximum performance of communication substrate will lay on the designer of software DSMs. The work in the dissertation just want to validate our idea, further energy should be put on this issue in the future. As the standardization of user level network interface, i.e., Virtual Interface Architecture (VIA)[22], study the specific communication layer for software DSMs is significant.

Currently, the application area of software DSMs is limited to scientific applications, which prevents the prevalence of software DSMs greatly. In fact, the single address space provided by software DSM system makes it be an ideal substrate for distributed services, such as file systems, databases, DNS, etc., and wide area applications, such as e-business, groupware, etc. These new applications will benefit greatly from the many mature techniques in software DSM area. For example, *automatic data distribution* makes location transparency and global memory management become possible; *transparent coherence management* makes shared data automatically kept coherent. Meanwhile, as the wide adoption of object-oriented paradigm in these applications, the task and corresponding data are tightly coupled and deterministic, so that high performance can be achieved easily.

I'd like to use following sentence as the conclusion of my dissertation: *"the goal of software DSM not noly depend on* **performance***, but also on* **Persistence, Interoperability, Security, Resource Management, Scalablity, and Fault Tolerance***"*.

Bibliography

[1] A. Agarwal, R. Bianchini, D. Chaiken, F. T. Chong, K. L. Johnson, D. Kranz, J. D. Kubiatowicz, B-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):430–444, March 1999.

[2] A. Agarwal, R. Simoni, J. L. Hennessy, and M. Horowitz. An evaluation of directory scheme for cache coherence. In *Proc. of the 15th Annual Int'l Symp. on Computer Architecture (ISCA'88)*, pages 280–289, May 1988.

[3] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[4] C. Amza, A. L. Cox, S. Dwarkadas, L-J. Jin, K. Rajamani, and W. Zwaenepoel. Adaptive protocols for software distributed shared memory. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):467–475, March 1999.

[5] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, and J. Philbin. Use-space communication: A quantitative study. In *Proc. of Supercomputing'98*, November 1998.

[6] J. Archibald and J. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4:273–298, November 1986.

[7] D. Bailey, J. Barton, T. Laninski, and H. Simon. The NAS parallel benchmarks. Technical Report TR #103863, NASA, USA, July 1993.

[8] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, and T. Ruhl. Performance evaluation of the Orca shared-object system. *ACM Trans. on Computer Systems*, 16(1):1–40, February 1998.

[9] H. E. Bal, R. A. F. Bhoedjang, R. Hofman, C. Jacobs, K. G. Langendoen, and K. Verstoep. Performance of a high-level parallel language on a high speed network. *Journal of Parallel and Distributed Computing*, 40(1):49–64, 1997.

[10] A. Basu, M. Welsh, and T. von Eicken. Incorporating memory management into user-level network interfaces. In *Hot Interconnects'97*, April 1997.

[11] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 528–537, February 1993.

[12] R. Bhoedjang, T. Rühl, and H. E. Bal. Efficient multicast on Myrinet using link-level flow control. In *Proc. of the 1998 Int'l Conf. on Parallel Processing (ICPP'98)*, August 1998.

[13] R. A. F. Bhoedjang, T. Rühl, and H. E. Bal. User-level network interface protocols. *IEEE Computer Magzine*, pages 53–68, November 1998.

[14] R. Bianchini, L. I. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C. L. Amorim. Hiding communication latency and coherence overhead in software DSMs. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, pages 198–209, October 1996.

[15] A. Bilas. *Improving the Performance of Shared Virtual Memory on System Area Networks.* PhD thesis, Dept. of Computer Science, Princeton University, November 1998.

[16] A. Bilas, C. Liao, and J. P. Singh. Accelerating shared virtual memory using commodity NI support to avoid asynchronous message handling. In *Proc. of the 26th Annual Int'l Symp. on Computer Architecture (ISCA'99)*, May 1999.

[17] M. A. Blumrich, R. D. Albert, Y. Chen, D. W. Clark, S. N. Damianakis, C. Dubnicki, E. W. Felten, L. Iftode, K. Li, M. Martonosi, and R. A. Shillner. Design choices in the SHRIMP system: An empirical study. In *Proc. of the 25th Annual Int'l Symp. on Computer Architecture (ISCA'98)*, June 1998.

