University of Alberta

Software Distributed Shared Memory: Issues and a Case Study

by

Mehmet Raşit Eskicioğlu ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Fall 2004

Your file Votre référence
ISBN: 0-612-95927-9
Our file Notre référence
ISBN: 0-612-95927-9

Perfection is not achieved when there is nothing left to add,
but when there is nothing left to take away.

St. Exupery's Dictum

To my dearest wife Hülya and my lovely daughters Pınar and Çağla,
for standing by me far too many years...

Acknowledgements

This has been a very long journey... So, I may not able to remember every name. I would like to start expressing my appreciation to my original supervisor, Dr. Wlodek Dobosiewicz for his endless and highly inspiring and motivating ideas. I also would like to thank my current supervisors and mentors Dr. Tony Marsland and Dr. Pawel Gburzynski. Dr. Marsland has been and still is my "virtual writing" coach. It is for his continuous advise that there is just a few if any sentence in this dissertation that ends with a word which also exists in the sentence itself. Along with Dr. Dobosiewicz, Dr. Gburzynski has been supportive from the very early days. I cannot forget their support.

I was the president of the 1989-90 CSGSA executive with Leigh Willard as VP and Jean Leroux as Treasurer. It was a great pleasure working with you guys. I also remember the good old days, when I shared offices with Franco Carlacci, George Ferguson, Yiang-Leng Chang, Iqbal Goralwalla, Dimitry Grodnichy, Andreas Junghanns, Yngvi Bjornsson, and various labs with Ken Barker, Randal Peters, Wade Holst, and Kalandhar Voruganti. Further, other friends and colleagues that I can remember are Mike MacGregor, Gordon Atwood, Abdul Sattar, Cesur Baransel, Afsal Upal, and Diego Novillo, all of whom have finished their studies long before me. I (shamefully) thank you all, and the others whom I forgot to mention here, for your company and friendship.

I also would like to recognize some of the department's personnel whom I shared many views academic or otherwise and received help in many occasions. From administrative services group: Edith Drummond, Karen Berg, Sharon Gannon, and Britta Nielsen; from hardware support group Rick Hughes, Charles Jobagy, and Rene Leiva; from instructional support group Rod Johnson, Chris Helmers, Carolyn Morris, and Roman Fedoriw; from systems support group Jim Easton, Bruce Folliot, and Rob Lake. Last, but not least, I would like to mention the support and friendship of Steve Sutphen, Carol Smith, and Catherine Descheneau.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software Distributed Shared Memory (S-DSM) is an abstract view of the collective memories on a set of loosely-coupled computers, as shown in Figure 1.1. This abstraction allows users to view the collection of memories of the component computers as a single large common (global) memory. Further, it allows a transparent replication and sharing of the application data over the distributed memory of the computers. This fundamental characteristic allows the deployment of networked *standalone* computers as one parallel multicomputer. The issues related to the transparent replication and sharing of application data make S-DSM quite a popular research topic. Such issues include transparency of memory among independent computers, consistency of shared data, and protocols that allow sharing.

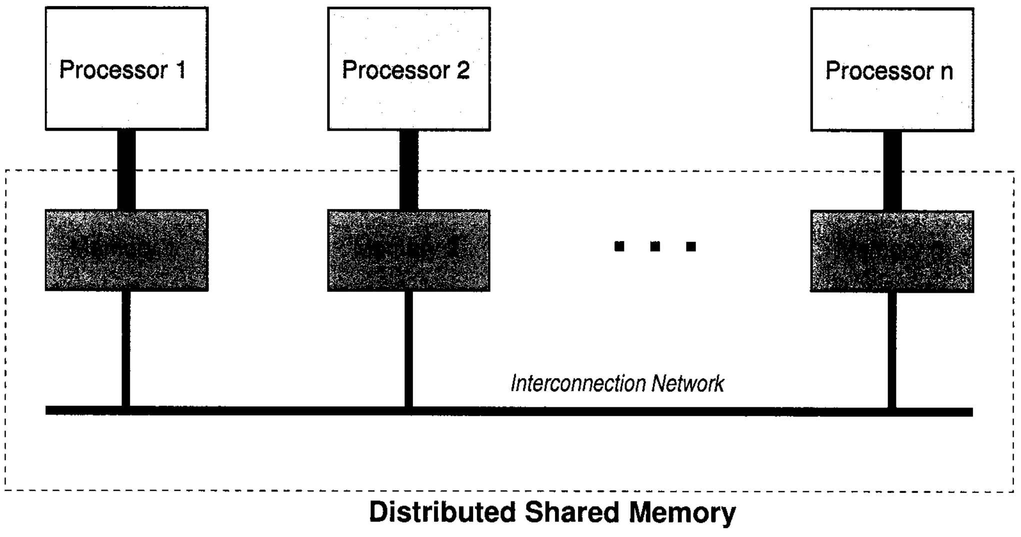


Figure 1.1: Software DSM Abstraction

This work is motivated by the observations that (i) with the advent of ever increasing availability of off-the-shelf personal computers (PCs) and the variety of network interconnects, powerful distributed systems can be built, and (ii) using S-DSM on such platforms provide the users with an easy programming model. The availability of inexpensive and powerful PCs as well as fast interconnect media increases not only the popularity of S-DSM research, but also a variety of work in other fields (e.g., caching in world wide web applications).

This thesis first examines the rationale behind S-DSM to give an insight for its development. The first few sections in this chapter give some background on loosely-coupled (distributed) computers, their basic underlying concepts covering two fundamental programming models (message passing and shared memory), and a brief description of the terminology. This introduction is followed by a classification of shared memory abstraction, a definition of S-DSM used in this thesis, and a short historical perspective. Finally, the I conclude the chapter with a brief description of the organization of the thesis and a summary of its contributions.

1.1 Background

During the past two decades, many kinds of distributed computing systems have been proposed and built, covering a wide range of design goals, scope, performance, and applications. The common point of all these systems is that they all consist of multiple processors. The collection of different architectures of multiple processor computer systems includes *vector computers, dataflow and deduction machines, multiprocessors, multicomputers,* and *workstation-LANs and -WANs.* Vector computers have many processors that execute the same arithmetic operations on different data. Dataflow and deduction computers execute different operations on different data. Multiprocessors have many autonomous processors that share a common single main memory. Multicomputers are similar to multiprocessors except that there is no shared memory but private memories dedicated to each processor. On the other hand, workstation-LANs (also known as *Network of Workstations (NOWs),* or *Clusters of Workstations (COWs),* or *Pile of Workstations (POWs)),* depending on their physical arrangements, and workstation-WANs are multicomputers consisting of workstations or minicomputers as their main processing units. Multiprocessors are commonly referred to as *tightly-* or *closely-coupled* systems, while multicomputers are referred to as *loosely-coupled* systems.

Because of this diversity, there is no agreement among researchers to date on the definition of a *distributed computing system*. Nevertheless, the following definition given by Bal *et al.* [BST89] is adopted, as it has the most generic form: *"A distributed computing system is a group of autonomous processors that do not share main memory but cooperate with each other over a communications network by exchanging messages."* This definition actually describes a loosely-coupled system, which is the common architecture where S-DSM systems have been generally implemented. I will use distributed computer system and loosely-coupled (computer) system interchangeably in this thesis.

With the recent advances in computer technology, the developments in operating systems have resulted in some form of a consensus: a *good* operating system must be capable of controlling a large number of individual machines communicating over a network.

The users should be given enough resources to solve their problems, even if it means creating *virtual resources* that share a (possibly smaller) set of *real resources*, and should not be forced to know how such a system implements various services that it provides. The users merely present to the system a set of cooperating processes and expect them to be executed by whatever means the system finds suitable.

While virtual memory becomes a *de facto* standard[1], the other real resources are not all transparent to the users. In particular, a user must *log on* a particular machine, use a file system that resides on a particular disk, use explicit commands to perform network communication (as opposed to local communication), and so on. On the other hand, a distributed computing system (or a loosely-coupled system) should be viewed as a single entity[2]: a computing server. Tanenbaum and van Renesse argue that the notion of a *host* should be abolished—the operating system should turn the collection of available resources into a number of *virtual processing units*. The underlying networks and protocols interconnecting these virtual processing units should be transparent to the users, unless the user explicitly wants to use them. Hence, a "distributed operating system" can be defined as *an operating system running on a collection of autonomous processing units that hides the multiplicity of the processing units and the underlying network from its users.*

[1]However, its need and usefulness has been questioned [Hag89].

[2]Tanenbaum and van Renesse in their excellent survey [Tv85] call this entity a *"virtual uniprocessor"* to emphasize its transparency and abstraction.

A few operating systems, fitting into the above definition, have been developed over the years. Most notably, Amoeba [MvT+90], Chorus [RMP+87], Mach [ABB+86], Sprite [OCD+88], Rhodos [WHG94], and Mos [BL85] (later became Mosix [BL88]) are some of the significant ones. However, only some of them have emerged outside the academia as commercial products, and some others have only been used in a few academic institutions.

1.2 Underlying Concepts

Interprocess communication (IPC) has been studied for many years, first in single processor systems, resulting in many communication and synchronization mechanisms [MOO87, BST89]. Two programming models that support distributed and parallel applications were introduced as a result of multiple processor computer systems: *message passing* (for loosely-coupled systems) and *shared memory* (for tightly-coupled systems).

Message passing has been the major model of IPC in distributed computing systems, since the computers forming a distributed computing system do not share main memory. The basis of the message passing model is Hoare's classic paper on Communicating Sequential Processes (CSP) [Hoa78]. Message passing is characterized by the data movement among the cooperating processes as the processes communicate and synchronize by *sending* and *receiving* messages. Many variations of message passing systems have been proposed in the literature. A complete list, discussing these variations, is provided elsewhere [BST89].

Unfortunately, message passing does not allow data sharing directly. One way to share data using message passing is to put the shared data in a special process and allow the other processes to send well-defined operations to the special process that operates on the data [Lib85]. Other methods that allow sharing data in a message passing environment may move data around explicitly in messages, but this approach needs special care since synchronization of the messages may become a problem. Moreover, data consistency may become a potential problem if several messages carry copies of the same data.

The message passing mechanism is usually becomes significantly burdensome for the application programmers, because they generally have to move data back and forth explicitly within programs [TSF90]. Earlier, *Remote Procedure Call (RPC)*, a mechanism for language-level transfer of control and data between processes [Nel81, BN84], was introduced to ease the burden on the application programmers. RPC has the basic semantics of shared memory, except parameters are passed only "by value", because of the separate memories (address spaces) on which the caller program and the remote procedure usually execute. Broadcast and multicast are other message-based communication mechanisms, where the interactions involve one sender and multiple receivers [Geh84].

Shared memory is among the earliest communications paradigms in programming. Many languages exist that use the shared memory paradigm. Many uniprocessor operating systems are constructed as a collection of processes communicating through a shared memory. The shared memory paradigm, that resides on the opposite end of the spectrum of communications mechanisms, provides direct support for data sharing as the mapping of data to a shared memory is natural.

Bal and Tanenbaum [BT91] describe the most important differences between message passing and shared memory paradigms as follows:

- In message passing, two processes should be alive when the interaction takes place because the message transfers information between two processes. Moreover, at least the sender should know the identity of the receiver before sending the message. In contrast, processes interacting through shared memory need not know the existence of each other, nor should they both be alive at the same time. They only need the address of the memory location they share.
- Also, in message passing there is a delay between sending a message and its reception. But an assignment to a shared memory location has immediate effect.
- One program module cannot affect the correctness of other modules in message passing. In contrast, a "wild store" through a wrong pointer in one module may cause a disaster in shared memory.
- Synchronization of the communicating processes is implicit with message passing. In other words, the receiver waits for a message to arrive. Similarly, with synchronous message passing, the sender waits for the receiver to be ready as well. With shared memory, however, synchronization must be provided explicitly, for example, by mutual exclusion using locks or semaphores.

Li [Li86] has noted some additional implementation problems of the message passing paradigm. They include the difficulty of passing complex data structures to a remote procedure and the problem of moving processes to other processors (process migration). These problems make process management, thus message passing, more complicated.

Earlier, researchers have observed the complementary role of memory and communication in the context of operating system kernel design as well as in the organization of distributed applications [YTR+87]. This observation reduced the problem to an investigation of how to extend this duality to a distributed environment. In recent years, the shared memory paradigm became a popular research topic among several researchers who exploited it as an alternative approach for IPC over a network, although it is not directly applicable to the current distributed computing systems. The abstraction of shared memory on a distributed computer system is known as *Distributed Shared Memory (DSM)*. In this context, a distributed shared memory can be viewed as a memory address space that spans multiple computers' memories, and is *logically* shared by processes running on a loosely-coupled system.

One other approach to ease the complexity of message passing is to build complex language compilers that would analyze the data dependencies in applications, and insert necessary constructs to automatically parallelize sequential code. Although extensive research has also been done in this area, it is beyond the scope of this thesis.

1.3 Classification and a Historical Perspective

Over the years, researchers used several different phrases to define the main topic of this thesis—shared memory abstraction. Raina [Rai92] gives a clear classification of the shared memory abstraction on distributed memory hardware as follows:

- *Virtual Shared Memory (VSM)*—systems that use hardware assistance (other than the memory management unit (MMU)), such as a hardware cache mechanism. A typical example is the DASH prototype [LLG+90]. The unit of sharing is usually a cache line.

- *Shared Virtual Memory (SVM)*—systems that implement shared memory on top of each node's virtual memory, allocating a range of address space shared by every node. Such systems use hardware page size as the unit of sharing.

- *Distributed Shared Memory (DSM)*—systems that are known as Non-Uniform Memory Architectures (NUMA) or "dancehall" architectures. In these systems, there is no replication, thus no hardware coherence problem.

This thesis is focuses on "software distributed shared memory" that fits into the definition of "shared virtual memory". Nevertheless, I will use the term *software distributed shared memory (S-DSM)* to mean "shared virtual memory".

Although Kai Li's work [LH86] in mid-80's is considered as the start of active research on S-DSM, the concept of sharing the memories of other computers in a local area network (which is fundamentally the essence of S-DSM) goes back to almost a decade earlier. Researchers at Monash University in Australia worked on the MONADS project [Kee78], whose purpose was to investigate the ways to build a network of personal computers based on uniform shared virtual memory. Within the MONADS project, Abramson [Abr81] introduced the idea of using hardware mechanisms to manage large[3] virtual memory, which in essence, is the very concept of software distributed shared memory. Later, Rosenberg and Keedy [RK81] described a method to manage a large virtual memory. Abramson [AK85] designed an novel bus architecture to achieve this goal, whereas Li's S-DSM work was solely based on the "already" existing support provided by the underlying hardware through the operating system. After all, the fundamental issue different S-DSM systems try to address is the fact that multiple copies of the application data must be identical at all (sometimes most) times. This issue, for practical purposes, is the cache coherence problem that arises in "multiprocessor" shared memory computer systems. The hardware communication platform in the latter case is the system bus, whose speed is close to that of the processors in the system.

1.4 Thesis Statement

The research activity on S-DSM steadily increased until the past decade. Most recently, research activities have reduced to developmental work trying to make

[3]Here "large" implies a capacity greater than the capacity of a single computer, as supported by its operating system.

S-DSM systems more efficient, run different benchmarks, and use variations of existing memory consistency protocols. Nevertheless, there are still different areas to explore: the development of *the killer application* and use of S-DSM concepts in other settings such as in storage area networks or wide area networks. The idea of caching has already been explored widely in the context of web browsers as well as Internet search engines.

Year	Number of Publications	Year	Number of Publications
2003	173 (as of 2/2004)	1996	341
2002	343	1995	316
2001	349	1994	239
2000	348	1993	253
1999	407	1992	160
1998	338	1991	118
1997	367	1990	106
		pre-1990	331

Table 1.1: S-DSM Related Publications over the Past 14 Years

To give a brief perspective of the past research relevant to S-DSM, I have searched one of the most frequently used citation service, INSPEC Database [IEE04]. I used the terms `((distributed shared memory) or (DSM) or (virtual shared memory) or (vsm) or (coherence protocol) or (consistency model))` in the search. Although there is a certain level of *contamination* in the search results (e.g., a record using one of the terms in a different context is hit by the search criteria), the results give a rough indication of relative intensity of S-DSM related publications over the years. The documents searched per year ranged from 258,862 in 1990 to 201,103 in 2003 (I believe this is still a low value since the collection is most likely be incomplete for the last year). For 1980–1989 period, the number of documents searched was 2,122,627. Table 1.1 shows the distribution of the search results over the last 14 years and beyond. Figure 1.2, which is extracted from the DSM section of the collection of computer science bibliographies [Ach94], depicts the yearly distribution of publications in [Esk95] as of February 6, 2004. Note that this online bibliography contains only a few references to some earlier key research in the hardware area. Although this bibliography represents considerable effort, it should still be considered incomplete, and there are about a hundred or so newer publications that have not yet made into the bibliography.

Based on the above evidence, I state that the software distributed shared memory is still a viable alternative to message passing for many applications. Indeed, all the "key" ideas have been explored, as Michael Scott claimed in his keynote speech at the Second Workshop on Distributed shared Memory [Sco00]. Nevertheless, most of the de-facto parallel benchmarks and well written applications can still achieve better speedups using S-DSM, compared to using message passing systems. I present a new software DSM system called JIA-R [Esk02] in this thesis to confirm Michael Scott's claims. Additionally, I show that such systems can benefit from high-speed network interconnects and further reduce the communication latency that is inherent in such systems. There is no work in the literature that presents an evaluation of a S-DSM on three networks.

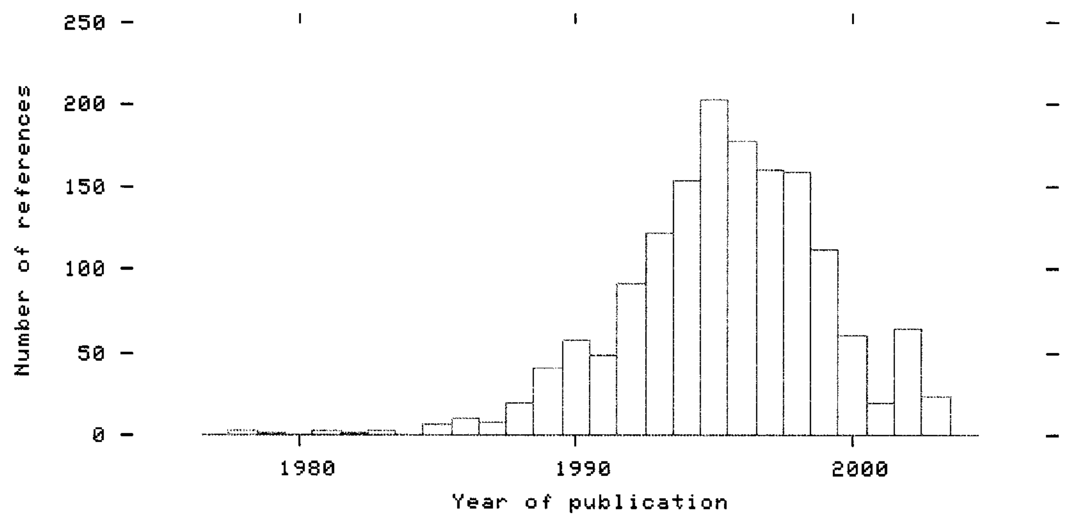


Figure 1.2: DSM Publications by Year

1.5 Organization of the Thesis

The rest of the thesis is organized as follows: Chapter 2 discusses the "foundations" of distributed shared memory. It reviews the two fundamental programming models, and argues how these models provide application programmers with an abstract view of distributed shared memory. Chapter 3 discusses various related issues, and gives a unified review of software distributed shared memory systems developed over the past decade or so. Chapter 4 presents JIA-R, a new software distributed system which is derived from JIAJIA [HST98], and describes its distinguishing characteristics over its predecessor. Chapter 5 evaluates JIAJIA and JIA-R on several platforms and network interconnects using the same set of benchmark applications[4]. Chapter 6, summarizes the contributions of my research, and presents a summary of future work.

1.6 Summary of Thesis Contributions

The principle contributions of this thesis are as follows:

- An overview of the state-of-the-art in software distributed shared memory systems, with a discussion of fundamental design and other research issues that are becoming more interesting and challenging to pursue.
- A unified review of earlier software distributed shared memory systems that appeared in the literature. Such systems are grouped into three distinct categories: page-based, object- or language-based, and hybrid.
- Demonstration of the advantages of using a software DSM system on networked off-the-shelf computers to run existing parallel shared memory applications or to develop new applications using this model, instead of explicit message passing systems such as PVM [Sun90] or MPI [Mes95].
- Introduction of a new S-DSM system called `JIA-R` and its comparative performance on three different network interconnect platforms: Fast Ethernet, Gigabit Ethernet, and Myrinet.

[4]With some exceptions described later.

Chapter 2

Foundations of DSM

The foundations of S-DSM, basically cache coherence and memory management, have been studied for many years, resulting in the development of several S-DSM systems. This chapter discusses design and implementation issues concerning the common characteristics of such S-DSM systems. These can be broadly identified as follows:

- Layer of abstraction
- Fundamentals
  - Memory consistency models
  - Coherency protocols and synchronization
  - Structure and granularity
  - Data location and access

Other less common issues addressed by various software DSM systems are heterogeneity and fault tolerance.

2.1 Layer of Abstraction

In parallel and distributed systems, S-DSM abstractions are integrated at different layers (Figure 2.1). Researchers have proposed three ways to provide DSM: hardware enhancements, operating system primitives and system libraries, and language (object) and application level mechanisms. These implementations, however, are not mutually exclusive. There are many hybrid approaches in the literature.

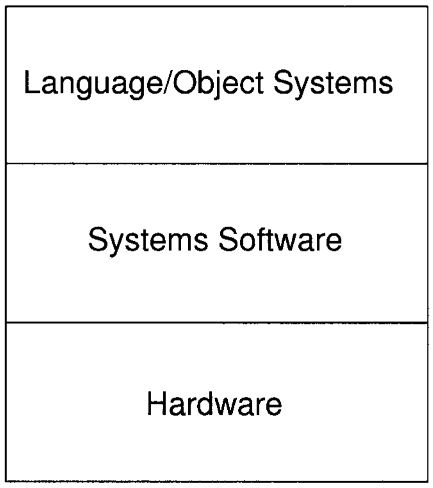


Figure 2.1: DSM Integration Layers

From the computer architecture point of view, distributed shared memory (DSM) usually refers to the memories of NUMA computers. However, over the years, software as well as hardware approaches have been developed to help overcome the "caching" problem of those computers.

2.1.1 Hardware Implementations

Introducing hardware support to S-DSM has been exploited in several ways. A common approach is to explore alternative communication structures to reduce the bus-memory contention in conventional multiprocessor architectures [PNB83]. Distributed global memory systems such as IBM RP3 [PBG+85] provide a distributed physical memory that is shared among the processors [SD88, Bro89]. The following summary describes a subset of research efforts regarding hardware DSM systems.

The "Data Diffusion Machine" (**DDM**) project [WH88] at the Swedish Institute of Computer Science aimed at developing a new scalable multiprocessor architecture based on a new notion of Cache Only Memory Architecture (COMA). This architecture relies on a hierarchical network structure. Each processor in this hierarchy has a "set-associative" local memory. These memories are connected to a local bus via a memory controller to construct a cluster. These clusters can then be connected, via another memory controller, to a higher bus, and up on the hierarchy. The higher level memory controllers are known as "directory controllers." Each processor holds the data it created in its own local memory, and that data can migrate automatically when it is needed, thus reducing access times and traffic. The DDM is primarily designed to support parallel execution of logic programs, but the architecture is sufficiently general that it can be applied to any class of applications.

The development of **MemNet** [Del88] is based on the observation that the network is always treated as an I/O device by the communication protocols. MemNet is a shared memory local area network, based on a high speed token ring, developed at the University of Delaware. A special purpose hardware unit and its software make this local network to appear as memory in the physical address space of each processor on the network.

The **VMP** [CGBG88], an experimental shared memory multiprocessor developed at Stanford University, uses software management of the processor caches and the design decisions in the cache. The project focuses on the problem of connecting multiple high-performance processors to a shared memory without significant degradation, rather than connecting a large number of processors with more modest capabilities.

The "Directory Architecture for SHared memory" (**DASH**) [LLG+89] prototype at Stanford University is a result of the research which showed that it is feasible to build scalable shared memory multiprocessors with hardware cache coherence. This prototype provided solid evidence that it is possible to build such a system and it allowed further research into the study of real workloads on an actual hardware. The DASH architecture is composed of two levels. At the first level, there is a set of processing nodes connected by a mesh network. These nodes in turn contain bus-based multiprocessors. The intra-node cache coherence implements a snoopy protocol, where each processor "listens" to the activity on the system bus all the time. The inter-node cache coherence is maintained by a distributed directory-based protocol. This prototype became an early version of Silicon Graphics Inc.'s scalable multiprocessor series.

The "Memory Hierarchy Network" (**MHN**) approach is based on the inclusion of data memories and dynamic routing capabilities in the switching elements of the multistage interconnection networks [MBLZ89].

PLUS [BNR89] is a multiprocessor system with a distributed memory topology, and supports memory coherence and synchronization in hardware and caching data among multiple memories in software.

The "Virtual Port Memory" (**VPM**) [Joh89] machine developed at New Mexico State University explores the idea of global memory machines that relies entirely upon message passing for interprocess communication and synchronization. This research machine is intended to evaluate the potential of global memory message passing architectures to combine the best features of both shared memory and message passing paradigms, while avoiding many of their drawbacks.

CAPNET [TF90] applies the shared memory paradigm onto a wider domain, namely the wide area networks. In this case, the inherent broadcast nature of a local area network is lost as the underlying network essentially becomes a collection of point-to-point interconnections between the nodes.

The **ALEWIFE** project [ACJ+92] at MIT is a large-scale multiprocessor design that integrates both cache-coherent, distributed shared memory, and user-level message passing in a single integrated hardware framework. Although the most recent implementation can scale up to 512 nodes, only a 32-node system has been prototyped so far. The ALEWIFE group also pursues some software issues, such as synchronization, compilation, various run-time systems, and operating system for the architecture.

The "FLexible Architecture for SHared memory" **FLASH** project [HKO+94] at Stanford University is to develop a scalable multiprocessor which is able to support a variety of communication models through the use of a programmable node controller. The main component of this design is a custom protocol engine MAGIC (Memory And General Interconnect Controller). FLASH is an umbrella project encompassing research into operating systems, simulation technology, applications, and compilers and languages.

The **AVALANCHE** project [SWCL95] at the University of UTAH aims at designing a scalable parallel computing environment with low communication latency, supporting both message passing and distributed shared memory programming models. The AVALANCHE prototype is designed to have 64 processing elements, using off-the-shelf hardware components as much as possible to achieve its goal. The main effort is the development of a "context sensitive" Cache and Communication Controller Unit (CCCU) to provide low message latency as well as support for flexible suite of cache coherence protocols. This project also makes use of the concept "Simple Cache Only Memory Architecture" or S-COMA [HSL94].

The **I-ACOMA** project [TP96] at the University of Illinois at Urbana-Champaign focuses on new processor, memory, and system technologies and organizations to build novel computer architectures. The emphasis is on architecture and software support for thread level speculation, design for reliability, ease of debugging and fault recovery, reconfigurable architectures, techniques for energy management and architectures integrating processor core components and memory on a chip.

The "Direct Interconnection of Computing Elements" (**DICE**) project [LQCK96] at the University of Minnesota is aimed at designing a shared-bus multiprocessor based on COMA architecture. It optimizes the COMA for a shared-bus to particularly reduce the side effects of the cache coherence.

The "Efficient Architecture for Running THreads" (**EARTH**) project [HMT+95] at McGill University is aimed at running both numeric and non-numeric parallel applications efficiently. It investigates compiler techniques and novel architectural features to support future high-performance architectures. The project has moved to the University of Delaware.

The **LIGHTNING** project [NSA97] is a multi-institution research aiming at developing optical interconnect system for high end workstation resource sharing. The unique characteristics of this project are: a fully scalable architecture allowing dynamic distributed network control and dynamic reallocation of communication bandwidth among various nodes as needed by an application. The operating system and computer interface cards are being developed at the Sarnoff Labs, the network architecture and network control hardware are being designed by SUNY Buffalo, and the optical components are being developed by the University of Maryland.

The **NOW** project [ACP95] at the University of Berkeley is somewhat different than the others described here. It aims at building a system support for using a network of workstations to act as a distributed supercomputer on a building-wide scale. Its complementary research efforts include developing an operating system and a communication architecture. The goal of the NOW Project is to demonstrate a 100 processor system that delivers better cost-performance for parallel applications than a massively parallel processing architecture for the same scale as well as better performance for sequential applications running on an individual workstation. To achieve these goals, the group are doing research and development into new network interface hardware, faster communication protocols, distributed file systems and distributed scheduling and job control.

The **NUMACHINE** project [GBC+98] at the University of Toronto aims at developing a modular, cost-effective and scalable shared-memory multiprocessor architecture. The NUMACHINE is designed as a cache-coherent architecture that is easy to program for efficient parallel applications. The node elements of the architecture are linked to each other by a hierarchy of unidirectional bit-parallel ring interconnects. This ring hierarchy provides efficient multicasting, order-preserving data transfers through cleverly intelligent cache coherence protocols that restricts the coherence traffic to local elements whenever possible. The architecture is particularly optimized for applications with good locality.

The **S3.MP** (Sun's Shared Memory Multiprocessor) [NAB+94] is a research project that implements a distributed cache-coherent (CC) shared memory computer. S3.MP uses a distributed directory-based protocol to achieve cache coherence. Similar to Scalable Coherent Interface (SCI) [Goo89], this protocol uses a linked list for its hardware supported overflow mechanism to hold the data blocks shared by the processing nodes.

The **TRAPEZE** project [YCGL97] at Duke University aims at developing new techniques for high-speed communication and fast access to stored data in workstation clusters. The primary platform for this project is a cluster of DEC/Compaq Alpha and Intel-based workstations linked by Myrinet [BCF+95] and Alteon [Nor00] interconnects.

Some of the above projects introduced above are only indirectly related to developing "distributed shared memory". Also, some of these projects are completed, yet some others are still on going.

2.1.2 Software Implementations

Abstraction of distributed shared memory at software level combines the scalability of loosely coupled multicomputers with the ease of programming of tightly coupled multiprocessors. Usually, this abstraction is achieved at various levels of systems software: the operating system level, the programming language level, or using a hybrid approach using the operating system and the communication substrate. As these implementations are the essence of this thesis, they are discussed in more detail in Chapter 3.

2.2 Fundamentals

The overall goal that the designers of software DSM systems face is to provide cost-effective algorithms to manage the "extended" memory (e.g., local and remote memories) so that data can be accessed efficiently, yet preserving a logically shared space to the programmer [MS99]. Although the design space of software distributed shared memory is large, the primary focus of the recent research has been concentrated in the following four categories:

- The consistency of the "extended" memory across different levels (e.g., local and remote memories). This basically means adopting an appropriate and efficient memory consistency model for the applications.
- The structure and granularity of shared data that is moved across the levels of the extended memory.
- The management of the extended memory with algorithms and mechanisms that will allow efficient data sharing.
- The initial placement of shared data on the extended memory and its location and access mechanisms.

