

C-36-659

NATIONAL SCIENCE FOUNDATION
Washington, D.C. 20550

FINAL PROJECT REPORT
NSF FORM 98A

PLEASE READ INSTRUCTIONS ON REVERSE BEFORE COMPLETING

PART I—PROJECT IDENTIFICATION INFORMATION

Institution and Address Georgia Institute of Technology Office of Contract Administration Atlanta, GA 30332-0420	2. NSF Program Microelectronic Information Processing Sys.	3. NSF Award Number MIP-8809268
	4. Award Period From 6/15/88 To 11/30/90	5. Cumulative Award Amount \$69,316

Project Title
Distributed Operating Systems and Machine Architecture: Reducing the Semantic Gap

PART II—SUMMARY OF COMPLETED PROJECT (FOR PUBLIC USE)

The focus of our work has been in exploring the interface between distributed operating system and the machine architecture, with an eye toward suggesting ways and means of reducing the semantic gap between the two. Operating systems structures for a distributed environment follow one of two paradigms: message-passing or object-invocation. Distributed message-based operating systems consist of a small message-passing kernel supporting a collection of system server processes that provide such services as resource management, file service, and global communications. For such an architecture to be practical, it is essential that basic messages be fast since they often replace what would be a simple procedure call or "kernel call" in a more traditional system. Distributed object-based operating systems view the resources of the system as a collection of objects. Clouds is an object-based distributed operating systems research project at Georgia Tech. System services are encoded in objects and access to system services are requested by invocations on the appropriate system object. The speed of object invocation is often used as a yardstick for measuring the performance of object-based systems.

The work completed under this grant set out to identify the software and hardware mechanisms for efficiently mapping the two paradigms of operating system structures in a distributed setting. The software mechanisms identified included coherence maintenance algorithms for distributed shared memory, and a software architecture for the implementation of distributed shared memory; the hardware mechanisms identified included queuing mechanisms in the memory bus for supporting message-passing, the design and analysis of memory management structures for object-based systems, and the design and analysis of synchronization primitives for multiprocessor caches.

PART III—TECHNICAL INFORMATION (FOR PROGRAM MANAGEMENT USES)

ITEM (Check appropriate blocks)	NONE	ATTACHED	PREVIOUSLY FURNISHED	TO BE FURNISHED SEPARATELY TO PROGRAM	
				Check (✓)	Approx. Date
Abstracts of Theses		X			
Publication Citations	X				
Data on Scientific Collaborators	X				
Information on Inventions	X				
Technical Description of Project and Results		X			
Other (specify)					

Principal Investigator/Project Director Name (Typed) Shore Ramachandran	3. Principal Investigator/Project Director Signature <i>[Signature]</i>	4. Date 5/21/91
--	--	--------------------

PART IV - SUMMARY DATA ON PROJECT PERSONNEL

NSF Division MIPS

The data requested below will be used to develop a statistical profile on the personnel supported through NSF grants. The information on this part is solicited under the authority of the National Science Foundation Act of 1950, as amended. All information provided will be treated as confidential and will be safeguarded in accordance with the provisions of the Privacy Act of 1974. NSF requires that a single copy of this part be submitted with each Final Project Report (NSF Form 98A); however, submission of the requested information is not mandatory and is not a precondition of future awards. If you do not wish to submit this information, please check this box

Please enter the numbers of individuals supported under this NSF grant.
Do not enter information for individuals working less than 40 hours in any calendar year.

*U.S. Citizens/ Permanent Visa	PI's/PD's		Post- doctorals		Graduate Students		Under- graduates		Precollege Teachers		Others	
	Male	Fem.	Male	Fem.	Male	Fem.	Male	Fem.	Male	Fem.	Male	Fem.
American Indian or Alaskan Native												
Asian or Pacific Islander	1				1							
Black, Not of Hispanic Origin												
Hispanic												
White, Not of Hispanic Origin												
U.S. Citizens												
U.S. Citizens	1				1							
U.S. & Non-U.S. . .	1				1							
Number of individuals who have a handicap that limits a major activity.												

*Use the category that best describes person's ethnic/racial status. (If more than one category applies, use the one category that most closely reflects the person's recognition in the community.)

AMERICAN INDIAN OR ALASKAN NATIVE: A person having origins in any of the original peoples of North America, and who maintains cultural identification through tribal affiliation or community recognition.

ASIAN OR PACIFIC ISLANDER: A person having origins in any of the original peoples of the Far East, Southeast Asia, the Indian subcontinent, or the Pacific Islands. This area includes, for example, China, India, Japan, Korea, the Philippine Islands and Samoa.

BLACK, NOT OF HISPANIC ORIGIN: A person having origins in any of the black racial groups of Africa.

HISPANIC: A person of Mexican, Puerto Rican, Cuban, Central or South American or other Spanish culture or origin, regardless of race.

WHITE, NOT OF HISPANIC ORIGIN: A person having origins in any of the original peoples of Europe, North Africa or the Middle East.

THIS PART WILL BE PHYSICALLY SEPARATED FROM THE FINAL PROJECT REPORT AND USED AS A COMPUTER SOURCE DOCUMENT. DO NOT DUPLICATE IT ON THE REVERSE OF ANY OTHER PART OF THE FINAL REPORT.

**Distributed Operating Systems and Machine
Architecture: Reducing the Semantic Gap**

Final Report for NSF Project MIP-8809268

Umakishore Ramachandran

College of Computing
Georgia Institute of Technology
Atlanta, Ga 30332-0280
Phone: (404) 894-5136
E-mail: rama@cc.gatech.edu

The final report summarizes the results of the research carried out under the NSF grant MIP-8809268.

The work proposed under the grant was the investigation of operating systems structures for distributed environments, with an eye toward reducing the semantic gap between the machine architecture and the operating systems. The specific work carried out under the grant may be categorized in three broad categories:

1. The design and implementation of a software architecture for supporting the operating system abstractions in a distributed object-based system.
2. The specification and analysis of hardware assists for enhancing the performance of both message-based and object-based operating system structures.
3. The specification, design, and analysis of synchronization mechanisms in hardware for assisting the efficient design of software systems in shared memory multiprocessors.

The NSF grant had provision for and was used for supporting the salaries of the PI, and one Ph. D. student: Joonwon Lee. A second Ph. D. student, Yousef Khalidi, (not directly paid from this grant, but paid from an institutional infrastructure grant from NSF) also worked with the PI in many of the issues addressed by this grant. The work carried out under this grant produced two Ph. D. theses, and the abstracts from these theses have been enclosed. Yousef Khalidi graduated with a Ph. D. in computer science in April 1989, and is currently working with Sun Microsystems. Joonwon Lee was supported for the entire period of the grant, and is graduating with a Ph. D. in computer science at the end of May. Upon graduation, he will be joining IBM Research Labs, Endicott, New York. It should be pointed out that Joonwon Lee has been supported by an NSF PYI award (September 1990 to present) since the expiration of this grant that was awarded to the PI in 1990.

We have presented the results from this research at several prestigious conferences (including International Symposium on Computer Architecture, and International Conference on Parallel Processing), research labs (including IBM Almaden Research Center, IBM Research in Yorktown Heights, Digital Equipment Corporation in Boston, and the Center for Development of Advanced Computing in India), and Universities (including University of Wisconsin - Madison, and University of Southern California, Los Angeles).

We briefly highlight the significant results from the work carried out under the grant. Following this a list of publication resulting from the grant is given. A set of selected publications and cover pages from all the publication resulting from this grant is also enclosed.

Distributed Shared Memory

Recently, the notion of “Distributed Shared Memory” has received considerable attention. It provides the abstraction of shared memory in a non-shared memory (distributed) architecture. Our contribution to this area has been in unifying the management of distributed shared memory with process synchronization. We proposed new algorithms for coherence maintenance of distributed shared memory [RAK89], an implementation of these algorithms in the Clouds distributed operating system [RK89c] to serve as the basic mechanism for page transport in the system, techniques for programming with distributed shared memory [RK89d], and low-level mechanisms for addressing fault-tolerance in distributed shared memory systems [MR89]. The key idea in our algorithms are the exploitation of application semantics in terms of synchronization information to reduce the overhead of coherence maintenance.

Hardware Support for Object-Based Systems

Object-based systems such as Clouds are very heavily reliant on efficient virtual memory management techniques. This in turn depends on the support that hardware memory management units provide as well as how efficiently remote page faults are serviced through the distributed shared memory facility that we mentioned earlier. Through an exhaustive measurement of the implementation we identified the impediments to the performance of object-based systems [RK89a], and suggested architectural solutions to these impediments. The main problem limiting the efficient implementation of object invocation is the insufficient support provided by hardware memory management units for modern operating systems abstractions. We proposed a set of criteria that memory management units ought to have [RK88], evaluated several commercial memory management units with respect to these criteria, and proposed a design of a memory management unit that satisfies these criteria [RK89b].

Shared Memory Multiprocessors

A key to designing software structures for parallel processors, requires an understanding of the capabilities of the underlying architecture. By the same token, the design of hardware mechanisms should exploit the semantic information forthcoming from the system software such as the operating system and the language runtime system. In this vein, we identified the features of large-scale shared memory multiprocessors that are important from the point of view of realizing parallel applications [LR91]. In particular, we argue for relaxing the memory model seen

by the programmer, and providing efficient synchronization mechanisms in hardware. Further, we identified the performance advantage of providing such synchronization support in hardware [LR90, RL88].

Hardware Support for Interprocess Communication

Lastly, the paper reference [RSV90] identifies hardware mechanisms for efficiently supporting message passing. While most of the work reported in [RSV90] was done prior to this grant period, it is appropriate to include this work in this final report since the PI had to do a sizable amount of work in extending his Ph. D. dissertation work to this final form.

The following is a *list of the publications* that resulted from this grant, and referred to in the final report.

References

- [LR90] Joonwon Lee and Umakishore Ramachandran. Synchronization with multiprocessor caches. In *Proc. 17th Int'l. Symp. on Computer Architecture*, pages 27–37, May 1990. (Also technical report GIT-ICS-89/47, School of ICS, Georgia Institute of Technology, Nov 1989).
- [LR91] Joonwon Lee and Umakishore Ramachandran. Locks, directories, and weak coherence - a recipe for scalable shared memory multiprocessors. In M. Dubois and S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. Also presented at the *1990 ISCA Workshop on Scalable Shared Memory Architectures*, Seattle, Wa, May 1990.
- [MR89] Ajay Mohindra and Umakishore Ramachandran. Implementing fault-tolerant transactions using distributed shared memory. Technical Report GIT-ICS-89/41, Georgia Institute of Technology, November 1989.
- [RAK89] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Coherence of distributed shared memory: Unifying synchronization and data transfer. In *18th International Conference on Parallel Processing, St. Charles, Ill*, pages 160–169, August 1989.

- [RK88] Umakishore Ramachandran and M. Yousef Amin Khalidi. An evaluation of memory management structures for object-based systems. Technical Report GIT-ICS-88/53, School of Information and Computer Science, Georgia Institute of Technology, December 1988. Submitted to *IEEE Transactions on Computers*.
- [RK89a] Umakishore Ramachandran and M. Yousef A. Khalidi. A measurement-based study of hardware support for object invocation. *SOFTWARE — Practice & Experience*, 19(9):809–828, September 1989. Also Technical Report GIT-ICS-88/21.
- [RK89b] Umakishore Ramachandran and M. Yousef Amin Khalidi. A design of a memory management unit for object-based systems. In *IEEE International Conference on Computer Design, ICCD'89, Cambridge, MA*, pages 512–517, October 1989.
- [RK89c] Umakishore Ramachandran and M. Yousef Amin Khalidi. An implementation of distributed shared memory. In *USENIX Workshop on Experiences in Building Distributed and Multiprocessor Systems*, Ft. Lauderdale, Florida, October 1989. To Appear in *Software Practice & Experience*, 1991. Also Technical Report GIT-ICS-88/50.
- [RK89d] Umakishore Ramachandran and M. Yousef Amin Khalidi. Programming with distributed shared memory. In *IEEE 13th International Computer Software and Applications Conference, COMPSAC'89*, pages 176–183, Orlando, Florida, September 1989. Also Technical Report GIT-ICS-88/38.
- [RL88] Umakishore Ramachandran and Joonwon Lee. Processor initiated sharing in multiprocessor caches. Technical Report GIT-ICS-88/43, Georgia Institute of Technology, 1988.
- [RSV90] Umakishore Ramachandran, Marvin Solomon, and Mary Vernon. Hardware support for interprocess communication. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):318–329, 1990.

Thesis Abstracts

Hardware Support for Distributed Object-based Systems

A Thesis
Presented to
The Academic Faculty

By

M. Yousef Amin Khalidi

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy
in the
School of Information and Computer Science

Georgia Institute of Technology
June 1989

Summary

The object-based approach is an attractive model for structuring distributed systems. Distributed object-based operating systems view the resources of the system as a collection of objects. The primary mechanism in object-based systems is the mapping of the invoked object into the invoking thread's address space. Hence the performance of object-based systems depends crucially on the efficiency of this memory mapping. The problem gets exacerbated with distribution since the invoked object may now be located on a remote node. This research concentrates on identifying local and network-wide memory management requirements and on providing software and hardware mechanisms to handle these requirements.