[18] N. J. Boden and et. al. Myrinet:a Gigabit-per-second local area network. *IEEE Micro*, pages 29–36, February 1995.

[19] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP-13)*, pages 152–164, October 1991.

[20] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *ACM Trans. on Computer Systems*, 13(3):205–243, August 1995.

[21] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache-coherence in large-scale multiprocessors. *IEEE Computer*, 23(6):49–58, June 1990.

[22] Compaq, Intel, and Microsoft. Virtual interface architecture specification version 1.0. Technical report, http://www.viarch.org, December 1997.

[23] A. Cox, E. de Lara, and W. Zwaenepoel Y. C. Hu. A performance comparison of homeless and home-based lazy release consistency protocols for software shared memory. In *Proc. of the 5th IEEE Symp. on High-Performance Computer Architecture (HPCA-5)*, January 1999.

[24] A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software versus hardware shared-memory implementation: A case study. In *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94)*, pages 106–117, April 1994.

[25] D. Culler, L. T. Liu, R. P. Martin, and C. Yoshikawa. LogP performance assessment of fast network interfaces. *IEEE Micro*, 1996.

[26] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient support for reliable, connection-oriented communication. In *Hot Interconnects'97*, April 1997.

[27] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and implementation of virtual memory-mapped communication on Myrinet. In *Proc. of the 11th Int'l Parallel Processing Symp. (IPPS'97)*, pages 388–396, April 1997.

[28] M. Dubois, J-C. Wang, L. A. Barroso, K. Lee, and Y-S. Chen. Delayed consistency and its effects on the miss rate of parallel programs. In *Proc. of Supercomputing'91*, pages 197–206, November 1991.

[29] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, pages 186–197, October 1996.

[30] S. Dwarkadas, A. Schaffer, R. W. Cottingham, A. L. Cox, P. Keleher, and W. Zwaenepoel. Parallelization of general linkage analysis problems. *Human Heredity*, 44:127–141, July 1994.

[31] D. L. Eager and J. Zahorjan. Adaptive guided self-scheduling. Technical Report CS-87-11, Department of Computer Science and Engineering, University of Washington, January 1987.

[32] T. Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP-15)*, December 1995.

[33] T. Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, May 1992.

[34] A. Erlichson, N. Nuckolls, G. Chesson, and J. L. Hennessy. SoftFLASH: Analyzing the performance of clustered distributed virtual shared memory. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, pages 210–220, October 1996.

[35] M. R. Eskicioglu, T. A. Marsland, W. Hu, and W. Shi. Evaluation of JIAJIA software DSM system on high performance computer architectures. In *Proc. of the 32st Hawaii Int'l Conf. on System Sciences (HICSS-32) CD-ROM*, pages 287, file: stdcr05.ps, January 1999.

[36] SGI et.al. OpenMP specification version 1.0. Technical report, http://www.openmp.org, October 1997.

[37] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood. Application-specific protocols for user-level shared memory. In *Proc. of Supercomputing'94*, pages 380–389, November 1994.

[38] B. Falsafi and D. A. Wood. Reactive NUMA: A design for unifying S-COMA and CC-NUMA. In *Proc. of the 24th Annual Int'l Symp. on Computer Architecture (ISCA'97)*, pages 229–240, June 1997.

[39] B. D. Fleisch. *Distributed Shared Memory in a Loosely Coupled Environment.* PhD thesis, Department of Computer Science, University of California in Los Angeles, 1989.

[40] K. Gharachorloo, A. Gupta, and J. L. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proc. of the 1991 Int'l Conf. on Parallel Processing (ICPP'91)*, volume I, pages 355–364, August 1991.

[41] K. Gharachorloo, D. E. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 15–26, May 1990.

[42] R. B. Gillett. Memory channel network for PCI. *IEEE Micro*, 16(1):12–18, February 1996.

[43] S. Gjessing, D. Gustavson, J. Goodman, D. James, and E. Kristiansen. *The SCI Cache Coherence Protocol*, pages 219–237. Kluwer Academic Publishers, 1991.