2.2.1 Memory Consistency Models

DSM systems make extensive use of "caching" to enhance overall performance. Caching requires that the data being shared must be kept consistent across the system all the time. A centralized shared memory system employs traditional, well-defined uniprocessor consistency model, called *Atomic Consistency (AC)* (or *strict consistency*), which requires that (1) each (shared) object has a unique copy, (2) all the write operations to such objects are totally ordered, (3) a read operation on an object always returns the last value written into the object, and (4) all the non-overlapping operations are performed in the order they are issued [HW90]. However, the AC model is not applicable to a distributed shared memory system as the set of operations on shared objects by each processor in such a system is only partially ordered. This observation directed DSM researchers to weaken the notion of memory consistency, and several less strict memory consistency models were subsequently adopted. I informally define these models in this section to help compare various S-DSM systems that are presented in Chapter 3.

Mosberger [Mos93] classifies the proposed memory consistency models as *uniform* and *hybrid*. The uniform models do not distinguish between the types of memory access. The hybrid models employ different ordering constraints depending on the type of memory access, such as shared or synchronizing. A detailed discussion of this classification can be found in [Mos93]. Figure 2.2 shows the hierarchy of uniform memory consistency models indicating their strictness on sharing:

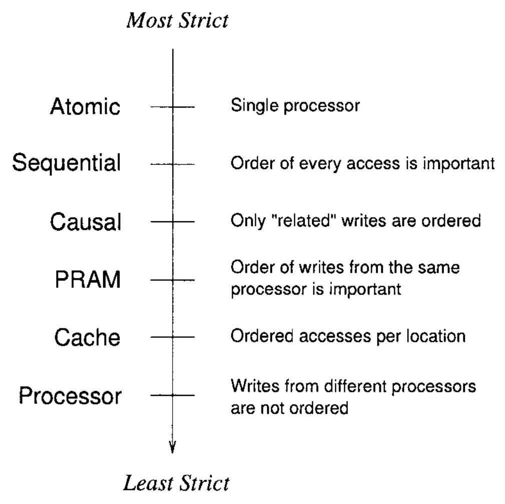

Most Strict
Atomic
Single processor
Sequential
Order of every access is important
Causal
Only "related" writes are ordered
PRAM
Order of writes from the same processor is important
Cache
Ordered accesses per location
Processor
Writes from different processors are not ordered
Least Strict

Figure 2.2: Uniform Consistency Models

- Lamport [Lam79] suggested a less strict model to be used in multiprocessor systems, called *Sequential Consistency (SC)*. The SC model guarantees that "the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."
- Hutto and Ahamad [HA90] applied Lamport's notion of *potential causality* [Lam78] to DSM, and defined *Causal Consistency (CC)* as "the agreement of all processors on the order of casually related events[1]."
- Lipton and Sandberg [LS88] suggested the *Pipelined RAM (PRAM)* consistency model, which requires that "all processors observe the writes from a single processor in the same order, but may disagree on the order of writes executed by different processors."

[1]The authors interpreted a *write* as a "message-sent" event and a *read* as a "message-received" event.

- Goodman [Goo89] weakened the SC relative to location and proposed *Cache Consistency* or *Coherence*. This scheme requires that "accesses are SC on a per-location basis."

- Goodman's [Goo89] definition of *Processor Consistency (PC)* can be interpreted as a combination of coherence and PRAM, i.e., writes from different processors may be observed in different orders, but writes from a single processor must be performed in the order they occurred.

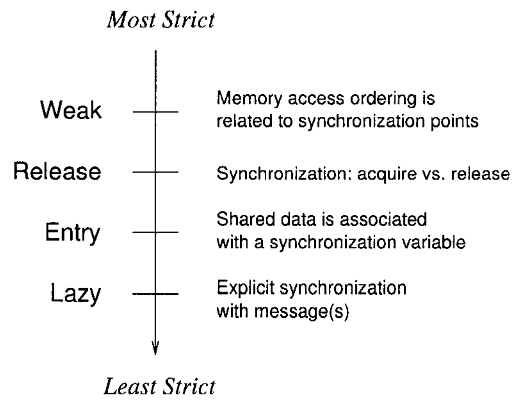

Most Strict
Weak
Memory access ordering is related to synchronization points
Release
Synchronization: acquire vs. release
Entry
Shared data is associated with a synchronization variable
Lazy
Explicit synchronization with message(s)
Least Strict

Figure 2.3: Hybrid Consistency Models

Hybrid memory consistency models reduce strictness even further as Figure 2.3 shows. These models take the advantage that most parallel and distributed applications enforce higher-level synchronization mechanisms within themselves, thus only require the enforcement of coherent shared memory during explicit synchronization operation(s). Cheriton [Che86] presents some examples where certain inconsistency levels are unavoidable, yet acceptable. Common hybrid models are listed below:

- *Weak Consistency (WC)* [DSB86] separates shared data accesses from synchronization accesses. It implies that all previous data accesses by a processor are performed *before* a synchronization access performed by that processor.

- *Release Consistency (RC)* [GLL+90] is an extension of WC where all previous data accesses (updates) are performed before a release of a synchronization access is observed by any processor.

- *Entry Consistency (EC)*, being weaker than RC, relates a synchronization variable with each shared data [BZ91]. In an entry consistent system, processors require consistency of shared data only at the beginning of a critical region. This model is used in MIDWAY S-DSM system.

- *Lazy Consistency (LC)* separates synchronization operations from the shared data [BH90]. Synchronization is achieved explicitly by other means, such as by sending and receiving messages. Updates and invalidations to the shared data may be postponed until the "new" values become visible.

Release consistency is further relaxed as follows:

- *Eager Release Consistency (ERC)*, where updates are buffered until the next *release* synchronization operation. This model is employed in MUNIN S-DSM system [BCZ91].
- *Lazy Release Consistency (LRC)*, where updates are buffered until the next *acquire* synchronization operation [KCZ92]. This model is used in TREADMARKS and several other S-DSM systems.

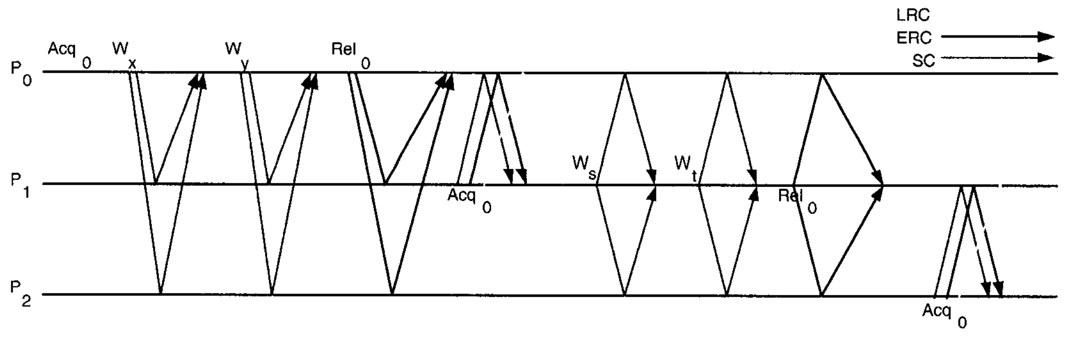


Figure 2.4: Common Consistency Models used in S-DSM Systems

Among the consistency models introduced above, only sequential consistency and hybrid consistency models are practical for S-DSM systems. Figure 2.4 shows the message traffic on these commonly used models. Note that the figure does not incorporate the messages for the first acquire operation on P_0, and that all the messages at the acquire point are for synchronization, except for LRC, which also contains all the buffered updates.

Other models usually require increased consistency message traffic resulting in a substantial amount of communication. This makes a S-DSM implementation inefficient and impractical.

Most recently, Iftode *et al.* [ISL96] introduced yet another consistency model, called *Scope Consistency (ScC)* for S-DSM systems. A *consistency scope* is a portion of the application code with respect to which memory is accessed, making the modifications to data only visible in that scope. The idea is similar to critical sections in that a scope consists of "all" critical sections protected by the same synchronization primitive.

In this model, consistency rules are relaxed in the following way: When a consistency scope is opened by a process, all the previous updates must be completed for the scope on that process, and a memory access by a process is allowed to be made only after all the other previously opened scopes are successfully entered.

Finally, an earlier consistency approach proposed by Cheriton [Che86] is called the *Problem-oriented Shared Memory*. The applications are required to handle any consistency issues through specific memory "fetch" and "store" operations. This approach is used by various implementations of Munin [BCZ90, CBZ91, CBZ95].

2.2.2 Structure and Granularity

The layout of the shared data (*structure*) and the size of the shared unit (*granularity*) are closely related [NL91]. These characteristics are usually controlled by the level of integration of DSM implementations. Hardware implementations typically support smaller granularity. For example, both DASH [LLW+92] and DDM [HLH92] have a granularity of 16 bytes, and this is the cache line size of their prototype processors MIPS R3000 and MOTOROLA MC88100/MC88200, respectively. MEMNET's [DSF88] cache size is 32 bytes. Some hardware implementations employ hybrid approaches: PLUS [BR90] uses caching (replication) granularity of a virtual page and 4 bytes for coherence.

In S-DSM systems implemented at the system level, a page is used as the unit of sharing. This allows designers to integrate the implementation with the virtual memory system of the underlying operating system. Such systems view shared data as an unstructured sequence of bytes and, also allow users to share multiple pages, if needed. For example, users can share a program code composed of several pages or a large array of integers occupying a couple of pages. IVY [LH86], MIRAGE [FP89], MUNIN [CBZ91], METHER [MF89] and many others use this approach. METHER also supports special "short pages", which are only 32 bytes long.

As the language and application level S-DSM implementations provide sharing at an "object" level, the granularity of sharing on such systems varies, and is basically determined by the size of the shared object. The run time system provides coherency by automatically serializing the accesses to the shared objects.

2.2.3 Coherence Protocols and Synchronization

S-DSM provides a global view of all memories to the users. The global view should be kept consistent according to the memory consistency model used, requiring that the access to the shared data must be closely controlled. Coherence protocols, similar to cache coherence protocols found in multiprocessors, are used to enforce this requirement. The protocol is usually trivial, provided there is no replication among the shared data. In that case, the coherence can easily be achieved by serializing the accesses to the data through the underlying interconnection network at the processor level. However, the traditional method severely reduces the major advantages of a S-DSM: scalability and parallelism. The easiest way to increase parallelism is to replicate data. Unfortunately, data replication complicates the coherence protocols, because the protocols must also deal with the replicas of the shared data.

In general, protocols that handle replication fall into two main categories: *write-invalidate* and *write-update*. Both of these groups are *snooping* protocols. A common alternative is *directory-based* protocols used in scalable SMPs, where the cache coherence is achieved at several (usually two) levels.

- A **write-invalidate protocol** broadcasts an invalidation request when a replica is modified by a processor. It allows multiple read-only copies and one write-only copy to exist, but before a write operation is actually performed, all the copies except one are invalidated. It is also known as the *multiple-readers-single-writer (MRSW)* protocol.
- A **write-update protocol** broadcasts the new value of the data when a replica is modified by a processor. It allows multiple write-only copies of shared data as well as multiple read-only copies. However, write operations are performed on all copies. It is also known as the *multiple-reader-multiple-writer (MRMW)* or *distributed write* protocol.

Generally, a *write-invalidate* protocol works better in "light sharing" applications, as write operations impose only one copy of data in the system. A *write* *update* protocol is preferred on "heavy sharing" situations, such as semaphores, as the write operations always impose coherent values in all caches.

Synchronized access to shared data is achieved by low level machine instructions such as *Test-and-Set* in shared memory multiprocessors, In S-DSM systems, however, the use of such instructions on arbitrary memory accesses is not practical. One solution is to provide the users with high level synchronization primitives, such as locks and barriers, implemented using message passing. Alternatively, applications may synchronize only when necessary (for example, to indicate the completion of computation). Most recent S-DSM implementations use the latter approach.

For the sake of completeness, I define a lock, a barrier, and a scope, as follows:

- A **lock** is a synchronization mechanism that allows an exclusive execution of a piece of program code that accesses some shared data. As such, a lock must be acquired to perform the operations, and then it must be released.

- A **barrier** is a mechanism that would block the execution of a parallel (SPMD) program until all the processors have reached a certain location in the code. In the context of S-DSM, one can assume the same program is executing on different computers.

- A **scope** is a "limited" view of (shared) memory where modification to data is only visible within.

2.2.4 Data Location and Access Algorithms

In addition to keeping the shared memory "consistent," a S-DSM system should also provide algorithms to locate and access shared data. Stumm and Zhou [SZ90a] categorize such algorithms based on whether the data are migratory and/or replicated:

Central-server algorithm:

Shared data resides in a fixed and known location, and is maintained by a server. The users (clients) of the shared data send requests to the server which responds to those requests. Although this algorithm is quite simple, it has a potential bottleneck where the server node may become overloaded by frequent requests. This algorithm is basically Li and Hudak's *Centralized Manager* algorithm, where the "ownership" of data is statically fixed [LH86].

Migration algorithm:

Shared data is grouped into blocks, and these blocks relocate to the requesting nodes as they are accessed. This approach eliminates the bottleneck of the central server algorithm by reducing the communication costs and allowing neighboring data to be accessed locally. If the data block is not local, a client broadcasts a location request message in order to locate the data block. After the data block is located, it is requested from the current holder by a second (migrate) message. This primitive two-phase algorithm causes unnecessary traffic on the network. More efficient methods to locate shared data are also known [LH86].

Read-replication algorithm:

Shared data is replicated on read operations in order to reduce the communication overhead of such operations. On a write operation, however, the requester has to multicast invalidate messages to the holders of the replicas of the shared data before performing the write operation in order to maintain consistency. This algorithm basically follows the write-invalidate protocol.

Full-replication algorithm:

This algorithm goes one step further, allowing multiple writable copies of the data blocks, but complicating the consistency maintenance of shared data. A global "sequencer" controls accesses to the shared data to ensure consistency.

The above replication algorithms are basically Li and Hudak's *Distributed Manager* algorithms. Although they reduce the message traffic for coherence, they are quite complex and non-trivial to implement. More discussion of these algorithms can be found in Li's PhD thesis [Li86].

2.2.5 Other Issues

Some other issues of S-DSM such as heterogeneity, recoverability, and fault tolerance have not been investigated as extensively as the issues described above. Because they are quite complex issues, these mechanisms are either not fully implemented, or they are only experimented with via simulations.

AGORA [BF88] and MERMAID [ZSM90] aim at extending an S-DSM to heterogeneous system environments. As a language level S-DSM, AGORA supports multi-language modules running on heterogeneous machines by providing a set of access functions to create and manipulate shared data structures. These functions can be accessed by different languages such as *C* and *CommonLisp*. MERMAID, is developed as a user-level S-DSM with some modifications to the underlaying operating system kernel. MERMAID only supports the *C* language. Both research efforts concluded that the major problem is data conversion, and for extreme values, different representations of floating point numbers make the conversion impossible. Zhou *et al.* [ZSLW92] argue that the number of different machines accommodating heterogeneous S-DSM can be extended to many, with the cost of providing separate conversion routines for basic data types and for each pair of machines. Admittedly, this approach does not scale well, because large number of heterogeneous machines would outweigh the benefits by yielding higher conversion overhead.

Another problem that has drawn interest in S-DSM research is the recoverability of shared data after processor failures. Wu and Fuchs [WF90] examine the problem of rollback recovery in S-DSM environments using checkpointing and a twin-page disk storage technique. Their checkpointing scheme is transparent to the user, and it is integrated into the S-DSM coherence protocol. Richard III and Singhal [RS93] use process checkpointing and read-shared pages for logging as a recovery technique. Their technique supports independent process recovery and, does not require active processes to rollback. Tam and Hsu [TH90b] extend their earlier token transaction method [TH90a] to achieve fast recovery in a database management system based on distributed shared memory. In this system, the database is mapped to distributed virtual memory which spans across the network [HT88]. The recovery is achieved as follows: each site on the network periodically checkpoints its token state to disk. On a failure, the site first restores its token directory using the most recent checkpoint, and then updates its sequence number information (about other sites) by communicating with the other sites. Finally, any lost token transaction message is replayed using these current sequence numbers.

Stumm and Zhou [SZ90b] extend the four basic S-DSM algorithms described in [SZ90a] to be resilient to single system faults. They argue that host failures are not frequent. Also, the failures in most cases are independent of each other (except the power failures which might affect many hosts), thus tolerating a single host failure is usually sufficient for most applications. Their study shows that the extended versions of the central-server and the full-replication algorithms do not introduce significant additional overhead. However, the overhead introduced by the migration and the read-replication algorithms may be substantial, and reduce the performance of these algorithms dramatically, depending on the access patterns of the applications.

There are other literature that studied a variety of formal models of shared memory consistency. Some of the notable related work includes, weak ordering [AH90], formalism for non-coherent distributed parallel memory [HS93, Sin93, HPS94], lazy caching [ABM93], sequential consistency in distributed systems [MRZ95a, MRZ95b], formal verification of cache coherence protocols [PD93, PD98] and consistency models [PD96, PD98]. These, however, are beyond the scope of this thesis.

2.3 Summary

The basic hardware organization of a system that provides the share memory at software level is a collection of otherwise independent computers connected by an interconnection network to support transparent exchange of messages. Such a system provides the necessary abstraction that allows programmers to utilize the simplicity of shared memory programming on distributed systems.

Software DSM is maturing, yet there are still open issues. The fundamental issues are addressed, and efficient mechanisms and algorithms are in place. The technological challenges are mostly known and companies have already demonstrated that these challenges can be overcome [MS99].

Chapter 3

A Review of Software DSM Systems

3.1 Introduction

There is a large body of literature about software DSM [Esk95]. Although somewhat dated, Hellwagner [Hel90] gives an excellent survey of this research area. Raina [Rai92] provides a survey of basic techniques and a review of architectures, including hardware, that provide distributed shared memory abstraction. Mohindra and Ramachandran [MR94] compare design issues of software DSM using simulation. Most recently, Judge *et al.* [JNT+99] give a broad overview of distributed shared memory; in fact, they review software DSMs. This chapter reviews systems based on their implementation at various software levels and on their fundamental characteristics described in Chapter 2. As such systems somewhat abstract the fundamental characteristics in different ways, and the literature describing them do not necessarily go into the same level of detail, I summarize a representative set of page-based software DSM systems, elaborating on the following common criteria: *Memory Organization*, *Coherence Protocol*, *Communication Mechanism*, and *Programming Interface*.

Memory organization deals with the way applications share the S-DSM address space. While most of the traditional page-based systems use the underlying operating systems' virtual memory primitives, some earlier systems had their own mechanisms to deal with this issue. Coherence protocol is the mechanism that provides the abstraction of consistency model in S-DSM systems. Similar to memory organization, most S-DSM system implementations use the underlying operating systems' networking functions, i.e., standard Internet protocols, such as UDP/IP.

Some other systems add another software layer and use traditional message passing systems, such as PVM or MPI, while earlier systems have their own communication mechanisms.

3.2 Software DSM Systems in the Literature

About three dozen software distributed shared memory systems have been cited in the literature over the past two decades. These systems are primarily categorized as page-based and object- or language-based. The other systems are considered as hybrid, because of their use of special hardware, their implementation platform, or because they are part of larger projects.

As the source code of the reported systems is not always publicly available for a variety of reasons, it is impossible to make an elaborate comparison. It is similarly difficult to replicate the results of the systems with their source code mainly due to the different nature of the hardware platforms used.

The development of software distributed shared memory systems can be grouped in three generations. The distinguishing milestones in each generation are summarized as follows:

- *First-generation*: The systems in this category include IVY, SHIVA, MIRAGE, and METHER. They were all developed on a network of single-CPU workstations, and they were not portable to other architectures due to their strong dependencies to the underlying systems. All these systems implemented *sequential* consistency.

- *Second-generation*: Although this generation of systems were also developed on single-CPU systems, they did not depend on the underlying architectures other than basic memory management and communication primitives and they relaxed memory coherence to reduce communication latency between the processors.

- *Third-Generation*: These systems continued using relaxed memory consistency models, and introduced various other mechanisms such as adaptation and home migration to further decrease the latency. Some of the systems in this generation such as BRAZOS use multithreading to achieve better application performance.

3.2.1 Page-based Software DSM Systems

Paged-based DSMs are the most studied category, because these systems generally (particularly if they run on UNIX or its derivatives) do not need modifications to the underlying operating system. These systems are built as user level runtime libraries where the applications are linked before they are deployed for execution. Moreover, these systems usually make use of the hardware features that are available through the operating system, such as basic virtual memory and message passing primitives.

Table 3.1 shows the page based-software DSM systems cited in the literature. I selected a small subset, marked with an asterisk, to elaborate their four basic characteristics in more detail. Other implementations are briefly summarized at the end of this section. I start discussing the page-based systems with IVY, because it was not only the very first system built, but also it stimulated more research in the DSM area and its contributions are highly recognized. Some of the systems are included in the table for their historical significance.

DSM System	Consistency Model	Coherence Protocol	Software/Hardware Requirements	Other Characteristics
ADSM [MB98]	LRC	Multiple	UDP/IP; UNIX Flavors	3rd gen.; adaptive
IVY* [Li86]	SC	WI; MRSW	Simple RPC, Aegis OS; Apollo Ring	1st gen.
Brazos* [SB97]	ScC	Multiple	Winsock, Windows NT	3rd gen.; multicast, threads
CarlOS [KFJ94]	LRC	WI; MRMW	UDP/IP, DEC OSF/1; DEC Alpha	2nd gen.
CVM [Kel96]	SC, LRC	WI; MRSW, MRMW	UDP/IP, UNIX Flavors	2nd gen.
KDSM [YLLM01]	LRC, ScC	WI; MRMW	TCP/IP, Linux	3rd gen.
KOAN [LP92]	SC	WI; MRSW	NX/2; iPSC/2	1st gen.
JIAJIA [HST98]	ScC	WI; MRMW	UDP/IP, UNIX Flavors	2nd gen.
JUMP [CWH00]	ScC	WI; MRMW	UDP/IP; SunOS, Linux	3rd gen.; fast sockets
Mether [MF89]	SC	WU; MRMW	SunOS 4.0	1st gen.
Mirage* [Fle87]	SC	WI, MRSW	System V IPC, Locus OS; DEC Vax	1st gen.
Munin [BCZ90]	SC, RC	Multiple	UDP/IP; V-System	2nd gen.
Nautilus [Md99]	ScC	WU; MRMW	UDP/IP, Linux	2nd gen.
Quarks [Kha96]	SC, RC	Multiple	UDP/IP, UNIX Flavors	2nd gen.
Shiva [LS89]	SC	WI; MRSW	NX/2; iPSC/2	1st gen.
Strings [RC98]	SC, RC	WU; MRMW	UDP/IP, UNIX Flavors	3rd gen.; SMP, threads
SVMLib [PS97]	SC	WI; MRSW, MRMW	SISCI[1], Windows NT; SCI[2]	1st gen.
TreadMarks* [KDCZ94]	LRC	WI; MRMW	UDP/IP, Unix Flavors	2nd gen.

[1] SISCI is a user level API for the SCI card below.

[2] Scalable Coherent Interconnect (SCI) network interface from Dolphin Interconnections, Inc.

Table 3.1: Page-based Software DSM Systems

IVY

Integrated shared Virtual memory at Yale (IVY) [LH86] is the first widely-acclaimed prototype of software distributed shared memory. Li [Li86] identifies three basic requirements for implementing the prototype: a fast communication link, a homogeneous set of computers, and a memory management unit (MMU) with page level protection mechanism. IVY prototype sits on top of an Apollo Domain system running a modified Aegis operating system. Nodes on the system are connected by a ring network [LLD+83]. The goal for implementing the prototype was to justify the use of "shared" virtual memory on loosely coupled multiprocessors.

IVY assumes a traditional model of a parallel program as a set of processes that share a single address space. However, the subtle difference is that these processes can run on "any" node in the Apollo Domain system. Thus, a parallel program can run on any number of processors on an Apollo ring network.

Memory Organization

Parts of a parallel program runs on several processors as closely-coupled processes. The address space of those processes can span a number of processors. Therefore, any process can directly access any memory location in this address space. Parts of this shared address space may exist on different real memories. *Memory Mapping Managers (MMMs)*, running on each node, handle the mapping between the local memories and the shared virtual memory address space. Similar to traditional virtual memory, IVY's shared virtual memory is also partitioned into "pages." Some of these pages are marked as *read-only*, if there are multiple copies of them on different nodes, whereas others are marked as *write* since they will exist on a single node.

The unit of sharing (replication) is a page that corresponds to the virtual memory page. The MMMs view the "shared" part of their local memory as a large cache of the shared address space. A reference to a shared memory location causes a page fault, and the memory mapping managers, with the help of the operating system's virtual memory, identify and fetch the missing page from the other processor's memory.

Coherence Protocol

IVY enforces strict consistency to maintain coherence between the processes. This means that each node must have a coherent view of the shared memory at all times. Memory mapping managers provide this functionality. Pages in the virtual shared memory can be in either of the two modes: (i) *read-only* or (ii) *write*. The coherence protocol is *write-invalidation*.

Each shared page is owned by a single node, and has one of the following access rights: *read*, *write*, or *nil*. The *owner* is always the node which has most recently modified the page. The owner maintains a *copyset* for each page it owns. A copyset contains the nodes which currently have an up-to-date (read-only) copy of the page. When a node references an address on a shared page, it first checks whether it has the proper access right. In the normal case, the location is accessed in the usual way. Otherwise, the memory mapping manager is invoked through a page fault. Read and write faults are handled differently. A shared page, whether it is local or not, is handled identically. The fault mechanisms are totally transparent from the user process.

Li has implemented several coherence maintenance algorithms which differ mainly in two aspects: (i) the way they locate the owner of a page, and (ii) the distribution of copysets. The details can be found in [Li86]. IVY's synchronization mechanism is based on the underlying Aegis Operating system's *eventcount* primitives.

Communication Mechanism

IVY only supports the Apollo Domain ring network as its underlying communication medium. A simple RPC mechanism handles all the remote operations. It is based on sending and receiving packets and, the exception handling mechanism. A process can either send a packet to another process or broadcast it. Li modified the Aegis operating system to handle the incoming packets more efficiently. As a result, a simple null RPC took about 10 milliseconds.

Programming Interface

IVY was implemented in Pascal. However, any programming language that could interact with procedure calls can also be used for developing applications. Programming convention requires that all shared data are grouped into a record.

None of the literature on IVY goes into details of parallelizing applications, though it is implicit in Li's thesis that he uses special primitives for the task. The programming model is not specified in the thesis explicitly either, but it is most likely to be SIMD.

The programmer is responsible for process synchronization and scheduling. There are two options for scheduling: manual or system. It the latter case, the user simply uses primitives to create and terminate processes. If the manual scheduling is chosen, the user has to take care of process migration as well. At startup, a special program initializes IVY on the nodes listed in a configuration file, and the parallelized application program starts its execution.

Other Properties

Li has implemented several algorithms to address various issues pertinent to DSM systems. They include the methods of process migration, memory allocation, and page replacement strategies. The IVY also had some shortcomings that triggered more research particularly into shared memory coherence algorithms. Overall, this work has shown that shared virtual memory can be implemented on a loosely-coupled system, and that it can achieve acceptable speedups for many parallel applications. Li's seminal work has opened a wide range of research opportunities, as seen in the vast amount of literature since 1986.

BRAZOS

BRAZOS [SB97] is the first software DSM system developed to run on Microsoft Windows NT operating system. It has a few features that are different from other software DSM systems: use of multi-threading both in itself and in applications, selective multicast, and the availability of several adaptive runtime performance tuning mechanisms.

Memory Organization

There is no special treatment to memory management in BRAZOS, except the modification of the Windows NT system call `mapmem()`. This was necessary to provide a mechanism to map two virtual pages onto the same physical page so that multiple threads could have possibly different access rights for the same page.

Coherence Protocol

BRAZOS uses selective multicast, scope consistency, and adaptive performance tuning at runtime to reduce the coherence protocol related network traffic. Multicast communication is used only during "global" synchronization. Global synchronization (or global scope) is the "arrival" at the next barrier by all the processes whereas local synchronization (local scope) is a section of code protected by a lock. Barrier synchronization occurs at two levels: (1) between threads of a process and (2) between processes. Scope consistency is the primary software-only memory model adopted by BRAZOS. It uses a distributed page management scheme, where each process maintains dirty portions of each shared data as "diffs" (see the TREADMARKS discussion for the details), since this approach is known to be superior to home-based protocols [Kel94]. Runtime performance tuning techniques used in BRAZOS include dynamic copyset reduction where dirty page diffs are not sent to unnecessary processes in cases when they are not needed anymore. BRAZOS also adopts an early update mechanism where redundant indirect diff messages at synchronization points are eliminated [SB97]. Finally, the runtime system adaptively changes the shared page management to distributed or home-based per shared page, depending on the behavior of a process.

Communication Mechanism

BRAZOS supports 100Mbps Ethernet, 1Gbps Ethernet, GigaNet cLAN, and ServerNet as the underlying communication media. Windows NT implements TCP/IP protocol stack through WinSock [Sta96] user level library. Thus, there is an additional overhead in WinSock socket calls as all the calls are made through the static library functions.

BRAZOS takes advantage of the WinSock library's multicast support to reduce coherence related message traffic during global synchronization as follows: when a process arrives at a synchronization point such as a barrier, it sends a message to the previously assigned barrier manager. This message includes a list of *dirty* pages. The manager collates this information, and sends a message to each process about dirty pages. When a process faults on an invalidated page, it sends a "multicast" message to those processes that are in the current copyset of the dirty page. A recipient of this message, in turn, sends back a multicast message containing the *diffs* for that particular page to all members of the page's copyset. In this way, processes receive indirect diffs for those pages that have not caused a page fault as yet. A disadvantage of this approach is that a process may not use the indirect diffs provided for a page it has. This restriction causes an unnecessary disruption to such processes when they receive the multicast message. BRAZOS runtime has a dynamic copyset reduction mechanism to reduce this side effect. When the number of unused diff messages count reaches a certain threshold, a process piggybacks the list of such pages to the next barrier arrival message, and removes itself from the copyset of those pages.

Programming Interface

In Windows NT, there is no function to start a process remotely similar to the Unix `rexecd` daemon. BRAZOS uses a special "service" on each node in the system to emulate this functionality.

One of the main components of BRAZOS is its GUI front end which is used to monitor the execution of parallel applications. Multiple DSM applications can be managed on a single computer using this graphical interface. It also provides performance feedback on each application by monitoring the traffic on the network.

Other Properties

BRAZOS DSM itself is multi-threaded in order to allow greater overlap between communication and computation. It also supports multi-threaded application execution, allowing programs to take advantage of the local tightly-coupled shared memory available on multiprocessor PC servers, while transparently interacting with remote "virtual" shared memory.

Current work on BRAZOS includes decreasing communication latency, multiprogramming, high availability (checkpoint/restart and thread migration), and support for MPI and OpenMP applications directly.

MIRAGE and MIRAGE+

MIRAGE [Fle87] is only the second DSM system developed. Unlike IVY, however, it is built into the Locus operating system kernel, using UNIX System V shared memory model. The Locus operating system [WPE+83] is a distributed version of UNIX that provides an enhanced set of standard UNIX services. Like IVY, having been developed more than a decade ago and built on an experimental operating system, neither MIRAGE nor the Locus operating system are operational today. MIRAGE was developed on a cluster of VAX 11/750 computers.