The contributions of this research are twofold. In the area of local memory management, this research reveals some of the inadequacies of commercial memory management units (MMUs) for supporting the requirements of object-based operating systems, and presents the design of an MMU that meets these requirements. The design presented in this thesis shows that it is possible to meet these requirements with a combination of the features available in commercial MMUs.

In the area of network-wide memory management, this work suggests structuring the distributed operating system around the concept of *distributed shared memory*. Since a thread can potentially invoke any object, the virtual address spaces of all objects can be viewed as constituting a global distributed shared memory. Such a view is attractive from the perspective of software architecture since it suggests a uniform implementation of a system-wide memory-mapping mechanism. An organization and mechanisms for

supporting this abstraction of a distributed shared memory are presented, and a software implementation is described. These mechanisms serve as the backbone for implementing object invocation, synchronization mechanisms, paging of memory segments, and process and object migration in object-based kernels.

Architectural Features for Scalable Shared Memory Multiprocessors

A Thesis
Presented to
The Faculty of the Division of Graduate Studies

By

Joonwon Lee

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy
in Information and Computer Science

Georgia Institute of Technology
May 1991

Summary

Cache memories are important in multiprocessors because of memory latencies caused by bus saturation and/or multi-hop interconnections. Traditional multiprocessor cache protocols treat synchronization accesses the same way as normal read/write memory accesses. This approach leads to inefficiencies in performing synchronization operations which ultimately limits the scalability of such systems. We propose synchronization schemes in multiprocessors using cache memories: one for shared bus, and another for general interconnection networks. Each cache line maintains states for synchronization as well as for cache coherence, and the cache protocol ensures correct synchronization operations.

Providing the ability to distinguish between synchronization requests and read/write memory accesses allows the software to defer global coherence for shared data until it is absolutely necessary. In this weak consistency model, strong memory coherence may be enforced only at synchronization point. We propose new cache primitives for the weak consistency model and show how they can be implemented with private caches.

To assess the performance gain due to our cache protocols, we have performed simulation studies. The workload model represents a dynamic scheduling paradigm which is the kernel of several parallel programs. There is a global work queue of tasks that represents the parallel computation. Each processor takes a task from the work queue and executes it. Queue accesses from the processors are mutually exclusive and

barrier synchronizations at the end of a phase of computation are used when needed. Results from simulation studies show that our protocol performs significantly better than the traditional approaches.

Cover Pages of Publications

Synchronization with Multiprocessor Caches *

PAPER [LR90] Joonwon Lee Umakishore Ramachandran

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332 USA
e-mail: joon@sybil.gatech.edu, rama@gatech.edu

Abstract

Introducing private caches in bus-based shared memory multiprocessors leads to the cache consistency problem since there may be multiple copies of shared data. However, the ability to snoop on the bus coupled with the fast broadcast capability allows the design of special hardware support for synchronization. We present a new lock-based cache scheme which incorporates synchronization into the cache coherency mechanism. With this scheme high-level synchronization primitives as well as low-level ones can be implemented without excessive overhead. Cost functions for well-known synchronization methods are derived for invalidation schemes, write update schemes, and our lock-based scheme. To accurately predict the performance implications of the new scheme, a new simulation model is developed embodying a widely accepted paradigm of parallel programming. It is shown that our lock-based protocol outperforms existing cache protocols.

1 Introduction

Cache memories have been used to reduce memory access latency in uniprocessors. In bus-based shared memory multiprocessors they may additionally reduce bus contention. However, private caches introduce the cache coherence problem [8]. The shared bus enables each cache controller to monitor the bus traffic and initiate appropriate actions to keep the shared data coherent. A group of cache coherence schemes called *snooping cache protocols* use this feature [13, 16, 18, 19, 24]. They are implemented in hardware and may not be visible to the programmer.

Most snooping cache protocols (Section 2) do not take into account synchronization requests that usually precede shared data accesses. Therefore, strong coherence among multiple copies is blindly enforced resulting in possibly unnecessary bus traffic for invalidation and data transfer. Usually, access to shared data is acquired via synchronization methods such as locks, semaphores, and barriers. Thus there is additional delay in accessing the synchronization variables and then acquiring the actual data.

In this paper, we present a cache coherence protocol supporting lock primitives (Section 3). Our scheme utilizes locking information provided by the software (e.g. compiler) to distinguish between shared-locks and exclusive-locks. We construct a distributed hardware-assisted FIFO queue of processors waiting for a given lock. With this scheme, the cache mechanism emerges as a visible part of the architecture since programmers should understand it to develop efficient parallel programs.

Efficient interprocessor synchronization and mutual exclusion are imperative to assure good performance for parallel programs. Therefore, we must evaluate the synchronization efficiency of cache protocols (Section 4).

Evaluating a multiprocessor system is a challenging task. Trace driven simulation has been used for multiprocessor evaluation [1, 11, 12, 25], but tracing parallel programs has so far been restricted to a small number of processors. Furthermore, the trace is affected by the host multiprocessor's architectural characteristics such as cache protocol, synchronization primitives, and interconnection network. As Bitar [4] points out using traces generated by software makes it difficult to verify the validity of the predicted results. Therefore, we have developed a new method for the evaluation of cache protocols. Our simulation model and some results are presented in Section 5.

2 Snooping Caches

Hardware cache coherence schemes for shared-bus multiprocessors have evolved into two categories, namely, *invalidation* schemes and *write update* schemes.

In invalidation schemes [13, 16], a write to a cached line results in invalidating copies of this line present in other caches. If writes to cached data always trigger invalidations, the bus is easily saturated with even a few participating processors [27]. To reduce the invalidation rate, Goodman developed the *write once* protocol [13] in which only the first write to a cached data updates main memory, and is used as a cue by other caches for invalidating their own copies. The Berkeley protocol [16] assumes an invalidation line on the bus to explicitly invalidate peer caches. Since an invalidation is induced even with a first

*This work is supported in part by NSF grant MIP-8809268

PAPER [LR-91]

Locks, Directories, and Weak Coherence - A Recipe for Scalable
Shared Memory Multiprocessors*

(FULL PAPER ENCLOSED UNDER "SELECTED PUBLICATIONS IN FULL")
Joonwon Lee Umakishore Ramachandran

Technical Report GIT-CC-90/68

December 1990

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

joon@cc.gatech.edu, rama@cc.gatech.edu

Abstract

Bus based multiprocessors have the limitation that they do not scale well to large numbers of processors due to the bus becoming a bottleneck with the current bus technology. Lock-based protocols have been suggested as a possible way of mitigating this bottleneck for single bus systems with snooping ability. In this research, we are interested in extending lock-based protocols to general interconnection networks. Directory based cache coherence schemes have been proposed for such networks. We are investigating a combination of locking with directory based schemes. Further, most protocols in the literature until now, assume a strong coherence requirement. However, recent research has shown that it is possible to weaken this coherence requirement. Such an approach is expected to reduce the coherence overhead even further, making it an appealing one for building scalable systems.

*This work is supported in part by NSF grant MIP-8809268.

PAPER [MR89]

Implementing Fault-tolerant Atomic Transactions Using Distributed Shared Memory*

Ajay Mohindra

Umakishore Ramachandran

School of Information and Computer Science

Georgia Institute of Technology

e-mail: ajay@ics.gatech.edu rama@gatech.edu

Phone: (404) 894-5136

TECHNICAL REPORT GIT-ICS-89/41

Abstract

The abstraction for supporting the notion of shared memory in a non-shared memory (distributed) architecture is referred to as distributed shared memory. We have implemented a set of mechanisms for maintaining the coherence of distributed shared memory. In this paper we show how distributed fault-tolerant atomic transactions can be implemented elegantly using these mechanisms. Since our mechanisms combine the tasks of synchronization and data transfer into one, transaction-level locking of segments comes for free. Fault-tolerance is achieved by using replication on top of distributed shared memory. Atomicity of transactions is guaranteed through use of a two-phase commit protocol. We show that the resulting implementation incurs much less overhead compared to message-passing schemes.

*This work has been funded in part by NSF grants CCR-8619886 and MIP-8809268.

PAPER [RAK 89]

Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer

Umakishore Ramachandran Mustaque Ahamad M. Yousef A. Khalidi

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Ga 30332-0280 USA

(FULL PAPER ENCLOSED UNDER "SELECTED PUBLICATIONS IN FULL")

Abstract

Clouds is a distributed operating systems research project at Georgia Tech. With threads and passive objects as the primary building blocks, *Clouds* provides a location-transparent protected procedure-call interface to system services. Mechanisms for synchronization within objects, and atomicity of computation are supported in *Clouds*. The primary kernel mechanism in object-based systems such as *Clouds* is the mapping of the object into the address space of the invoking thread. Hence the performance of such systems depends crucially on the efficiency of memory mapping. The problem gets exacerbated with distribution since now the object invoked by a thread may be located on a remote node. Since a thread can potentially invoke any object, the virtual address spaces of all objects can be viewed as constituting a "global distributed shared memory". Such a view is attractive from the perspective of software architecture since it suggests a uniform implementation of a system-wide memory-mapping mechanism. We present an organization and mechanisms for supporting this abstraction of a distributed shared memory. We propose a *distributed shared memory controller* that provides mechanisms for efficient access and consistency maintenance of the distributed shared memory. The novel feature of our approach is the exploitation of process synchronization to simplify consistency maintenance. The distributed shared memory mechanisms serve as the backbone for implementing object invocation, synchronization mechanisms, and network-wide memory management in the *Clouds* system.

1 Introduction

We are exploring hardware support to improve the performance of object-based distributed operating systems. The hardware environment consists of a collection of computing nodes interconnected by a local area network. There are one or more processors and a certain amount of memory in each node. Nodes do not share memory; message exchange across the network is the only mechanism for communication between them.

Many operating systems designs for such an environment [4,14,20,32] place a *message-passing kernel* on each node, supporting processes and communication between them via explicit messages. This kernel supports both *local communication*—communication between processes on the same node—and *non-local or remote communication* (sometimes implemented via a distinguished network manager process). Access to system services are requested via protected *procedure calls* in a traditional system, whereas in a message-based operating system they are requested via *message passing*. While a simple procedure call costs just a few instructions, and a protected procedure call (kernel call) costs a few hundred instructions, IPC costs a few

This work has been funded in part by NSF grants CCR-8619886 and MIP-8809268.

thousand instructions in several systems that we studied [28]. Message-based operating systems are attractive for structuring distributed systems due to the separation of policy (encoded in server processes) from mechanism (in the kernel).

Object-based distributed operating systems [7,2,39,34,26] view the resources of the system as a collection of objects. *Clouds* [7] is an object-based distributed operating system being developed at Georgia Tech. In *Clouds*, system services are encoded in *passive objects* (syntactic units that are similar in flavor to the server processes of message-based systems) that occupy distinct virtual address spaces in the system. Access to system services are requested by invocations (similar to the protected procedure calls of traditional systems) into the appropriate system object. The speed of object invocation is often used as a yardstick for measuring the performance of object-based systems. In passive object-based systems, invocation performance depends on the efficiency of object memory management (see §2). The problem gets exacerbated with distribution since now the invoked object may be located on a remote node.

In this paper we suggest mechanisms (that can be implemented in hardware/firmware) for supporting the abstraction of "globally distributed shared memory". In §2, we give the relevance of our work and motivate the need for supporting this abstraction. Related work in the area of memory coherence is presented in §3. With the *Clouds* operating system as our target application, we present our ideas on customizing the memory coherence requirements in §4. Our proposed hardware organization, the primitives provided by the distributed shared memory controller, and the algorithms for maintaining the consistency of the distributed shared memory are discussed in §5. In §6 we describe a software implementation of the proposed primitives. A performance evaluation of our scheme is presented in §7. Finally, our conclusions are presented in §8.

2 Relevance

Clouds [7] is a distributed operating system that is intended to provide a unified environment over distributed hardware. Location independence for data as well as processing, atomicity of distributed computation, and fault-tolerance are some of the research goals of *Clouds*.

Objects and *threads* are the basic building blocks of *Clouds*. Objects are passive entities and specify a *distinct* and *disjoint* piece of the global virtual address space that spans the entire network. An object is the encapsulation of the *code* and *data* needed to implement the *entry points* in the object. Thus a *Clouds* object can be considered syntactically equivalent to an abstract data type in the programming language parlance. Access to entry points in the object are accomplished through a capability mechanism in the kernel.

Threads are the only active entities in the system. A thread is a unit of activity from the user's perspective. Upon creation, a thread starts executing in an object. A thread enters an ob-

SUBMITTED TO IEEE TRANSACTIONS ON COMPUTERS.

PAPER [RK 88]

An Evaluation of Memory Management Structures for Object-based Systems*

M. Yousef Amin Khalidi
Sun Microsystems, Inc.
2550 Garcia Ave., MS 5-44
Mt. View, CA 94043 U.S.A.
Tel. (415) 336-5392
e-mail: yak@eng.sun.com

Umakishore Ramachandran
College of Computing
Georgia Institute of Technology
Atlanta, Ga 30332-0280 U.S.A.
Tel. (404) 894-5136
e-mail: rama@cc.gatech.edu

Abstract

Object-based operating systems have the desirable property of separating policies from mechanisms, while providing a protected procedure call interface for accessing system services. However, the kernel mechanisms in such systems rely very heavily on efficient memory management. A set of criteria for supporting the kernel mechanisms in object-based systems is presented. Five commercial memory management units (MMUs) are surveyed with respect to these criteria. We then present the design of a simple MMU tailored for object-based systems. The proposed design is an engineering solution combining the features available in commercial MMUs.