[44] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherence Interface Working Group, March 1989.

[45] C. Holt, M. Heinrich, J. P. Singh, E. Rothberg, and J. L. Hennessy. The performance effects of latency, occupancy and bandwidth in cache-coherent DSM multiprocessors. In *Proc. of the Fifth Workshop on Scalable Shared Memory Multiprocessors*, June 1995.

[46] W. C. Hsieh, P. Wang, and W. E. Weihl. Computation migration: Enhancing locality for distributed-memory parallel systems. In *Proc. of the Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'93)*, May 1993.

[47] W. Hu, W. Shi, and Z. Tang. A framework of memory consistency models. *Journal of Computer Science and Technology*, 13(2):110–124, March 1998.

[48] W. Hu, W. Shi, and Z. Tang. JIAJIA software DSM system. Technical report, Institute of Computing Technology, Chinese Academy of Sciences, TR980004, October 1998.

[49] W. Hu, W. Shi, and Z. Tang. A lock-based cache coherence protocol for scope consistency. *Journal of Computer Science and Technology*, 13(2):97–109, March 1998.

[50] W. Hu, W. Shi, and Z. Tang. Home migration in home-based software DSMs. In *Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*, June 1999.

[51] W. Hu, W. Shi, and Z. Tang. JIAJIA: An SVM system based on a new cache coherence protocol. In *Proc. of the High-Performance Computing and Networking Europe 1999 (HPCN'99)*, pages 463–472, April 1999.

[52] W. Hu, W. Shi, and Z. Tang. Reducing system overheads in home-based software DSMs. In *Proc. of the Second Merged Symp. IPPS/SPDP 1999)*, pages 167–173, April 1999.

[53] S. E. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A practical and robust method for scheduling parallel loops. *Communication of ACM*, 35(8):90–101, August 1992.

[54] L. Iftode. *Home-based Shared Virtual Memory.* PhD thesis, Dept. of Computer Science, Princeton University, June 1998.

[55] L. Iftode, M. Blumrich, C. Dubnicki, D. L. Oppenheimer, J. P. Singh, and K. Li. Shared virtual memory with automatic update support. In *Proc. of the 13th ACM-SIGARCH Int'l Conf. on Supercomputing (ICS'99)*, June 1999.

[56] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving release-consistent shared virtual memory using automatic update. In *Proc. of the 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA-2)*, February 1996.

[57] L. Iftode and J. P. Singh. Shared virtual memory: Progress and challenges. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):498–507, March 1999.

[58] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pages 277–287, June 1996.

[59] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. *Theory of Computing Systems*, 31(4):451–473, July/August 1998.

[60] S. Ioannidis and S. Dwarkadas. Compiler and run-time support for adaptive load balancing in software distributed shared memory systems. In *Proc. of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, May 1998.

[61] A. Itzkovitz and A. Schuster. Multiview and millipage: Fine-grain sharing in page-based DSMs. In *Proc. of the 3rd Symp. on Operating Systems Design and Implementation (OSDI'99)*, pages 215–228, February 1999.

[62] A. Itzkovitz, A. Schuster, and L. Shalev. Thread migration and its applications in distributed shared memory systems. *The Journal of Systems and Software*, 47(1):71–87, July 1998.

[63] D. Jiang, B. Cokelley, X. Yu, A. Bilas, and J. P. Singh. Scalability of home-based shared virtual memory on clusters of SMPs. In *Proc. of the 13th ACM-SIGARCH Int'l Conf. on Supercomputing (ICS'99)*, June 1999.

[64] D. Jiang, H. Shan, and J. P. Singh. Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors. In *Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 217–229, June 1997.

[65] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP-15)*, pages 213–228, December 1995.

[66] A. Karlin, M. Manasse, L. Rudolph, and D. Sleator. Competitive snoopy caching. In *Proc.of the 27th Annual Symposium on Foundations of Computer Science*, pages 244–254, 1986.