Memory Organization

Because MIRAGE processes use UNIX System V shared memory semantics, they access shared memory through the use of *segments*. A segment is used only to store raw data, not the program code. Processes can share segments at a page granularity, which is 512 bytes long on the VAX system. MIRAGE also provides upward compatible system calls to create, delete, or access a shared segment. For example, `shmat()` is used to create (and "attach") a shared segment to the process address space or `shmget()` is used to access a shared segment.

Coherence Protocol

The coherence model of MIRAGE is similar to IVY, i.e., multiple-readers-single-writer sequential consistency. Also, the parallel programs are assumed responsible for the proper data access synchronization.

The nodes in the system are identified as the *library*, the *clock* (i.e., the current writer), or the *requesting* site. Any other node is called the *reader* node for a page, if they share that page as read-only. The library site is associated with a particular segment whereas all the other "sites" are associated with a particular page of a segment. All the access requests are sent, and processed by the library site sequentially.

The key part of the implementation is the *delta* (Δ) time value which controls the page thrashing. The clock mechanism ensures that all the readers or the current writer "holds on" to a page for the Δ period of time. In a traditional sense, Δ is the time slice each process gets from the CPU. This value can be tuned statically or dynamically to improve overall application performance.

The library site maintains the Δ values but the clock sites deal with the page invalidations. Typically, when an invalidation request arrives at a clock site, the site (1) invalidates the (local) page, (2) invalidates the other read-only copies, if the page was read-only, and finally (3) distributes the page to the readers and the new writer.

A new system call `yield()` is created to allow processes to release the ownership of a page before their Δ time has expired. Although the new system call improves the performance in some worst-case scenarios, the use of it might become a burden to programmers.

Communication Mechanism

Unfortunately, the literature on MIRAGE does not discuss any details about its underlying communication mechanism. As the underlying operating system Locus is a distributed operating system based on UNIX System V, however, it is safe to assume that MIRAGE uses System V messages for communication.

Programming Interface

As mentioned above, parallel programs use System V "Shared Memory Interface". This simple interface is composed of four primitives: `shmat()` to attach a shared memory to the address space of a process; `shmctl()` to manipulate permissions of a shared memory segment; `shmdt()` to remove a shared segment from the address space; and `shmget()` to access a shared segment with an application.

Other Properties

Fleisch continued working on this research. His research group at UC Riverside ported MIRAGE to a network of IBM PS/2 computers, and incorporated additional features to it. The new system is called MIRAGE+ [FHJ94]. The work on this system primarily focused on reliability and fault tolerance, and various new protocols are developed using this system [JF95, TF95a, TF95b, TF99]. The group has also developed a fault-tolerant distributed storage system called OASIS+ [WLF01], that is based on the new algorithms and protocols mentioned above.

TREADMARKS

TREADMARKS [KDCZ94] is a distributed shared memory system developed at Rice University. Although its public availability was restricted due to commercial licensing, it is by far the most widely used DSM system in the academic community. TREADMARKS was developed as an experimental distributed shared memory system to study parallel computing on a network of computers. Currently it runs on a variety of UNIX flavors.

Memory Organization

TREADMARKS uses the heap address space provided by the underlying operating system and uses its own `malloc()` function to allocate shared data. Allocation of shared data is done by the "master" process before copies of it are spawned on other processors. Since the shared data allocated on the heap do not necessarily have the same address on different processors, the master process is also responsible for distributing the pointer of the shared data to other processes using `Tmk_distribute()` function.

Coherence Protocol

TREADMARKS adopts *lazy release memory consistency (LRC)* [KCZ92] to reduce coherence related communication overhead. In LRC, the modifications to the local copy of the shared data are not propagated *until the next acquire*, when the acquiring processor determines the modifications it needs based on the *release consistency (RC)*. The novel approach in LRC is the execution *intervals* that start and end with a release or acquire (lock) operation performed on a processor. The intervals on different processors are partially ordered [AH90] such that: (1) the intervals on a single processor are in program order, and (2) an interval on processor p precedes an interval on processor q if the interval on processor q begins with an acquire that corresponds to the release that completed the interval on processor p.

TREADMARKS uses a multiple-writer protocol to address the *false sharing* problem that is common on all page-based DSM systems. Initially, all the shared pages are write-protected. On the first write operation to a page, the protocol creates a *twin* of the page and changes the page's protection to read-write. Subsequent writes are made on the original page without any intervention of the protocol. A run length encoding of the modifications to the current page (i.e. the differences between the twin copy and itself), called a *diff* is created only when another processor requests for the modifications to a page or when a "write-notice" (a message indicating that a page has been modified in a particular interval) arrives from another processor for that page. The arrival of a write-notice causes the invalidation of a particular page, and subsequent accesses cause the propagation of the modifications to the local copy.

Communication Mechanism

Similar to other second generation and newer page-based DSM systems, TREADMARKS uses standard UNIX libraries for remote process creation, interprocess communication, and memory management. Interprocess communication operations use UDP/IP protocol on an Ethernet or AAAL3/4 on an ATM network. Since both protocols are unreliable, TREADMARKS uses a thin layer of operation-specific protocols for reliable delivery of messages.

Programming Interface

TREADMARKS has a very simple API that allows easy process creation and termination, synchronization, and shared memory allocation. This API includes the following functions: `Tmk_startup()` is used to initialize the DSM system and create remote processes. `Tmk_malloc()`, `Tmk_distribute()`, and `Tmk_free()` are used to allocate shared memory (by the master process), data distribution, and memory release, respectively. `Tmk_lock_acquire()`, `Tmk_lock_release()`, and `Tmk_barrier()` are used for synchronization. A few additional functions are also used in applications, but I will not list them here.

Other Properties

Since TREADMARKS uses the heap of the process address space, it is important to claim back the space that is no longer used by various system data structures. Garbage collection operation is triggered when the free space for the system drops below a threshold. This operation is usually performed while the application is blocked on a barrier [KDCZ94].

Other Page-based Systems

ADSM [MB98] extends TREADMARKS by enhancing its consistency protocol to adopt to the application's data sharing patterns. ADSM categorizes the shared data pages as: (i) falsely-shared, (ii) migratory, and (iii) producer/consumer(s). Based on this classification, each page is managed either in "single-writer" or "multiple-writer" mode. Coherence is achieved by invalidation for every page, whereas migratory pages are protected by locks, and producer/consumer(s) pages use barriers to protect the pages' respective modes. This type arrangement is reported to outperform original TREADMARKS by up to 155% on some applications. However, the additional classification for adaptation requires some application code modifications, which is somewhat a fall back of the basic goals of software DSM systems.

CARLOS [KFJ94] is one of the earlier systems that is based on TREADMARKS. The memory coherence actions are triggered by special causality annotations exchanged by special messages. The authors called this approach *message-driven coherency mechanism*, and it is intended to provide a foundation for building systems that integrates message passing and shared memory paradigms. The ultimate goal is to provide a mixed platform for which either paradigm can be used to best suit the applications. CARLOS is built on top of OSF/1 operating system using traditional UDP/IP datagrams, supplemented with a custom sliding window protocol for reliability and orderly delivery of messages.

CVM [Kel96] supports multiple coherence protocols. Initially, it was intended to include four models: single- and multiple-writer versions of lazy release consistency, sequential consistency, and eager release consistency. However, the last public release CVM 0.2 does not include eager release consistency implementation. This version runs on Sun Sparc, DEC Alpha, and IBM RS6K architectures. It is written in C++, and supports applications written in C, C++, and Fortran. One of the key features of CVM is its extensibility. Since it is written in C++, new classes can easily be derived from a master Protocol class, allowing new protocols to be easily incorporated. Similar to other newer software DSM systems, multi-threading, which allows overlap of computation and communication through context switching, is also supported. CVM has its own threading mechanism as a user-level library. However, CVM itself is not multi-threaded. Further, newer features such as heterogeneity, on-the-fly configuration, race detection [PK00], and "tapeworms" based synchronization libraries [Kel99] are not publicly available.

KDSM [LYLM02] is built on the Linux operating system. Like many other systems, it is implemented as a user level library using TCP/IP for communication and SIGIO for signal handling among processes. KDSM uses a page-based multiple-writer invalidation protocol, and also supports home-based lazy release consistency (HLRC) [ZIL96]. Apparently, this work has borrowed many ideas from JIAJIA [HST98] and HLRC. Nevertheless, the group has also developed several communication mechanisms based on different high speed networks: VIA [VIA97] and Myrinet [BCF+95].

KOAN [LP92] is built on an Intel iPCS/2 hypercube computer embedded into its the NX/2 operating system. It adopts sequential consistency implemented by an invalidation protocol using the fixed distributed manager algorithm introduced by Li [Li86]. Later, the system was ported to Paragon XP/S multi computer as an external page manager of the underlying Mach microkernel.

JIAJIA [HST98] is a second generation DSM system that uses scope consistency model. It implements a multiple-reader, multiple-writer protocol using a simple lock-based mechanism. JIAJIA is described in more detail in Chapter 4.

JUMP [CWH99] (short for JIAJIA Using Migrating-Home Protocol) extends JIAJIA by adopting a *migrating-home protocol*, and using an efficient BSD sockets-based communication protocol called *Socket-DP* [CWH00]. It has been shown in [ZIL96] that home-based memory coherence protocols usually outperform home-less protocols. Nevertheless, the JUMP system's migrating home protocol gains performance on certain applications by migrating a page to a new process P, when processor Q is serving a remote page fault from processor P if these conditions are met: (i) processor Q is the home of faulted page X, and (ii) the page X is up-to-date, i.e., processor Q has received all the updates to the page. This scheme eliminates the transmission of diff notices to the new "home" of the page. However, the new home still sends messages to all other processors on the held lock's release as short "migration notices".

METHER [MF89] is one of the older software DSM systems, and it runs on Sun workstations under SunOS 4.0 operating system. It is composed of two components: a kernel driver that maintains a set of shared pages and their states, and an event-driven user level server. Basic UNIX communication-related system calls are used to implement the two components. `ioctl()` is used to control the kernel driver, and `mmap()`, `select()` are used for shared page allocation, and the message arrival and shared page related events (such as page wanted or page freed). Although a page based DSM system, METHER supports both regular page size of 8KB, and a mini page size of 32 bytes. The latter size is to support the memory mapped network MEMNET [Del88] that originally inspired the METHER project.

MUNIN [BCZ90] is an early second-generation software DSM system. It incorporates several novel techniques that were not seen in the earlier systems. These techniques include use of multiple consistency protocols and support for complex protocols to allow concurrent multiple writers and to reduce the communication overhead among processes. It also implements a distributed locking mechanism to further reduce the network traffic load. The users of MUNIN system can annotate their applications to chose one of several consistency protocols. The latest version of the system recognizes one of *read-only*, *migratory*, *write-shared*, and *conventional* annotations. The consistency protocol uses release consistency that delays updates to share data until a synchronization location, at which point the modifications from different processors are merged [CCD+93].

NAUTILUS [Md99] also uses scope consistency implemented by a lock-based protocol. In addition, NAUTILUS uses threads, but only to handle synchronization services. Although several publications compare it with other software DSM systems, none of those publications give sufficient details as to how and why the system almost always achieves better performance than any other DSM system.

QUARKS [Kha96] system consists of two major components: a user-level run-time library that is linked to parallel applications and a centralized *Shared Memory Server* that manages the shared memory and the synchronization primitives. When a parallel application starts, it first registers itself to the server, and then spawns the other remote processes which in turn register themselves to the server as well. Each parallel application process is composed of an *Application* and a *DSM_Server* thread. QUARKS supports a write-update protocol for sequential consistency and a delayed write update protocol for (eager) release consistency. Adding new protocols is also relatively easy. The system has been ported to SunOS 4.1, HP-UX and IRIX 5.2 operating systems, running on SPARC, PA-RISC, and MIPS architectures, respectively.

SHIVA [LS89] is a followup project to Li's IVY software DSM system. It is designed and built on a the Intel iPSC/2 hypercube multicomputer. Since the Inter hypercube is primarily used by a single user at a time, SHIVA does not have to deal with any address space protection among multiple users. This approach greatly simplifies the design of SHIVA. The main components of SHIVA are a shared memory mapping and memory management mechanism, a synchronization and thread control module, a message passing implementation, and a language-independent RPC facility. Unlike IVY, however, the synchronization primitives are independent of shared memory mechanism.

STRINGS system [RC98] is based on QUARKS, but uses POSIX threads instead of QUARK's original Cthreads implementation. STRINGS also allows multiple application threads as it is designed to work on an SMP cluster running Solaris operating system. One unique characteristics of STRINGS are its use of multiple kernel-level threads for the applications and its design to run primarily on a cluster of SMP computers.

SVMLIB [PS97] is one of the few software DSM systems that run on Windows NT operating system. The primary focus of the project is to integrate scalable distributed synchronization algorithms into DSM systems since the use of such mechanisms at the lower implementation layer is necessary to implement particularly weaker consistency models efficiently. In addition to using traditional UNIX network protocols, TCP/IP, SVMLIB also makes use of a high performance network interconnect based on SCI [Goo89]. SVMLIB implements a user-configurable sequential consistency (SC) as well as synchronization support for the lazy release consistency (LRC) protocol. It is also possible to run shared memory applications on several flavors of the Unix operating system using a simple NT emulation library [PBS98].

The page-based systems mostly use the underlying operating systems' support for basic memory management and message passing features. These systems use the virtual memory trap handling mechanisms for the management of shared data and BSD sockets using (TCP,UDP)/IP protocol stack for communication between the nodes. A few systems use high performance network interconnects such as Myrinet and SCI that allow the system designers to develop more efficient communication protocols. With a few exceptions, alt the systems are built as user level libraries that are linked to parallel applications and the runtime spawns the processes on the other nodes as needed.

Other not widely known or new research efforts on page-based software DSM include CABLES [JB02], MOODY [LYL95], THE REGION TRAP LIBRARY [BS99], and WIND [SHU+00].

3.2.2 Object/Language-based Software DSM Systems

This section overviews some of the object- and language-based systems that implement distributed shared memory at the highest level of abstraction. Some of the systems are included in Table 3.2 only for their historical significance.

DSM System	Structure	Coherence Model	Software Requirements	Other Characteristics
Agora [BF87]	Object	SC	—	Heterogeneity
Amber [CAL+89]	Object	SC	—	Programming system
Adsmith [Lia94]	Object	RC	PVM	—
Aurora [Lu97]	Object	ScC	—	Based on std C++ classes
Cilk [BJK+95]	Language	Dag	—	—
Concert [KC93]	Language	Multiple	—	—
DiSOM [CGSC96]	Object	EC	—	Heterogeneity
Dist. Filaments [FLA94]	Language	SC	—	Stackless threads
Dosmos [BL94]	Object	Weak; MRSW	PVM	Hierarchical structure
Midway [BZS93]	Object	EC	—	First EC implementation
Orca [BTK90]	Language	SC	—	Strongly-typed, distributed PL
PComp [BAFR96]	Language	SC	ParC	—
Problem-Oriented SM [Che86]	Object	Problem oriented	—	—
SAM [SL94]	Language	SC	Jade; PVM	Runs on SMPs and NOWs
Tuple Space [Gel85]	Language	SC	—	Programming system

Table 3.2: Object/Language-based Software DSM Systems

AGORA [BF87] is a system that supports the development of multi-language parallel applications for heterogeneous machines. It provides two types of support for heterogeneous parallel programming: operating system level mechanisms that can be used to implement heterogeneous parallelism and programming environment functionalities that facilitates the management of parallel programs.

AMBER [CAL+89] is an earlier programming system that permits a single application program to use a homogeneous network of computers in a uniform way, making the network appear to the application as an integrated multiprocessor. It is specifically designed for high performance in the case where each node in the network is a multiprocessor.

ADSMITH [Lia94] is an object-based system built on top of PVM. Its user-level C++ API provides primitives to create and manipulate shared objects. This API supports release consistency (RC) model using object-based multiple-writer protocol with bulk transfer, prefetching, non-blocking and other specialized access capabilities.

AURORA [Lu97] is a distributed shared data (DSD) system based on a standard C++ class library and a run-time system that provides a shared data abstraction on a distributed system. It includes a unique property, called "scope behavior", that can be used in applications for various data sharing optimizations. AURORA does not provide any language extensions and it does not require any support from the underlying hardware. Hence, it is highly portable.

CILK [BJK+95] is a multithreaded algorithmic language. Its philosophy is that a programmer should concentrate on the structure of the program for efficient parallelism, leaving the scheduling, load balancing, and other execution issues to the runtime system of the compiler. Version 3.0 implemented a new shared memory consistency model called "DAG (Directed Acyclic Graph) Consistency" [BFJ+96], where the memory model is defined in terms of the computation DAG only. This version was implemented on a CM5 system. The newer version 5.1 is designed to run on a cluster of SMPs.

CONCERT [KC93] is a compiler and runtime support system for fine-grained concurrent object-oriented languages. Among other advanced techniques, it provides a global shared name space supported by object-based concurrency control. This approach, called *view caching* [KC97], provides a framework to construct customized asynchronous coherence protocols that require less synchronization among processes even with fine-grained sharing.

DiSOM [CGSC96] runs on heterogeneous networks. It provides a fine grained data sharing by using a simple shared memory model that implements update-based entry consistency. It also supports data sharing between applications running a shared memory multiprocessor.

DISTRIBUTED FILAMENTS [FLA94] is a C runtime library that runs on variety of different shared- and distributed-memory machines. It has been designed to support parallel scientific applications, and is intended to be a target for a compiler. Filaments provide two abstractions: fine-grain parallelism and shared variable communication. Each filament is a lightweight thread with no stack, and are managed by a server thread on every node. A barrier mechanism supported by reduction operations is used for the synchronization is among multiple filaments. A reliable datagram protocol is used to handle message traffic for retrieving non-local pages. The system provides an iterative filament used in loop based applications and a fork/join filament used in recursive applications.

DOSMOS [BL94] (stands for Distributed Objects Shared MemOry System) is based on a set of distributed passive objects that can be grouped as necessary. The objects that are handled by a client-server protocol can be shared by applications transparently. The system supports a weak MRSW model, although applications can also use strong consistency by declaring the objects as such. Typical PVM functions are used to handle object, group, and process creation as well as their distribution. Users can create hierarchical processes which can share the same set of objects.

The **MIDWAY** [BZS93] is best known for its "Entry Consistency" memory model. In order to provide this model, MIDWAY uses locks that are *explicitly* bound to shared variables. This allows applications acquire locks for the proper synchronization of the shared data, thus limits the movement of shared data to the lock acquisition messages. To reduce communication overhead, it uses specialized communication protocols built on Mach kernel's low-overhead interfaces for both ATM and Ethernet networks.

ORCA [BTK90] is a strongly typed programming language, based on "shared data objects" model [BT88]. It contains primitives to support concurrent programming as well as mechanisms for process creation and synchronization on remote nodes. The implementation of ORCA consists of a compiler and a run-time system.

The main goal of the **PCOMP** project is to apply compiler technology to increase the performance of parallel programs using distributed shared memory paradigm. Unlike the traditional compiler optimizations seen in automatic parallelization of sequential loops or partitioning shared arrays, the high level language of the PCOMP project, called PARC, optimizes a parallel program by analyzing the page movements of the underlying distributed shared memory.

PROBLEM-ORIENTED SHARED MEMORY [Che86], proposed by David Cheriton, is a somewhat different paradigm for building sophisticated distributed applications. A problem-oriented shared memory implements fetch and store operations that are specific to the particular application it is supporting. This shared memory can be regarded as a system service implemented on multiple processors. The semantics of the problem-oriented shared memory is relaxed such that "stale" data can exist in the system.

SAM [SL94] is a run-time system that supports a shared name space in software on distributed-memory multiprocessors. SAM uses variable granularity, based on the size of the shared data used in applications. An implementation of the Jade [RSL93] parallel programming language for distributed memory machines is also included. SAM has been implemented on a variety of platforms, including the Intel iPSC/860 and Paragon, the Thinking Machines CM-5, the IBM SP1, and on heterogeneous network of workstations running PVM.

The **TUPLE SPACE** [Gel85] is a novel synchronization mechanism developed for the Linda language. The Tuple Space is a global memory containing *tuples*, constructs similar to records in Pascal language. A tuple is denoted by providing both actual and formal parameters for every field. Unlike the other conceptual shared memory systems, the Tuple Space is addressed associatively (by contents).

There are other language- or object-based software DSM implementations or high level languages that provide shared memory abstraction in the literature. These include EMERALD [JLHB88], JADE [RSL93], LOCUST [Ver96], and SENSE [Joh99].

3.2.3 Hybrid DSM Systems

The systems listed below are considered "hybrid" for a variety of reasons. A few of these systems are not exactly a DSM system, but developed as a distributed application supporting weak consistency of shared data (e.g., BAYOU system) and are part of a larger, not necessarily a software DSM, project (e.g., SciFS). Some of them describe an architecture, and may not be fully implemented (e.g., the Π Architecture). Others are parts of an operating system (e.g., CHORUS CONSISTENCY SERVER), or a "hardware" prototype, but introduced new consistency mechanisms (e.g., the SHRIMP project). Yet some others are explicitly built on top of a message passing system (e.g., PHOSPHORUS). Nevertheless, all of those systems contributed in a substantial way to DSM research over the years. Although the list is not complete, it includes most of the systems that are frequently referenced in the literature. Some systems are included in Table 3.3 for their historical significance.

DSM System	Consistency Model	Coherence Protocol	Other Characteristics
Bayou [DPS+94]	Weak	MRMW	Disconnected applications
CASHMERe [SLD+96]	Weak	MRMW	Uses DEC Memory Channel
Chorus Consistency Server [AAO92]	SC	MRSW	Built on top of Chorus Nucleus
Clouds [RAK89]	SC	MRSW	OO distributed OS prototype
CRL [JKW95]	SC	MRMW	Uses PVM for process handling features
DSM-PM2[AB01a]	SC, RC, Java	MRMW	Various high-performance NICs
DSM-Threads [Mue97]	Multiple	MRMW	POSIX threads
Hamster [Sch01]	Multiple	MRMW	SCI interconnect
Larchant [FS94]	Multiple	MRMW	Garbage collection
Millipede [ISW97]	SC	MRSW	Minipages
Mach Shared Memory server [FBYR89]	SC	MRSW	Built on top of Mach Kernel
PAMS [Myr95]	SC	MRSW	Hardware assisted
Phosphorous [CDM94]	Multiple	MRMW	Built on top of PVM
Π Architecture [KBCC93]	Various	Various	Design architecture
Plurix [STS98]	SC	MRSW	Java-based distributed OS
Rthreads [DZU98]	SC	MRSW	Supports PVI, PMI, and DCE
SciFS [KCR98]	SC	MRSW	Distributed file system; SCI interconnect
Shasta [SGT96]	RC	MRMW	Code re-writing and instrumentation
Shared Regions [SGZ93]	EC and RC variations	MRMW	Partial implementation and simulation
SHRIMP [BAC+98]	AURC, HLRC	MRMW	Hardware assisted
StarDust [CP96]	SC	MRSW	Heterogeneous parallel programming env.
Unify [GYF93]	Multiple	MRMW	Spatial consistency
Vote [Cor94]	SC	MRSW	Part of the Peace parallel OS
WindTunnel [HLW95]	Various	Various	Sub-projects: Blizzard, Tempest, Typhoon

Table 3.3: Hybrid Software DSM Systems

The **BAYOU** [DPS+94] system is primarily designed to support "disconnected" applications such as mobile users. Among other things, its emphasis is on developing new replication algorithms that would allow weak consistency. Some typical applications of the system include calendars, databases, or documents "shared" among co-workers that usually allow distant collaboration. Hence, BAYOU is focused on supporting application specific mechanisms instead of generic ones.

The "Coherence Algorithms for SHared MEmory aRchitectures" **CASHMERE** project [SLD+96] is an effort to provide efficient, scalable, shared memory with minimal hardware support. Early simulation results have shown that the performance of non-cache coherent, non-uniform memory access (NCC-NUMA) architectures can be close to the performance of totally hardware coherent multiprocessors. For this reason, the CASHMERE project tries to bridge the performance gap between shared memory emulations on networks of workstations and tightly-coupled cache-coherent multiprocessors with a minimal hardware support. As a proof of concept, the group has successfully built an eight 4-processor NCC-NUMA prototype using DEC 4100 SMPs connected by DEC's proprietary high speed *Memory Channel* network. The success of the project has lead to further research into data sharing. The followup project, called INTERACT, aims at providing efficient and transparent data sharing in a client-server environments such as a datamining application. Most recently, the work on the INTERWEAVE project [CDP+00] complements message passing by allowing users to share data segments across distributed platforms. The unique features of this work are that it allows (i) sharing on "heterogeneous" architectures, and (ii) use of a variety of programming languages. Currently Alpha, Sparc, x86, MIPS, and Power series processors, and C, C++, Java, Fortran 77, and Fortran 90 languages are supported.

CHORUS CONSISTENCY SERVER [AAO92] is implemented as a subsystem running outside the Chorus Nucleus [RMP+87]. It is a part of the Chorus/MiX operating system that implements a distributed version of the System V UNIX on top of the Chorus Nucleus. Chorus/MiX provides a single system image on multicomputer architectures. The consistency server uses sequential consistency algorithms introduced by IVY.

CLOUDS [RAK88] is an object-oriented, micro kernel based distributed operating system developed at Georgia Institute of Technology in the late 80's. Clouds is built on passive *objects* and active entities called *threads*. Each object occupy a distinct location in the Clouds' global virtual address space. Multiple *segments* encapsulating code and data of an application makes an object. Threads provide flow of control in the system. The node's processor, *distributed shared memory controller* (implemented as a software prototype) and a network interface provides the necessary functionality of DSM.

The C Region Library (**CRL**) [JKW95] is an all-software DSM system. The key features are (i) it is both hardware and software architecture, and language independent (therefore it is portable) and (ii) unlike other DSM systems, CRL does not use any features of the underlying hardware and software (e.g., the page fault mechanism) mainly to avoid drawbacks of inefficient interfaces to access them. Its finer granularity avoids false sharing, a common problem of page-based systems, by annotating shared data. CRL was originally implemented on CM-5 and Alewife. The most recent, publicly available version (CRL 1.0) was implemented on a network of Sun Sparc workstations using Berkeley sockets for interprocess communication, and PVM [Sun90] for its group and process management features. Despite the advocated goals for simplicity and its efficiency for some applications, its special API makes parallel benchmark suites such as SPLASH [SWG92] and SPLASH2 [WOT+95] or any other existing parallel application difficult to adopt. The requirement of PVM, which is no longer a popular message passing system, makes the system dependent on *other* software.

DSM-PM2 [AB01a] is not a complete DSM system but rather a portable implementation platform for multithreaded consistency protocols for software distributed shared memory. It provides generic building blocks, allowing easy protocol implementation and comparison within a unified framework. Currently, DSM-PM2 supports three consistency models: sequential consistency (SC), release consistency (RC) and Java consistency [ABM+00]. Different models can be used by a runtime switch without re-compiling applications. DSM-PM2 runtime is available on a variety of clusters operated by UNIX-like operating systems and based on Myrinet, SCI and Ethernet networks.

DSM-THREADS [Mue97] is a distributed multithreaded runtime system that adopts the distributed shared memory approach using a POSIX Threads-like API. The system supports several memory consistency models where both the consistency and the synchronization mechanisms are based on decentralized algorithms.

The **HAMSTER** [Sch01], which stands for Hybrid-dsm based Adaptive and Modular Shared memory archiTEctuRe, project is a shared memory framework which combines the traditional software DSM mechanisms with the advanced hardware capabilities of a SCI based interconnection medium. The underlying low-level API, called SCI-VM, exports a transparent global shared memory abstraction that provides necessary infrastructure for the efficient implementation of different shared memory models.

LARCHANT [FS94] project primarily deals with garbage collection (GC) in shared distributed stores. One particular interest in the project is to provide an algorithm that enables GC in distributed shared memory. The project identified the interactions between coherence and GC, and implemented a proof-of-concept prototype. Further work in the project includes improving performance, and adding heuristics for various issues, such as clustered objects and interaction of GC with multiple consistency mechanisms.

MILLIPEDE [ISW97] is a software platform that builds a "virtual parallel machine" (VPM) on top of a network of computers running Windows NT operating system. Each computer on the network acts as a processor of an SMP multiprocessor, sharing all the memory in the system. The transparent sharing of the network's memory is provided by the DSM component of the system, called MILLIPAGE that implements Sequential Consistency through MRSW protocol.

MACH SHARED MEMORY SERVER (MSMS) [FBS89] is an external paging service built on top of the Mach operating system kernel. Originally implemented on a shared memory system, the MSMS is basically a virtual memory manager that allows virtual pages to reside on different nodes in the system.

The "Parallel Application Management System" (**PAMS**) [Myr95] from Myrias Software Corporation is a hardware-assisted software package that turns a distributed system into a shared memory multicomputer. It was originally developed on the Myrias supercomputer for FORTRAN applications. The most important PAMS "directive" is *PARDO (PARallel DO)*, which automatically invokes the parallel management functions. FORTRAN programs using such directives are preprocessed by the PAMS driver to include the necessary parallel execution functionality. Most of the PAMS directives are similar to OpenMP [Ope97] primitives.

PHOSPHOROUS [CDM94] is a S-DSM developed on top of PVM system. The system has evolved as part of an ongoing research in parallel architectures. The unit of sharing is a "variable," and the data types are handled by packing/unpacking functions of PVM. PHOSPHOROUS uses distributed and dynamic ownership scheme suggested by Li [Li86], and supports data sharing schemes similar to those of MUNIN.

The Π **ARCHITECTURE** [KBCC93] is a new system software design approach that allows its subsystems to be easily tailored for the needs of applications and hardware configurations. The idea behind this new design principle is that the system software should be flexible to accommodate varying needs of the applications. One of the subsystems of the Π ARCHITECTURE is the distributed shared memory, which is based on the idea of making implicit concepts explicit by a process known as *reification*, or materialization. The architecture creates a mapping of an "abstract" application model to the underlying "computational" domain by defining a uniform object model for both. The DSM subsystem is composed of several subcomponents where each subcomponent has two interfaces: a *Function* exporting the functionality of the subcomponent and a *System* interface that controls the actual implementation. The components of the DSM subsystem that sits on top of the Π ARCHITECTURE are the *Hardware Resource Virtualization Component, HVRC*, the *DSM Meta-Object Subsystem*, and the *Language Support*. The DSM Meta-Object Subsystem is used to reify (materialize) the implementation of a DSM system.

PLURIX [STS98] is a native high-speed operating system for PC clusters developed from scratch using Java. It has an implicit DSM storage which directly supports intra-network cooperation. This storage unit implements an optimistic transaction scheme that guarantees consistency of shared data automatically using "Java Objects" technology.

The **RTHREADS** (short for Remote Threads) [DZU98] is based on the POSIX threads (pthreads) model. A pre-compiler transforms a pthreads program into an RThreads program automatically by replacing "all" the global variables into shared variables. Since the translated programs are free from any specific DSM runtime, they can run on heterogeneous clusters and they can also use PVM, MPI, and DCE message passing systems. Rthreads is object-based and primarily supports sequential consistency.