Keywords: Operating systems, object-based systems, memory management, object invocation.

*This work has been funded in part by NSF grants CCR-8619886 and MIP-8809268.

PAPER : [RK89a]

A Measurement-based Study of Hardware Support for Object Invocation

(FULL PAPER ENCLOSED UNDER "SELECTED PUBLICATIONS IN FULL")

UMAKISHORE RAMACHANDRAN AND M. YOUSEF AMIN KHALIDI
School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA
30332-0280, U.S.A.

SUMMARY

The object invocation paradigm is attractive for structuring distributed systems. Distributed object-based operating systems view the resources of the system as a collection of objects. Object invocation is the primary mechanism in such systems, and is often used as a yardstick for measuring the system performance. However, existing systems of this flavour exhibit poor performance due to the mismatch between the requirements of the object invocation mechanism and the machine architecture. Through measurements of an existing object-based kernel, we present a breakdown of the costs involved in implementing the object invocation mechanism. The measurements suggest architectural solutions to improve the performance of such systems. We present our preliminary studies towards providing hardware support for the object invocation mechanism.

KEY WORDS Object invocation Measurements Hardware support Distributed operating systems

INTRODUCTION

This paper explores architectural support to improve the performance of object-based distributed operating systems. The hardware environment consists of a collection of computing nodes interconnected by a local area network. There are one or more processors and a certain amount of memory in each node. Nodes do not share memory; message exchange across the network is the only mechanism for communication between them.

Operating system structures for a distributed environment follow one of two paradigms: *message passing* or *object invocation*. Message-based operating systems¹⁻⁴ place a *message-passing kernel* on each node, supporting processes and communication between them via explicit messages. This kernel supports both *local communication*—communication between processes on the same node—and *non-local* or *remote* communication, sometimes implemented via a distinguished network manager process. In a traditional system such as Unix, access to system services is requested via *protected procedure calls*, whereas in a message-based operating system it is requested via *message passing*. Message-based operating systems are attractive for structuring distributed systems due to the separation of policy, encoded in server processes, from mechanism in the kernel.

0038-0644/89/090809-20\$10.00
© 1989 by John Wiley & Sons, Ltd.

Received 9 June 1988
Revised 3 January 1989

CONCLUSIONS AND FUTURE WORK

The Clouds kernel served well as a prototype, and enabled us to gain important insight into the requirements of the object-based model of computation. However, it has several drawbacks. It is a monolithic kernel that is hard to modify and maintain, and the implementation is very machine-dependent. In addition, message transmission in Clouds is slow. Sending messages from one process running in a local kernel to a process running in a remote kernel takes roughly 10 ms. Most of this cost can be attributed to a poor network interface.³⁵ Other systems, such as QuickSilver³ and V,¹ report numbers in the range of 6 to 1.5 ms on faster hardware.

Our notions on structuring object-based operating systems has matured since the prototype design was begun. A new kernel for Clouds, called the *Ra* kernel, has been designed and implemented.^{5,9} *Ra* runs on the Sun-3 architecture, and incorporates support for distributed shared memory. The DSMC has been implemented in software and we are currently in the process of evaluating the implementation,^{36,37} Further refinement and implementation of an MMU tailored to object-based systems that incorporates our TLB scheme, and a hardware module that implements the DSMC protocol are some of the work we have identified for future research.

ACKNOWLEDGEMENT

This work has been funded in part by NSF grants CCR-8619886 and MIP-8809268.

REFERENCES

1. D.R. Cheriton and W. Zwaenepoel, 'The distributed V kernel and its performance for diskless workstations', *Operating Systems Review*, 17,(5), 123-140 (1983).
2. Mike Accetta, Roberta Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian and Michael Young, 'Mach: a new kernel foundation for Unix development', *Proceedings Summer Usenix*, July 1986.
3. R. Haskin, Y. Malachi, W. Sawdon and G. Chan, 'Recovery management in QuickSilver', *Proceedings of the 11th Symposium on Operating Systems Principles*, November 1987.
4. R. Rashid and G. Robertson, 'Accent: a communication oriented network operating system kernel', *Proceedings of the 8th Symposium on Operating Systems Principles*, December 1981, pp. 64-75.
5. José M. Bernabéu Aubán, Phillip W. Hutto, M. Yousef A. Khalidi, Mustaque Ahamad, William F. Appelbe, Partha Dasgupta, Richard J. LeBlanc, and Umakishore Ramachandran, 'The architecture of Ra: a kernel for Clouds', *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, January 1989.
6. Guy T. Almes, Andrew P. Black, Edward D. Lazowska and Jerre D. Noe, 'The Eden system: a technical review', *IEEE Trans. Software Engineering*, SE-11, (1), 43-58 (1985).
7. B. Liskov and R. Scheifler, 'Guardians and actions: linguistic support for robust, distributed programs', *Ninth Conference on Principles of Programming Languages*, 1982.
8. J. Duane Northcutt, *Mechanisms for Reliable Distributed Real-Time Operating Systems*, Volume 16 of *Perspectives in Computing*, Academic Press, 1987.
9. José M. Bernabéu Aubán, Phillip W. Hutto, M. Yousef A. Khalidi, Mustaque Ahamad, William F. Appelbe, Partha Dasgupta, Richard J. LeBlanc and Umakishore Ramachandran, 'Clouds—a distributed, object-based operating system: architecture and kernel implementation', *European UNIX systems User Group Autumn Conference*, EUUG, October 1988.
10. Andrew D. Birrell and Bruce Jay Nelson, 'Implementing remote procedure calls', *ACM Transactions on Computer Systems*, 2, (1), 39-59 (1984).
11. Digital Equipment Corporation, *VAX Architecture Handbook*, 1986.
12. E.H. Spafford, 'Kernel structures for a distributed operating system', *Ph.D. Thesis*, School of Information and Computer Science, Georgia Institute of Technology, 1986. Available as *Technical Report GIT-ICS-86/16*.

A Design of a Memory Management Unit for Object-based Systems

Umakishore Ramachandran

M. Yousef Amin Khalidi

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Ga 30332-0280 USA

Abstract

Object-based operating systems have the desirable property of separating policies from mechanisms, while providing a protected procedure call interface for accessing system services. However, the kernel mechanisms in such systems rely very heavily on efficient memory management. A set of criteria for supporting the kernel mechanisms in object-based systems is presented. We then present the design of a simple MMU tailored for object-based systems. The proposed design is an engineering solution combining the features available in commercial MMUs.

1 Introduction

Traditional systems such as Unix, provide a protected procedure call interface for system services. However, in such systems the kernel tends to be monolithic and bulky since the policies and the mechanisms are all embedded in the kernel. In recent times, especially with the advent of local area networks, message-based operating systems have become popular [8,1,21,11]. Policies are encoded in server processes and the kernel is lean and supports simply a message passing mechanism between processes. System services are requested by sending and receiving messages. However, the major drawback in such systems is the fact that a round-trip message is involved for requesting system services which can be an order of magnitude more expensive than a protected procedure call [18].

Object-based systems [5,2,14,17] view the resources of the system as a collection of objects. An object is the encapsulation of the code and data needed to implement the entry points in the object. These entry points provides the procedural interface for an activity to execute the code in the object. System services are requested via invocations (similar to a protected procedure call) on these entry points. At the same time the kernel is lean since it provides simply the invocation mechanism with the policies encoded in system objects outside the kernel. Thus object-based systems, at least in principle, combine the advantages of traditional and message-based systems.

However, implementation of object-based systems on conventional machine architectures can result in performance that is worse than message-based systems [19]. Object-based operating systems are critically dependent on efficient memory management. Specifically, an object specifies a virtual space. Virtual spaces are composed of memory segments. Independent virtual spaces can share memory segments or parts thereof. An object invocation entails entering a different virtual space. In a passive object system, there are no processes associated with an object. On invocation, the virtual space of the invoked object is mapped into the invoking process.

This work has been funded in part by NSF grants CCR-8619886 and MIP-8809268.

Clouds [5] is an example of such an object-based system. Ra [4] is an operating system kernel designed to be the nucleus of Clouds. The abstractions that Ra uses for supporting memory management are segments and virtual spaces. A Ra virtual space is described by a segment called the *virtual space descriptor* (VSD) that contains a collection of *windows*. Each window is a data structure that identifies the segment that backs a range of addresses in the virtual space (Figures 1 and 2).

An object virtual space is called an O space while a process virtual space is called a P space. Ra views the machine address space as consisting of three distinct regions that are called K, O, and P hardware spaces for kernel, object, and process, respectively. The kernel is always mapped in the K hardware space. A process P space is mapped (or *installed*) into the P hardware space and unmapped on context switch. On object call, the O space of the invoked object is installed, and on object return, the O space of the caller object is installed.

Ra advocates a model of programming with a large number of possibly small-sized segments. Current implementation of Ra on the Sun-3 allows around 100 windows.

2 Comparison Criteria

As can be seen from the description of Clouds and Ra, an efficient implementation relies very heavily on effective management of virtual memory. In particular, the following five MMU features are required to support an efficient implementation of Clouds/Ra:

- C1. *The ability to cache TLB information across object invocation.* The cost of flushing TLB entries at object call/return is an implicit cost that affects mostly user mode time. Systems with one large virtual address space do not require flushing the TLB across object invocations. Examples of such systems include the IBM RT/PC, HP Precision Architecture, and SPUR machine [22].
- C2. *The ability to implement an object call/return by performing a small number of operations on the MMU.* The cost of required MMU operations is an explicit part of object invocation. This cost is quantified by the number of MMU registers and page table entries that need to be modified on each object call/return. In our proposed MMU (see §3), changing one register is the only operation that is required to effect an object call/return. A paper design of object invocation implementation on segmented single virtual address space systems, such as IBM RT/PC and HP Precision Architecture, reveals that only a small number of MMU operations is required [20]. However, our experience in implementing Ra on the Sun-3 MMU [4] reveals that many MMU operations are needed on this machine.
- C3. *The ability to represent sparse address spaces efficiently.* Inefficient representation of sparse address spaces results in page tables that have large memory requirements and

PAPER [RK89C]

An Implementation of Distributed Shared Memory*

Umakishore Ramachandran
School of ICS
Georgia Institute of Technology
Atlanta, Ga 30332
Phone: (404) 894-5136
E-mail: rama@gatech.edu

M. Yousef A. Khalidi
Sun Microsystems
2550 Garcia Avenue
Mountain View, Ca 94043
Phone: (415) 336-5392
E-mail: yak@sun.com

Abstract

Shared memory is a simple yet powerful paradigm for structuring systems. Recently, there has been an interest in extending this paradigm to non-shared memory architectures as well. For example, the virtual address spaces for all objects in a distributed object-based system could be viewed as constituting a global *distributed shared memory*. We propose a set of primitives for managing distributed shared memory. We present an implementation of these primitives in the context of an object-based operating system as well as on top of Unix.

1 Introduction

Programming with shared memory is well-understood and despite the interest in distributed and parallel systems for reasons of availability, fault-tolerance, and increased computational power, the style of programming these systems has not changed drastically. Even in non-shared memory architectures researchers have proposed a style that presents to the programmers an abstraction of a logical shared memory [19, 14, 8, 23]. Other researchers have proposed algorithms for maintaining the consistency of such a logically shared memory in non-shared memory architectures [17, 18, 21]. The abstraction for supporting the notion of shared memory on a non-shared memory (distributed) architecture is referred to as *distributed shared memory* (DSM) in this paper.

A second motivation for DSM is the current trend in structuring distributed systems using a collection of diskless computational servers, namely workstations, and a few data servers or file servers. In such an environment the code and data for program execution has to be paged-in from the data server. There are two issues here: The first one is a scheduling decision of 'where' to execute the program, one that is best left to a higher level policy making entity. The second one is the chore of bringing in the required data and code, i.e., remote paging. If sharing is coupled with this second issue, then we see that DSM presents itself as a natural facility for combining the two.

Several other researchers have proposed software architectures based on the shared memory paradigm, in different settings:

*This work has been funded in part by NSF grants CCR-8619886 and MIPS-8809268.

TO APPEAR IN SOFTWARE PRACTICE + EXPERIENCE, 1991
PAPER [RK89C]

An Implementation of Distributed Shared Memory*

Umakishore Ramachandran
School of ICS
Georgia Institute of Technology
Atlanta, Ga 30332
Phone: (404) 894-5136
E-mail: rama@gatech.edu

M. Yousef A. Khalidi
Sun Microsystems
2550 Garcia Avenue
Mountain View, Ca 94043
Phone: (415) 336-5392
E-mail: yak@sun.com

Abstract

Shared memory is a simple yet powerful paradigm for structuring systems. Recently, there has been an interest in extending this paradigm to non-shared memory architectures as well. For example, the virtual address spaces for all objects in a distributed object-based system could be viewed as constituting a global *distributed shared memory*. We propose a set of primitives for managing distributed shared memory. We present an implementation of these primitives in the context of an object-based operating system as well as on top of Unix.

Keywords: Distributed shared memory, distributed operating systems, object-based systems.