[67] M. Karlsson and P. Stenstrom. Effectiveness of dynamic prefetching in multiple-writer distributed virtual shared memory systems. *Journal of Parallel and Distributed Computing*, 43(2):79–93, June 1997.

[68] P. Keleher. The relative importance of concurrent writers and weak consistency models. In *Proc. of the 16th Int'l Conf. on Distributed Computing Systems (ICDCS-16)*, pages 91–98, May 1996.

[69] P. Keleher. Symmetry and performance in consistency protocols. In *Proc. of the 13th ACM-SIGARCH Int'l Conf. on Supercomputing (ICS'99)*, June 1999.

[70] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.

[71] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[72] L. I. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, Jr., S. Dwarkadas, and M. L. Scott. VM-based shared memory on low-latency, remote-memory-access networks. In *Proc. of the 24th Annual Int'l Symp. on Computer Architecture (ISCA'97)*, pages 157–169, June 1997.

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

[74] G. Lathtop and et.al. Strategies for multilocus analysis in humans. *PNAS*, 81:3443–3446, 1994.

[75] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proc. of the 24th Annual Int'l Symp. on Computer Architecture (ISCA'97)*, pages 241–251, June 1997.

[76] D. E. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. L. Hennessy, M. Horowitz, and M. Lam. Design of the Stanford DASH multiprocessor. Technical Report CSL-TR-89-403, Computer Systems Laboratory, Stanford University, December 1989.

[77] D. E. Lenoski, J. Ludon, K. Gharachorloo W-D. Weber, A. Gupta, J. L. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[78] K. Li. Ivy: A shared virtual memory system for parallel computing. In *Proc. of the 1988 Int'l Conf. on Parallel Processing (ICPP'88)*, volume II, pages 94–101, August 1988.

[79] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems*, 7(4):321–359, November 1989.

[80] T-Y. Liang, C-K. Shieh, D-C. Liu, and W. Zhu. Dynamic task scheduling on multithreaded distributed shared memory systems. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, volume II, pages 1058–1065, July 1998.

[81] J. Liu, V. A. Saletore, and T. G. Lewis. Safe self-scheduling: A parallel loop scheduling scheme for shared memory multiprocessors. *Int'l Journal of Parallel Programming*, 22(6):589–616, June 1994.

[82] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Message-passing vs. distributed shared memory on networks of workstations. In *Proc. of Supercomputing'95*, December 1995.

[83] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Quantifying the performance differences between PVM and treadmarks. *Journal of Parallel and Distributed Computing*, 43(2):65–78, June 1997.

[84] E. Markatos and T. Le Blanc. Using processor affinity in loop scheduling on shared memory multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[85] E. P. Markatos and T. J. LeBlanc. Load balancing vs locality management in shared memory multiprocessors. In *Proc. of the 1992 Int'l Conf. on Parallel Processing (ICPP'92)*, August 1992.

[86] W. Meira Jr., T. J. LeBlanc, N. Hardavellas, and C. Amorim. Understanding the performance of DSM applications. In *Proc. of the First Int'l Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC'97)*, pages 198–211, February 1997.

[87] T. C. Mowry, C. Chan, and A. Lo. Comparative evaluation of latency tolerance techniques for software distributed shared memory. In *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, pages 300–311, February 1998.

[88] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstation: Illinois fast messages(FM) for Myrinet. In *Proc. of Supercomputing'95*, December 1995.

[89] M. Papamarcos and J. Patel. A low overhead coherence solution for multiprocessors. In *Proc. of the 11th Annual Int'l Symp. on Computer Architecture (ISCA'84)*, pages 348–254, June 1984.

[90] C. Polychronopoulos and D. Kuck. Guided self-scheduling: A practical self-scheduling scheme for parallel supercomputers. *IEEE Transcations on Computers*, 36(12):1425–1439, December 1987.

[91] L. Prylli and B. Tourancheau. BIP: A new protocol designed for high performance netwroking on Myrinet. In *Proc. of the First Merged Symp. IPPS/SPDP 1998)*, March 1998.

[92] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94)*, pages 325–337, April 1994.