SCIFS [KCR98] is a distributed shared virtual memory built on top of SciOS using the memory mapped file concept. As the name implies, the whole system is built on Dolphin SCI network adapter. A driver module handles all the low level memory operations on this adapter. The system is based on the Linux Virtual File

System (VFS).

SHASTA [SGT96] is a S-DSM system that supports varying granularity. The system re-writes the parallel applications with hooks that intercept the load (read) and store (write) instructions. It uses a flexible cache coherence protocol that supports multiple relaxed memory models to handle the shared data consistency at software level with various run-time overhead reducing techniques.

SHARED REGIONS (SR) [San95] is a framework which provides strategies for efficient cache management in SMPs. The framework allows shared data to be managed at varying granularity required by the application. Further, the coherence decisions are made dynamically and at the software level. The SR provides a run-time system that is located between the operating system and the compilers.

The **SHRIMP** (stands for Scalable High-performance Really Inexpensive Multi-Processor) project investigates how to construct high-performance servers with a network of commodity PCs, and commodity operating systems...

STARDUST [CP96] provides an environment for parallel computing on networks of heterogeneous workstations, supporting both message passing and shared memory programming paradigms. It also supports load balancing, application re-configuration, and fault tolerance.

The objective of the UNIFY project [GYF93] is to build a scalable multicomputer system that is capable of shared memory applications on many nodes that are at geographically distant locations. It is a segment based system supporting three abstractions for shared data. Random access memory is directly addressable. Sequential access memory is accessible in two ways: the front end of the memory is accessed by a read and the back end of the memory is accessed by a write operation. Associative memory is accessed using <key, value> pairs. Sequential access and associative memory uses a new memory model called *spatial consistency*, which determines the relative order of the contents of the replicas of a segment.

VOTE [Cor94] is a communication system that provides support for both message passing and shared memory programming paradigms. VOTE is part of the Peace project [Sch94].

WINDTUNNEL [WWT01] is a large umbrella project that focused on designing cost-effective parallel machines supporting shared memory. The project had three phases developing ideas and research papers on: a simplified shared memory hardware called "cooperative shared memory" [HLRW93] that allowed software managed data movement, a generic "memory interface" [HLW95] that enabled applications to use message passing, shared memory, or combination of both through programming interfaces, and improving performance through more hardware support [MHW03]. The third phase is evolved into a new project called MULTIFACET [Mul03].

Other notable hybrid software DSM implementations include PVMSYNCH [Pit00].

3.3 Summary

I summarized the software distributed systems developed in various forms that range from pure page-based systems to object- or language-based systems to hybrid systems. For a more complete list of DSM implementations please refer to the online DSM bibliography [Esk95]. Javid Huseynov has recently taken over the DSM implementations web site from Peter Keleher. This new web site is currently maintained at `http://www.ics.uci.edu/~javid/dsm.html`.

Most of the software DSM systems in the literature provide only a command-line interface for the execution of application. An exception is the 3rd generation BRAZOS system which provides a graphical user interface for applications on the Windows NT operating system.

In the software distributed system research and development mentioned above, the main focus have been one of the following:

- Design of new and more efficient memory models.
- Approach from a purely theoretical perspective with some prototyping.
- Development of various latency hiding techniques.
- Use of user-level network interfaces.

There are other aspects of research as in the literature that are used to support various issues software DSM systems, such as execution visualization and profiling, post-mortem or algorithm analysis tools. I will not include them here, because this thesis does not focus on such issues.

As mentioned in Chapter 2, research on some other aspects of software DSM is not widely and sufficiently done. These areas of research include architectural issues from both hardware-assisted and software-only perspectives, characterization and possible restructuring of applications based on the memory models. Most of the research describing development and comparative evaluation of analysis of software DSM systems, including the one reported in this thesis, use the de-facto benchmark suites SPLASH [SWG92], SPLASH2 [WOT+95], and NAS [BBLS94]. Albeit it is difficulty to study the performance of other applications for a variety of reasons (e.g., being proprietary or confidential software), the advantages of software DSM systems are yet to be seen on *Grand Challenge Problems*, such as ocean or weather modeling.

Chapter 4

JIA-R—A Case Study

4.1 Overview

This chapter describes the JIA-R software DSM system as a case study. This system is derived from an early version of the JIAJIA software DSM system. I was involved in the development of JIAJIA in it early stages (version 0.9). I introduced many improvements after its first public release, and I have changed its name to JIA-R. Further, unlike its predecessor, JIA-R fully supports the M4 macros that are commonly used in many shared memory applications. The most noteworthy improvements made to the original code are enhanced automated startup procedure, highly efficient message structure, and optimized communication sub-system.

In the following sections, I introduce the common characteristics of JIAJIA and JIA-R, and describe the enhancements of the JIA-R system. I will refer to each processor of the underlying hardware as a *node* and use both words interchangeably in the rest of the thesis.

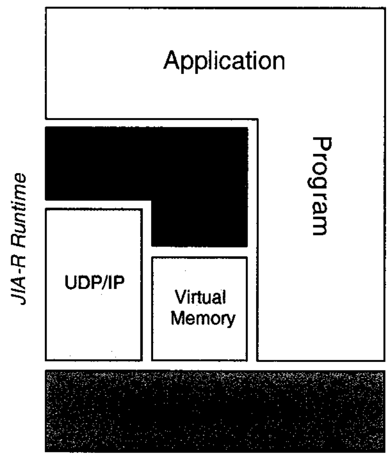


Figure 4.1: Architecture of JIAJIA

4.2 JIAJIA and JIA-R

I have started working with JIAJIA when it was still preliminary. It was being tested on four Sun Sparc workstations, where some of the workstations were also being used to develop the system. The discussion of JIAJIA in the next four sections reflects its very first public release, version 0.9. During the its development, I have ported JIAJIA to about half a dozen different platforms. I have written the handler code for SIGSEGV and SIGIO signals on all the ports, and simplified the original code for the SunOS operating system. My initial involvement goes to a point where I proof-read the original developers many papers and the JIAJIA documentation.

Similar to TREADMARKS, JIAJIA is a software DSM system that is designed to run at user level on a network of Sun workstations. It is originally built onto Solaris (a UNIX flavor) as a runtime library and it uses standard libraries for remote program invocation, interprocess communication, and memory management. JIAJIA implements scope consistency [ISL96] with a lock-based protocol and uses a write-invalidation scheme to handle dirty data. The system also allows multiple writers to alleviate false sharing.

4.3 Memory Organization

As Figure 4.2 shows, JIAJIA organizes the shared memory in an unconventional way. The global shared memory is distributed across the processors. Each processor acts as the home of a portion of the shared memory. Users can specify home size of each processor in a configuration file and hence control initial distribution of shared data. A page is accessed ordinarily when referenced by its home processor. A remote page, on the other hand, is first fetched from its home processor and cached locally for subsequent accesses. A page is always kept at the same user space address, in other words, the logical address of a page is identical on all processors, whether it is a home page or has been cached by the processor. This approach eliminates any address translation upon a remote access and provides a uniform view of the shared memory across the processors. Furthermore, each processor uses a local page table to keep information only about its "cached" pages. It contains the address, current state and a twin (if in RW state) for each cached page.

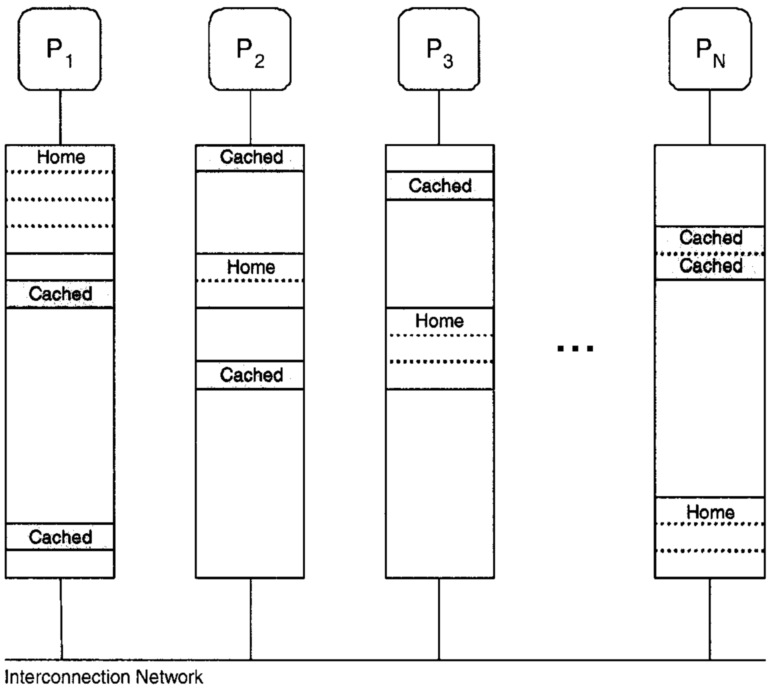


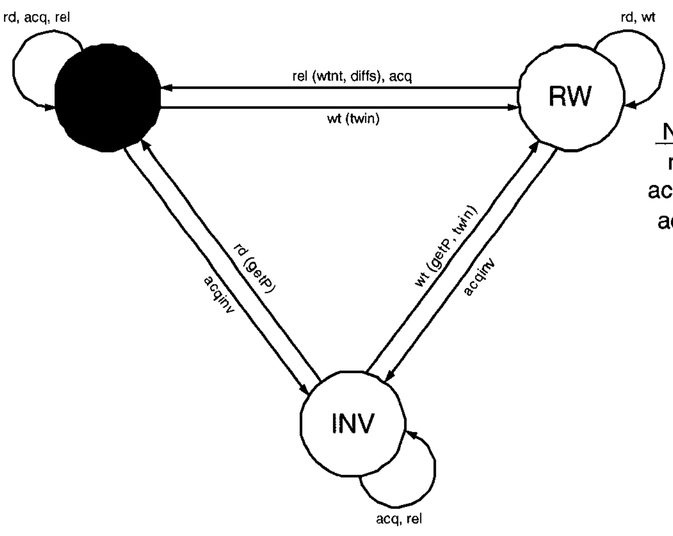
Figure 4.2: Memory Architecture of JIAJIA

With the above memory organization, JIAJIA is able to support shared memory that is much larger than the physical memory of any single processor in the system. Hence, the total size of the shared memory is not limited by the physical memory of a single processor, but only by the virtual memory settings (e.g., maximum allowable user-mappable address range) of the underlying hardware and operating system.

4.4 Coherence Protocol

Currently, JIAJIA provides two synchronization operations (though, others can easily be added): *lock/unlock* and *barrier*. Either one of these operations can be used in an application to control a critical section. A barrier can be viewed as a combination of a lock–unlock pair, but in reverse order: arriving at a barrier *exits* from the "previous" critical section and leaving a barrier *enters* the "next" (new) critical section. Since two barriers are needed to enclose a critical section, the start of an application is considered an implicit entry to the first critical section.

Based on the observation that the overhead of a complex software DSM system may easily offset its benefits, the coherence protocol of JIAJIA, as summarized in



Notes

rd, wt :	read, write
acq, rel :	acquire, release
acqinv :	invalidate the page on acquire
getp :	get the page from its home
wtnt :	send write-notices to lock
diffs :	send page diffs to home(s)
twin :	create a twin of the page

Figure 4.3: Coherence Protocol of JIAJIA

Figure 4.3, is designed to be simple.

The shared pages in an application can either be "local" or "cached" on a given processor. In the former case, the processor is the *home* of the page. During the execution, these pages can be in one of three states: `Invalid(INV)`, `Read-Only(RO)`, and `Read-Write(RW)`. The initial state of the pages at their home processors is `RO`. Since multiple writers are allowed, a page may be in different states after several processors cache it.

Ordinary read and write accesses to a `RW` page, or read access to a `RO` page, or acquire and release on an `INV` or a `RO` page do not cause any change in the page's state. Like the shared pages, each lock is assigned a home processor in a round-robin fashion during system initialization.

On a release, the processor generates "diffs" (run-length encoding of the changes made to a page) for all modified pages and eagerly sends them to their respective homes. Also, the processor sends a release request to the lock's home processor along with the write-notices (basically, a list of modified pages) for the associated critical section. Similarly, the acquiring processor sends a request to the lock's owner and waits until it receives a lock-grant message. Multiple acquire requests for a lock are queued at the lock's home processor. When the lock becomes (or is) available, a lock-grant message is sent to the first processor in the queue, piggy-backed with the current existing write-notices. After receiving the lock-grant message, the acquiring processor invalidates the pages listed in the write-notices and continues with its normal execution.

On reaching a barrier, processors send write-notices along with diffs to the homes of the modified pages. Home processors, in turn, apply the diffs to the original copy of the pages. Thus, each processor resumes execution with an up-to-date view of the shared memory after a barrier.

In summary, the protocol propagates all the modifications (as diffs) to the home processor of a page on a release and to the next processor on the following acquire. This approach keeps the diffs only for a short period of time, hence reducing local diff keeping overhead.

Unlike other DSM systems, JIAJIA does not keep a separate global directory structure, instead, only a lock structure keeps the necessary information, such as ownership, for the relevant pages. This approach further reduces the overall space overhead of the system. JIAJIA's shared memory allocation scheme is only bound by the virtual memory management limitations of the underlying (UNIX or its flavors) operating system. This feature of JIAJIA allows parallelization of applications that require large amounts of shared data.

4.5 Communication Mechanism

Like most second generation page-based DSM systems, JIAJIA uses traditional UDP/IP protocol for both control and data messages. The runtime system keeps track of the messages being sent on each processor, and provides a naive, yet functional, reliability for the communication by inserting sequence numbers to each message. The `SIGIO` signal handler installed during the initialization phase verifies the ordering when a message arrives.

4.6 Programming Interface

JIAJIA implements the SPMD programming model, in which each processor runs the same program on different parts of the shared data. It provides functions for system initialization, shared memory allocation, and synchronization using the following exported programming interface (API):

- `jia_init(int argc, char** argv)`—initializes JIAJIA runtime system. It must be called from every shared memory application.

- `jia_alloc(int size)`—allocates shared memory. The parameter `size` indicates the number of bytes allocated.
- `jia_lock(int lockid)`—acquires a global lock specified by `lockid`.
- `jia_unlock(int lockid)`—releases a global lock. `jia_lock()` and `jia_unlock()` should appear in pairs for obvious reasons.
- `jia_barrier()`—performs a global barrier by preventing any process from proceeding until all processes reach the barrier.
- `jia_wait()`—similar to `jia_barrier()` except that `jia_wait()` does not enforce any coherence operations across processors.
- `jia_clock()`—returns elapsed time since the start of application in seconds.
- `jia_error(char *str)`—prints out the error string `str` and terminates the application.
- `jia_exit()`—prints statistics (optional) and terminates the application.

Additionally, two variables, `jiapid` and `jiahosts`, specify the host identification number and the total number of hosts of a parallel program, respectively. This simple interface is defined in a header file, which must be included by the application.

4.7 Details of JIA-R Enhancements

Typically, software DSM systems follow the general flow as shown in Figure 4.4. This is quite common in S-DSM systems, with slight differences. For example, in TREADMARKS, the master node allocates the global variables and distributes their addresses to the other nodes.

The original system used a simple but time consuming remote process startup procedure. This code was developed on four networked Sun workstations. If there is no Network File System (NFS) mounted directories, then the primary node "copies" the executables to other nodes before starting the execution. This approach was an inefficient as a system startup. JIA-R uses a dynamic mechanism, adopted from TREADMARKS, for remote process creation and invocation. This approach has reduced the startup time to half for all applications. Even larger applications benefit from this mechanism.

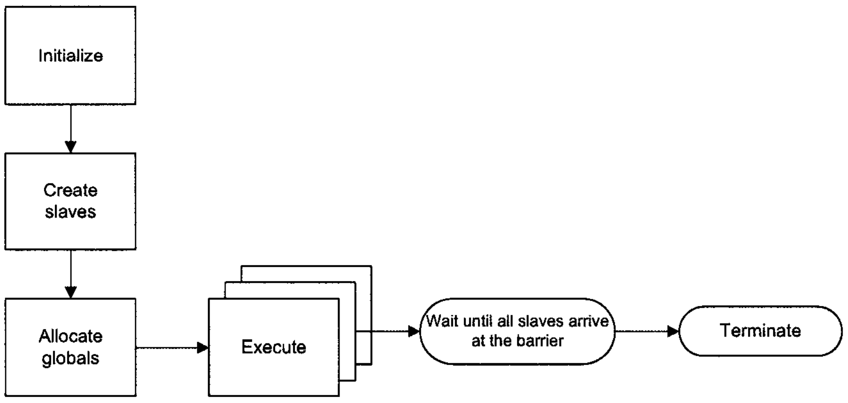


Figure 4.4: General Execution Flow of Parallel Applications

In addition to dynamic port allocation and assignment enhancements in `JIA-R`, I also restructured the messaging data structures and message transfer mechanisms. For example, instead of having a set of buffers for incoming messages and another set for outgoing messages, I introduced a common pool of constant size (currently there are 32 message buffers[1]) for both message types. This approach reduced the memory requirements in the runtime code. In the original code, when a control message or a shared page is sent to other nodes, a message would be sent to the local node if it is involved in the communication scheme. This communication would incur unnecessary delay on the local machine, because the message would travel down and up through the protocol stack, as well as produce an unnecessary I/O interrupt. I optimized message delivery system such that if a message is to be sent to the local machine, I simply remove the message from the outgoing queue and insert it into the incoming queue.

In the original system, each JIAJIA node allocates two ports (channels), one for control messaging and the other for the transfer of the shared pages. UNIX port numbers are used to "name" these communication channels and the numbers are assigned statically and stored in a fix-sized two-dimensional array, large enough to hold ports for maximum number of nodes (it was set to 16) during the startup. The disadvantages of this approach are the static nature of the assignments and the potential conflict of the port numbers on various UNIX operating system flavors. Further, JIA-R dynamically allocates the port numbers by requesting them from the operating system. This approach avoids any potential conflict on ports

[1]This number is not special and it is found by trial-and-error while testing the applications in the benchmark suite.

with other applications and services. The port numbers are reclaimed after the application's execution is completed. In JIA-R, the port allocations are done automatically, based on the number of available nodes.

I amalgamated multiple controls and internal structures of the shared pages that allowed the elimination of several time consuming operations that had to be executed every time a page fault on a shared page occurs. I introduced many metrics into the software to help evaluate the system. They include counters for messages being sent and received, memory operations (including SIGSEGV handling), I/O operations (including SIGIO handling), diff related operations, and synchronization primitives. In addition, timings for these counters are also recorded internally and output at the end of each execution. I also experimented with buffering the output of the programs until the end of their execution. However, the performance gains of doing so were negligible. Lastly, I have developed additional code to allow the users debug their applications by an X-based debugging facility.

The current directory structure of the JIA-R source code base is shown below:

```
.../JIA-R/
         /src
             /...
         /doc
         /appls
               /barnes/src
                          /...
                      /null
                           /Makefile
                      /linux
                      /solaris
                      /aix41
                      /Makefile.common
                      /datafiles
               /...
         /lib
             /linux
                   /libjia.a
                   /Makefile
             /solaris
             /aix41
             /Makefile.common
         /m4.macros
         JIA-R-clean
         JIA-R-make
         README
         README.JIA-R
```

I identified the more complicated and important additions and modifications made to the original JIAJIA code as three groups: *automatic startup and initialization*, *new messaging system*, and *communication subsystem optimizations*.The following sections describe the details of these enhancements.

4.7.1 Automated Startup and Initialization

When a parallel application is submitted to the system, it starts executing on a node called the *master*. The master node then spawns all the other nodes called the *slaves*. Nodes run an exact copy of the application, except the master node does a few additional tasks related to the setup, initialization, and termination. The master node is mainly responsible for starting the slaves, in addition to execute the application itself. All nodes read a small configuration file and learn about their peers, including the master. For simplicity, the first node in this file is also the master.

The first initialization step, done by all the nodes, is to initialize memory, communication, and synchronization sub-systems. During the memory initialization, a `SIGSEGV` handler is installed to deal with "shared memory" accesses and necessary data structures for the shared memory are created. Each node then allocates a message common buffer pool for all incoming and outgoing message traffic. A `SIGIO` handler is also installed during the communication sub-system initialization. I have completely re-written this startup and initialization component.

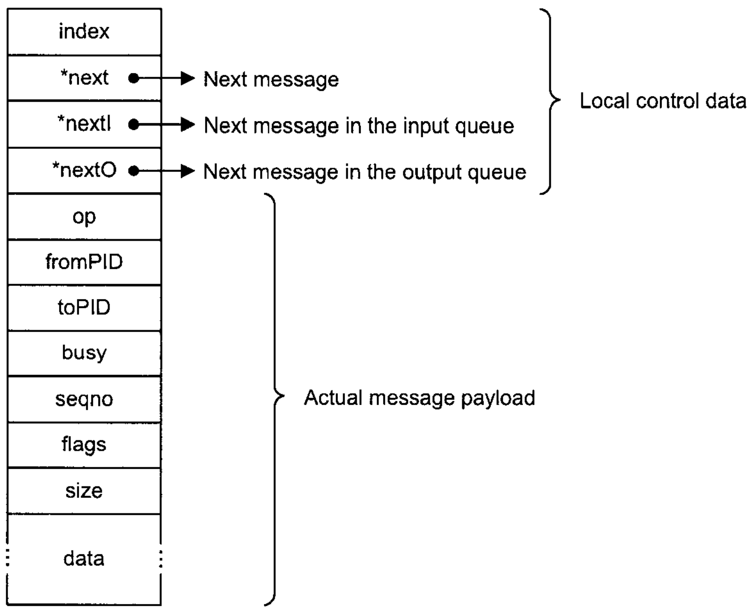


Figure 4.5: JIA-R Message Structure

4.7.2 New Messaging System

I have re-designed the message structure as Figure 4.5 shows. By introducing 16 bytes of additional control structure, I was able to use the same structure for both incoming and outgoing messages. Further, with this small additional data, which is only used locally, I was able to eliminate redundant message structures in the system.

4.7.3 Communication Subsystem Optimizations

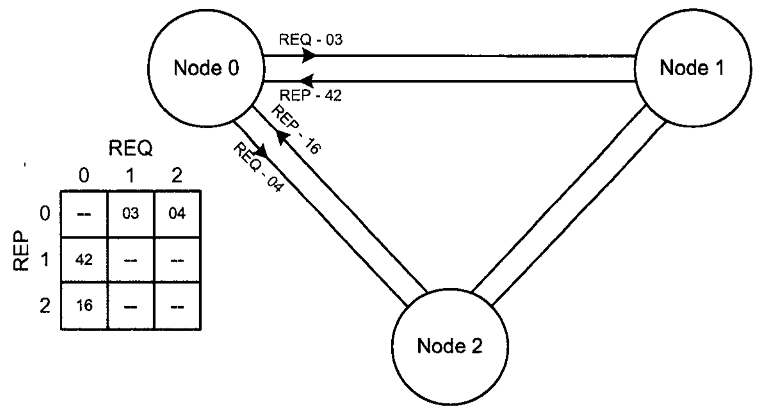


Figure 4.6: JIA-R Communication Structure

I have re-designed and implemented the communication subsystem of JIAJIA. In JIA-R, not only there are two communication channels between a pair of nodes, but also those channels (sockets) are created dynamically, eliminating port number conflicts in the UNIX operating system. The port numbers are assigned dynamically each time a channel is created. This is done by the `socket()` system call. The send operations for both the control messages and the data transfers are done on one socket. Similarly, the receive operations of the same are done on another socket. Figure 4.6 shows the communication channels assigned on a three-node system. Each node has a pair of sockets for every other node in the system. One socket is dedicated for sending messages and the other is for receiving messages. They are illustrated as REQ - 03 and REP - 42 between node 0 and node 1, respectively. The REP - XX sockets are set to receive asynchronous messages, thus trigger SIGIO interrupts, because messages may arrive at random. The 3×3 matrix below node 0 illustrates the data structures used to hold the socket numbers between the pairs of nodes.

4.8 Summary

I have detailed the important features of the original JIAJIA S-DSM software and discussed the optimizations and improvements introduced to the new system. Unfortunately, I was not able to locate unmodified original version 0.9 of JIAJIA. Moreover, the partially modified version did not compile on the newer platform. Nevertheless, it is a conservative assumption that the new "internal" structure and optimizations have improved the performance of the applications by as much as 15%. Moreover, these enhancements have increased readability of the source code.

I believe additional improvements to JIA-R software DSM system can be made to further improve its performance. Possible improvements include re-designing a more efficient locking mechanism of the coherence protocol, incorporating process migration and/or page migration, using threading to hide communication latencies, developing more efficient communication library, such as Fast Messages [PLC95] or Active Messages [vCGS92]. Both JIA-R and its predecessor JIAJIA are single-threaded systems. Related research has shown that using multi-threading techniques and hiding communication latencies, application performance can be improved in many cases [AB01b, SB97, ISS98, JB02, KC98, LR00].

Apparently, some of these suggestions have been incorporated into the newer version of JIAJIA. However, the details presented in related publications are rather sketchy. Also, I have been unable to acquire the new code with those additions, thus I was unable to compare JIA-R with the newer versions of JIAJIA.

Chapter 5

Evaluation of JIAJIA and JIA-R

This chapter presents the performance results of JIAJIA, CVM, TREADMARKS, and JIA-R. I started this research by evaluating various software DSM systems whose source code is freely available. At the time, there were three possibilities. I omitted TREADMARKS, because it had strict licensing rules and CVM because of its immature status at the time. As it turned out, the developer of CVM has not released a distribution beyond version 0.2. Hence, I decided to work with JIAJIA, despite its early development stage and I acquired the full source code of JIAJIA without any licensing issues. I then compared the performance of several applications with JIAJIA, CVM and TREADMARKS. The first part of the evaluation component of this chapter is devoted to these comparisons. In the latter part, I evaluate the new JIA-R DSM system and present the results of my experiments on three different networks.

5.1 Experimental Platforms

The platforms for testing JIAJIA and JIA-R with a set of applications has varied over the years. Initially, I used a 64-node IBM SP2 cluster at the Center for High Performance Computing at the University of Utah [EM98]. Although there were two types of nodes with slightly different characteristics on the cluster, I used up to 16 identical "thin nodes" for the experiments. Each thin node was equipped with 120 MHz POWER2 Superchip processor and 128 MB physical memory. All the cluster nodes were interconnected with a high performance multi-stage Omega switch (SP2 Switch) which provided a minimum of four simultaneous paths (with a bandwidth of 80 megabits each) between any pair of nodes. The nodes were also connected to the outside world by both Ethernet and FDDI links. A full version of

AIX 4.1.5 operating system was running on each node. Although all the nodes were available for public use with no restrictions, I ran my experiments on dedicated nodes, i.e., with no other user process, thus utilizing the full capacity of each node. I will refer to this platform as the **SP2 Cluster**. I compared the performances of JIAJIA, TREADMARKS, and CVM on that platform.

The second, and the most recent platform, consists of a 16-node cluster of IBM M-Pro PCs. Each node has an 866Mhz P-III Coppermine processor with 256KB L2 cache, 256MB RAM, and 18GB wide-SCSI disk drive. All nodes run Redhat Linux 7.2 with the 2.4.9-31 kernel. The nodes are interconnected with three networks: 100 Mbps (Fast) Ethernet, Gigabit Ethernet, and Myrinet (the third generation interface, a.k.a. Myrinet-2000). Each network has its own dedicated switch, and the cluster is connected to outside world on their Fast Ethernet via a switch. I refer to this platform as the **16-SingleCPU Cluster**.

In both platforms, I used the public-domain C compiler `gcc` with option `-O2` for the compilation of both the DSM systems (JIAJIA, CVM, TREADMARKS, and JIA-R) and the applications.

5.2 Benchmark Suite

There are several benchmarks for a variety of environments on the Internet. However, only a few of them suit the purpose of this research. I have selected applications from three widely used parallel benchmarks, namely, SPLASH [SWG92], SPLASH2 [WOT+95], and NAS [BBLS94], covering a broad range of problem domains and varying data access patterns.

SPLASH is a collection of parallel applications developed primarily for use in the design of shared memory multiprocessors, and in the study of centralized and distributed shared memory multiprocessors. Consequently, these applications are tailored for fine-grain, hardware (sequential) cache-coherent systems. SPLASH2 is the successor of the SPLASH suite of applications. Applications in the NAS Parallel Benchmark suite are developed for evaluating high performance computers.

I have experimented with a total of seven applications: Barnes, EP, LU, SOR, TSP, Water, and Matmul. Barnes and Water are from SPLASH. EP is from the NAS. LU is from SPLASH-2. The three remaining programs extended the applications selected from the above benchmark suites. Matmul is a locally developed simple matrix multiplication program, manipulating large matrices. SOR is an implementation of a method of solving partial differential equations. Finally, TSP is developed at the Rice University in conjunction with their TREADMARKS system.

Table 5.1 lists relevant characteristics of the applications in the test suite. Note that for simplicity, both JIAJIA and JIA-R allocate a new page for each `jia_alloc()` call. So, the page count in the last column of the table does not necessarily reflect the actual size of the shared data.

Appl.	**Sync.**	**Dataset** *Sm/Md/Lg*	**Shared Mem** *(4K-pages)*
Barnes	B	8192 bodies	499
		16384 bodies	995
		32768 bodies	1987
EP	B	2^{24} numbers	1
		2^{26} numbers	1
		2^{28} numbers	1
LU	B	1K×1K	2060
		2K×2K	8206
		3K×3K	18450
Matmul	B	1K×1K	2060
		2K×2K	8206
		3K×3K	18450
SOR	B	1022×511	2048
		2046×1023	4096
		3070×1535	12288
TSP	L	18 cities	197
		20 cities	197
		19 cities (big)	197
Water	B, L	343 mols	27
		1000 mols	71
		1728 mols	121
B=barrier, L=lock			

Table 5.1: Application Characteristics

Below is a summary of the benchmark applications used in my evaluations. For most of them, more detailed descriptions can be found in the literature [SWG92, WOT+95, BBLS94]. Please note that not all of them are used in my different experiments presented in the next sections. Nevertheless, I included a brief summary of them all below for completeness.

Barnes

Barnes is the implementation of Barnes-Hut hierarchical N-body algorithm which simulates the interaction of a system of particles in 3-dimensions over a number of time steps. Every body in the program is modeled as a point of mass and exerts forces on all other bodies in the system where the space is represented by an octree (octal tree) in three dimensions. The root of the tree represents a space cell containing all bodies in the system. Depending on the user input of the number of bodies, the tree is built by adding particles into the initially empty root cell, and subdividing a cell into its eight children as soon as it contains more than a single body. The leaves of the resulting tree are individual bodies, and the internal nodes are cells. There are five phases for each time step:

1. `MakeTree`: Construct the tree.
2. `MassCompute`: Compute center of mass of each cell.
3. `GetBodies`: Partition bodies among processes.
4. `ForceCompute`: Compute forces on partial (self) bodies.
5. `Update`: Update position and velocity of partial bodies.

where the tree is traversed once for each body to compute the net force acting upon that body.