*This work has been funded in part by NSF grants CCR-8619886 and MIPS-8809268. An earlier version of this paper was presented at the USENIX workshop on experiences in building distributed and multiprocessor systems, Ft. Lauderdale, Florida, October 1989.

Programming with Distributed Shared Memory

Umakishore Ramachandran

M. Yousef Amin Khalidi

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Ga 30332-0280 USA

Abstract

In a distributed system, remote services may be provided by either remote procedure call (RPC) mechanism, or by paging in the required memory segments and performing the services locally. The latter approach, termed *Distributed Shared Memory* (DSM) has several benefits given the current trend of structuring computing systems using diskless computational servers (workstations) and data servers (file servers). We propose a set of distributed shared memory mechanisms that handle network-wide memory management for an object-based system. We discuss the implementation of these mechanisms and provide examples of their use in implementing the programming language *Linda*, process migration, two-phase commit, and a distributed game.

the segment. A partition is an entity that realizes, maintains, and manipulate segments (see §4).

Virtual spaces abstract the notion of an addressing domain. A Ra virtual space is described by a segment called the *virtual space descriptor* that contains a collection of *windows*. Each window is a data structure that identifies the segment that backs a range of addresses in the virtual space. A virtual space is composed (or built) by a sequence of attach operations, each of which defines one of the virtual space's windows. An object is a virtual space (called O space) that consists of code and data segments. The code segment of an O space has entry points that can be invoked by user processes.

Ra isibas are an abstraction of the fundamental notion of computation or activity and can be thought of as lightweight processes. Isibas may be used as daemons within the kernel or they may be associated with a virtual space (a P space) to implement a user process.

The machine virtual address space consists of three distinct regions that are called K, O, and P hardware spaces for kernel, object, and process, respectively. The kernel is always mapped in the K hardware space. (System objects, which we discuss in §6, are also mapped into the K space, but may be installed and removed dynamically.) A Process is mapped into the P hardware space and unmapped on context switch. A process consists of an isiba and a virtual space (P space). The object in which a process is currently executing is mapped into the hardware O space.

Local object invocation involves mapping the required memory segments of the object into the address space of the invoking process by installing the object as the current O space with the process's P space. The current trend in structuring distributed systems is to use a collection of diskless computational servers (workstations) and a few data servers (file servers). In such an environment, the code and data for the (local) invocation has to be paged-in from the data server. Further, for remote object invocation we have one of two choices: The first choice is to perform the computation at the node where the object resides (remote procedure call). The second choice is to make the invocation appear local by bringing in the segments required for the invocation. While we have to support the former for immovable objects (such as an object that reads disk blocks), we believe that the latter may be a better choice for movable objects. There are two reasons to support this belief:

- the principle of locality [14] that suggests an invocation (or other invocations in the same object) may be repeated
- the reduction in computational overhead due to the elimination of slave process management to support remote invocation at the node where the object resides [16].

Each host has a distributed shared memory controller (DSMC) that together with the network interface assists the host in mapping remote memory segments into virtual address spaces on the local host. The system memory is (logically) partitioned

1 Introduction

The resources in a loosely-coupled distributed system can be viewed as a collection of objects. The name space of all objects constitute a "global distributed shared memory". In this paper, we suggest mechanisms for supporting the abstraction of a globally distributed shared memory, and applications using these mechanisms. These mechanisms were designed in the context of the *Clouds* [7,13] object-based distributed operating system.

The paper is organized as follows: We start with an overview of *Ra*—a kernel for *Clouds* and its relation to the shared memory module which we call *distributed shared memory controller* (DSMC). In §3 we discuss related work, and in §4 we discuss the status of our implementation. The three sections that follow present applications that use the DSMC mechanisms in implementing the *Linda* parallel programming language [15], process migration, and 2-phase commit. Section 8 investigates the use of weak memory coherence semantics in the implementation of a distributed game. Section 9 evaluates the use of distributed shared memory vs. remote procedure call for invoking remote objects. Finally, conclusions and future work follow in §10.

2 Ra and DSMC

Ra [5,7,6] is an operating system kernel designed to be the nucleus of *Clouds* operating system. It currently runs on the Sun-3 architecture. Ra defines and manages three primitive abstractions: *segment*, *virtual space*, and *isiba*. Segments serve as containers of data and may be viewed as uninterpreted sequences of bytes. The contents of a segment may only be accessed when the segment is *attached* to a range of virtual addresses. Segments persist until explicitly destroyed. Each segment resides in a *partition* that is responsible for providing backing store for

This work has been funded in part by NSF grants CCR-8619886 and MIP-8809268.

PAPER [RL 88]

Processor Initiated Sharing in Multiprocessor Caches *

Umakishore Ramachandran

Joonwon Lee

School of Information and Computer Science

Georgia Institute of Technology

Atlanta, Georgia 30332 USA

Ph: (404) 894-5136

rama@gatech.edu, joon@pyr.gatech.edu

Abstract

We present a lock-based multiprocessor cache consistency protocol. This protocol requires the processes to provide information about the data they access: read or write, shared or non-shared. This information enables caches to avoid frequent invalidations and data transfers. Shared data can be accessed only through lock operations. These lock operations entail queues for the processes awaiting a lock. A distributed hardware-assisted queue which ensures fairness and efficient waiting scheme is proposed. Implementation of the protocol with a current computer bus is discussed.

Keywords: Shared Memory Multiprocessor, Cache Coherency, Process Synchronization

*This work is supported in part by NSF grant MIP-8809268

Hardware Support for Interprocess Communication

UMAKISHORE RAMACHANDRAN, MARVIN SOLOMON, AND MARY K. VERNON, MEMBER, IEEE

PAPER [RSV 90]

(FULL PAPER ENCLOSED UNDER "SELECTED PUBLICATIONS IN FULL")

Abstract—In recent years there has been increasing interest in message-based operating systems, particularly in distributed environments. Such systems consist of a small message-passing kernel supporting a collection of *system server* processes that provide such services as resource management, file service, and global communications. For such an architecture to be practical, it is essential that basic message exchanges be fast, since they often replace what would be a simple procedure call or "kernel call" in a more traditional system. Careful study of several operating systems shows that the limiting factor, especially for small messages, is typically not network bandwidth but processing overhead. Therefore, we propose using a special-purpose coprocessor to support message passing. Our research has two parts. First, we partitioned an actual message-based operating system into *computation* and *communication* parts, executing, respectively, on a *host* and a *message coprocessor* interacting through shared queues, and measured its performance on a multiprocessor. Second, we designed hardware support in the form of a special-purpose *smart bus* and *smart shared memory* and demonstrated the benefits of these components through analytical modeling using *Generalized Timed Petri Nets*. Our analysis shows good agreement with the experimental results and indicates that substantial benefits may be obtained from both the partitioning of the software between the host and the message coprocessor and the addition of a small amount of special-purpose hardware.

Index Terms—Architecture support, bus protocol, distributed systems, measurements, message-based operating systems, performance Petri nets, system architecture.

I. INTRODUCTION

WE are interested in providing hardware support to improve the performance of distributed systems. The hardware environment consists of a collection of computing nodes interconnected by a local area network (LAN). There are one or more processors and a certain amount of memory in each node. The nodes do not share memory; message exchange across the network is the only mechanism for communication between them. Many recent operating systems designs for such an environment [6], [25]–[27] place a *message-passing kernel* on each node, supporting processes and communication between them via explicit messages. This kernel supports both *local communication*—communication between processes on the same node—and *nonlocal* or *remote* communication (sometimes implemented via a distinguished network manager process).

Manuscript received April 21, 1989; revised March 17, 1990. This work was supported in part by the Defense Advanced Research Projects Agency, the National Science Foundation, and International Business Machines. An earlier version was presented at the 14th International Symposium on Computer Architecture.

U. Ramachandran is with the School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332.

M. Solomon and M. K. Vernon are with the Department of Computer Science, University of Wisconsin-Madison, Madison, WI 53706.

IEEE Log Number 9036217.

Access to system services is requested via protected *procedure calls* in a traditional system, whereas in a message-based operating system it is requested via *message passing*. While a simple procedure call costs just a few instructions, and a protected procedure call (kernel call) costs a few hundred instructions, IPC costs a few thousand instructions in several systems that we studied [20]. Since message exchange is the basic kernel mechanism in message-based operating systems, the performance of the system depends crucially on the rate of message exchange. Our measurements (see Section II and [24]) as well as the measurements of others [6], [11] indicate that for small messages (which make up the vast majority of all messages sent [6]), the limiting factor is the high processing overhead that is incurred in message passing rather than limited network bandwidth or the time to copy messages from buffer to buffer.

There are two important figures of merit in this environment: *roundtrip time*, and *message throughput*. Roundtrip time is the elapsed time seen by an application between sending a message and receiving a reply from the intended receiving process. This figure of merit affects an individual application's performance. Message throughput is a global figure of merit that determines the performance of the entire system. Informally, it is the number of messages that the system handles per unit time. We show that a modest amount of additional hardware can significantly improve message throughput and average roundtrip time in a multiprogramming environment. We also show that additional hardware support in the form of high-level bus primitives affords even greater improvement in communication subsystem performance.

A. Background and Related Work

Available hardware support [1], [14] and previous modeling studies [12], [29] address the issue of off-loading communication protocols onto front-end processors, and provide evidence that this approach can have a significant performance payoff. However, these and other previous proposals of hardware support for interprocess communication (see survey in [21]) are more limited than the study reported in this paper in several respects.

First, previous work generally assumes "communication" to be the work that is performed by the operating system to satisfy nonlocal requests. However, for message-based operating systems, measurements by us (see Section II and [24]) and others [6] show that there is a high processing overhead for local communication as well. Existing hardware support for interprocess communication takes the form of

1) operations on structured data types in the instruction set architecture of the processor (often through microcode) such

Selected Publications in Full

Synchronization with Multiprocessor Caches *

PAPER [LR90]

Joonwon Lee

Umakishore Ramachandran

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332 USA
e-mail: joon@sybil.gatech.edu, rama@gatech.edu

Abstract

Introducing private caches in bus-based shared memory multiprocessors leads to the cache consistency problem since there may be multiple copies of shared data. However, the ability to snoop on the bus coupled with the fast broadcast capability allows the design of special hardware support for synchronization. We present a new lock-based cache scheme which incorporates synchronization into the cache coherency mechanism. With this scheme high-level synchronization primitives as well as low-level ones can be implemented without excessive overhead. Cost functions for well-known synchronization methods are derived for invalidation schemes, write update schemes, and our lock-based scheme. To accurately predict the performance implications of the new scheme, a new simulation model is developed embodying a widely accepted paradigm of parallel programming. It is shown that our lock-based protocol outperforms existing cache protocols.

1 Introduction

Cache memories have been used to reduce memory access latency in uniprocessors. In bus-based shared memory multiprocessors they may additionally reduce bus contention. However, private caches introduce the cache coherence problem [8]. The shared bus enables each cache controller to monitor the bus traffic and initiate appropriate actions to keep the shared data coherent. A group of cache coherence schemes called *snooping cache protocols* use this feature [13, 16, 18, 19, 24]. They are implemented in hardware and may not be visible to the programmer.

Most snooping cache protocols (Section 2) do not take into account synchronization requests that usually precede shared data accesses. Therefore, strong coherence among multiple copies is blindly enforced resulting in possibly unnecessary bus traffic for invalidation and data transfer. Usually, access to shared data is acquired via synchronization methods such as locks, semaphores, and barriers. Thus there is additional delay in accessing the synchronization variables and then acquiring the actual data.

In this paper, we present a cache coherence protocol supporting lock primitives (Section 3). Our scheme utilizes locking information provided by the software (e.g. compiler) to distinguish between shared-locks and exclusive-locks. We construct a distributed hardware-assisted FIFO queue of processors waiting for a given lock. With this scheme, the cache mechanism emerges as a visible part of the architecture since programmers should understand it to develop efficient parallel programs.

Efficient interprocessor synchronization and mutual exclusion are imperative to assure good performance for parallel programs. Therefore, we must evaluate the synchronization efficiency of cache protocols (Section 4).

Evaluating a multiprocessor system is a challenging task. Trace driven simulation has been used for multiprocessor evaluation [1, 11, 12, 25], but tracing parallel programs has so far been restricted to a small number of processors. Furthermore, the trace is affected by the host multiprocessor's architectural characteristics such as cache protocol, synchronization primitives, and interconnection network. As Bitar [4] points out using traces generated by software makes it difficult to verify the validity of the predicted results. Therefore, we have developed a new method for the evaluation of cache protocols. Our simulation model and some results are presented in Section 5.

2 Snooping Caches

Hardware cache coherence schemes for shared-bus multiprocessors have evolved into two categories, namely, *invalidation schemes* and *write update schemes*.

In invalidation schemes [13, 16], a write to a cached line results in invalidating copies of this line present in other caches. If writes to cached data always trigger invalidations, the bus is easily saturated with even a few participating processors [27]. To reduce the invalidation rate, Goodman developed the *write once* protocol [13] in which only the first write to a cached data updates main memory, and is used as a cue by other caches for invalidating their own copies. The Berkeley protocol [16] assumes an invalidation line on the bus to explicitly invalidate peer caches. Since an invalidation is induced even with a first

*This work is supported in part by NSF grant MIP-8809268

	2.67	1.0	Snoopy
8	1.0	1.0	1.0
9	0.89	1.0	1.0
10	0.8	1.0	1.0
20	0.4	1.0	1.78
		1.0	1.6
			0.8

Table 1: Cost for cache coherency.

write to data that is not being shared, there still exist redundant invalidations. These schemes lead to repeated invalidations and frequent data transfers when data is actively shared by many processors.