[93] D. P. Rodohan and S. R. Scuinders. Parallel implementations of the finite-difference time- domain (FDTD) method. In *Proceedings of the 2nd Int. Conference Computation in Electromagnetics*, pages 367–370, April 1994.

[94] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. Home-based SVM protocols for SMP clusters: Design and performance. In *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, February 1998.

[95] H. S. Sandhu, T. Brecht, and D. Moscoso. Multiple writers entry consistency. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, volume I, pages 355–362, July 1998.

[96] R. Santos, R. Bianchini, and C. L. Amorim. A survey of messaging software issues and systems for Myrinet-based clusters. *Parallel and Distributed Computing Practices, Special Issue on High Performance Computing on Clusters*, May 1999.

[97] A. Saulsbury, T. Wilkinson, J. B. Carter, and A. Landin. An argument for simple coma. In *Proc. of the 1st IEEE Symp. on High-Performance Computer Architecture (HPCA-1)*, pages 276–285, January 1995.

[98] D. J. Scales and K. Gharachorloo. Shasta: a system for supporting fine-grain shared memory across clusters. In *Proc. of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[99] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-grain software distributed shared memory on SMP clusters. In *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, February 1998.

[100] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, pages 174–185, October 1996.

[101] A. Schaffer and et.al. Avoiding recomputation in generic linkage analysis. *Human Heredity*, 44:225–237, 1994.

[102] I. Schoinas, B. Falsafi, M. D. Hill, J. R. Larus, and D. A. Wood. Sirocco: Cost-effective fine-grain distributed shared memory. In *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT'98)*, October 1998.

[103] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *Proc. of the 6th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVI)*, pages 297–307, October 1994.

[104] C. B. Seidel, R. Bianchini, and C. L. Amorim. The affinity entry consistency protocol. In *Proc. of the 1997 Int'l Conf. on Parallel Processing (ICPP'97)*, pages 208–217, August 1997.

[105] W. Shi, W. Hu, and Z. Tang. Where does the time go in software DSM system?–experiences with JIAJIA. *Journal of Computer Science and Technology*, 14(3):193–205, May 1999.

[106] W. Shi, W. Hu, Z. Tang, and M. R. Eskicioglu. Dynamic task migration in home-based software DSM systems. Technical Report TR990004, Institute of Computing Technology, Center of High Performance Computing, February 1999.

[107] W. Shi and J. Ma. High efficient parallel computation of resonant frequencies of waveguided loaded cavetives on JIAJIA software DSM system. In *Proc. of the High-Performance Computing and Networking Europe 1999 (HPCN'99)*, April 1999.

[108] W. Shi, Y. Mao, and Z. Tang. Communication substrate for software DSMs. In *Proc. of the 11th IASTED Int'l Conf. on Parallel and Distributed Computing and Systems (PDCS'99)*, November 1999.

[109] W. Shi and Z. Tang. Affinity-based self scheduling for software shared memory systems. In *Proc. of the 6th Int'l Conf. on High Performance Computing (HiPC'99)*, December 1999.

[110] W. Shi and Z. Tang. Using confidence interval to summarize the evaluting results of DSM systems. *Journal of Computer Science and Technology*, 14(6), November 1999.

[111] W. Shi and Z. Tang. A more practical loop scheduling for home-based software dsms. *Accepted by Journal of Scheduling*, 2000.

[112] W. Shi, Z. Tang, and W. Hu. A more practical loop scheduling for home-based software DSMs. In *Proc. of the ACM-SIGARCH Workshop on Scheduling Algorithms for Parallel and Distributed Computing–From Theory to Practice*, June 1999.

[113] R. Simoni and M. Horowitz. Dynamic pointer allocation for scalable cache coherence directories. In *Proceedings of International Symposium on Shared Memory Multiprocessors*, pages 72–81, April 1991.

[114] J. P. Singh, Wolf-Doetrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared memory. *ACM Computer Architecture News*, 20(1):5–44, March 1992.

[115] W. E. Speight, H. Abdel-Shafi, and J. K. Bennett. Realizing the performance potential of the virtual interface architecture. In *Proc. of the 13th ACM-SIGARCH Int'l Conf. on Supercomputing (ICS'99)*, June 1999.