In the program, both body and cell arrays are shared and the first two phases are executed by the master process to reduce extensive data exchange traffic. Barriers are used to synchronize the computations after `MassCompute` and `ForceCompute` steps. In this version, there are no locks. Modifications to cell structures during the steps are done on local copies and then merged together at barriers. This version was slightly modified by the TREADMARKS group at Rice University. Basically, they've eliminated one barrier from the code.

EP

EP (embarrassingly parallel) kernel benchmark is heavily computational. The program generates 2×2^n Gaussian random numbers, where n is given as a command line argument, and tabulates them in successive annuli.

After the random numbers r_i are generated in the interval (0,1) for $1 \leq i \leq 2n$, the following algorithm is executed:

```
for $j = 1$ to $n$ do
    $x_j = 2r_{2j-1} - 1$;
    $y_j = 2r_{2j} - 1$;
end for
$k = 0; j = 1$;
while $k \leq n$ do
```

```
    if $(t_j = x_j^2 + y_j^2) \leq 1$ then
      $X_k = x_i\sqrt{-2logt_j/t_j}$;
      $X_k = x_i\sqrt{-2logt_j/t_j}$;
      $k = k + 1$;
    else
      $j = j + 1$;
    end if
  end while
```

Finally, the program tabulates ten (Q_l) values as the count of (X_k, Y_k) pairs, where $l \leq Max(|X_k|, |Y_k|) \leq l + 1$.

Since each process can generate part of the uniform random numbers independently, this program can easily be parallelized (hence the name embarrassingly parallel) such that each process can accumulate the Q_l sums independently, and only pass the partial sums to the master process at the end of the computations. So, the only communication between the processes occur toward the very end when all processes participate in a reduction operation, protected by a lock, to generate a global sum. In this sense EP provides an estimate of the upper achievable limit for floating-point performance on a particular system.

LU

LU is a matrix decomposition program that factors a dense matrix into the product of a lower triangular matrix and an upper triangular matrix. The dense $n \times n$ matrix is divided into an $N \times N$ array of $B \times B$ blocks $(n = NB)$ to exploit temporal locality on sub-matrix elements. This version of the kernel (LU-Contiguous) factors the matrix as an array of blocks, allowing blocks to be allocated contiguously and entirely at the processes that own them, even though these blocks are not contiguous in the original array. The algorithm factors the matrix in several steps separated by barriers.

Matmul

Matmul is a simple implementation of the inner product algorithm used to multiply two $N \times N$ matrices. Both the multiplicand matrices and the product matrix are shared. The work is divided among processes, where each process computes the result for a certain number of rows. The partial results are then merged at a barrier after the computations. The algorithm is only timed for the multiplication part, i.e., initialization of A and B are not taken into consideration for timing. Two barriers, before and after, are used by the worker processes while they multiply band of rows of A with columns of B.

SOR

SOR is an iterative algorithm for solving discretized Laplace equations on a grid of points represented by a matrix. For each iteration, each element of the matrix is updated for the next iteration by some *function* of the neighboring elements. For testing purposes, this function is the average of the element's nearest neighbors. The two matrices, *red* and *black* are used to hold both temporary and current values at each iteration, taking turns. Each iteration has two phases. Only one of the colored matrices is calculated during each phase. This scheme interchanges for each iteration during the whole process. For each iteration, parallel processes operate on a band of rows of both matrices and two barriers are used for synchronization. Due to the function of the algorithm, the size of the matrices are $2M + 2 \times 2N + 2$.

TSP

TSP solves the classical traveling salesman problem using a branch-and-bound algorithm to find the shortest path (tour). The cities are represented as the nodes of a directed graph in the program. The program starts with an initial partial path and recursively permutes over the remaining nodes, updating the partial path, if and when necessary, until it finds the shortest path between two cities. In this sample solution, if the length of a partial path plus a lower bound of the remaining portion of the path is longer than the current shortest path, the search stops as the solution cannot lead to a shorter path than the current maximum length path.

This version of the program maintains a shared priority queue of partially evaluated paths, where the head contains the path with the shortest lower bound on its length. Other shared data structures are an array of tour data structures, stack pointers to unused tour structures, and the current global maximum tour and its length.

A lock is used to guard this queue and the recursion stack. The process that finds a shorter path than the global minimum gets the lock to set a new global minimum value for the path length.

Water

Water is an N-body molecular simulation program that evaluates forces and potentials in a system of water molecules in the liquid state using an $O(n^2)$ brute force method with a cutoff radius. Water simulates the state of the molecules over a user defined number of steps. Both intra- and inter-molecular potentials are computed in each step. The most computation- and communication-intensive part of the program is the inter-molecular force computation phase, where each processor computes and updates the forces between each of its molecules and each of the $n/2$ following molecules in a wrap-around fashion. The main data structure used in this application is a single one dimensional array representing a water molecule. Each molecule structure contains information as the center of mass, the forces of its three atoms, their displacements, and the first six derivatives of the displacements. I used the slightly revised TREADMARKS version [LDCZ95] of Water in this study.

5.3 Experimental Methodology

My approach to evaluate both JIAJIA and JIA-R was similar to that of others. In the following first section, I compared the performance results of JIAJIA with CVM and TREADMARKS. In the second section, I present the results of my experiments with JIA-R on three different network interconnects.

Since ensuring reproducible experiments is the most crucial aspect of the evaluation, I ran my experiments on a dedicated cluster and collected speedups. All the results were logged using a simple scripting procedure. The reported results are the best times out of five executions of each application, since I believe that the other slightly higher times were caused by the NFS traffic on the network. In any case, the observed variance between the five execution times is within 0.5–2% margin. Further, the part of the runtime code to collect the statistics had a negligible overhead (less than 1%) on the execution times of each application. Unless otherwise noted, I report all the performance results described below in seconds. Previous work in the literature reported short periods of execution times. This may not necessarily reveal the actual execution pattern of an application. I have executed each application for longer periods to allow them stabilize in terms of execution patterns. Also, I ran each application five or more times and selected the best time among them.

5.4 JIAJIA on SP2

This section summarizes my early work [EM98, EMHS99] on JIAJIA. I present a comparison of JIAJIA with CVM and TREADMARKS. All the experiments were done on the **SP2 Cluster**.

5.4.1 Comparison of JIAJIA and CVM

In this part of the study I used six benchmark applications: Barnes, EP, LU, Matmul, TSP, and Water. Table 5.2 shows a summary of the overall results. The numbers in the table indicate the execution time of each application in seconds.

Appl.	Size	Number of Cluster Nodes									
		1		2		4		8		16	
		JIAJIA	CVM	JIAJIA	CVM	JIAJIA	CVM	JIAJIA	CVM	JIAJIA	CVM
Barnes	8192 bodies	51.60	49.69	28.39	29.62	19.65	20.03	19.62	16.33	29.33	18.62
	16384 bodies	118.86	114.34	65.07	66.97	43.53	44.75	40.20	34.37	59.78	38.03
	32768 bodies	270.56	260.16	148.24	–	98.15	97.24	86.15	72.83	120.61	80.25
EP	2^{24} numbers	74.72	78.42	37.58	39.17	19.27	19.60	9.37	9.84	4.73	4.96
	2^{26} numbers	300.46	314.46	151.24	157.27	75.05	78.94	37.51	39.76	19.27	19.63
	2^{28} numbers	1203.83	1260.50	606.96	630.44	301.15	515.24	150.66	158.02	75.62	79.36
LU	1K×1K	6.59	15.62	6.00	9.93	3.33	6.15	2.65	4.00	2.52	3.03
	2K×2K	53.55	–	40.52	–	19.64	–	14.85	–	10.04	–
	3K×3K	182.28	–	136.22	–	62.03	–	42.73	–	26.01	–
Matmul	1K×1K	45.82	34.96	24.57	19.99	13.58	10.94	8.24	7.68	10.97	7.15
	2K×2K	366.92	280.69	196.37	160.82	104.81	86.97	58.72	53.75	45.26	46.66
	3K×3K	1243.07	952.55	669.63	770.53	352.91	391.22	190.98	389.47	118.09	370.07
TSP	18 cities	42.80	122.34	22.74	66.01	12.44	34.09	7.34	20.88	4.92	13.48
	20 cities	277.07	799.24	149.61	407.13	80.22	211.88	47.03	116.98	36.08	64.19
	19 cities (big)	434.92	1235.71	226.41	637.37	118.75	337.92	59.38	154.33	33.56	91.32
Water	343 mols	43.03	51.09	30.98	34.79	15.92	22.77	14.76	17.74	26.07	16.35
	1000 mols	363.93	439.43	195.13	250.22	102.52	140.45	60.91	86.06	55.09	63.51
	1728 mols	1115.76	1316.22	575.32	716.93	294.60	390.53	158.53	221.13	110.99	143.40

Table 5.2: Performance of Applications with JIAJIA and CVM

Despite the elimination of the critical and only lock from the code, the overall performance of Barnes was not good. This application suffers from not only excessive but also irregular fine-grain sharing, which causes invalidation and refetching of whole pages. Barnes only achieved speedups 1.76, 1.99, and 2.24 on 16 processors for 8192, 16384, and 32768 particles, respectively. This is an example of an application that is not suited for JIAJIA.

EP achieved an excellent performance as expected and scales well. The speedups are near linear (for example, 15.92 on 16 processors with 2^{28} random numbers) because the only communication among the processors, which is compensated by the high computation rate, occurs at the end of the number generation phase to accumulate the tabulated results.

The speedups obtained from LU ranged from 2.62 for a 1K×1K matrix to 7.01 for a 3K×3K matrix. The results confirmed my expectations that higher speedups and better performance can be achieved with larger problem sizes. I used a block size of 64 bytes, because (after performing some additional tests) I observed that this size (*as opposed to 16 recommended by the developers of the application*) yields the best performance on SP2 cluster, probably because of bigger cache lines on SP2 nodes.

The locally developed matrix multiplication program Matmul also achieved good speedups, particularly for larger data sets. As mentioned earlier, 1-processor version of this application showed an execution anomaly. For this reason, I used the execution times of the sequential version (**SEQ** column) to calculate relative speedups. Although, the speedup on 16 processors is low (4.18) for 1K×1K matrices, it is very good (10.53) for 3K×3K matrices. Matmul clearly benefits from initial distribution of shared data among processors.

TSP uses only locks for synchronization while executing the branch-and-bound algorithm. There are also two barriers in the application, before and after the recursive evaluation of the tours. I tested TSP with 18, 19, and 20 cities with recursion levels (`-r` option) of 14, 14, and 15, respectively. Incidentally, the program finds the minimum tour length for 20 cities faster than for 19 cities which can be attributed to the setup of the input data. The speedup for all three data sets up to four processors is near linear. As the number of processors increases beyond four, the larger data sets are penalized by the lock-based coherence protocol. JIAJIA transfers mostly whole pages because the accumulation of lock-releases unnecessarily invalidates more pages a lock is acquired. Nevertheless, it achieved good speedups, despite this deficiency.

I simulated 343, 1000, and 1728 molecules with Water, each for 25 steps. The amount of shared data in the revised Water code is smaller because the molecule data is split into shared and non-shared parts in this version. With the small data set, the speedup is not good, the performance degrading for eight or more processors. The major cause of this problem, which is usually more detrimental with the 343-molecule case, is extensive fine-grain sharing, because the algorithm requires that each processor fetches modified data from half of the other processors. However, with the large data set, this overhead is compensated for by higher computation rate, leading to better speedups. In the test runs, I obtained speedups 1.65, 6.72, and 10.05 on 16 nodes for 343, 1000, and 1728 molecules, respectively.

Overall, the applications in the test suite achieved speedups from 1.34 (LU) to 1.99 (EP) on 2 processors with small data set, from 1.83 (Barnes) to 8.13 (TSP) with medium data set, and from 1.65 Water to 15.92 EP with large data set on 16 processors. The major causes for the variance between speedups may be irregular shared data access patterns, low computation to communication ratio and, to some extent that JIAJIA protocols have not yet been optimized.

I ran the same applications with CVM software DSM. CVM is a protocol testbed with a variety of other models such as sequential consistency and single-writer LRC. I only tested multiple-writer LRC protocol. Also, JIAJIA and CVM use totally different allocation schemes for shared memory. JIAJIA distributes the shared pages among processors whereas CVM replicates them on each and every processor. Although static data distribution through home processors improves performance of certain applications, the positive effect of this approach is not always possible.

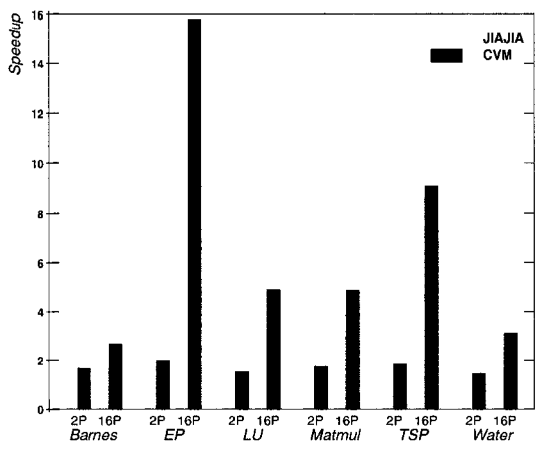
Figure 5.1 summarizes the comparison of speedups achieved by three different data sets with JIAJIA and CVM. This figure also shows that most applications achieve higher speedups with JIAJIA as the data set size increases and that JIAJIA performs better than CVM in all applications with larger data sets, except Barnes and TSP.

Unfortunately, not all applications ran successfully under CVM. I was unable to run LU with a 2K×2K matrix on more than 4 processors, and a 3K×3K matrix on more than a single processor. This application ran successfully, however, on 1–16 processors with smaller (up to 1K×1K) matrices. Similarly, for unknown reasons, Barnes did not run to its completion with 32768 particles on 2 processors.

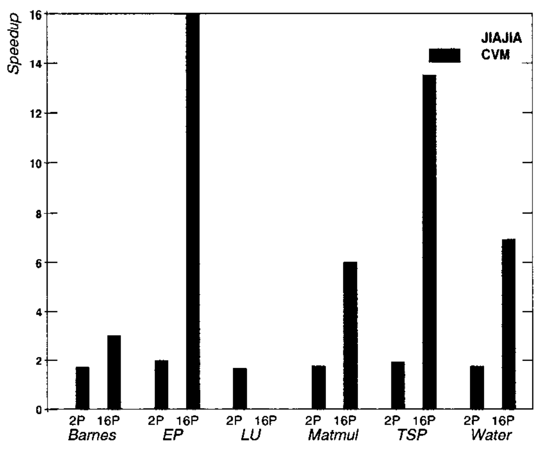
Overall, the applications generally ran faster with JIAJIA, mainly because of the simpler coherence protocol and low overhead of the system.

5.4.2 Comparison of JIAJIA and TMK

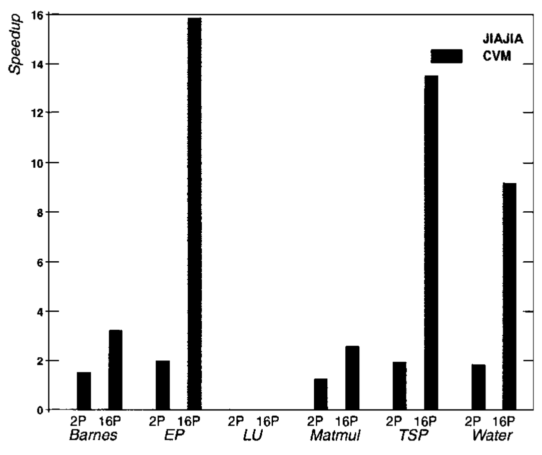
In this part of the study, five benchmark applications are used: EP, LU, Matmul, TSP, and Water. Table 5.3 shows a summary of the overall results. The numbers in the table indicate the execution time of each application in seconds.



(a) Small Data Set



(b) Medium Data Set



(c) Large Data Set

Figure 5.1: Comparative Speedups: JIAJIA vs CVM

Figure 5.2 shows the speedups of applications on 2, 4, 8, and 16 processors for both JIAJIA and TREADMARKS.

EP performed similarly with both JIAJIA and TREADMARKS, as expected. LU and Matmul achieved better speedups with JIAJIA, while some problem sizes of Water and TSP performed only slightly better with TREADMARKS. Overall, JIAJIA versions of the applications showed comparable performance to TREADMARKS despite its un-optimized and primitive protocols. I also collected the to-

Appl.	Size	Number of Cluster Nodes									
		1		2		4		8		16	
		JIAJIA	TMK	JIAJIA	TMK	JIAJIA	TMK	JIAJIA	TMK	JIAJIA	TMK
EP	2^{24} numbers	74.72	78.23	37.58	39.13	19.27	19.62	9.37	9.82	4.73	4.91
	2^{26} numbers	300.46	314.34	151.24	157.38	75.05	78.88	37.51	39.38	19.27	19.61
	2^{28} numbers	1203.83	1261.43	606.96	630.10	301.15	315.19	150.66	157.98	75.62	79.06
LU	1K×1K	6.59	15.74	6.00	11.11	3.33	7.27	2.65	6.23	2.52	5.18
	2K×2K	53.55	126.39	40.52	86.16	19.64	55.42	14.85	42.31	10.04	36.33
	3K×3K	182.28	427.31	136.22	299.44	62.03	222.06	42.73	178.01	26.01	151.79
Matmul	1K×1K	45.82	48.44	24.57	26.28	13.58	16.01	8.24	13.52	10.97	13.54
	2K×2K	366.92	390.64	196.37	254.09	104.81	162.09	58.72	156.72	45.26	168.57
	3K×3K	1243.07	–	669.63	–	352.91	–	190.98	–	118.09	–
TSP	18 cities	42.80	55.34	22.74	30.06	12.44	16.10	7.34	9.47	4.92	6.10
	20 cities	277.07	356.22	149.61	182.01	80.22	100.07	47.03	52.15	36.08	29.48
	19 cities (big)	434.92	563.64	226.41	291.17	118.75	155.35	59.38	76.18	33.56	42.64
Water	343 mols	43.02	53.07	30.98	35.10	15.93	22.05	14.76	15.94	26.07	16.09
	1000 mols	369.93	454.67	195.13	257.29	102.52	143.38	60.91	85.78	55.09	61.27
	1728 mols	1115.76	1360.65	575.32	736.83	294.60	398.11	158.53	224.12	110.99	144.72

Table 5.3: Performance of Applications with JIAJIA and TMK

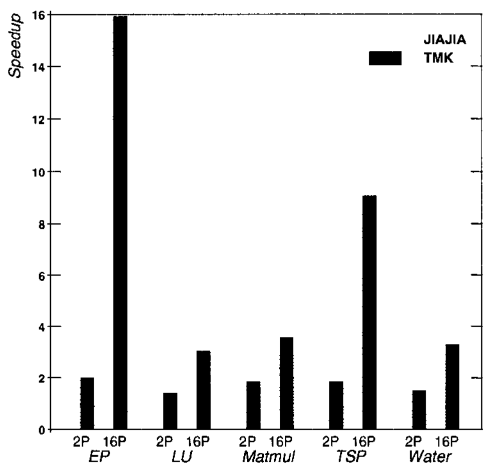
tal number of messages and data exchanged by the processors with JIAJIA and TREADMARKS. Table 5.4 shows these statistics.

Although the number of messages in the TREADMARKS version of Water is an order of magnitude more than that of the JIAJIA version, the total data transferred is only twice as much. The reason for the higher data transfer rate is that JIAJIA usually sends whole pages because of the write-invalidate protocol, but causes less diff accumulation. TREADMARKS, however, sends many small diff messages.

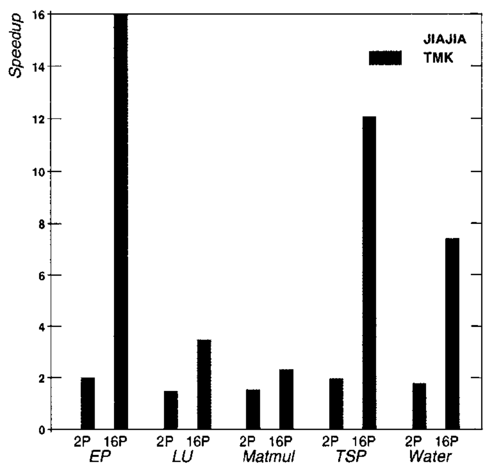
The amount of transferred messages and data are quite similar in both JIAJIA and TREADMARKS versions of LU on 2 processors. The data amount quadruples with TREADMARKS when I scale to 16 processors, while it only doubles with JIAJIA on the same number of processors. The message count on 16 processors is more with TREADMARKS, even though it is less on 2 processors. This roughly indicates that JIAJIA's protocol scales better in applications like LU.

EP performs nearly identical with both JIAJIA and TREADMARKS. JIAJIA sends more messages and data because of unnecessary invalidation of the only shared page.

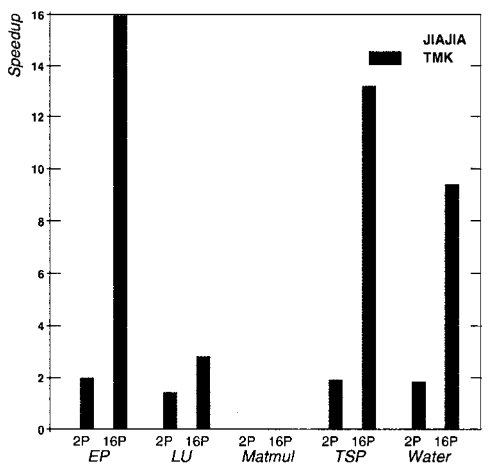
JIAJIA's lower speedup in TSP is caused by the extensive amount of message (5 times more) and data (75 times more) transfers. Nevertheless, the speedups achieved by TREADMARKS does not reflect this advantage because it also suffers from the higher overhead of its diff management.



(a) Small Data Set



(b) Medium Data Set



(c) Large Data Set

Figure 5.2: Comparative Speedups: JIAJIA vs TREADMARKS

Matmul transfers slightly more data and messages with JIAJIA, but again, its simple protocol helps achieve better speedups. TREADMARKS cannot execute Matmul, for example, with 3K×3K matrices, even when the shared page pool has 65,636 pages. This is a side effect of the fact that the shared memory in TREADMARKS is allocated on the system's heap and therefore limited to a certain value which is different for different operating systems. JIAJIA's shared memory allocation scheme

Appl.	Size	No. of Procs	JIAJIA		TreadMarks	
			No. of Messages	Total Data	No. of Messages	Total Data
EP	Sm	2	18	8.6 KB	9	4.3 KB
		16	270	128.6 KB	181	90.8 KB
	Lg	2	18	8.6 KB	9	4.3 KB
		16	270	128.8 KB	177	88.5 KB
LU	Sm	2	4,496	8.6 MB	4,302	8.3 MB
		16	19,400	34.2 MB	29,196	54.8 MB
	Lg	2	38,040	72 MB	37,513	73 MB
		16	152,288	144 MB	155,777	298 MB
Matmul	Sm	2	4,100	8.1 MB	3,079	6.0 MB
		16	61,470	121.6 MB	58,169	116.9 MB
	Lg	2	16,388	32.4 MB	12,302	24.1 MB
		16	245,760	486.4 MB	236,336	468.2 MB
TSP	Sm	2	1,848	2.3 MB	551	70.1 KB
		16	5,409	6.5 MB	2,423	1.4 MB
	Lg	2	13,000	20.3 MB	2,763	268.4 KB
		16	27,357	43.1 MB	10,759	4.9 MB
Water	Sm	2	5,496	8.6 MB	44,033	12.1 MB
		16	139,911	163.6 MB	324,472	145.9 MB
	Lg	2	12,100	33.9 MB	200,526	59.5 MB
		16	210,299	324.5 MB	1,546,692	718.9 MB

Table 5.4: Message count and data sizes with JIAJIA and TREADMARKS

is only bound by the virtual memory management limitations (e.g., less limited than the heap size itself) of the underlying UNIX operating system. This feature of JIAJIA allows parallelization of applications that require large amounts of shared data, practically as large as the size of the available address space in the virtual shared memory.

I observed that JIAJIA is sensitive to load imbalance of the component processors on the SP2 cluster. This is also true for the TREADMARKS system.

5.5 JIA-R on the 16-SingleCPU Cluster

This section extends my recent work [Esk02] and presents some new results. The characteristics of the network interconnects used in the experiments are given in Table 5.5. To the best of my knowledge, this work is unique in that it evaluates the performance of a software DSM system on identical hardware using three different network interconnects.

Interconnect	Type	Brand/Model
Fast Ethernet	on-board	Intel EPro 100
Gigabit Ethernet	32-bit PCI	Syskonnect SK-9843
Myrinet	64-bit PCI	M3S-PCI64B-4

Table 5.5: Network Interconnects

I ran the applications on Myrinet using their GM-Sockets[1] interface, both with 1518-byte standard and 9000-byte "jumbo" MTUs. As the page size on Pentium class computers is 4096 bytes, I anticipated that the results will show at least modest improvements using a larger MTU, since it captures an entire message carrying a page. To my surprise, the results have only shown minimal and negligible difference between each run. I attribute this unexpected behavior to GM-sockets software's the first, perhaps then not optimized, public release.

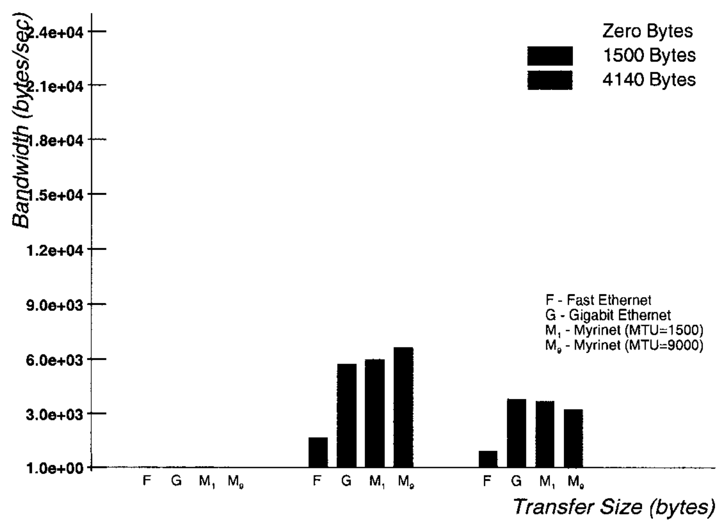


Figure 5.3: Bandwidths of three Network Interconnects

I did some micro-benchmark tests on Fast (100Mbit) Ethernet, Gigabit Ethernet, and 3rd generation Myrinet (a.k.a. Myrinet-2000) networks and measured their sustained user level performance. I experimented with 2 different buffer sizes: 1500 bytes for the default MTU and 4140 bytes for a full page transfer. The additional 44 bytes in the latter buffer size is for the header information used by JIA-R to transfer data between the processes. For the user level times, I ran a simple buffer transfer program (based on UDP sockets) 500000 times and observed the following average

[1]This software runs on top of Myrinet's firmware, called GM, and it provides BSD-Sockets interface to the applications running on the PCs.

times for each of the three networks (all times are given in microseconds):

Interconnect	Buffer Size	
	1500-byte	4140-byte
Fast Ethernet	389	593
Gigabit Ethernet	129	190
Myrinet (MTU=1500)	179	237
Myrinet (MTU=9000)	155	234

Basically, these numbers show one-way communication latency on the system. Other relevant system overhead are 123 μsecs for servicing a (native) page fault, 0.6μsec for requesting current time, 4.2 μsecs for SIGIO, and 5.5 μsecs for SEGV handling on the host Linux operating system.

The results of Netperf[2] runs on the three networks are shown in Figure 5.3. The reported results are for `UDP REQUEST/RESPONSE TEST` with a confidence level of 99% and a $\pm$5.0% confidence interval width.

Appl.	Size	SEQ	Number of Cluster Nodes			
			2	4	8	16
Barnes	8192 bodies	117.36	65.69	35.73	25.73	26.80
	16384 bodies	371.90	199.84	105.98	67.14	62.13
	32768 bodies	1040.48	553.08	289.45	171.73	143.76
EP	2^{24} numbers	19.13	9.57	4.79	2.40	1.22
	2^{26} numbers	76.61	38.33	19.17	9.59	4.81
	2^{28} numbers	306.83	153.48	76.76	38.39	19.20
LU	1K×1K	90.37	42.13	23.49	18.17	10.31
	2K×2K	790.38	382.48	201.03	95.08	52.92
	3K×3K	2746.65	1369.69	694.67	333.47	179.20
SOR	1022×511	3.72	4.54	2.57	1.71	1.60
	2046×1023	14.11	12.76	6.77	3.88	2.76
	3070×1535	33.79	32.49	16.97	9.41	6.17
TSP	18 cities	9.39	5.03	2.82	1.74	1.25
	20 cities	62.59	35.14	20.36	15.44	14.79
	19 cities (big)	95.28	49.82	26.66	13.85	8.16
Water	343 mols	11.00	6.55	4.94	6.84	13.73
	1000 mols	95.17	50.63	27.88	20.58	26.00
	1728 mols	286.41	149.85	–	48.50	45.10

Table 5.6: Performance of Applications on Fast Ethernet

The major overhead inherent to software DSM systems is communication la-

[2]Netperf is a benchmark that can be used to measure the performance of many different types of networking. It provides tests for both unidirectional throughput, and end-to-end latency [Jon02]. I have used version 2.2alpha of Netperf for the benchmarks.

tency. One of the ways to reduce this overhead is to send fewer and smaller messages. I explored this by developing an additional experiment. Borrowing the idea from IBM's new Memory Expansion Technology (MXT) [TFR+01], I used a fast RAM algorithm [Riz97] to compress the "full" shared pages before they are transferred. This algorithm was originally developed to compress SWAP files and is reportedly faster and more efficient than LZRW1 [Wil91]. The results showed only minimal (approximately 2–5% on the average) performance gain. The main reason for this poor performance is that the algorithm is only efficient when the pages contain mostly zero values (as it is commonly the case in swap files), but shared data pages of the applications are almost always contain non-zero values. I also tried using the compression for the diff messages, but for the same reason, there was no improvement. IBM's MXT technology doubles the memory size. Thus, memory constrained programs, such as web applications, benefit from this technology. Those applications can cache more pages into the main memory and reduce the disk I/O activity. I added the RAM algorithm in JIA-R and ran some of the benchmark applications. Basically, the marginal reduction in the communication latency was absorbed by the compression time of the transferred pages, because these pages contained mostly non-zero values.

Tables 5.6, 5.7, and 5.8 summarize the results of the experiments on Fast Ethernet, Gigabit Ethernet, and Myrinet (with 9000-byte MTU), respectively. The **SEQ** column shows the execution times of the pseudo-sequential runs. JIA-R runtime reduces the system overhead to a bare minimum for most of the applications when the number of hosts is one.

5.6 Overview of Results

Table 5.9 shows the speedups of applications on 2, 4, 8, and 16 processors on all three network interconnects. These tables show the execution times of the applications in seconds for each of the small, medium, and large data sets, respectively.

In addition to execution times for each application, I collected timing statistics and grouped them as:

IDLE—is the combined time spent on locks and barriers.

OS—is the time spent on sending and receiving messages and `mprotect()` system call.