In write update schemes [18, 24], writes are broadcast and caches with matching entries update their corresponding cache lines. Using a probabilistic simulation model, Archibald and Baer [2] have shown that write update schemes outperform invalidation schemes. However, using trace driven simulation, Eggers and Katz [11] have shown that neither scheme dominates entirely. The performance of both schemes is sensitive to the sharing pattern. Particularly important is the length of the uninterrupted sequence of write requests interspersed with reads to a shared cache line by one processor (referred to as a *write-run* in [11]). The length of a write-run is the number of writes in that write-run. A write-run is terminated when another processor reads or writes the same cache line. Every new write-run requires an invalidation and scheme generates frequent updates. Since the data transfer is much higher than the update of one word, invalidation schemes are inferior for this sharing pattern. For long write-runs, write update schemes perform poorly in comparison to invalidation schemes because of frequent write broadcasts. Repeated updates are clearly redundant given long write-runs. The optimal strategy would be to drop cache lines from all processors other than the writing processor. Thus the performance of both schemes critically depends on the sharing pattern.

In [15], a dynamic cache scheme called *competitive snoopy caching* is introduced. Among several variants presented in the paper, "Limited Block Snoopy Caching" is the most practical. First, the protocol begins by operating like a write update scheme. But, when the length of the write-run exceeds a threshold the cache scheme switches to the invalidation strategy. An important feature of this protocol is *read snarfing* which allows caches to grab a cache line that is being transferred on the bus. This feature allows several readers to acquire a cache line in a single bus transaction following a long write-run. This cache scheme guarantees a lower bound for any sequence of memory requests when the threshold is equal to the block size.

Let p be the block size, which is also the threshold. Let

n be the number of writes among l_i be the address, l_i be to a single shared address, l_i be a run, and \bar{l} be the average write-run, and $\bar{l} = n/k$. For simple k write-runs $\bar{l} = n/k$. Since there be a multiple of k . Since there a validation scheme costs $k \times C_{IX}$ for invalidating and transferring a sors. The subscripts of C denote the for invalidation, X for block transfer Note that C_I is usually much smaller update scheme, regardless of the write nC_W where C_W is the cost for one wo ing cache lines. Since C_{IX} is roughly p of the invalidation scheme is p/\bar{l} times update scheme. Hence when the write- and the block size is large, the invalida erates more bus traffic for cache coheren invalidation scheme outperforms the writ by the factor \bar{l}/p . Let a be the number of v lengths are larger than the threshold p . If length of such write-runs then the total nu in these write-runs is $a\bar{l}$. Since n is the to writes in the whole sequence σ and since p that $ap \leq n$. The upper bound for the tot competitive snoopy scheme is

$$nC_W + aC_{IX} = nC_W + apC_W \leq 2nC_W$$

Thus in the worst case, the competitive snoopy inferior by a constant factor, namely 2, for an request sequence. Table 1 shows the cost relati write update scheme for processing the sequence c of the schemes when the block size p is 8.

The protocols considered thus far treat cache co problem in isolation and do not consider synchron issues. Bitar and Despain [5] propose a scheme in the cache accepts lock and unlock commands from processor in addition to the traditional read and write requests. On receiving a lock request, the cache broad a write on the bus. If the write is allowed, the cache the state to locked and allows the processor to use the li Otherwise, the line is in use by another cache. So, the requesting cache stores the address of the line in a speci register called the *busy-wait* register and the cache hold ing the lock sets its cache line state to *lock-wait* indicating there is at least one waiting processor. Upon receiving an unlock request, a cache changes its cache line state to in- valid and broadcasts unlock if its state was *lock-wait*. This lock scheme combines lock-based synchronization with the lock scheme does not distinguish between read lock and write lock requests. Since reads are a large portion of shared data access, this scheme limits potential concurrency.

In [14], Goodman et al. suggest a synchronization prim- itive, QOSB, which can be used by the programmer for high level synchronization operations. They present an ef- ficient implementation of this primitive by exploiting the hardware cache consistency protocol of the multiproces- sor. The first lock requester becomes the owner of the

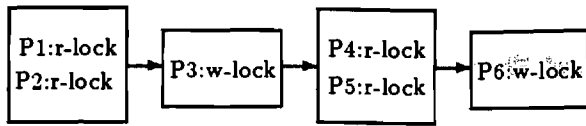


Figure 1: A waiting queue of lock requests

lock. The next requester allocates a new cache line (the *shared line*) and the address of the requesting processor is stored in main memory as the head of a queue of lock requesters. Since the data field of the shared line is invalid, this space can be used to store the address of the next requester. To know when to respond to a new requester, a tail queue pointer is also needed. Therefore, whenever a new requester appears, memory accesses are required to maintain these two queue pointers. These memory accesses can be eliminated if the primitive is implemented for a single bus multiprocessor. Like Bitar and Despain, Goodman only implements exclusive locks.

3 Lock Based Cache Protocol

At about the same time the QOSB primitive was developed, we designed a lock-based cache protocol [22] supporting exclusive and non-exclusive locks. Our lock based protocol (LBP) improves on the schemes discussed above. We chose lock primitives because of their generality. Like the queueing mechanism used in QOSB, a distributed queue is constructed using participating cache lines. In our scheme, a sharable lock is distinguished from an exclusive lock. Sharable locks enhance concurrency and are needed to efficiently implement other synchronization operations.

The processor and the cache together form a node of the shared memory multiprocessor. Each node is assigned a unique id which we refer to in this paper as *node-id*. The handshake between the cache and the bus is explained as follows. The cache entertains six requests from the processor: read, write, read-lock, write-lock, read-unlock, and write-unlock. Read and write are deemed as accesses to non-shared data and the cache processes them as would a uniprocessor cache. The granularity of a lock is a cache line. Processors must wait until the current request is satisfied before generating new requests.

Consider the sequence of lock requests, P1:read-lock, P2:read-lock, P3:write-lock, P4:read-lock, P5:read-lock, P6:write-lock, as shown in Figure 1. The first requester (P1) obtains a read-lock, and the following requester (P2) shares the lock since the lock type is *read*. P3 waits for the lock because its lock type is *write*. P4 and P5 wait after P3 to ensure fairness, even though the current lock held by P1 and P2 is sharable. A *peer-group* is a group of read-lock requesters who concurrently share a lock ({P1,P2} and {P4, P5} are peer-groups). To implement a queue, each directory entry of a cache line has a *next-node* field

States	Meaning
INVALID	The line is invalid
WO	Write lock owner.
WOT	WO at the tail.
WOV	Waiting for a write lock.
WOVT	WOV at the tail.
R	Read lock holder.
RV	Waiting for a read lock.
RO	Read lock owner.
ROT	RO at the tail.
ROV	Waiting for a read lock ownership.
ROVT	ROV at the tail
O	Unlocked, but still an owner.
OT	O at the tail.

Table 2: States of a cache line

containing the node-id of the next waiting cache if any. When a lock is released, the cache sends a *wake* signal to the next waiting cache (if any). Caches with waiting states must monitor the signals on the bus for the line address and their node-id. Note that the protocol described here assumes a single process per processor. In [22], we describe the multiprocessing case and discuss implementation issues on standard buses.

The possible states of a cache line are summarized in Table 2, where R,W are used to specify the lock type, T to signify the tail of the queue, V to indicate waiting state, and O for the ownership. State transitions are triggered by processor requests and/or bus activities. Note that the cache controllers only respond to lock and unlock requests on the bus since simple reads and writes are deemed to be for private lines. Therefore, the states in Table 1 apply only for shared lines obtained through lock requests. In the discussion to follow, we use lock and line interchangeably since lock acquisition is merged with the cache line transfer.

An owner cache has the latest copy of the line, so it provides the line to the other caches when requested. The line is written back to memory when a write-lock owner releases the lock. There is at most one owner of a lock even when the lock is shared. A lock state with a T suffix denotes that the cache is at the tail of the waiting queue and should respond to subsequent requests for that lock. Only the first requester within a peer-group can be a tail or an owner. A shared lock is released when the size of the peer-group reaches zero, so caches with read-lock ownership or awaiting ownership keep the size of the peer-group in a *count* field. The ownership persists even after the line is unlocked at the owner cache. Assigning read-lock ownership to the first requester may result in unnecessary cache entries since it is likely that the read-lock owner will be the first to release the lock. However, the alternative choice of giving ownership to the last one in a peer-group could generate more bus traffic to transfer the count variable to the new owner. The width of the count field is determined by the number of nodes in the system. Alternatively, the width may be determined by the maximum

state	next-node	count
-------	-----------	-------

Figure 2: Tag fields of a cache line

membership we would like to allow in a peer group. Each cache line has a directory entry (tag) with the fields as shown in Figure 2.

Figure 3 illustrates the state transitions. When a processor requests a read-lock, the cache broadcasts it on the bus resulting in one of following responses:

- The block came from the main memory (denoted as *hit(M)* in the state transition diagram). It means that the line is not locked by any cache. The memory system provides the line, and the cache changes the state to ROT since it is the first requester.
- The block came from another cache (denoted as *hit* in the state transition diagram). The current read-lock owner sends the cache line allowing the requester to share the lock. The receiving cache changes its state to R.
- A *wait* signal is detected on the bus. A peer cache in state ROVT sends this signal with its own node-id. Since the *wake* signal (to be discussed shortly) is addressed to the first requester in a peer-group, this node-id is necessary for the waiting nodes to receive the signal correctly. The receiving cache stores this node-id in the next-node field, and changes its state to RV.
- A *wait(T)* signal is detected on the bus. The signal comes from a cache in state WOVT or WOT, and signifies that the tail state is transferred to the requester. Therefore, the receiving cache changes its state to ROVT.

When a cache receives *read-unlock* from the processor, the state of the line is one of R, ROT, or RO. A cache line in the R state is simply changed to the state INVALID, and a *read-unlock* signal is broadcast on the bus to inform the owner to decrease the count. If the state is ROT or RO, it is changed to OT or O respectively after decrementing the count. The cache is still the owner even after its own processor releases the lock and is responsible for sending a wake signal when the count goes to zero. Even though we assume a single process per processor, a processor may request a lock after releasing a lock, i.e. it may request a lock when the state of the line is OT or O. This case is not shown in Figure 3 since it is treated as a sub-case of multiple processes per processor. Another simple solution without increasing hardware complexity is to allow the processor to be an owner again. In this case, the fairness between processors cannot be guaranteed.

In case of a write lock, it is not necessary for the owner to keep the count since only one writer is allowed at a time. If a *wait(T)* signal is detected after broadcasting a write-lock, the state is changed from INVALID to WOVT. However, it ceases to be at the tail when any subsequent

R : Bus Response
 B : Bus Activity
 P : Processor Request

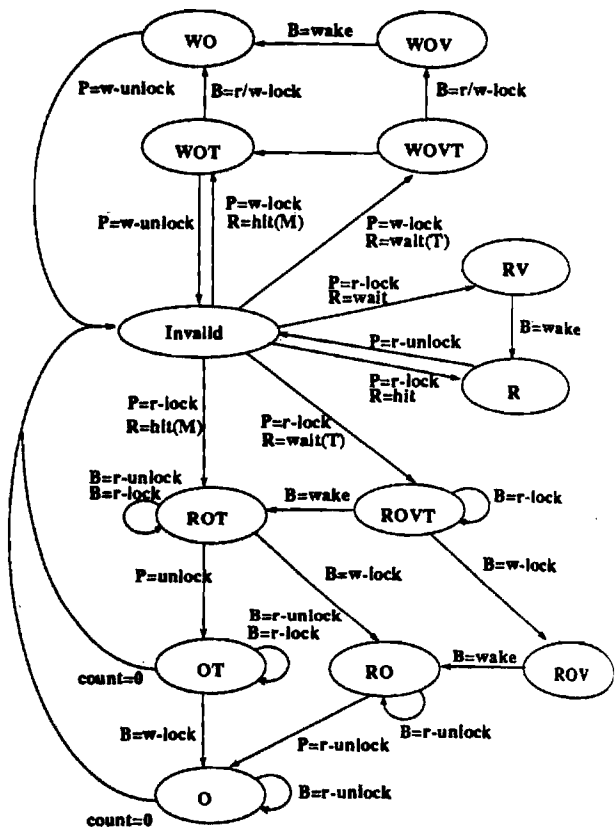


Figure 3: State Transitions

request for a lock is observed on the bus. On receiving an appropriate wake signal, the cache controller changes waiting states, WOV, WOVT, to owner states, WOT, WO respectively, and allows the processor to use the line. It is not necessary to broadcast a write-unlock. On receiving a write-unlock request from the processor, the cache changes the state of the line to INVALID, sends a wake signal enclosing the cache line to the next requester (if any) as indicated by the next-node field, and writes the line back to memory.