[116] W. E. Speight and J. K. Bennett. Brazos: A third generation DSM system. In *Proc. of the USENIX Windows NT Workshop*, August 1997.

[117] W. E. Speight and J. K. Bennett. Using multicast and multithreading to reduce communication in software DSM systems. In *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, pages 312–322, February 1998.

[118] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.

[119] P. Stenstrom, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 80–91, May 1992.

[120] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and Michael Scott. Cashmere-2l: Software coherent shared memory on a clustered remote-write network. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, October 1997.

[121] S. Subramaniam and D. L. Eager. Affinity scheduling of unbalanced workloads. In *Proc. of Supercomputing'94*, pages 214–226, November 1994.

[122] M. Swanson, L. Stroller, and J. B. Carter. Making distributed shared memory simple, yet efficient. In *Proc. of the 3rd Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'98)*, pages 2–13, March 1998.

[123] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation, 2nd Edition*, chapter 3. Prentice Hall. Inc., 1997.

[124] P. Tang and P.-C. Yew. Processor self-scheduling for multiple nested parallel loops. In *Proc. of the 1986 Int'l Conf. on Parallel Processing (ICPP'86)*, pages 528–535, August 1986.

[125] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: An operating system coordinated high performance communication library. In *Proc. of the High-Performance Computing and Networking Europe 1997 (HPCN'97)*, pages 708–717, April 1997.

[126] C. Thacker, L. Stewart, and E. Satterthwaite. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[127] M. Thapar, B. A. Delagi, and M. J. Flynn. Scalable cache coherence for shared memory multiprocessors. In *Proc. of the 1st Int'l ACPC Conference*, pages 1–12, September 1992.

[128] K. Thitikamol and P. Keleher. Multi-threading and remote latency in software DSMs. In *Proc. of the 17th Int'l Conf. on Distributed Computing Systems (ICDCS-17)*, May 1997.

[129] K. Thitikamol and P. Keleher. Thread migration and communication minimization in DSM systems. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):487–497, March 1999.

[130] J. Torrellas and D. Padua. The Illinois aggressive coma multiprocessor project (I-ACOMA). In *Proc. of the 6th Symp. on the Frontiers of Massively Parallel Computing (Frontiers'96)*, October 1996.

[131] T. H. Tzen and L. M. Ni. Trapezoid self-scheduling:a practical scheduling scheme for parallel compilers. *IEEE Trans. on Parallel and Distributed Systems*, 4(1):87–98, January 1993.

[132] T. v. Eicken and W. Vogels. Evaluation of the virtual interface architecture. *IEEE Computer Magzine*, pages 61–68, November 1998.

[133] V. Varadarajan and R. Mittra. Finite-difference time-domain analysis using distributed computing. *IEEE Microwave Guided Wave Letters*, 4(5):144–145, May 1994.

[134] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Charachterization and methodological considerations. In *Proc. of the 22nd Annual Int'l Symp. on Computer Architecture (ISCA'95)*, pages 24–36, June 1995.

[135] Y. Yan, C. Jin, and X. Zhang. Adaptively scheduling parallel loops in distributed shared memorysystems. *IEEE Trans. on Parallel and Distributed Systems*, 8(1):70–81, January 1997.

[136] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: A multigrain shared memory system. In *Proc. of the 23rd Annual Int'l Symp. on Computer Architecture (ISCA'96)*, pages 45–55, May 1996.

[137] W. M. Yu and A. L. Cox. Java/DSM: a platform for heterogeneous computing. In *Proc. of Java for Computational Science and Engineering–Simulation and Modeling Conf.*, pages 1213–1224, June 1997.

[138] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad. Software write detection for a distributed shared memory. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation (OSDI'94)*, pages 87–100, November 1994.

[139] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared memory virtual memory systems. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, pages 75–88, October 1996.

[140] Zhiyu Zhou, Weisong Shi, and Zhimin Tang. A novel multicast algorithm in parallel systems. Technical Report TR990009, Division of Computer Systems, Institute of Computer Technology, CAS, September 1999.