Appl.	Size	SEQ	Number of Cluster Nodes			
			2	4	8	16
Barnes	8192 bodies	117.36	54.13	33.77	20.21	14.77
	16384 bodies	371.90	198.31	101.70	56.41	38.22
	32768 bodies	1040.48	549.80	280.73	150.17	96.05
EP	2^{24} numbers	19.13	9.57	4.79	2.40	1.21
	2^{26} numbers	76.61	38.33	19.17	9.59	4.80
	2^{28} numbers	306.83	153.48	76.75	38.39	19.20
LU	1K×1K	90.37	41.75	23.07	18.03	9.61
	2K×2K	790.38	380.58	199.66	92.19	50.69
	3K×3K	2746.65	1365.54	691.85	330.91	175.00
SOR	1022×511	3.72	4.36	2.36	1.40	1.11
	2046×1023	14.11	12.76	6.56	3.53	2.24
	3070×1535	33.79	32.22	16.61	8.81	5.15
TSP	18 cities	9.39	4.95	2.72	1.55	1.06
	20 cities	62.54	33.21	17.67	10.06	7.62
	19 cities (big)	95.28	49.23	25.97	12.96	7.35
Water	343 mols	11.00	6.08	3.70	3.40	5.48
	1000 mols	95.17	49.60	25.96	15.19	13.36
	1728 mols	286.41	147.47	–	40.93	27.87

Table 5.7: Performance of Applications on Gigabit Ethernet

Appl.	Size	SEQ	Number of Cluster Nodes			
			2	4	8	16
Barnes	8192 bodies	117.36	64.89	33.81	20.62	15.33
	16384 bodies	371.90	198.58	102.03	57.08	39.29
	32768 bodies	1040.48	550.84	281.30	151.48	98.18
EP	2^{24} numbers	19.13	9.57	4.79	2.40	1.21
	2^{26} numbers	76.61	38.33	19.17	9.59	4.80
	2^{28} numbers	306.83	153.48	76.76	38.39	19.20
LU	1K×1K	90.38	41.81	23.12	17.71	9.61
	2K×2K	790.38	380.79	199.82	92.40	50.86
	3K×3K	2746.65	1365.75	692.14	329.78	175.58
SOR	1022×511	3.72	4.39	2.40	1.46	1.22
	2046×1023	14.11	12.61	6.59	3.63	2.37
	3070×1535	33.79	32.11	16.57	8.88	5.25
TSP	18 cities	9.38	4.97	2.73	1.59	1.07
	20 cities	62.54	33.38	17.92	10.50	8.30
	19 cities (big)	95.28	49.33	26.05	13.01	7.33
Water	343 mols	11.00	6.16	3.92	3.97	6.79
	1000 mols	95.17	49.72	26.17	15.78	14.75
	1728 mols	286.41	147.76	–	41.51	29.66

Table 5.8: Performance of Applications on Myrinet

Appl.	Size	Number of Cluster Nodes											
		2			4			8			16		
		F	G	M	F	G	M	F	G	M	F	G	M
Barnes	8192 bodies	1.79	2.17	1.81	3.28	3.48	3.47	4.56	5.81	5.69	4.38	7.95	7.66
	16384 bodies	1.86	1.88	1.87	3.51	3.66	3.65	5.54	6.59	6.52	5.99	9.73	9.47
	32768 bodies	1.88	1.89	1.89	3.59	3.71	3.70	6.06	6.93	6.87	7.24	10.83	10.60
EP	2^{24} numbers	2.00	2.00	2.00	3.99	3.99	3.99	7.97	7.97	7.97	15.68	15.81	15.81
	2^{26} numbers	2.00	2.00	2.00	4.00	4.00	4.00	7.99	7.99	7.99	15.93	15.96	15.96
	2^{28} numbers	2.00	2.00	2.00	4.00	4.00	4.00	7.99	7.99	7.99	15.98	15.98	15.98
LU	1K×1K	2.14	2.16	2.16	3.85	3.92	3.91	4.97	5.01	5.10	8.77	9.40	9.40
	2K×2K	2.07	2.08	2.08	3.93	3.96	3.96	8.31	8.57	8.55	14.94	15.59	15.54
	3K×3K	2.01	2.01	2.01	3.95	3.97	3.97	8.24	8.30	8.33	15.33	15.70	15.67
SOR	1022×511	0.82	0.85	0.85	1.45	1.58	1.55	2.18	2.66	2.55	2.33	3.35	3.05
	2046×1023	1.11	1.12	1.12	2.08	2.15	2.14	3.64	4.00	3.89	5.11	6.30	5.95
	3070×1535	1.04	1.05	1.05	1.99	2.03	2.04	3.59	3.84	3.81	5.48	6.56	6.44
TSP	18 cities	1.87	1.90	1.89	3.33	3.45	3.44	5.40	6.06	5.91	7.51	8.86	8.78
	20 cities	1.78	1.88	1.87	3.07	3.54	3.49	4.05	6.22	5.96	4.23	8.21	7.53
	19 cities (big)	1.91	1.94	1.93	3.57	3.67	3.66	6.88	7.35	7.32	11.71	12.96	13.00
Water	343 mols	1.68	1.81	1.79	2.23	2.97	2.81	1.61	3.24	2.77	0.80	2.01	1.62
	1000 mols	1.88	1.92	1.91	3.41	3.67	3.64	4.62	6.27	6.09	3.66	7.12	6.45
	1728 mols	1.91	1.94	1.94	–	–	–	5.91	7.00	6.90	6.35	10.28	9.66
F=Fast Ethernet, G=Gigabit Ethernet, M=Myrinet													

Table 5.9: Application Speedup on 3 Networks

DSM—is the combined time spent for the DSM protocol, including times for page twinning, diff creation, shared data related segmentation fault handling and page invalidation, and several other internal protocol functions.

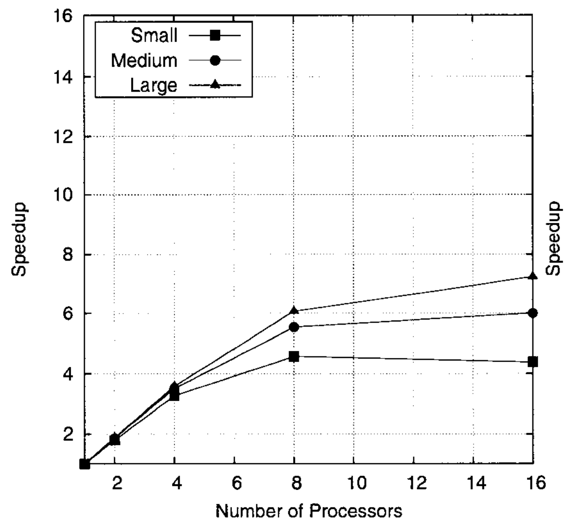
USER—is the time spent for the application execution.

Below, I give detailed performance results of the benchmark applications using three network interconnects. Table 5.1 shows the amount of shared pages allocated for each application. Note that JIA-R allocates a separate page for each shared memory request. This allocation might seem a wasteful approach, but I believe it potentially reduces the false sharing. Though I did not investigate this point further, it only effects Barnes, TSP, and Water, because these applications have smaller (than a page) units of shared structures.

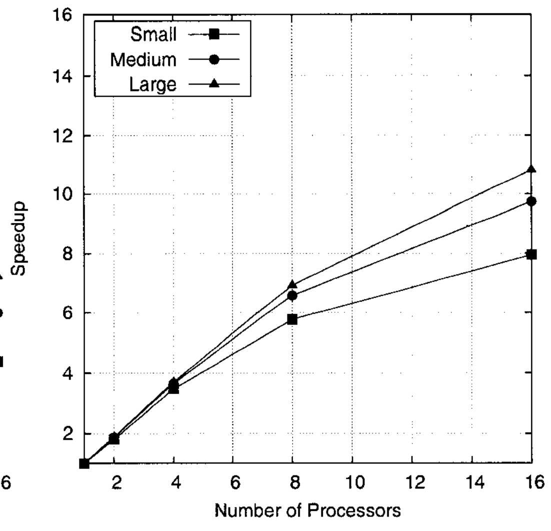
Obviously, the speedup of the applications on faster networks are better. With a few exceptions, there is not much difference on the performance on two Gigabit Ethernet and Myrinet networks. The gap between those two and the Fast Ethernet, which also carries NFS traffic, albeit locally, is more noticeable.

Barnes

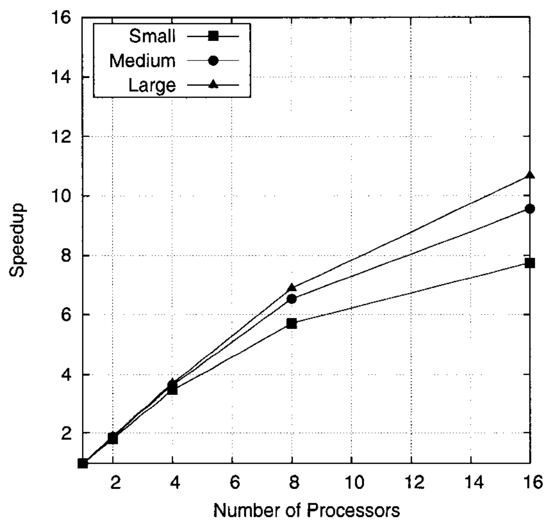
Barnes is an application with irregular data access patterns. For this and practical purposes, all the shared pages are allocated on the first (master) node. The number of pages to hold the shared data structures increases as the data set size (particle count) increases. The effect of irregular and fine-grained data sharing in the application becomes obvious as the number of nodes increases (Figure 5.4). As a result, the number of messages exchanged grow accordingly.



(a) Fast Ethernet



(b) Gigabit Ethernet



(c) Myrinet

Figure 5.4: Performance of Barnes on the 16-SingleCPU Cluster

The overheads incurred by the **DSM** and **OS** operations are nearly identical for all data sets and network interconnects. The communication latency hiding is more clear on fast networks (Figure 5.5). The results show that Barnes can achieve very good speedups on up to four nodes. For Fast Ethernet, the speedup on 8 nodes is the only achievable maximum.

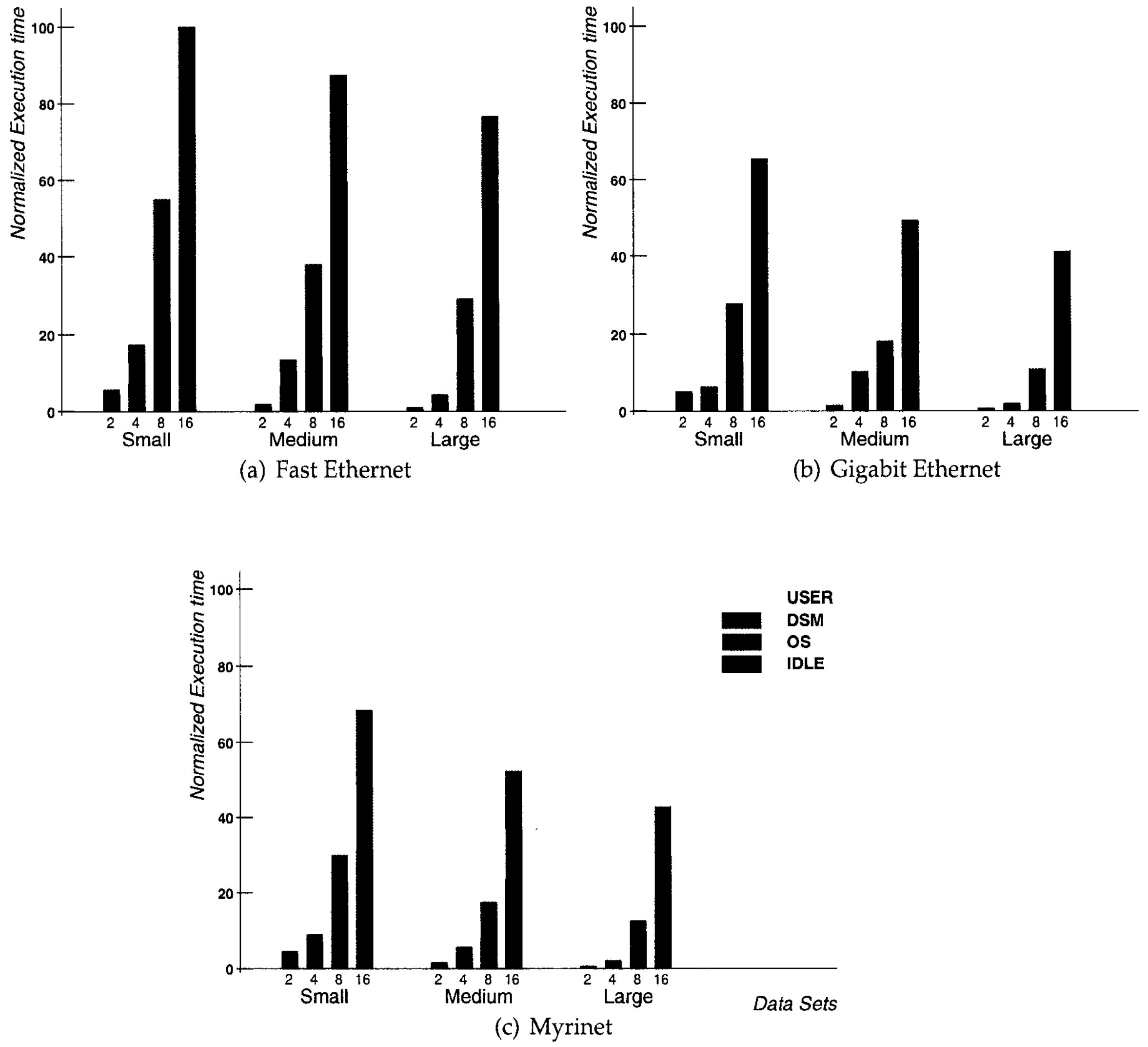
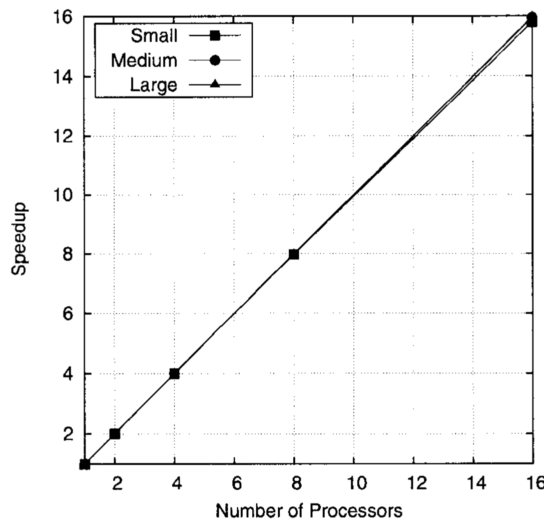


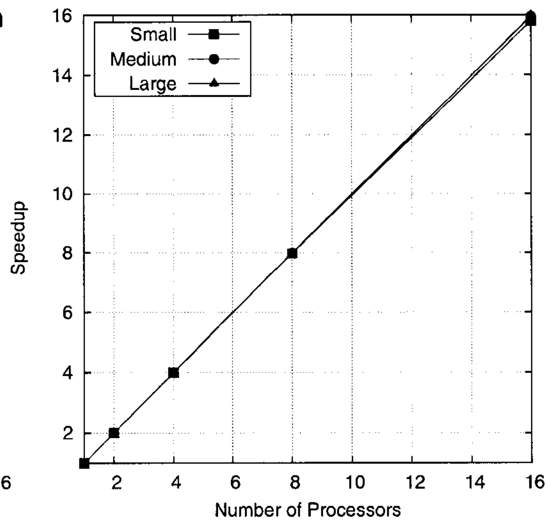
Figure 5.5: Relative Times of Barnes on the 16-SingleCPU Cluster

EP

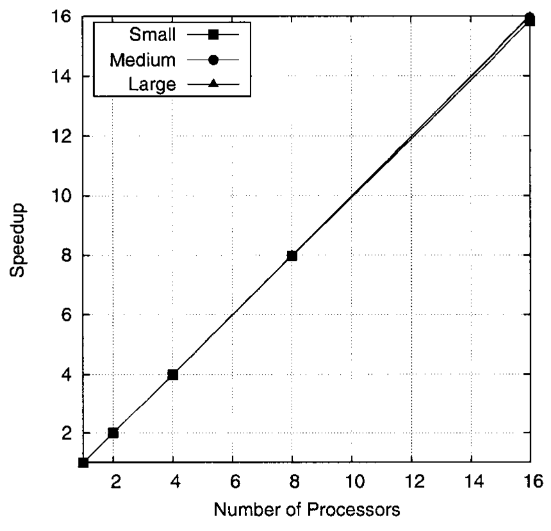
As discussed earlier and as the name implies, EP is a highly parallel benchmark application. It shares only an array of ten elements that contains the counts of Gaussian pairs calculated on each node. A total of 18 messages are exchanged in a 2-processor run, and 270 in a 16-processor run. The relative cost of this communication is less than %1 in both cases. This application shows almost a linear speedup, as expected (Figure 5.6).



(a) Fast Ethernet



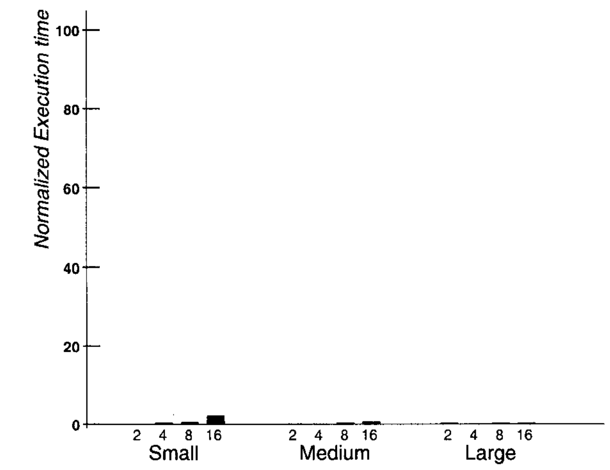
(b) Gigabit Ethernet



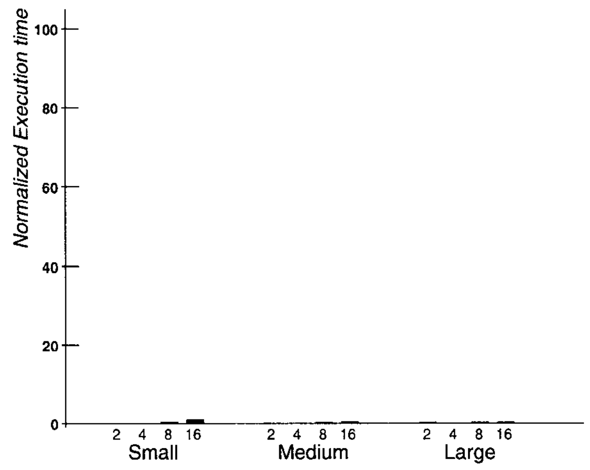
(c) Myrinet

Figure 5.6: Performance of EP on the 16-SingleCPU Cluster

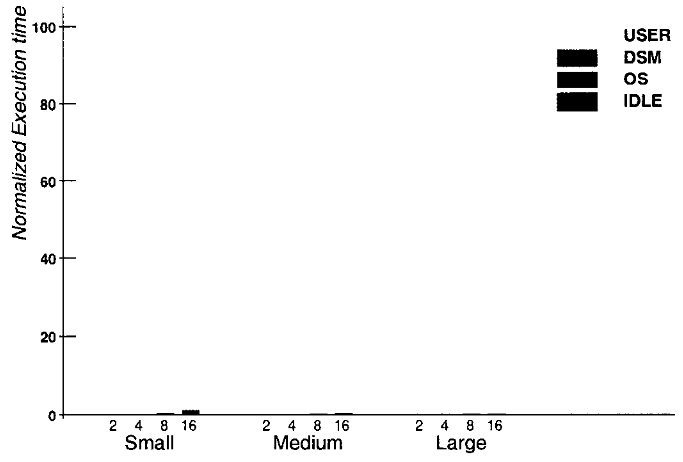
The effect of this linear speedup is also reflected in the relative times graphs. The only observable (communication) overhead is with the small data set on 16 processors using the Fast Ethernet connection. Obviously, this shows that the computation of the Gaussian pairs on 16 processors is fast that the messaging time is only visible on the Fast Ethernet (Figure 5.7). Other than this observation, there is no noticeable advantage in using any one of the three network interconnects for this particular application.



(a) Fast Ethernet



(b) Gigabit Ethernet

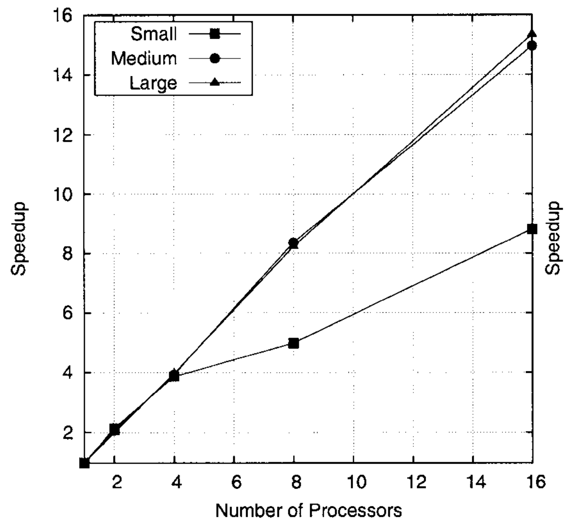


(c) Myrinet

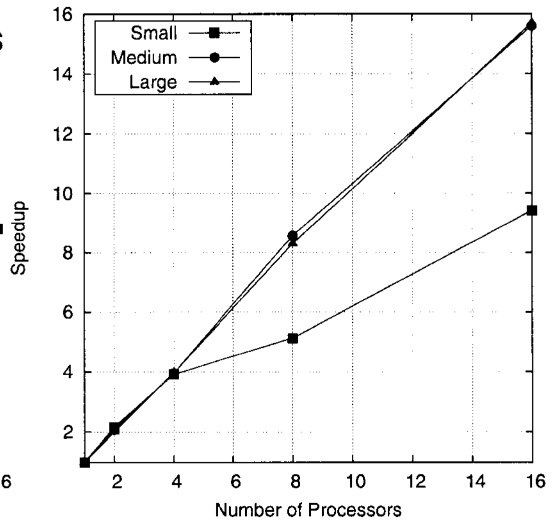
Figure 5.7: Relative Times of EP on the 16-SingleCPU Cluster

LU

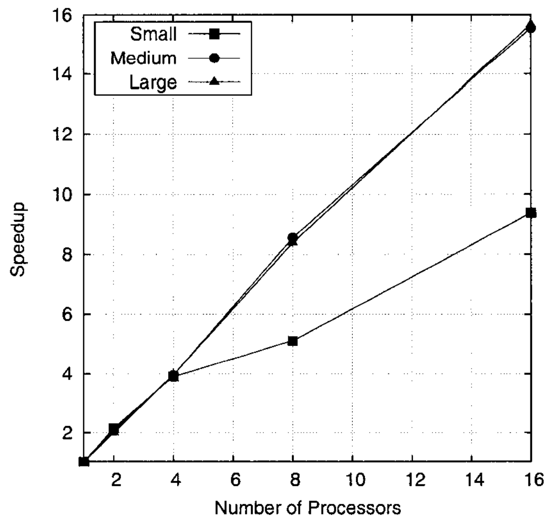
Because LU manipulates large dense matrices, the shared data for this application is also large. I carefully allocated matrix blocks to individual pages and divide these pages equally to the nodes. Consequently, the data traffic is substantially reduced and the application achieved near optimal speedups for larger data sets (Figure 5.8).



(a) Fast Ethernet



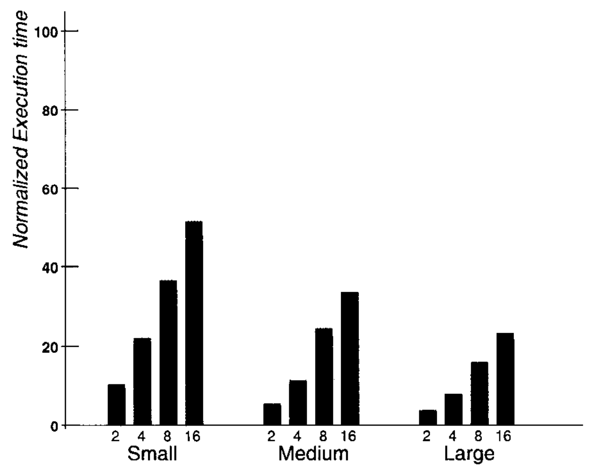
(b) Gigabit Ethernet



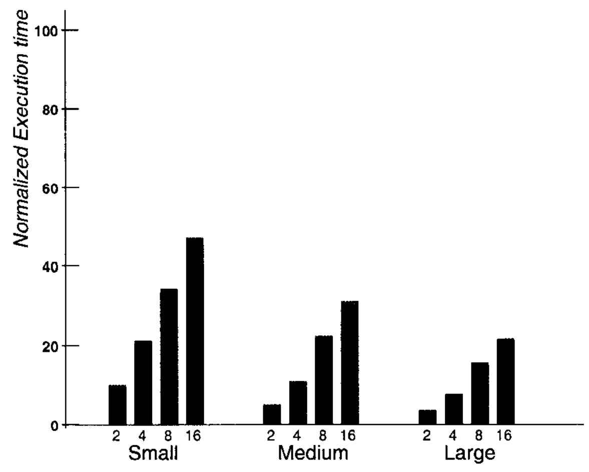
(c) Myrinet

Figure 5.8: Performance of LU on the 16-SingleCPU Cluster

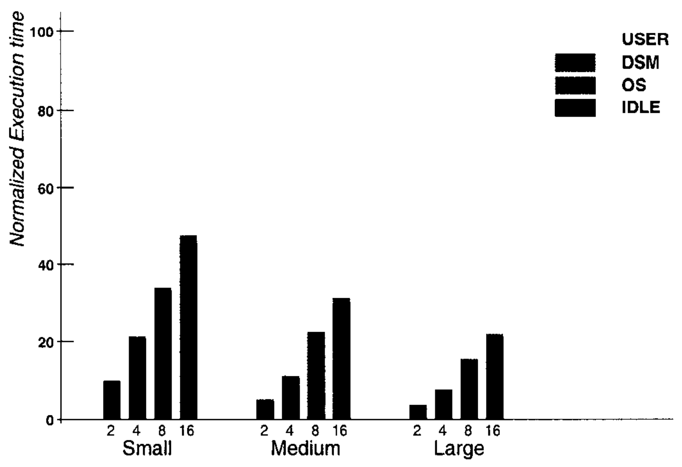
However, as the size of the matrices, hence the number of blocks increases, the data traffic also goes up, limiting the speedup of the application for small data set to approximately 50%, even for faster networks. On the other hand, the application with medium and large data sets obtains fair amount of computation time, thus better speedup (Figure 5.9).



(a) Fast Ethernet



(b) Gigabit Ethernet

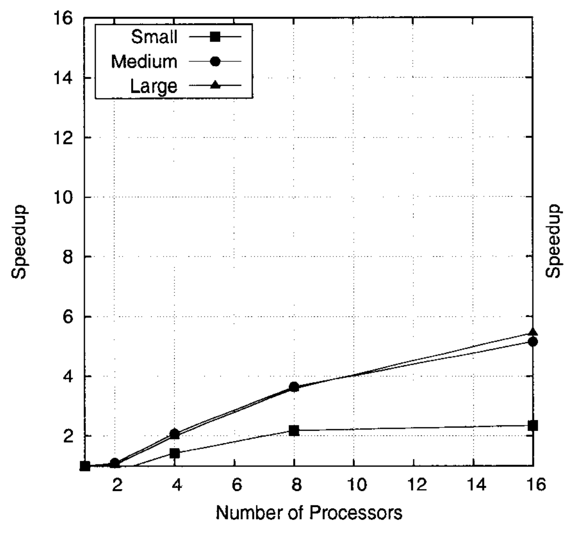


(c) Myrinet

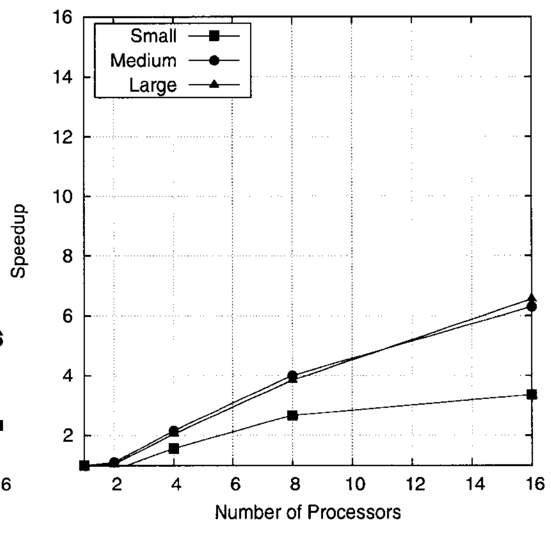
Figure 5.9: Relative Times in LU on the 16-SingleCPU Cluster

SOR

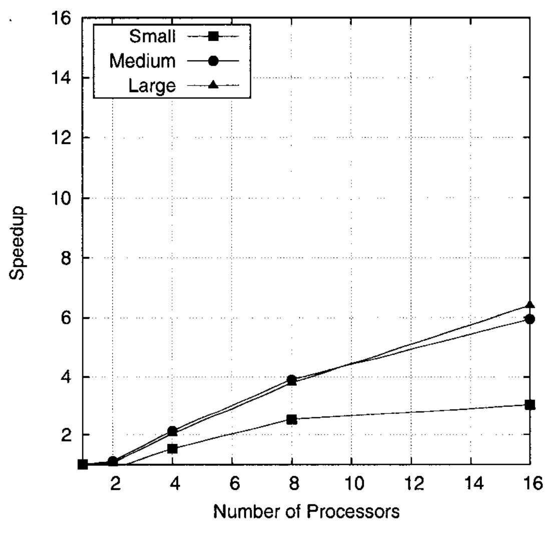
SOR accesses the same data blocks for each iteration. I ran this application for 101 iterations. I distributed the row blocks of the matrix evenly to each node. This approach reduced the page fault rate per iteration, because the nodes processed the local part of the matrix for a while. Despite of the careful distribution of row blocks, the attainable speedup was limited to approximately 6.5 on even the fast networks on 16 nodes (Figure 5.10).