Bus signals include: read-lock, write-lock, read-unlock, hit, hit(M), wait, wait(T), and wake. The wait signal is sent from the tail cache to the lock requester when the lock is unavailable. The wait(T) signal additionally transfer the tail state to the requester. The wake signal is sent to notify that the lock is released to a cache whose node-id was stored in the next-node field of the tag entry for the line. Our scheme requires 13 states in the cache controller compared to 4 in most other protocols, and a larger cache tag memory to implement the distributed hardware queue.

4 Synchronization Issues

Cache line replacement becomes tricky for lock-based protocols including ours. If the line being replaced either owns a lock or is waiting for a lock then special attention is required. In the latter case, the best strategy would be to delete this node from the queue of waiting requesters for this lock. In a bus-based scheme such as ours this strategy can be fairly easily implemented. Prior to replacement, the cache controller broadcasts this event on the bus along with the address of the successor. The predecessor (cache controller) in the queue updates its cache line entry on receiving this event.

Replacing a cache line that owns a lock is a bit more complicated. The most simple and straightforward solution is to disallow replacement of a cache line that owns a lock. However, this solution may be feasible only if a fully associative or a set-associative caching strategy is in use. If a direct mapped caching scheme is used then it is necessary to modify the memory system to hold the lock status, tag and count fields of the cache line in addition to the data field. We believe this can be done with a little added complexity to the cache controller, minimal change to the memory system, and support from the compiler. The compiler can allocate additional memory space for every shared read/write data structure for holding the auxiliary information. For instance, if the data block is 4 words, one word may be reserved by the compiler for storing the auxiliary information. The memory system has a bit for every block that indicates whether the block is locked or not, which is returned with every memory request. When a cache line is replaced, the cache controller checks the status of the cache line. If it is locked, then the cache controller writes back the data field along with the auxiliary information to the memory system. When the block is reloaded from the memory, the cache controller extracts the auxiliary information returned in the data field and stores it in the cache line. When a processor makes a lock request and the block comes from the memory, the cache controller checks the lock bit. If it is set then it is an indication that some other cache currently holds the lock. Therefore, the requesting cache has to retry the lock request at a later time.

The preceding description is only conceptual. In an actual implementation, it is not necessary that the lock bit in memory be out of band data. It is perfectly reasonable, and in fact practically feasible, to keep the lock bit in band. The compiler has to preallocate this space in the cache line and the cache controllers have to agree on the location of this lock bit in the cache line. Keeping the lock bit in band allows the use of conventional RAMs and eliminates the need for any special purpose memory system design.

In the next section, we see how our lock-based protocol aids synchronization and mutual exclusion in a parallel programming environment.

Efficient synchronization is imperative for multiprocessors since parallel programs tend to generate repeated requests for mutual exclusion, barrier, and operations on shared data structures. The inefficiency caused by synchronization is twofold: wait times at synchronization points and the intrinsic overhead of the synchronization operations. Reducing waiting time is the province of the programmers. Reducing synchronization overhead is a task for the computer architect. Hardware support for synchronization comes in various forms such as a special-purpose coprocessor (e.g. Sequent SLIC [3]), a combining network [6, 10], and a special bus for interprocessor communication [26]. Recently, researchers have been interested in incorporating synchronization into snoopy cache schemes [5, 14, 22].

In the following subsections, we consider how efficiently various cache schemes (including our own) support synchronization. Three synchronization scenarios are considered. In the first (parallel lock) we assume n processors are simultaneously competing for the same lock. The second scenario (serial lock) assumes that locks are requested only serially. Finally we consider barrier notify. Similar to Goodman's QOSB primitive, a binary semaphore can be implemented very efficiently with our scheme while other schemes require mutual exclusion and queue construction.

4.1 Lock

If the machine supports an atomic `test_and_set` primitive then mutual exclusion can be implemented as in [20]. However, implementation of the `test_and_set` on traditional cache schemes can create additional penalties due to the *ping-pong* effect [9]. Since the 'set' part of the test-and-set primitive involves a write to a shared data, a spin-lock may cause each contending processor to invalidate (or update) other caches continuously. The following method avoids the ping-pong effect by busy-waiting on the cache memory without modification.

```
repeat
  while(LOAD(lock_variable) = 1) do nothing;
  /* spin without modification */
until(test_and_set(lock_variable) = 0);
```

But, it still generates considerable bus traffic when a lock is released since all the waiting processors try to modify the cache line, thus invalidating (or updating) the corresponding cache line of the other caches.

Suppose n processors are competing for a lock at the same time (labeled 'parallel lock' in Table 3). After loading the lock variable, each processor executes `test_and_set`, thus generating n invalidations and $n - 1$ block transfers if the invalidation scheme is used. When the first lock holder releases the lock, it invalidates other copies of the lock variable. This invalidation is followed by $(n - 1)$ block transfers. Now $n - 1$ processors are competing for the lock. So, total cost to service all the n processors with

Scenario	LBP	W-Invalidate	W-Update	Competitive
parallel lock	$2nC_b$	$n^2C_x + \frac{n(n+3)}{2}C_I$	$\frac{n(n+3)}{2}C_w$	$2n(C_x + C_I)$
serial lock	$2nC_b$	$n(C_x + 2C_I)$	$2nC_w$	$n(C_x + 2C_I)$
barrier notify	C_b	$C_I + (n-1)C_x$	C_w	$C_I + C_x$

Table 3: Overhead of Synchronization Primitives.

the invalidation scheme is

$$\sum_{i=1}^n (iC_x + iC_I + (i-1)C_x + C_I)$$

where the first term inside the summation is for loading the lock variable; the second and the third terms are the result of `test_and_set` executed simultaneously by i processors, and the last term is for unlocking. Simplifying the summation we get $n^2C_x + \frac{n(n+3)}{2}C_I$. With the write update scheme, invalidation is replaced with write update. The cost in this case is given by

$$\sum_{i=1}^n (iC_w + C_w)$$

where the first term is for `test_and_set` and the next one is for unlocking. Both of these schemes have a complexity of $O(n^2)$. The constant factor of the invalidation scheme is larger than the write update scheme. The total cost with our scheme is $2n$ primitive bus transactions ($2nC_b$) since each processor issues lock and unlock commands on the bus. The read snarfing feature of the competitive snoopy caching allows an efficient implementation of the parallel lock. An unlock operation results in one invalidation followed by one line transfer to all the waiting requesters. The first cache which successfully performs `test_and_set` invalidates all the other caches. This invalidation triggers another line transfer for loading the lock variable into all the remaining caches. So, the total cost for this scheme is $2n(C_x + C_I)$.

The other extreme of lock competition is when all the n processors are serialized, i.e., only one processor requests the lock at one time. With the invalidation scheme this situation costs $n(C_x + 2C_I)$ since each processor executes load, `test_and_set`, and unlock. For the write update scheme it costs $2nC_w$, and for the competitive snoopy caching it costs $n(C_x + 2C_I)$. This case is termed as 'serial lock' in Table 3.

A lock request is with or without an argument. When a process needs mutual exclusion for a specific variable, the variable is specified as an argument. Even when there is not a specific variable to be locked, an argument is used if there are more than one critical sections in an epoch of parallel computation. A lock request with an argument needs twice the accesses to cache lines, one for the synchronization variable, and one for the actual data. The QOSB primitive and our cache scheme merge the two

```

init_barrier(cnt, flag);
  cnt = n;
  flag = 0;
end init_barrier;

barrier(cnt, flag)
  lock(cnt)
  cnt = cnt - 1;
  if (cnt = 0) then
    write(flag, 1);
    unlock(cnt);
  else
    unlock(cnt);
    while (flag = 0) do;
  endif;
end barrier;

```

Figure 4: Barrier implementation with snooping cache schemes

accesses into one for a cache line, and hence would show even more impressive performance than the other schemes for such requests. In Table 3 lock requests are assumed to be without arguments.

4.2 Barrier

Barrier requires all participating processors to synchronize at a certain point (the barrier). Traditional implementation of the barrier uses a counter which may become a hot-spot¹. In [28] a software combining method is suggested to eliminate the hot-spot where processors contend to increment the counter. This method has a nice scalability property ($O(\log_2 n)$ overhead). But, considering the limit² on the scalability of bus-based systems (certainly below 100), the benefit of dispersing the hot-spot is offset by the overhead for shared memory access with the software combining approach. In [23], Sohi et al. describe a restricted form of fetch-and-add, namely, fetch-and-increment that allows all the participating pro-

¹A specific memory location is contended for by many processors simultaneously.

²The electrical characteristics of the bus as well as the traffic, limits the number of processors which can be attached to a shared bus.

5 Performance Evaluation

```
init_barrier(cnt, flag);
  cnt = n;
  write_lock(flag);
end init_barrier;

barrier(cnt, flag)
  write_lock(cnt)
  cnt = cnt - 1;
  if (cnt = 0) then
    write_unlock(flag);
    write_unlock(cnt);
  else
    write_unlock(cnt);
    read_lock(flag);
  endif;
end barrier;
```

Figure 5: Barrier implementation with LBP

processors to perform the operation in a single bus transaction. Brooks implemented the "Butterfly barrier" with busy-waiting locks [7]. This scheme requires $2 \log_2 n$ lock operations for each processor. A more realistic implementation of the barrier with bus-based snooping caches is given in Figure 4. Processors arriving before the last one keep reading their copies of *flag* variable in their respective cache memories. When the last arrival induces a write to the flag variable, other copies are invalidated or updated according to the cache coherency protocol being used. So, the cost for notifying other processors of the last arrival is $C_I + (n - 1)C_X$ for the invalidation scheme and C_W for the write update scheme. With the read snarfing feature, the competitive snoopy caching executes barrier notification with a cost of $C_I + C_X$.

Barrier implementation with our cache scheme is given in Figure 5. In our scheme the 'wake' operation for a peer-group enables the *notify* phase to execute in one bus transaction. For the *counting* phase simple mutual exclusion is used since it is unlikely that the contention for the counting variable would cause too much overhead with an efficient lock operation. However, note that the degree of contention for the counting phase depends to a large extent on the type of work that has been farmed out to the processors before the barrier. If all the processors take roughly the same amount of time to complete their respective computations then it is likely that the contention will be high. Under such circumstances the scheme proposed by Sohi et al. [23] may prove to be quite efficient.

While synchronization issues are important, it is somewhat artificial to look at them in isolation from the point of view of system performance. To know the relative performance of cache protocols, we need to study the completion times of parallel programs with a workload comprised of private and shared data accesses interspersed with synchronization requests.

Evaluating multiprocessors is difficult because of the interaction between the processors and the lack of a standard suite of benchmark applications. Recently real multiprocessor traces have become available. In [1] a tracing facility called ATUM2 is used for capturing the traces of parallel programs. However, it does not generalize for a variety of architectures due to its requirement of special hardware support and its limited scalability. Eggers and Katz [11, 12] measured multiprocessor cache protocols with traces of a set of parallel programs. Since the traces are generated on a per processor basis by software method, they are more scalable than the ATUM2. Traces gathered by software tools are prone to problems such as omission of operating system calls, oversimplification of the execution times of various instructions, and manifestation of architectural characteristics in the traces. Such problems may result in distorting the execution pattern of parallel algorithms. Note that such distortions may not necessarily lead to erroneous results in so far as measuring invalidation rate or performance related to the invalidation rate. However, as Bitar [4] points out, trace driven simulation may not be effective in capturing the high-level interactions between processors for hypothetical architectures.

We present a new simulation model for the measurement of multiprocessors. This model represents a dynamic scheduling paradigm believed to be the kernel of several parallel programs [21]. The basic granularity is a task. A large problem is divided into atomic tasks, and dependencies between tasks are checked. Tasks are inserted into a work queue of executable tasks honoring such dependencies, thus making the work queue non-FIFO in nature. Each processor takes a task from the queue and processes it. If a new task is generated as a result of the processing, it is inserted into the queue. All the processors execute the same code until the task queue is empty or a predefined finishing condition is met. If there is a need to synchronize all the processors at some point, then a barrier operation is used. Figure 6 shows the pseudo code that each processor executes.

To simulate the memory reference pattern of each processor during task execution, a probabilistic model similar to the one developed by Archibald and Baer [2] is incorporated into our model. Additional features in our model are synchronization primitives, differentiation of synchronization variables from other variables, and different evaluation metrics. Many parameters are fixed not only because their effects are well studied in [2] but also our primary concern was to measure the effect of various synchronization mechanisms on protocol performance. The values of the parameters used in the simulation are summarized in Table 4. The degree of sharing will be fairly low during the execution of a task than during queue operation. So, 0.03 and 0.5 are assumed for the degree of sharing for these two cases respectively. Real traces [11, 25] show that the read ratio ranges from 0.6 to 0.9 depending on the application of traced programs. In [2] simulation was done varying

```

finished = false;
new_task = empty;
loop
  lock(queue);
  if (new_task ≠ empty) then
    insert_queue(new_task);
  end if;
  task = delete_queue();
  unlock(queue);
  new_task = execute(task);
  if (need_synchronization) barrier();
  check_if_finished(finished);
until(finished)

```

Figure 6: Programming paradigm with dynamic scheduling

Parameters	value
ratio of shared accesses	0.03, 0.5*
number of shared blocks	32
cache hit-ratio	0.95
read ratio	0.85
main memory cycle time	4 cache cycles
block size	4 words
cache size	1024 blocks
lock ratio	50%

* 0.03: task execution, 0.5: queue access

Table 4: Summary of parameters used in simulation

the read ratio from 0.7 to 0.85. In our simulation the read ratio is set to 0.85. Shared accesses are secured by lock primitives with a probability *lock-ratio*. The nature of the lock can be read (sharable) or exclusive depending on the type of shared access. The ratio of lock requests to shared accesses is from 50% to 70% in some applications [1, 17] and below 10% in other applications [11]. In the results reported a lock-ratio of 50% is assumed³. For invalidation and write update schemes, the lock primitive is implemented as *test_test_and_set*. Another important parameter is the grain size of parallelism. The grain size is decided by the number of data memory references during the execution of a task. Cache schemes evaluated are Berkeley protocol [16] as an invalidation scheme, Dragon [18] protocol as a write update scheme, Bitar and Despain's method [5], and our lock-based protocol. The problem size is 160 tasks, and the results are the average of 5 runs. Measured metrics are completion time in number of cycles, processor utilization, and the average length of the bus queue.