(a) Fast Ethernet



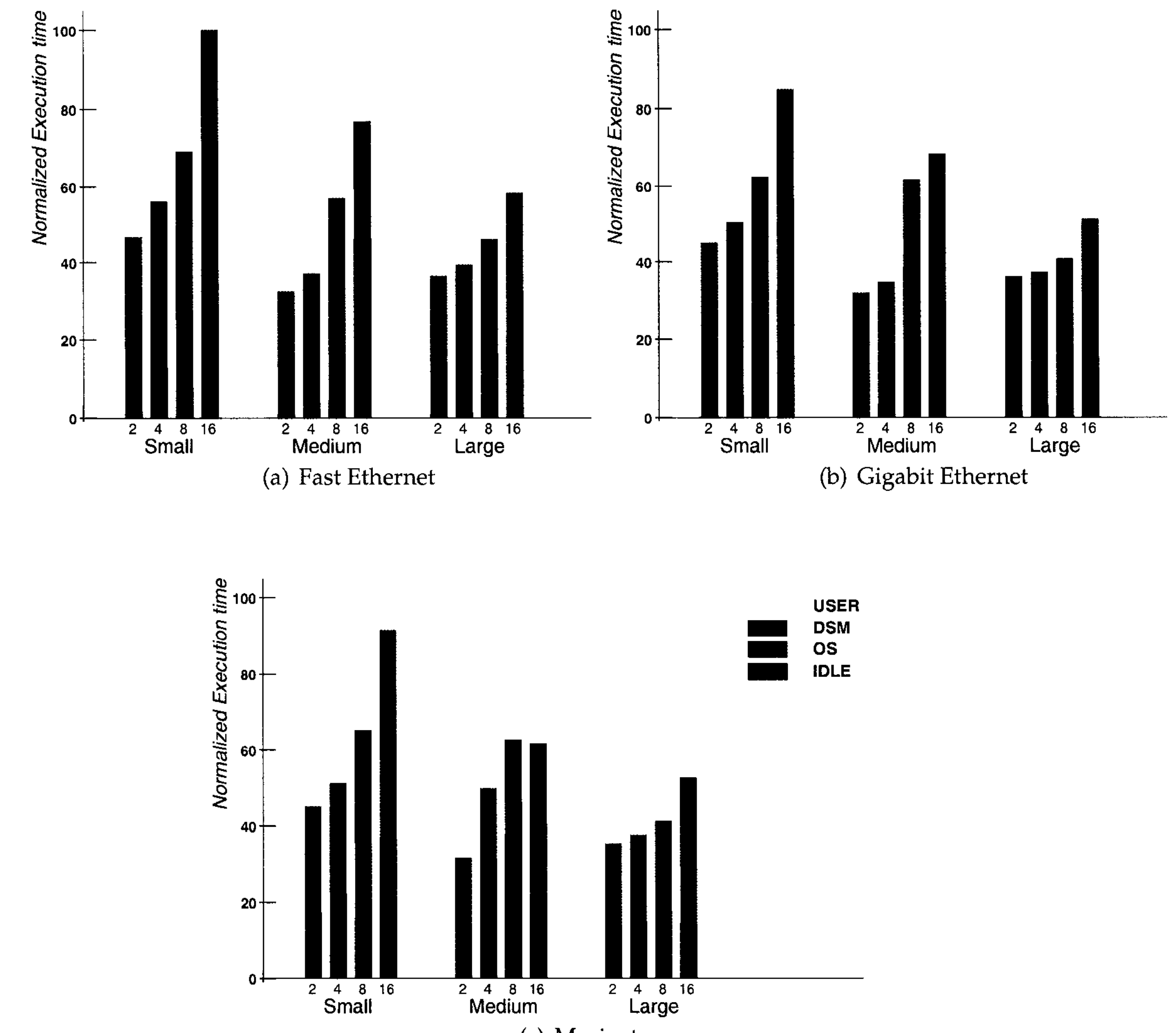
(b) Gigabit Ethernet



(c) Myrinet: MTU 1500

Figure 5.10: Performance of SOR on the 16-SingleCPU Cluster

Please note that the even distribution of the matrix blocks substantially increased the number of shared pages. This behavior is also valid for all applications that manipulates matrices. On Fast Ethernet, the **IDLE** time generally dominates, as the nodes wait on the barriers after each iteration. This time somewhat reduced on the faster networks (Figure 5.11).



(a) Fast Ethernet

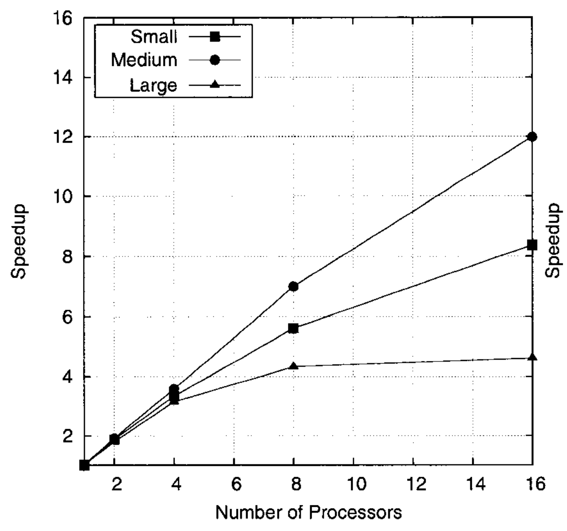
(b) Gigabit Ethernet

(c) Myrinet

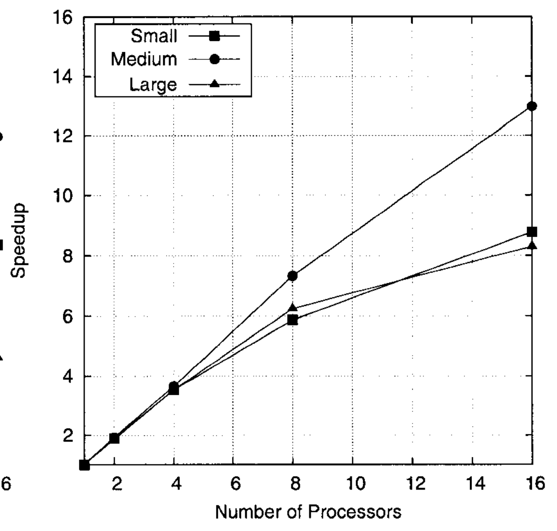
Figure 5.11: Relative Times in SOR on the 16-SingleCPU Cluster

TSP

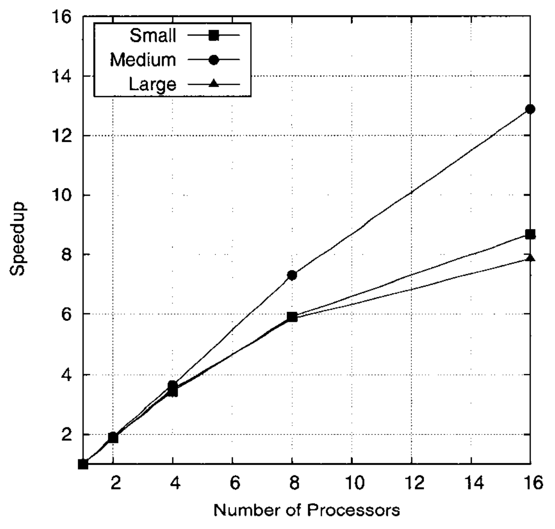
The TSP application has a fixed amount of shared data, independent of the data sets. Since the max size of intermediate solution tours is constant for each data set and relatively small size of shared data, TSP is quite scalable and it achieves good speedups (Figure 5.12).



(a) Fast Ethernet



(b) Gigabit Ethernet



(c) Myrinet

Figure 5.12: Performance of TSP on the 16-SingleCPU Cluster

For small problems, the synchronization overhead increase noticeably with the number of nodes. Nevertheless, for larger problems, this is a non-issue (Figure 5.13).

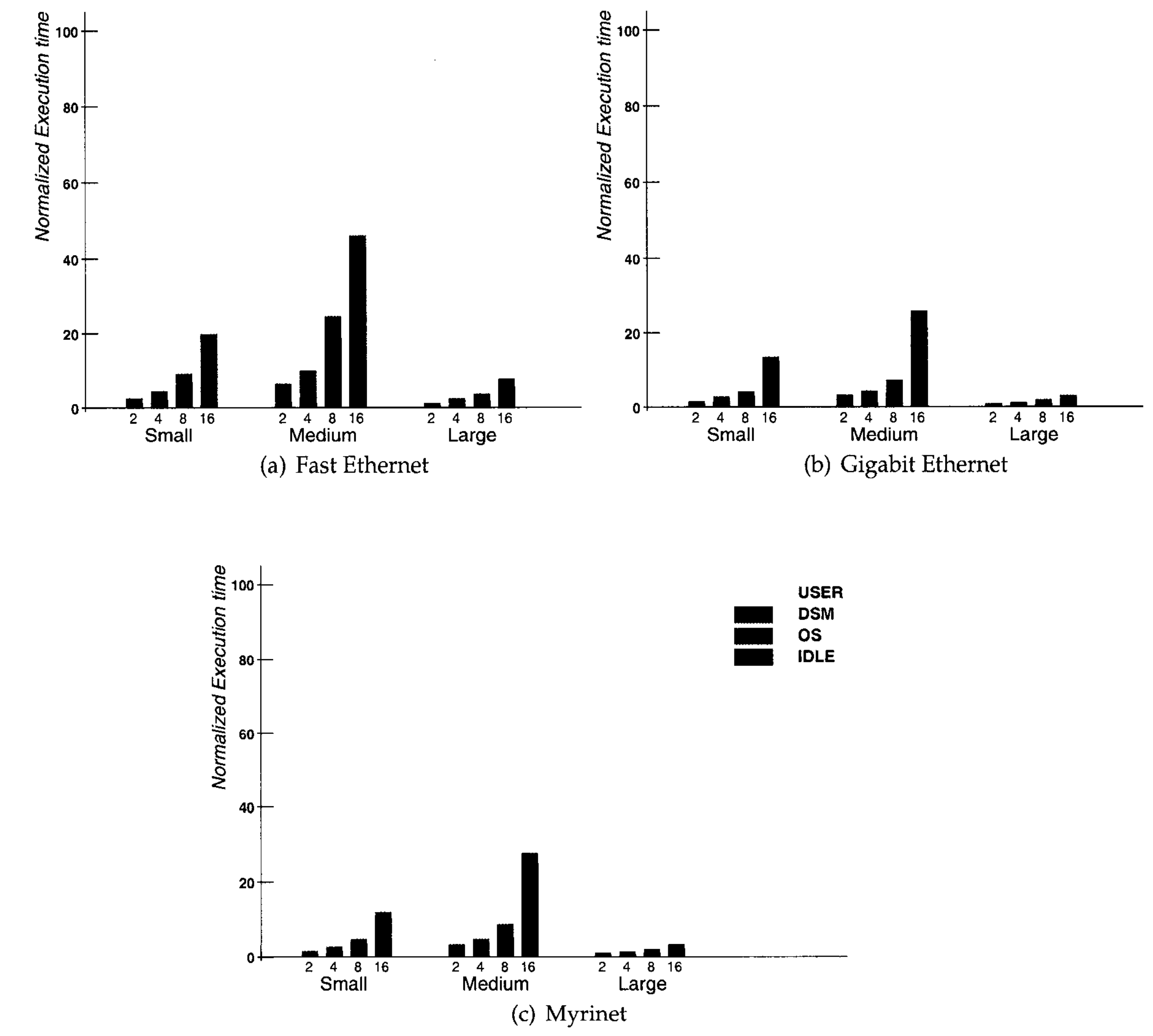
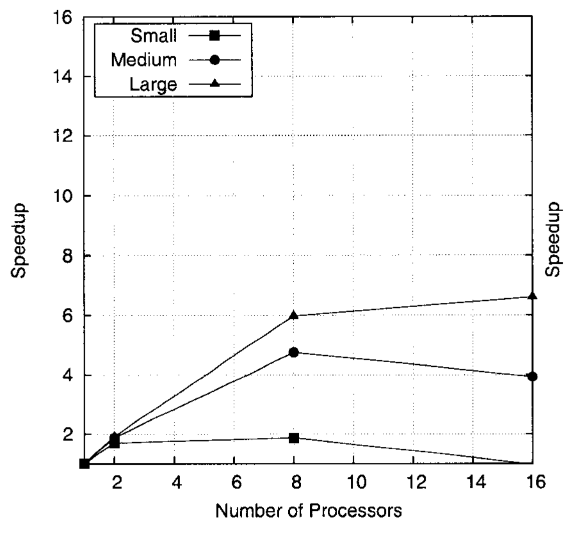


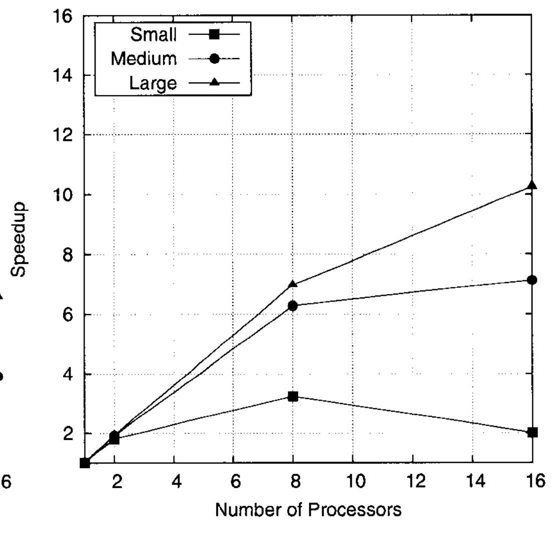
Figure 5.13: Relative Times in TSP on the 16-SingleCPU Cluster

Water

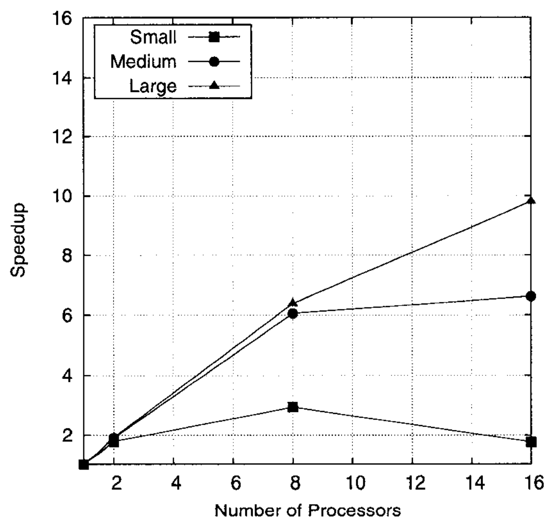
Like Barnes, Water also has an irregular sharing pattern. For unknown reasons, this application did not run successfully on 4 nodes with the large data set. Nevertheless, it achieved reasonable speedups for 8 and 16 nodes on fast networks (Figure 5.14).



(a) Fast Ethernet



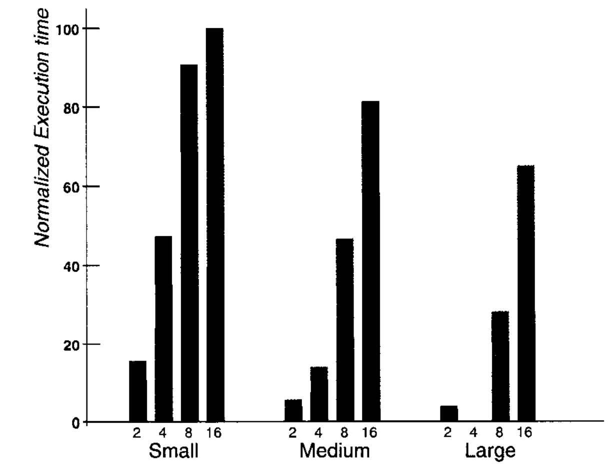
(b) Gigabit Ethernet



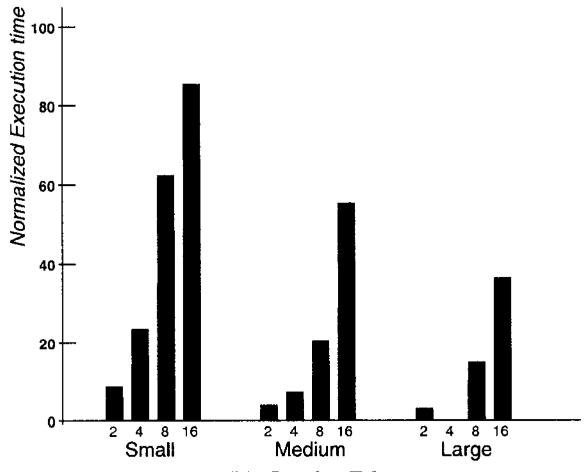
(c) Myrinet

Figure 5.14: Performance of Water on the 16-SingleCPU Cluster

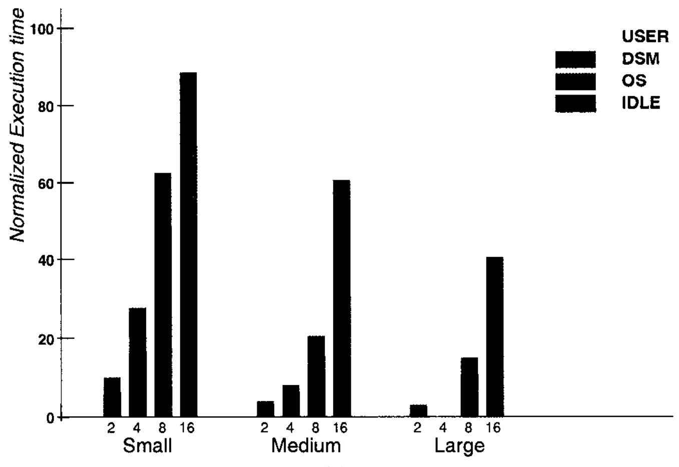
The speed of the communication media helped reduce the messaging overhead for even medium data set (Figure 5.15).



(a) Fast Ethernet



(b) Gigabit Ethernet



(c) Myrinet

Figure 5.15: Relative Times in Water on the 16-SingleCPU Cluster

5.7 Summary

I gave a detailed analysis of the application performance of JIAJIA and JIA-R. The advantages of software distributed shared memory can be seen clearly in the above performance results. It is also clear that not all the benchmark applications can achieve a desirable speedup. For each application, I discussed the reasons for such anomalies. I conclude that the current version of JIA-R achieves moderate to good results on the applications tested. Better performance is not feasible mainly because of the inherent overhead of the underlaying operating system for virtual memory trapping and network protocol stack. I believe that employing user accessible firmware mechanisms of today's high performance network interconnects could not only improve performance, but also it may increase scalability.

Chapter 6

Conclusions and Future Work

In this thesis, I reviewed the state-of-the-art in distributed shared memory, emphasizing page-based systems. I also introduced JIA-R, a page based-software DSM system as a case study, and analyzed the performance of JIA-R with several commonly used benchmark applications from a wide variety of areas and domains.

The *killer application* for software DSM systems is yet to be developed. However, such an application needs to be designed from scratch. Usually, a better performance can be achieved if an application is developed from scratch with S-DSM in mind. Admittedly, trying to improve an existing large application requires a thorough understanding of it, as well as efficient performance tuning tools supporting them.

6.1 Conclusions

The idea of implementing a shared memory over a network of computers using software techniques (software DSM) was proposed more than a decade ago by Kai Li in his seminal work [LH86]. Other researchers have expanded on the idea by studying a variety of areas, which can be categorized as: (1) DSM concepts, (2) memory coherence protocols and algorithms, (3) implementation type: hardware, (low-level) software, and language primitives, (4) performance, and (5) other issues such as synchronization, fault tolerance, heterogeneity, and persistence [Esk96]. Although it is hard to find new key ideas, there are many engineering details that can still be utilized to make production quality software distributed shared memory systems.

In this thesis, I have introduced a new software distributed shared memory system called JIA-R. Using this software, I have demonstrated that such systems are still viable alternatives to message passing systems, such as MPI. For the first time in the literature, I have evaluated parallel application performance on three different network interconnects. I have also presented an extensive survey of the state-of-the-art in software distributed shared memory, and provided a unified classification of earlier systems.

6.2 Future Work

Over the last decade, software DSM research has focused on four basic areas: consistency models (e.g., [AG96, HKV97, Kin99]), protocols (e.g., [KDCZ94, ISL96, BZ91, CBZ91]), architectural support (e.g., [ALK95, OAS95]), and application-driven approaches (e.g., [BIS96]). The majority of recent research has concentrated on consistency models and protocols, with far fewer contributions in the latter two areas. In their work, Iftode and Singh [IS99] summarize the current progress and challenges in software DSM. They identify the current focus as: (i) performance analysis and application restructuring for DSM, (ii) protocol enhancements driven by application bottlenecks, (iii) architectural support and interactions with the communication architecture, (iv) comparison with alternative software shared memory approaches, and (v) software tools. They also predict that the areas for future advances will be in: (a) reducing the still considerable performance gap between hardware and software DSM, (b) improving the protocols and system support further, and (c) in understanding programmability.

Until recently, there were several active workshops attracting research papers on DSM and related issues. For example, the theme of the 1999 Workshop on Software Distributed Shared Memory was *"What remains to be done in software DSM?"* and more specifically, *"Will software DSM ever move into the mainstream?"* The consensus there was that *"There still remain many issues to be researched in this area"* and that *"A killer application or better support from the underlying network interconnect fabric are still needed"*. These are obviously some of the targets for future research on distributed shared memory.

Since the fundamental idea behind DSM is "caching", there are potentially endless opportunities to apply the idea in other fields of computing. One such field is the storage area networks. For example, IBM's *StorageTank* and *SANFS* products [IBM00] use the caching technology to their advantage [Hen04].

I conclude the thesis with a summary of Michael Scott's excellent keynote address, titled *"Is S-DSM Dead?"* at the 2nd Workshop on Software Distributed Shared Memory [Sco00]:

- *Why we study S-DSM?* Shared memory is an attractive model and arguably simpler especially for non-performance-critical applications. Hardware coherence is faster, but providing coherence in software is cheaper as it can be built faster. More complex protocols can be used in software and it is easy to enhance, tune and customize. It is the only option on distributed systems.
- *Is S-DSM dead?*

  Yes, because the key ideas are all explored and applications are still written using MPI.

  No, because speedups for well-written applications are usually good, but serious users wait for "production quality" systems. Furthermore, the ideas are still valuable in wider domain.
- *What has been done so far?* There is no major operating system that contains a S-DSM. TREADMARKS is the only commercially available system as a separate package[1]. Many researchers still prefer message passing, particularly MPI.
- *Where do we stand today?* Relaxed memory models, virtual memory based protocols, multiprocessor nodes and use of advance network interfaces such as VIA or Myrinet are right choices. Performance of even well-tuned applications are OK on small number of nodes, but the real scalability is yet to be seen.
- *A look at the future.* S-DSM is not going to run on systems with many nodes, nor it can match the performance of well-tuned MPI applications. However, S-DSM is "good" for modest sized clusters and the applications they run. We need a single system image with good debuggers and efficient process and memory management mechanisms, better compiler integration, non-scientific applications such as games or e-commerce, and wide area distribution providing heterogeneity and fault tolerance (for functionality and not performance).

[1]Author's note: Even the work on TREADMARKS is currently stalled.

Bibliography

[Abr81] D. A. Abramson. Hardware Memory Management of a Large Virtual Memory. In *Proc. of the 4th Australian Computer Science Conf. (ACSC-4)*, pages 1–13, January 1981.

[AK85] D. A. Abramson and J. L. Keedy. Implementing a Large Virtual Memory in a Distributed Computer System. In *Proc. of the 18th Hawaii Int'l Conf. on System Sciences (HICSS-18)*, pages 515–522, January 1985.

[AAO92] V. Abrossimov, F. Armand, and M. I. Ortega. A Distributed Consistency Server for the CHORUS System. In *Proc. of the Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMS-III)*, pages 129–148, March 1992.

[ABB+86] M. Acetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proc. of Summer USENIX 1986 Technical Conf.*, pages 93–112, June 1986.

[Ach94] A.-C. Achilles. The Collection of Computer Science Bibliographies. `http://liinwww.ira.uka.de/bibliography/index.html`, Jan 1994.

[AH90] S. V. Adve and M. D. Hill. Weak Ordering—A New Definition. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 2–14, May 1990.

[AG96] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[ABM93] Y. Afek, G. Brown, and M. Merritt. Lazy Caching. *ACM Trans. on Programming Languages and Systems*, 15(1):182–205, January 1993.

[ACJ+92] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.-H. Lim, G. Maa, and D. Nussbaum. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In M. Dubois and S. S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*, pages 239–261. Kluwer Academic Publishers, 1992.

[ALK95] J.-H. Ahn, K.-W. Lee, and H.-J. Kim. Architectural Issues in Adopting Distributed Shared Memory for Distributed Object Management Systems. In *Proc. of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'95)*, pages 294–300, August 1995.

[ACP95] T. E. Anderson, D. E. Culler, and D. Patterson. A Case for NOW (Network of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[ABM+00] G. Antoniu, L. Bouge, M. MacBeth, K. McGuigan, and R. Namyst. Implementing Java Consistency Using a Generic, Multi-threaded DSM

Runtime System. In *Proc. of the Int'l Workshop on Java for Parallel and Distributed Computing*, pages 560–567, May 2000.

[AB01a] G. Antoniu and L. Bouge. DSM-PM2: A Portable Implementation Platform for Multithreaded DSM Consistency Protocols. In *Proc. 6th Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01)*, pages 55–70, April 2001.

[AB01b] G. Antoniu and L. Bouge. Implementing Multithreaded Protocols for Release Consistency on Top of the Generic DSMPM2 Platform. In *Proc. Int'l Workshop on Cluster Computing (IWCC '01)*, pages 179–188, September 2001.

[BAFR96] Y. Ben-Asher, D. G. Feitelson, and L. Rudolph. ParC—An Extension of C for Shared Memory Parallel Processing. *Software—Practice and Experience*, 26(5):581–612, May 1996.

[BBLS94] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[BT88] H. E. Bal and A. S. Tanenbaum. Distributed Programming with Shared Data. In *Proc. of the 1988 Int'l Conf. on Computer Languages*, pages 82–91, 1988.

[BST89] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3), September 1989.

[BTK90] H. E. Bal, A. S. Tanenbaum, and M. F. Kaashoek. Orca: A Language for Distributed Programming. *ACM SIGPLAN Notices*, 25(5):17–24, May 1990.

[BT91] H. E. Bal and A. S. Tanenbaum. Distributed Programming with Shared Data. *Computer Languages*, 16(2):129–146, 1991.

[BL85] A. Barak and A. Litman. MOS: A Multicomputer Distributed Operating System. *Software Practice and Experience*, 15(8):725–738, August 1985.

[BL88] A. Barak and O. La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Future Generation Computer Systems*, 13(4–5):361–372, April 1988.

[BCZ90] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proc. of the Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'90)*, pages 168–176, March 1990.

[BCZ91] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Using Multi-Protocol Release Consistency. In A. I. Karshmer and J. Nehmer, editors, *Operating Systems of the 90s and Beyond*, number 563 in Lecture Notes in Computer Science, pages 56–60. Springer-Verlag, July 1991.

[BZ91] B. N. Bershad and M. J. Zekauskas. Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science, Carnegie-Mellon University, September 1991.

[BZS93] 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.

[BIS96] A. Bilas, L. Iftode, and J. P. Singh. Supporting A Coherent Shared Address Space Across SMP Nodes: An Application-Driven Investigation. In *Proc. of IMA Workshop on Parallel Algorithms and Parallel Systems*, November 1996.

[BN84] A. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BF87] R. Bisiani and A. Forin. Architectural Support for Multilanguage Parallel Programming on Heterogeneous Systems. In *Proc. of the second Int'l Conf. on Architectual Support for Programming Languages and Operating Systems*, pages 21–30, April 1987.

[BF88] R. Bisiani and A. Forin. Multilanguage Parallel Programming of Heterogeneous Machines. *IEEE Transactions on Computers*, 37(8):930–945, August 1988.

[BNR89] R. Bisiani, A. Nowatzyk, and M. Ravishankar. Coherent Shared Memory on a Distributed Memory Machine. In *Proc. of the 1989 Int'l Conf. on Parallel Processing (ICPP'89)*, volume I, pages 133–141, August 1989.

[BR90] R. Bisiani and M. Ravishankar. PLUS: A Distributed Shared-Memory System. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 115–124, May 1990.

[BJK+95] R. D. Blumofe, C. F. Joerg, B. Kuszmaul, C. E. Leiserson, and K. H. Randall. Cilk: An Efficient Multithreaded Runtime System. In *Proc. of the 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 207–216, July 1995.

[BFJ+96] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-Consistent Distributed Shared Memory. In *Proc. of the 10th Int'l Parallel Processing Symp. (IPPS'96)*, pages 132–141, April 1996.

[BAC+98] 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.

[BCF+95] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit per Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[BH90] L. Borrmann and M. Herdieckerhoff. A Coherency Model for Virtually Shared Memory. In *Proc. of the 1990 Int'l Conf. on Parallel Processing (ICPP'90)*, volume II, pages 252–257, August 1990.

[BS99] T. Brecht and H. Sandhu. The Region Trap Library: Handling Traps on Application-Defined Regions of Memory. In *Proc. of the USENIX Annual Technical Conference*, pages 85–99, June 1999.

[Bro89] M. Brorsson. A Decentralized Virtual Memory Scheme Implemented on an Emulated MIMD Multiprocessor. In *Proc. of the 22nd Hawaii Int'l Conf. on System Sciences (HICSS-22)*, pages 286–295, January 1989.

[BL94] L. Brunie and L. Lefevre. DOSMOS : A Distributed Shared Memory Based on PVM. In *Proc. of the First European PVM Users Group Meeting*, October 1994.

[CP96] G. Cabillic and I. Puaut. Stardust: An Environment for Parallel Programming on Networks of Heterogeneous Workstations. In *Proc. of the Second Int'l Euro-Par Conf.*, volume I, pages 114–119, August 1996.

[CDM94] R. Cabrera-Dantart, I. Demeure, and P. Meunier. Phosphorus: Adding Shared Memory to PVM. Presented at the First European PVM Users' Group Meeting, Rome, Italy, October 1994.

[CBZ91] 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.

[CCD+93] J. B. Carter, A. L. Cox, S. Dwarkadas, E. N. Elnozahy, D. B. Johnson, P. J. Keleher, S. Rodrigues, W. Yu, and W. Zwaenepoel. Network Multicomputing Using Recoverable Distributed Shared Memory. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 519–527, February 1993.

[CBZ95] 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.

[CGSC96] M. Castro, P. Guedes, M. Sequeira, and M. Costa. Efficient and Flexible Object Sharing. In *Proc. of the 1996 Int'l Conf. on Parallel Processing (ICPP'96)*, volume 1, pages 128–137, August 1996.

[CAL+89] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proc. of the 12th ACM Symp. on Operating Systems Principles (SOSP-12)*, pages 147–158, December 1989.

[CDP+00] D. Chen, S. Dwarkadas, S. Parthasarathy, E. Pinherio, and M. L. Scott. Interweave: A Middleware System for Distributed Shared State. In S. Dwarkadas, editor, *Proc. of the Fifth Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR'00)*, LNCS. Springer-Verlag, May 2000.

[Che86] D. R. Cheriton. Problem-oriented Shared Memory: A Decentralized Approach to Distributed System Design. In *Proc. of the 6th Int'l Conf. on Distributed Computing Systems (ICDCS-6)*, pages 190–197, May 1986.

[CGBG88] D. R. Cheriton, A. Gupta, P. D. Boyle, and H. A. Goosen. The VMP Multiprocessor: Initial Experience, Refinements and Performance Evaluation. In *Proc. of the 15th Annual Int'l Symp. on Computer Architecture (ISCA'88)*, pages 410–421, June 1988.

[CWH00] D. W.-L. Cheung, C.-L. Wang, and K. Hwang. JUMP-DP: A Software DSM System with Low-Latency Communication Support. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'00)*, pages 821–827, June 2000.

[CWH99] D. W.-L. Cheung, C.-L. Wang, and K. Hwang. A Migrating-Home Protocol for Implementing Scope Consistency Model on a Cluster of Workstations. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, June 1999.

[Cor94] J. Cordsen. Basing Virtually Shared Memory on a Family of Consistency Models. In *Proc. of the Int'l Workshop on Support for Large Scale Shared Memory Architectures*, pages 58–72, April 1994.

[Del88] G. S. Delp. *The Architecture and Implementation of MemNet: a High-Speed Shared-Memory Computer Communication Network*. PhD thesis, Department of Electrical Engineering, University of Delaware, 1988.

[DSF88] G. S. Delp, A. S. Sethi, and D. J. Farber. An Analysis of MemNet: An Experiment in High-Speed Shared-Memory Local Networking. In *Proc. of the ACM Symp. on Communications Architectures, Protocols and Applications (SIGCOMM'88)*, pages 165–174, August 1988.

[DPS+94] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Therry, M. M. Theimer, and B. B. Welch. The Bayou Architecture: Support for Data Sharing among Mobile Users'. In *Proc. of the Workshop on Mobile Computing Systems and Applications*, pages 2–7, December 1994.

[DZU98] B. Dreier, M. Zahn, and T. Ungerer. The Rthreads Distributed Shared Memory System. In *Proc. of the 3rd Int'l Conference on Massively Parallel Computing Systems (MPCS'98)*, April 1998.

[DSB86] M. Dubois, C. Scheurich, and F. A. Briggs. Memory Access Buffering in Multiprocessors. In *Proc. of the 13th Annual Int'l Symp. on Computer Architecture (ISCA'86)*, pages 434–442, June 1986.

[Esk02] M. R. Eskicioglu. Software DSM's Are Still Alive and Well! In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, pages 90–96, June 2002.

[Esk95] M. R. Eskicioglu. An Online Comprehensive Bibliography of Distributed Shared Memory. `http://dsmbiblio.cs.umanitoba.ca/WEB`, January 1995.

[Esk96] M. R. Eskicioglu. A Comprehensive Bibliography of Distributed Shared Memory. *ACM Operating Systems Review*, 30(1):71–96, January 1996.

[EM98] M. R. Eskicioglu and T. A. Marsland. Shared Memory Computing on SP2: JIAJIA Approach. In *Proc. of the IBM Centre for Advanced Studies Conf. (CASCON'98)*, pages 235–245, November 1998.

[EMHS99] 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 32nd Hawaii Int'l Conf. on System Sciences (HICSS-32) CD-ROM*, pages 287, file: stdcr05.ps, January 1999.

[FS94] P. Ferreira and M. Shapiro. Garbage Collection and DSM Consistency. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation (OSDI'94)*, pages 229–241, November 1994.

[Fle87] B. D. Fleisch. Distributed Shared Memory in a Loosely Coupled Distributed System. In *Proc. of the ACM SIGCOMM'87 Workshop on Frontiers in Computer Communications Technology*, pages 317–327, August 1987.

[FP89] B. D. Fleisch and G. J. Popek. Mirage: A Coherent Distributed Shared Memory Design. In *Proc. of the 12th ACM Symp. on Operating Systems Principles (SOSP-12)*, pages 211–223, December 1989.

[FHJ94] B. D. Fleisch, R. L. Hyde, and N. C. Juul. MIRAGE+: A Kernel Implementation of Distributed Shared Memory on a Network of Personal Computers. *Software—Practice and Experience*, 24(10):887–909, October 1994.

[FBYR89] A. Forin, J. Barrera, M. Young, and R. Rashid. Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach. Technical Report CMU-CS-88-165, Carnegie-Mellon University, School of Computer Science, January 1989.

[FBS89] A. Forin, J. Barrera, and R. Sanzi. The Shared Memory Server. In *Proc. of the Winter 1989 USENIX Conference*, pages 229–243, January 1989.

[FLA94] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews. Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation (OSDI'94)*, pages 201–213, November 1994.

[Geh84] N. H. Gehani. Broadcasting Sequential Processes (BSP). *IEEE Transactions on Software Engineering*, SE-10(4):343–351, July 1984.

[Gel85] D. Gelertner. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[GLL+90] 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.

[Goo89] J. R. Goodman. Cache Consistency and Sequential Consistency. Technical Report 61, IEEE Scalable Coherence Interface Working Group, March 1989.

[GBC+98] A. Grbic, S. Brown, S. Caranci, R. Grindley, M. Gusat, G. Lemieux, K. Loveless, N. Manjikian, S. Srbljic, M. Stumm, Z. Vranesic, and Z. Zilic. Design and Implementation of the NUMAchine Multiprocessor. In *Proc. of the 35th IEEE Design Automation Conference*, pages 66–69, June 1998.

[GYF93] J. N. Griffioen, R. Yavatkar, and R. Finkel. Unify: A Scalable, Loosely-Coupled, Distributed Shared Memory. Technical Report CS226-93, Department of Computer Science, University of Kentucky, January 1993.

[HLH92] E. Hagersten, A. Landin, and S. Haridi. DDM—A Cache-Only Memory Architecture. *IEEE Computer*, 25(9):44–54, September 1992.

[HSL94] E. Hagersten, A. Saulsbury, and A. Landin. Simple COMA Node Implementations. In *Proc. of the 27th Hawaii Int'l Conf. on System Sciences (HICSS-27)*, volume I, pages 522–533, January 1994.

[Hag89] R. Hagmann. Comments on Workstation Operating Systems and Virtual Memory. In *Proc. of the 2nd Workshop on Workstation Operating Systems*, September 1989.

[HS93] A. Heddaya and H. Sinha. An Overview of Mermera: A System and Formalism for Non-coherent Distributed Parallel Memory. In *Proc. of the 26th Hawaii Int'l Conf. on System Sciences (HICSS26)*, pages 164–173, January 1993.

[HPS94] A. Heddaya, K. Park, and H. Sinha. Using Warp to Control Network Contention in Mermera. In *Proc. of the 27th Hawaii Int'l Conf. on System Sciences (HICSS-27)*, volume II, pages 96–105, January 1994.

[HKO+94] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J. P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proc. of the 6th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 274–285, October 1994.

[Hel90] H. Hellwagner. A Survey of Virtually Shared Memory Schemes. Technical Report TUM-19056, Institute for Informatics, Technical University of Munich, Germany, December 1990.

[Hen04] B. Henderson. StorageTank: Delivering on the SAN Promise. *LINUX Magazine*, 6(2):28–32, February 2004.

[HW90] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[HKV97] L. Higham, J. Kawash, and N. Verwaal. Defining and Comparing Memory Consistency Models. In *Proc. of the 10th Int'l Conf. on Parallel and Distributed Computing Systems (PDCS-97)*, pages 349–356, October 1997.

[HLRW93] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Trans. on Computer Systems*, 11(4):300–318, November 1993.

[HLW95] M. D. Hill, J. R. Larus, and D. A. Wood. Tempest: A Substrate for Portable Parallel Programs. In *Proc. of the 40th IEEE Int'l Computer Conf. (COMPCON Spring'95)*, March 1995.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8), August 1978.

[HT88] M. Hsu and V.-O. Tam. Managing Databases in Distributed Virtual Memory. Technical Report TR-07-88, Aiken Computation Laboratory, Harvard University, March 1988.

[HST98] 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.

[HMT+95] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, G. R. Gao, P. Cupryk, N. Elmasri, L. J. Hendren, A. Jimenez, S. Krishnan, A. Marquez, S. Merali, S. S. Nemawarkar, P. Panangaden, X. Xue, and Y. Zhu. A Design Study of the EARTH Multiprocessor. In *Proc. of the IFIP WG 10.3 Working Conf. on Parallel Architectures and Compilation Techniques, PACT '95*, pages 59–68, June 1995.

[HA90] P. W. Hutto and M. Ahamad. Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories. In *Proc. of the 10th Int'l Conf. on Distributed Computing Systems (ICDCS-10)*, pages 302–311, May 1990.

[IBM00] IBM International Business Machines. StorageTank and SANFS. `http://www.storage.ibm.com/software/virtualization/sfs`, 2000.

[ISL96] 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.

[IS99] 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.

[IEE04] IEE. INSPEC Database on Axiom, Institute of Physics Publishing (IOP), 1968–2004.

[ISW97] A. Itzkovitz, A. Schuster, and L. Wolfovich. Supporting Multiple Parallel Programming Paradigms on Top of the Millipede Virtual Parallel Machine. In *Proc. of the 2nd Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'97)*, April 1997.

[ISS98] 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.

[JB02] R. Jamieson and A. Bilas. CableS: Thread Control and Memory Management Extensions for Shared Virtual Memory Clusters. In *Proc. of the 29th Annual Int'l Symp. on Computer Architecture*, pages 236–247, May 2002.

[Joh99] A. S. Johansen. SENSE—An All-Software, Portable DSM System with a Variable Unit of Coherence. `http://hjem.get2net.dk/dduck/sense.html`, 1999.

[Joh89] E. E. Johnson. The Virtual Port Memory GMMP Multiprocessor. In *Proc. of the 2nd Int'l Conf. on Computing and Information (ICCI'89)*, volume 2, pages 127–130, May 1989.

[JKW95] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proc. of the Fifth Workshop on Scalable Shared Memory Multiprocessors*, June 1995.

[JNT+99] A. Judge, P.A. Nixon, B. Tangney, S. Weber, and V. Cahill. *Distributed Shared Memory*, volume 1, chapter 17, pages 412–441. Prentice Hall, 1999.

[JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Trans. on Computer Systems*, 6(1):109–133, February 1988.

[JF95] N. C. Juul and B. D. Fleisch. A Memory Approach to Consistent, Reliable DSM. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 108–112, May 1995.

[KC93] V. Karamcheti and A. A. Chien. Concert–Efficient Runtime Support for Concurrent Object-oriented Programming Languages on Stock Hardware. In *Proc. of the 1993 ACM/IEEE Conference on Supercomputing*, pages 598–607, December 1993.

[KC97] V. Karamcheti and A. A. Chien. View Caching: Efficient Software Shared Memory for Dynamic Computations. In *Proc. of the 11th Int'l Parallel Processing Symp. (IPPS'97)*, pages 483–489, April 1997.

[KC98] K. M. Kavi and W. E. Cohen. Memory Latency and Thread Migration Challenges for Distributed Shared Memory Systems. In *Proc. of the 31st Hawaii Int'l Conf. on System Sciences (HICSS-31)*, volume VII, pages 772–773, January 1998.

[Kee78] J. L. Keedy. The MONADS Operating System. In *Proc. of the 8th Australian Computer Conf.*, pages 903–910, 1978.

[KCZ92] P. J. 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.

[KDCZ94] P. J. 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.

[Kel94] P. J. Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Department of Computer Science, Rice University, December 1994.

[Kel96] P. J. 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.

[Kel99] P. J. Keleher. Tapeworm: High-Level Abstractions of Shared Accesses. In *Proc. of the 3rd Symp. on Operating Systems Design and Implementation (OSDI'99)*, pages 201–214, February 1999.

[Kha96] D. R. Khandekar. QUARKS: Distributed shared Memory as a Building Block for Complex Parallel and Distributed Systems. Master's thesis, Department of Computer Science, The University of Utah, March 1996.

[Kin99] E. Kindler. A Classigfication of Consistency Models. Technical report, Institute of Informatics, Berlin Free University, October 1999.

[KFJ94] P. T. Koch, R. J. Fowler, and E. B. Jul. Message-Driven Relaxed Consistency in a Software Distributed Shared Memory. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation (OSDI'94)*, pages 75–85, November 1994.

[KCR98] P. T. Koch, E. Cecchet, and X. Ronsset de Pina. Global Management of Coherent Shared Memory on an SCI Cluster. In *Proc. of the European Multimedia, Multiprocessor Systems and Electronic Commerce Conference (SCI Europe'98)*, September 1998.

[KBCC93] D. C. Kulkarni, A. Banerji, M. R. Casey, and D. L. Cohn. Structuring Distributed Shared Memory with the Π Architecture. In *Proc. of the 13th Int'l Conf. on Distributed Computing Systems (ICDCS-13)*, pages 93–100, May 1993.

[LP92] Z. Lahjomri and T. Priol. KOAN: A Shared Virtual Memory for iPSC/2 Hypercube. In *Proc. of the 2nd Joint Int'l Conf. on Vector and Parallel Processing (CONPAR'92)*, pages 441–452, September 1992.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam79] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LLD+83] P. J. Leach, P. H. Levine, B. P. Douros, J. Hamilton, D. L. Nelson, and B. L. Stumpf. The Architecture of an Integrated Local Network. *IEEE Journal on Selected Areas in Communications*, SAC-1(5):842–856, November 1983.

[LQCK96] G. Lee, B. Quattlebaum, S. Cho, and L. Kinney. Global Bus Design of a Bus-based COMA Multiprocessor DICE. In *Proc. of the Int'l Conf. on Computer Design: VLSI in Computers and Processors*, pages 231–240, October 1996.

[LYLM02] S.-K. Lee, H.-C. Yun, J. Lee, and S. Maeng. Design and Implementation of KDSM (KAIST Distributed Shared Memory) System. *Journal of KISS: Computer Systems and Theory*, 29(5–6):257–64, June 2002.

[LR00] L. Lefevre and O. Reymann. Combining Low-Latency Communication Protocols with Multithreading for High Performance DSM Systems on Clusters. In *8th Euromicro Workshop on Parallel and Distributed Processing*, pages 333–338, January 2000.

[LLG+89] 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.

[LLG+90] D. E. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 148–159, May 1990.

[LLW+92] 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.

[LH86] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proc. of the 5th Annual ACM Symp. on Principles of Distributed Computing (PODC'86)*, pages 229–239, August 1986.

[Li86] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Department of Computer Science, Yale University, September 1986.

[LS89] K. Li and R. Schaefer. Shiva: An Operating System Transforming A Hypercube into a Shared-Memory Machine. Technical Report CS-TR-217-89, Department of Computer Science, Princeton University, April 1989.

[Lia94] W.-Y. Liang. Adsmith: A Structure-Based Heterogeneous Distributed Shared Memory on PVM. Master's thesis, Institute of Computer Science, National Tsing Hua University, June 1994.

[Lib85] D. Libes. User-Level Shared Variables. In *1985 USENIX Summer Conf. Proceedings*, June 1985.

[LYL95] Y.-W. Lin, S.-M. Yuan, and D. Liang. Design and Implementation of Moony: A Fault Tolerant Distributed Shared Memory System. *Int'l Journal of Computer Systems Science and Engineering*, 10(2):111–119, April 1995.

[LS88] R. J. Lipton and J. S. Sandberg. PRAM: A Scalable Shared Memory. Technical Report CS-TR-180-88, Department of Computer Science, Princeton University, September 1988.

[LDCZ95] 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.

[Lu97] P. Lu. Aurora: Scoped Behaviour for Per-Context Optimized Distributed Data Sharing. In *Proc. of the 11th Int'l Parallel Processing Symp. (IPPS'97)*, April 1997.

[MOO87] M. Maekawa, A. E. Oldehoeft, and R. R. Oldehoeft. *Operating Systems—Advanced Concepts*. Benjamin/Cummings Publication Company, Inc., Menlo Park, CA, 1987.

[Md99] M. D. Marino and G. L. de Campos. A Preliminary DSM Speedup Comparision: JIAJIA x NAUTILUS. In *Proc. of the 13th Annual Int'l Symp. on High Performance Computing Systems and Applications (HPCS'99)*, pages 713–722, June 1999.

[MHW03] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proc. of the 3oth Annual Int'l Symp. on Computer Architecture (ISCA'30)*, pages 182–193, June 2003.

[MS99] V. Milutinovic and P. Stenstrom. Scanning the Issue: Special Issue on Distributed Shared Memory Systems. *Proc. of the IEEE*, 87(3):399–404, March 1999.

[MF89] R. G. Minnich and D. J. Farber. The Mether System: Distributed Shared Memory for SunOS 4.0. In *Proc. of the 1989 Summer USENIX Conference*, pages 51–60, June 1989.

[MBLZ89] H. E. Mizrahi, J.-L. Baer, E. D. Lazowska, and J. Zahorjan. Introducing Memory into the Switch Elements of Multiprocessor Interconnection Networks. In *Proc. of the 16th Annual Int'l Symp. on Computer Architecture (ISCA'89)*, pages 158–166, May 1989.

[MRZ95a] M. Mizuno, M. Raynal, and J. Z. Zhou. Sequential Consistency in Distributed Systems: Theory and Implementation. Technical Report RR-2437, INRIA, France, March 1995.

[MRZ95b] M. Mizuno, M. Raynal, and J. Z. Zhou. Sequential Consistency in Distributed Systems. In K. Birman, F. Mattern, and A. Schiper, editors, *Proc. of the Int'l Workshop on Theory and Practice in Distributed Systems*, number 938 in Lecture Notes in Computer Science, pages 224–241. Springer-Verlag, July 1995.

[MR94] A. Mohindra and U. Ramachandran. A Comparative Study of Distributed Shared Memory System Design Issues. Technical Report GIT-CC-94/35, College of Computing, Georgia Institute of Technology, August 1994.

[MB98] L. R. Monnerat and R. Bianchini. Efficiently Adapting to Sharing Patterns in Software DSMs. In *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, pages 289–299, February 1998.

[Mos93] D. Mosberger. Memory Consistency Models. *ACM Operating Systems Review*, 27(1):18–26, January 1993.

[Mes95] Message Passing Interface (MPI) Forum. *A Message-Passing Interface Standard*. version 1.1, June 1995.

[Mue97] F. Mueller. On the Design and Implementation of DSM-Threads. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, pages 315–324, June 1997.

[MvT+90] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A Distributed Operating System for the 1990's. *IEEE Computer*, 23(5):44–53, May 1990.

[Mul03] Multifacet Research Group. Wisconsin Multifacet Project. `http://www.cs.wisc.edu/multifacet/`, 2003.

[TFR+01] R. B. Tremaine, P.A. Franaszek, J. T. Robinson, C. O. Schultz, T.B. Smith, M. E. Wazlowski, and P. M. Band. IBM Memory Expansion Technology (MXT). *IBM Research and Development Journal*, 45(2):271–284, March 2001.

[Myr95] Myrias Computer Technologies. *Parallel Application Management System (PAMS) V2*, 1995.

[Nel81] B. J. Nelson. *Remote Procedure Call*. PhD thesis, Carnegie-Mellon University, May 1981.

[Jon02] R. Jones. NetPerf: A Network Performance Measurement Tool. `http://www.netperf.org/netperf/NetperfPage.html`, 2002.

[NL91] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[Nor00] Nortel Networks, Ltd. Alteon Web Systems. `http://www.nortelnetworks.com`, 2000.

[NAB+94] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, D. Lee, and M. Parkin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proc. of the 27th Hawaii Int'l Conf. on System Sciences (HICSS-27)*, volume I, pages 144–153, January 1994.

[NSA97] National Security Council NSA. Lightning Project, 1997.

[OAS95] M. Oguchi, H. Aida, and T. Saito. A Proposal for a DSM Architecture Suitable for a Widely Distributed Environment and Its Evaluation. In *Proc. of the Fourth IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-4)*, pages 32–39, August 1995.

[Ope97] OpenMP Architecture Review Board OpenMP. The OpenMP Application Programming Interface Specifications. `http://www.openmp.org`, 1997.

[OCD+88] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, February 1988.

[PS97] S. M. Paas and K. Scholtyssik. Efficient Distributed Synchronization within an all-software DSM System for Clustered PCs. In *Proc. of the 1st Workshop on Cluster-Computing, TU Chemnitz*, November 1997.

[PBS98] S. M. Paas, T. Bemmel, and K. Scholtyssik. Win32 API Emulation on UNIX for Software DSM. In *Proc. of the 2nd USENIX Windows NT Symposium*, August 1998.

[PLC95] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proc. of the 1995 ACM/IEEE Conference of Supercomputing (CDROM)*, December 1995.

[PK00] D. Perkovic and P. J. Keleher. A Protocol-Centric Approach to On-The-Fly Race Detection. *IEEE Trans. on Parallel and Distributed Systems*, 11(10):1058–1072, October 2000.

[PBG+85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. L. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Prototype (RP3): Introduction and Architecture. In *Proc. of the 1985 Int'l Conf. on Parallel Processing (ICPP'85)*, pages 764–771, August 1985.

[PNB83] L. Philipson, B. Nilsson, and B. Breidegard. A Communication Structure for a Multiprocessor Computer with Distributed Global Memory. In *Proc. of the 10th Annual Int'l Symp. on Computer Architecture (ISCA'83)*, pages 334–340, June 1983.

[Pit00] A. J. Pitman. PVMsynch Client Library. `http://elvis.rowan.edu/~pitman/pvmsync/index.shtml`, March 2000.

[PD93] F. Pong and M. Dubois. The Verification of Cache Coherence Protocols. In *Proc. of the 5th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'93)*, pages 11–20, June 1993.

[PD96] F. Pong and M. Dubois. Formal Verification of Delayed Consistency Protocols. In *Proc. of the 10th Int'l Parallel Processing Symp. (IPPS'96)*, pages 124–131, April 1996.

[PD98] F. Pong and M. Dubois. Formal Verification of Complex Coherence Protocols Using Symbolic State Models. *JACM*, 45(4):557–587, April 1998.

[Rai92] S. Raina. Virtual Shared Memory: A Survey of Techniques and Systems. Technical Report CSTR-92-36, Department of Computer Science, University of Bristol, 1992.

[RAK88] U. Ramachandran, M. Ahamad, and M. Y. A. Khalidi. Unifying Synchronization and Data Transfer in Maintaining Coherence of Distributed Shared Memory. Technical Report GIT-ICS-88/23, College of Computing, Georgia Institute of Technology, June 1988.

[RAK89] U. Ramachandran, M. Ahamad, and M. Y. A. Khalidi. Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer. In *Proc. of the 1989 Int'l Conf. on Parallel Processing (ICPP'89)*, volume II, pages 160–169, August 1989.

[RS93] G. G. Richard III and M. Singhal. Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory. In *Proc. of the 12th Symp. on Reliable Distributed Systems (SRDS'93)*, pages 58–67, October 1993.

[RSL93] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A High Level Machine-Independent Language for Parallel Programming. *IEEE Computer*, 26(6):28–38, June 1993.

[Riz97] L. Rizzo. A Very Fast Algorithm for RAM Compression. *ACM Operating Systems Review*, 31(2):36–45, April 1997.

[RK81] J. Rosenberg and J. L. Keedy. Software Management of a Large Virtual Memory. In *Proc. of the 4th Australian Computer Science Conf. (ACSC-4)*, pages 173–181, January 1981.

[RC98] S. Roy and V. Chaudhary. Strings: A High-Performance Distributed Shared Memory for Symmetrical Multiprocessor Clusters. In *Proc. of the Seventh IEEE Int'l. Symp. on High Performance Distributed Computing*, pages 90–97, July 1998.

[RMP+87] M. Rozier, J. L. Martins, Y. Paker, J.-P. Banatre, and M. Bozyigit. *CHORUS Distributed Operating System: Some Design Issues*, volume 28 of *NATO ASI Series F: Computer and Systems Sciences*, pages 261–288. Springer-Verlag, 1987.

[SHU+00] S. Saito, A. Hayashi, T. Uehara, K. Joe, and Y. Kunieda. Wind : A Low Cost Communication Module for a Software DSM System. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'00)*, pages 729–735, June 2000.

[SGZ93] H. S. Sandhu, B. Gamsa, and S. Zhou. The Shared Region Approach to Software Cache Coherence on Multiprocessors. In *Proc. of the Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'93)*, pages 229–238, July 1993.

[San95] H. S. Sandhu. *Shared Regions: A Strategy for Efficient Cache Management in Share-Memory Multiprocessors*. PhD thesis, Graduate Department of Computer Science, University of Toronto, July 1995.

[SWCL95] 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.

[SL94] D. J. Scales and M. S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation (OSDI'94)*, pages 101–114, November 1994.

[SGT96] 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 (ASPLOS-VII)*, pages 174–185, October 1996.

[SD88] C. Scheurich and M. Dubois. Dynamic Page Migration in Multiprocessors with Distributed Global Memory. In *Proc. of the 8th Int'l Conf. on Distributed Computing Systems (ICDCS-8)*, pages 162–169, June 1988.

[STS98] M. Schoettner, S. Traub, and P. Schulthess. A Transactional DSM Operating System in Java. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, volume I, pages 99–106, July 1998.

[Sch94] W. Schroeder-Preischat. *The Logical Design of Parallel Operating Systems.* Prentice-Hall, Upper Saddle River, N. J., 1994.

[Sch01] M. Schultz. *Shared Memory Programming on NUMA-based Clusters Using a General and Open Hybrid Hardware / Software Approach.* PhD thesis, Department of Informatics, Technical University of Munich, July 2001.

[Sco00] M. L. Scott. Is S-DSM Dead? *Keynote address at the 2nd Workshop on Software Distributed Shared Memory*, May 2000.

[SLD+96] M. L. Scott, W. Li, S. Dwarkadas, L. Kontothanassis, G. Hunt, M. Michael, R. Stets, N. Hardavellas, W. Meira, A. Poulos, M. Cierniak, S. Parthasarathy, and M. Zaki. Implementation of Cashmere. In *Proc. of the Sixth Workshop on Scalable Shared Memory Multiprocessors*, October 1996.

[SWG92] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[Sin93] H. S. Sinha. *Non-Coherent Distributed Shared Memory for Parallel Computing.* PhD thesis, Department of Computer Science, Boston University, May 1993.

[SB97] W. E. Speight and J. K. Bennett. Brazos: A Third Generation DSM System. In *Proc. of the USENIX Windows NT Workshop*, August 1997.

[Sta96] Stardust Technologies, Inc. Windows Sockets 2 Application Programming Interface, 1996.

[SZ90a] M. Stumm and S. Zhou. Algorithms Implementing Distributed Shared Memory. *IEEE Computer*, 23(5):54–64, May 1990.

[SZ90b] M. Stumm and S. Zhou. Fault Tolerant Distributed Shared Memory. In *Proc. of the Second IEEE Symp. on Parallel and Distributed Processing*, pages 719–724, December 1990.

[Sun90] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice&Experience*, 2(4):315–339, December 1990.

[TF90] M.-C. Tam and D. J. Farber. CapNet–An Approach to Ultra High Speed Network. In *Proc. of the IEEE Int'l Conf. on Communications (ICC'90)*, pages 955–961, April 1990.

[TSF90] M.-C. Tam, J. M. Smith, and D. J. Farber. A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems. *ACM Operating Systems Review*, 24(3), July 1990.

[TH90a] V.-O. Tam and M. Hsu. Token Transactions: Managing Fine-Grained Migration of Data. In *Proc. of the 9th ACM Symp. on Principles of Database Systems*, April 1990.

[TH90b] V.-O. Tam and M. Hsu. Fast Recovery in Distributed Shared Virtual Memory Systems. In *Proc. of the 10th Int'l Conf. on Distributed Computing Systems (ICDCS-10)*, pages 38–45, May 1990.

[Tv85] A. S. Tanenbaum and R. van Renesse. Distributed Operating Systems. *ACM Computer Surveys*, 17(4):419–470, December 1985.

[TF95a] O. E. Theel and B. D. Fleisch. Design and Analysis of Highly Available and Scalable Coherence Protocols for Distributed Shared Memory Systems Using Stochastic Modeling. In *Proc. of the 1995 Int'l Conf. on Parallel Processing (ICPP'95)*, volume I, pages 126–130, August 1995.

[TF95b] O. E. Theel and B. D. Fleisch. Analysis of a Fault-Tolerant Coherence Protocol for Distributed Shared Memory Under Heavy Write Loads. In *Proc. of the 1995 IEEE Pacific Rim Conf. on Fault Tolerant Systems (PRFTS'95)*, pages 146–151, December 1995.

[TP96] 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.

[TF99] J. Turk and B. D. Fleisch. DBRpc: A Highly Adaptable Protocol for Reliable DSM Systems. In *Proc. of the 19th Int'l Conf. on Distributed Computing Systems (ICDCS-19)*, pages 340–348, May 1999.

[Ver96] M. Verma. *Compiler-Directed Distributed Shared Virtual Memory*. PhD thesis, Department of Computer Science, State University of New York at Stony Brook, 1996.

[VIA97] Virtual Interface Architecture Forum VIA. Virtual Interface Architecture Specification, Draft Version 1.0, December 1997.

[vCGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Sym. on Computer Architecture (ISCA'92)*, pages 256–266, May 1992.

[WPE+83] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. In *Proc. of the 9th ACM Symp. on Operating Systems Principles*, pages 49–70, October 1983.

[WH88] D. H. D. Warren and S. Haridi. Data Diffusion Machine—A Scalable Share Virtual Memory Multiprocessor. In *Proc. of the Int'l Conf. on Fifth Generation Computer Systems (ICOT'88)*, pages 943–952, 1988.

[WLF01] D. Watson, Y. Lou, and B. D. Fleisch. Experiences with Oasis+: A Fault Tolerant Storage System. In *Proc. of the IEEE Int'l Conf on Cluster Computing*, pages 29–36, October 2001.

[WHG94] G. Wickham, M. Hobbs, and A. Goscinski. The Logical Design of the RHODOS Multithreaded Microkernel. Technical Report TR C94/06, School of Computing and Mathematics, Deakin University, April 1994.

[Wil91] R. N. Williams. An Extremely Fast ZIV-Lempel Data Compression Algorithm. In *Proc. of the Data Compression Conference*, pages 362–371, April 1991.

[WOT+95] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22th Annual Symp. on Computer Architecture*, pages 24–36, June 1995.

[WF90] K.-L. Wu and W. K. Fuchs. Recoverable Distributed Shared Memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.

[WWT01] WWT Research Group. Wisconsin Wind Tunnel Project. `http://www.cs.wisc.edu/~wwt/`, 2001.

[YCGL97] K. G. Yokum, J. S. Chase, A. J. Gallatin, and A. R. Lebeck. Cut-Through Delivery in Trapeze: An Exercise in Low Latency Messaging. In *Proc. of the IEEE Symp. on High-Performance Distributed Computing (HPDC'97)*, pages 243–252, August 1997.

[YTR+87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proc. of the 11th ACM Symp. on Operating Systems Principles (SOSP-11)*, pages 63–76, November 1987.

[YLLM01] H.-C. Yun, S.-K. Lee, J. Lee, and S. Maeng. An Efficient Lock Protocol for Home-based Lazy Release Consistency. In *Proc. of the 1st IEEE/ACM Int'l Symp. on Cluster Computing and the Grid (CCGrid'01)*, pages 527–532, May 2001.

[ZSM90] S. Zhou, M. Stumm, and T. McInerney. Extending Distributed Shared Memory to Heterogeneous Environments. In *Proc. of the 10th Int'l Conf. on Distributed Computing Systems (ICDCS-10)*, May 1990.

[ZSLW92] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous Distributed Shared Memory. *IEEE Trans. on Parallel and Distributed Systems*, 3(5):540–554, September 1992.

[ZIL96] 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.

Abbreviations

COMA	:	Cache Only Memory Architecture
DSM	:	Distributed Shared Memory
EC	:	Entry Consistency
ERC	:	Eager Release Consistency
LRC	:	Lazy Release Consistency
MRSW	:	Multiple Reader Single Writer
MRMW	:	Multiple Reader Multiple Writer
MTU	:	Maximum Transmission Unit
RC	:	Release Consistency
SC	:	Sequential Consistency
ScC	:	Scope Consistency
SPMD	:	Single Program Multiple Data
WC	:	Weak Consistency
WI	:	Write Update
WU	:	Write Invalidate