Figure 7 shows the completion time of each cache scheme for the grain size 100 (fine to medium grain) with-

³Varying the lock-ratio produces only a small difference in performance because the degree of sharing is quite low during task execution.

out barrier synchronization. BD denotes the Bitar and Despain's scheme, and LBP is our scheme. The unit metric is 10,000 cycles. The Berkeley protocol shows an anomalous loss of efficiency as the number of processors n increases beyond 8. This loss of efficiency happens because the bus is already saturated (the measured average length of the bus queue supports this argument) and thus, useful work is delayed by queue access activity of each processor. This effect is the multiprocessor equivalent of thrashing induced by the greedy scheduling discipline. All the other schemes also show slowdown in speed-up with more than 8 processors. With 8 processors LBP completes in 20480 cycles while BP, the second best one next to LBP, completes in 25149 cycles. This performance gap grows as n increases. The performance gap between BP and LBP is due to increased concurrency provided by read-locks of LBP during the task execute phase of the simulation model (Figure 6).

Figure 8 depicts the performance with a large grain size of 500. The steep loss of efficiency with Berkeley protocol disappears in this case because the long processing time outweighs the overhead of queue access. Though the grain size is multiplied by 5, the completion times for all the schemes do not scale down by quite as much, which is to be expected because there is a constant overhead for accessing the queue. When n is 8 the performance gap between LBP and BP is 14303 cycles, which is more than three times the gap when the grain size is fine to medium.

The effect of barrier synchronization is shown in Figures 9 and 10. Irrespective of the protocol, the net effect of the barrier is to synchronize the queue access of all the processors thus aggravating the contention for this shared resource (see Figure 6). Hence for a given grain size, the inclusion of the barrier results in longer completion times for all the protocols (compare Figures 7 and 9, and Figures 8 and 10). With respect to Figures 8 and 10, Dragon protocol outperforms Berkeley protocol by larger gaps with barrier than when there is no barrier. The same is true in Figures 7 and 9, except for the anomalous behavior of Berkeley protocol beyond 8 processors in Figure 7. The reason for these gaps is evident from the cost functions developed in Section 4.1: Even though the barrier itself does not lead to significant performance gap, simultaneous lock requests (see 'parallel lock' in Table 3) after the barrier entails considerable expense for the Berkeley protocol as compared to the Dragon protocol, especially with large n . With $n = 8$ and fine to medium grain parallelism, Dragon is better than Berkeley by 8% when barrier is not used, and by 32% when barrier is used. The case when the grain size is large (500) shows a similar trend. LBP outperforms BD by a larger margin with barrier synchronization. When the grain size is fine to medium (100), LBP is better than BP by 4669 cycles if barrier is not required, and by 9249 cycles with barrier. The efficiency of LBP for barrier operation comes from the sharable lock that enables the notification to be done in one bus transaction.

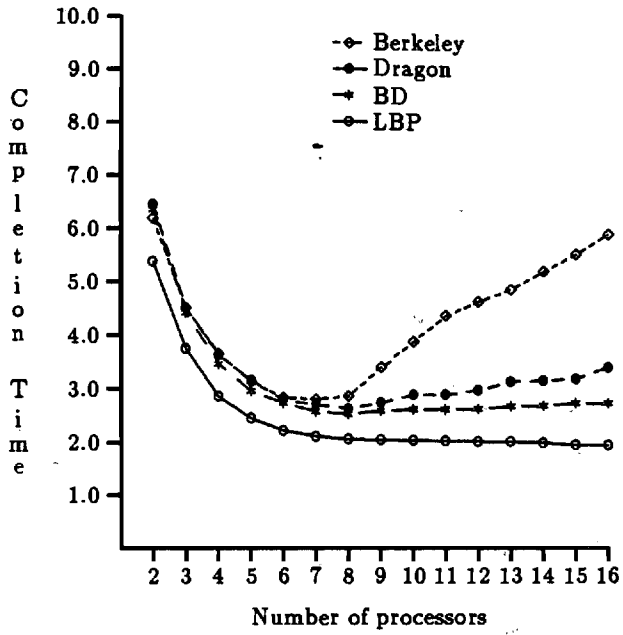


Figure 7: Performance of cache schemes (grain size = 100, without barrier)

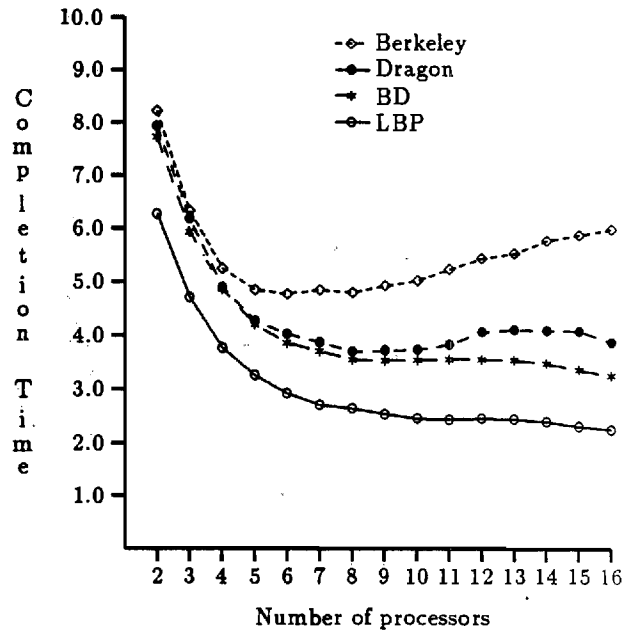


Figure 9: Performance of cache schemes (grain size = 100, with barrier)

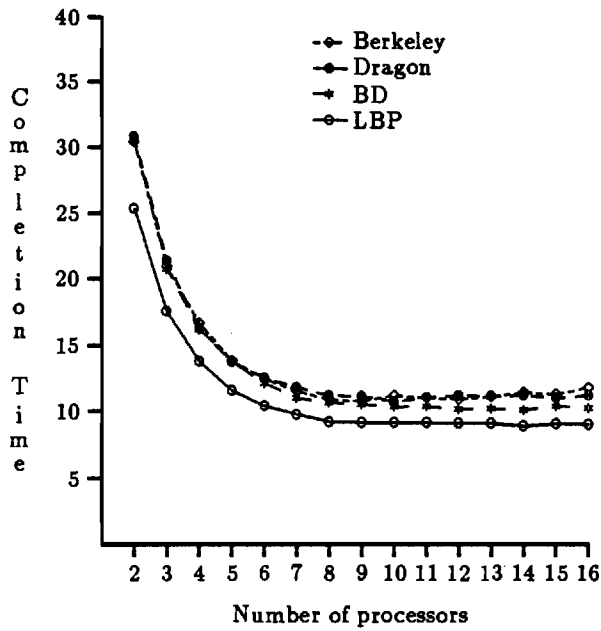


Figure 8: Performance of cache schemes (grain size = 500, without barrier)

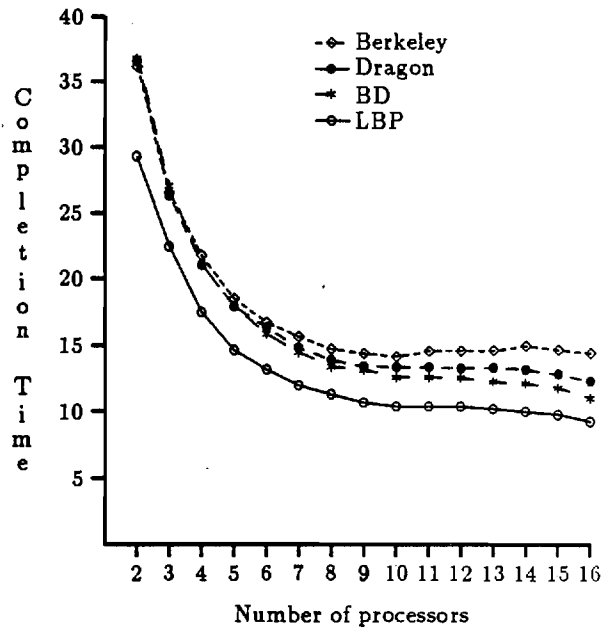


Figure 10: Performance of cache schemes (grain size = 500, with barrier)

6 Discussion and Conclusions

We presented a new lock-based scheme which incorporates cache coherency strategy with synchronization. Lock operations were used as the underlying primitives for cache coherency. As a waiting mechanism, a distributed queue was constructed in hardware using the cache line of each lock requester. Memory accesses for queue pointers were eliminated by storing the link information in the tag of the cache memory. The protocol distinguishes sharable lock from exclusive locks thus allowing increased concurrency for simultaneous readers. The invalidation, the write update, and the lock-based schemes were analyzed for synchronization operations to identify the source of inefficiency. For the evaluation of each cache scheme a new simulation model was developed which represents a widely used paradigm for parallel programming. The simulation results show that our scheme outperforms others by a considerable margin in the test cases.

There are several dimension in which our work can be extended in the future. The fetch-and-add [10] primitive is a powerful synchronization primitive and its utility has been demonstrated in combining multistage networks. It would be interesting to compare the performance and hardware complexity of the fetch-and-add primitive in a bus-based environment against our protocol. We showed earlier that read snarfing is a useful feature for synchronization. Detailed simulation study for examining the overall performance of read snarfing on snoopy cache protocols is another area of future research. VLSI implementation of cache schemes such as invalidate, update, Bitar and Despain, QOSB primitive, and ours is also being pursued to quantify the cost in terms of circuit complexity and understand the delay characteristics of the different cache protocols.

Acknowledgment

We thank Jim Goodman for his several constructive criticisms and suggestions. We thank our colleagues, H. Venkateswaran and Phil Hutto for reading earlier drafts of this paper.

References

- [1] Anant Agarwal and Anoop Gupta. Memory-reference characteristics of multiprocessor applications under MACH. In *Proceedings of ACM SIGMETRICS Conference*, pages 215-225, May 1988.
- [2] James Archibald and Jean-Loup Baer. Cache coherence protocols: evaluation using a multiprocessor model. *ACM Transactions on Computer Systems*, pages 278-298, Nov. 1986.
- [3] Bob Beck, Bob Kasten, and Shreekanth Thakkar. VLSI assist for a multiprocessor. *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 10-20, 1987.
- [4] Philip Bitar. A critique of trace-driven simulation for shared-memory multiprocessors. *ISCA '89 Workshop: Cache and Interconnect Architectures in Multiprocessors*, 1989.
- [5] Phillip Bitar and Alvin M. Despain. Multiprocessor cache synchronization: Issues, innovations, evolution. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 424-433, June 1986.
- [6] W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 processor-memory element. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 782-789, Aug. 1985.
- [7] E. D. Brooks. The Butterfly barrier. *International Journal of Parallel Programming*, pages 295-307, Aug. 1986.
- [8] Lucien M. Censier and Paul Feautrier. A new solution to coherence problem in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112-1118, Dec. 1978.
- [9] M. Dubois, C. Scheurich, and F. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, pages 9-21, Feb. 1988.
- [10] Jan Edler, Allan Gottlieb, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, Marc Snir, Patricia J. Telen, and James Wilson. Issues related to MIMD shared-memory computers: the NYU Ultracomputer approach. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 126-135, June 1985.
- [11] Susan J. Eggers and Randy H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 373-382, June 1988.
- [12] Susan J. Eggers and Randy H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 2-15, June 1989.
- [13] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124-131, June 1983.
- [14] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessor. Technical Report TR-814, Univ. of Wisconsin at Madison, Jan. 1989.
- [15] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:79-119, 1988.

- [16] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 276-283, June 1985.
- [17] Zhiyuan Li and Walid Abu-sufah. A technique for reducing synchronization overhead in large scale multiprocessor. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 284-291, 1985.
- [18] E. McCreight. *The Dragon Computer System: An early overview*. Xerox Corp., Sept. 1984.
- [19] M. Papamarcos and J. Patel. A low overhead solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348-354, June 1984.
- [20] James L. Peterson and Abraham Silberschatz. *Operating System Concepts*, pages 337-340. Addison-Wesley, 1983.
- [21] Constantine D. Polychronopoulos. *Parallel Programming and Compilers*, pages 113-158. Kluwer Academic Publishers, 1988.
- [22] Umakishore Ramachandran and Joonwon Lee. Processor initiated sharing in multiprocessor caches. Technical Report GIT-ICS-88/43, Georgia Institute of Technology, Nov. 1988.
- [23] G. S. Sohi, J. E. Smith, and J. R. Goodman. Restricted fetch-and- ϕ operations for parallel processing. In *International Conference on Supercomputing*, June 1989. Crete, Greece.
- [24] C. P. Thacker and L. C. Stewart. Firefly: A multiprocessor workstation. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164-172, Oct. 1987.
- [25] Wolf-Dietrich Weber and Anoop Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243-256, 1988.
- [26] W. A. Wulf and C. G. Bell. C.mmp - a multi-mini processor. In *Proceedings of the Fall Joint Computer Conference*, pages 765-777, Dec. 1972.
- [27] W. C. Yen, D. W. L. Yen, and King-Sun Fu. Data coherence problems in multicache system. *IEEE Transactions on Computers*, c-34, No. 1:56-65, Jan. 1985.
- [28] P. C. Yew, N. F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessor. *IEEE Transactions on Computers*, pages 388-395, April 1987.

PAPER [LR91]

Locks, Directories, and Weak Coherence - A Recipe for Scalable Shared Memory Multiprocessors *

Joonwon Lee

Umakishore Ramachandran

Technical Report GIT-CC-90/68

December 1990

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

joon@cc.gatech.edu, rama@cc.gatech.edu

Abstract

Bus based multiprocessors have the limitation that they do not scale well to large numbers of processors due to the bus becoming a bottleneck with the current bus technology. Lock-based protocols have been suggested as a possible way of mitigating this bottleneck for single bus systems with snooping ability. In this research, we are interested in extending lock-based protocols to general interconnection networks. Directory based cache coherence schemes have been proposed for such networks. We are investigating a combination of locking with directory based schemes. Further, most protocols in the literature until now, assume a strong coherence requirement. However, recent research has shown that it is possible to weaken this coherence requirement. Such an approach is expected to reduce the coherence overhead even further, making it an appealing one for building scalable systems.

*This work is supported in part by NSF grant MIP-8809268.

1 Introduction

In shared memory multiprocessors there is a need for defining a consistency model that specifies the order of execution of memory accesses from multiple processors. Lamport [Lam79] has proposed sequential consistency as the ordering constraint for correct execution of a multiprocess program: The multiprocessor execution of the program should have the same effect as a sequential execution of any interleaving of the operations of all the processes (that comprise the program). The allowed interleavings are those that preserve the program order of operations of each individual process. With the sequential consistency model, read and write operations are sufficient to implement synchronization operations correctly. However, this model is inherently inefficient since it imposes a strong ordering constraint for all memory accesses regardless of the usage of shared data. Further, each memory access has to wait until the previous memory access is completed. Thus large scale shared memory multiprocessors are expected to incur long latencies for memory accesses if this ordering constraint is imposed, leading to poor performance. Further, such long latencies seriously hamper the scalability of shared memory multiprocessors.

In parallel program design, it is not unusual to use synchronization operations to enforce a specific ordering of shared memory accesses. Based on this observation Dubois et al. [DSB86] have proposed *weak ordering* that relaxes the ordering constraint of sequential consistency by distinguishing between accesses to synchronization variables and ordinary data. Their model requires (a) that synchronization variables be strongly consistent, (b) that all global data accesses be globally notified before synchronization operations, and (c) that all global data accesses subsequent to the synchronization operation be delayed until the operation is globally performed. Thus this model requires strong consistency of global data accesses with respect to synchronization variables.

A synchronization operation usually consists of *acquire* and *release* steps, e.g. lock and unlock, P and V of semaphores, barrier-request and barrier-notify. The acquire step (such as P or lock) need not be strongly consistent with respect to global memory accesses issued before it. Likewise, global memory accesses following the release step (such as V or unlock), need not wait until the synchronization operation is globally performed. This observation enables several extensions [AH90, CLLG89] to weak ordering, and thus provides more flexibility in machine design.

Private caches significantly reduce memory latencies and network contention in shared memory

multiprocessors, provided an efficient cache coherence scheme is devised. For bus-based multiprocessors snooping cache protocols have been popular since they provide the sequential consistency model without much overhead. However, snooping caches rely on a fast broadcast capability, not available in more scalable interconnections such as a multistage interconnection network. However, private caches are indispensable for reducing memory latencies, despite the fact that scalable interconnections complicate coherence maintenance. Because of their scalability, directory-based cache coherence protocols have been proposed for large scale shared memory multiprocessors [CGB89, CFKA90]. Until recently, such protocols implement the strong sequential consistency constraint used in the snoopy cache protocols. However, given that global operations such as invalidation and updates are expensive in scalable interconnects, it is important to incorporate weak consistency models in such protocols.

In this paper, we present a directory-based caching scheme based on a consistency model in which strong consistency is enforced only for cache lines accessed by lock operations. Other accesses are deemed to be for private data. This protocol is similar to write-broadcast protocol [TS87, McC84] with the difference that updates are only sent to processors that request them. Lock requestors wait in a FCFS queue organized with pointers in each cache line and the directory¹. In the next section we consider the effect of caching schemes on consistency models. The motivation for and applications of our new caching scheme are presented in Section 3. Section 4 describes our cache protocol and its implementation issues. Finally, the performance potential of our scheme is discussed in Section 5.

2 Cache Consistency

For the purposes of this paper the multiprocessor model is a uniform shared memory one: each processor has a private cache; an interconnection network connects the processors with the memory module(s); and a cache coherence scheme assures consistency using some consistency model. The sequential consistency model for cache coherence is the most straightforward one to comprehend from the point of view of programming. This model parallels the database transaction model of single copy serializability in the way it ensures correctness of concurrent execution of operations.

¹Similar ideas have been proposed in our previous work [LR90], and IEEE SCI protocol [IEE89]

Therefore, it is easy to see that this consistency model relieves the programmer from having to worry about the order of execution of individual operation in writing parallel programs. Thus, until recently multiprocessor cache coherence schemes have used this consistency model.

The implementation cost (in time, network traffic, and circuit complexity) of a cache protocol depends on the choice of interconnection network. When the interconnection network is a single shared bus it is relatively easy to implement the sequential consistency model for cache coherence since (a) each processor has the ability to observe the (read/write) events from all the other processors by *snooping* on the bus, and (b) the bus provides a fast broadcast capability. This observation has led to a group of cache coherence protocols often referred to as *snooping cache* protocols [Goo83, KEW⁺85, PP84, KMRS88, TS87, McC84, LR90] for bus-based systems. The snooping ability of the processors allows the directory information to be distributed among the private caches.

It is fairly well-known that single bus systems are incapable of scaling to very large numbers of processors due to the bus becoming a bottleneck beyond a few processors [AB86]. Therefore, there has been a resurgence of interest in developing efficient cache coherence schemes for more scalable interconnection networks [MHW87, GW88, Wil87, BW87]. While some of these have been extensions of single bus snooping cache protocols to multiple buses, some researchers have been investigating an alternative approach, namely, *directory-based protocols*. In this approach, the directory information is centralized at the memory module(s) instead of being distributed in the private caches, thus making it is possible to implement the sequential consistency model for non bus-based networks. However, the lack of a fast broadcast capability in such networks leads to an inefficient implementation of the sequential consistency model.

All of the protocols until recently have used the sequential consistency model for assuring correctness. However, recently researchers have observed that the sequential consistency model is too strong, and have developed weaker models of consistency [SD87, AH90, LS88, HA89, ABM89]. Researchers have shown that it is possible to achieve a more efficient implementation of multiprocessor cache protocols with weaker models of consistency [ABM89, AH90, CLLG89].

3 Consistency and Synchronization

The evolution of multiprocessor cache protocols has three distinct aspects: development of different models of consistency, study of trade-offs between centralized and distributed directories, and innovation in the choice of cache primitives. We discussed the first two aspects in the previous section. In this section we discuss the motivation for developing new cache primitives. Early cache protocols assumed the traditional uniprocessor interface between the processor and the cache, namely, the cache responds to read and write requests from the processor. However, in a multiprocessor data is potentially shared between processors. To account for this, the cache protocols extend the functionality of the private caches for read/write accesses from other processors via the network, satisfying some chosen model of consistency.

However, when there is sharing there is synchronization, that usually governs such sharing. By ignoring the synchronization aspect, most of the early protocols treated the coherence problem for shared data in isolation. In reality, synchronization and data coherence are intertwined. Realizing this, researchers have proposed primitives for synchronization via the caches. [GW88, BD86, LR90].

The data accesses from a processor can be grouped into three categories: access to private data, access to synchronization variables, and access to shared data. If permission to access to shared data is acquired through some synchronization mechanism then strong coherence would be required only for the access to the synchronization variables. Synchronization operations may be used in a program for one of three purposes:

- Case 1 The operation guarantees the atomicity of a set of operations (transaction) on shared data. A lock is usually associated with the shared data for this purpose. The lock governs access to the shared data.
- Case 2 The operation provides mutual exclusion to a critical section in the program.
- Case 3 The operation signals the completion of an epoch of a computation to other processes (e.g. barrier synchronization).

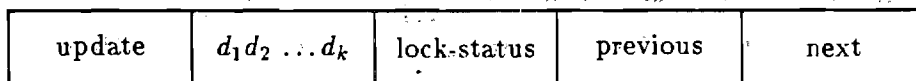
In each case the synchronization operation itself requires strong coherence enforcement. If the cache is capable of recognizing a synchronization request distinct from other requests, then it is

relatively straightforward to enforce the strong coherence requirement: stall the processor until the synchronization operation is performed globally. However, once the synchronization operation is performed globally there is latitude in how other accesses are performed. For instance in case 1, if the hardware has knowledge of the shared data associated with each lock then it can "batch" the global propagation of all the updates to shared data that happened in this transaction upon release of the lock. Similarly, in case 2 the synchronization variable governing access to the critical section needs to be strongly coherent but accesses inside the critical section need not be globally notified until exit from the critical section. Case 3 is similar to case 2 in that accesses done during the epoch of the computation need not be globally propagated until the notification phase (similar to exiting a critical section) of the barrier operation. These arguments are along the lines of those put forward by Scheurich et al. [SD87], and Adve and Hill [AH90] for justifying weak ordering.

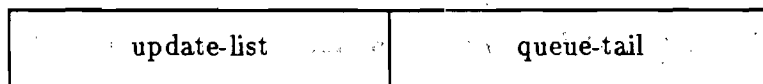
Thus if the hardware is capable of recognizing a set of synchronization operations, then subsequent data accesses can be treated as not requiring any consistency. Therefore, any cache protocol that performs consistency maintenance for *all* shared accesses is doing more work than it has to. This argument is the motivation for our cache coherence protocol that is described in the next section.

In the discussion thus far we have assumed that a process always acquires permission to access data or code through some synchronization mechanism. This need not always be true. There are applications in which it may be sufficient that a read of a shared location return the most recent value. Further it may not be necessary that the value be updated or invalidated if other processors write that location subsequently. An example is a monitoring process that wants to read the value just once. Parallel game programs are another example where it may be sufficient to perform the updates more *lazily*. Once again any such optimizations to reduce the network traffic requires that the hardware (cache controller) support additional primitives, and that the software be written to take advantage of such primitives. Needless to say such optimizations allow the scalability of such machines by reducing the network traffic.

Our proposed cache protocol (to be described next) has four features: (a) it provides hardware recognized locking primitives to handle all three cases discussed above; (b) it exploits the weak ordering principle by performing updates to shared locations more lazily; (c) it distinguishes be-



a. Tag fields of a cache line



b. Tag fields of a directory entry

Figure 1: Tag fields of a cache line and a directory entry

tween a read (that returns the value once) and a read-update (that returns the value and provides updates); and (d) it uses a centralized directories for update propagation and distributed directories in the caches for lock implementation.

4 Protocol Description

In this section, we present a directory-based cache protocol based on the consistency requirements discussed in section 3. Our scheme assumes cooperation with the software in generating appropriate requests for the desired consistency.

The cache entertains seven commands from the processor: read, read-update, write, read-lock, write-lock, unlock, and flush. Read and write are regarded as requests for private data requiring no consistency maintenance. Read-update is similar to read except that it requests updates for the cache line. This is a dual to the write-update schemes [TS87, McC84] in that the updates are receiver initiated as opposed to sender initiated. Read-lock, write-lock, and unlock requests combine processor synchronization with data coherence. Read-lock provides the requesting processor with a non-exclusive copy of the cache line that is guaranteed by the protocol not to change. Similarly, an exclusive copy of the cache line is provided to the write-lock requester. The processors have to explicitly perform unlock to release the cache lines acquired under a lock. The flush operation purges a cache line updating memory words that are modified in the cache line. The main memory, in turn, sends updates of this line to other processors if need be.

Figure 1-a illustrates tag fields of a cache line. The *update* bit of a cache line indicates whether updates have been requested for this line. The next k bits denote the modified word(s) of the cache line respectively, where k is the number of words per cache line. Only the modified words are written back to memory when the cache line is replaced. The *lock-status* denotes if the cache line is locked, and if so, the kind (read or write) of lock. The next two fields are pointers which are used to construct a doubly-linked list for processors waiting for the same lock. If the lock-status is not locked and the *previous* pointer is not null, this cache line is waiting for a lock. If the *next* pointer is null, this processor is at the *tail* of the queue.

Figure 1-b shows tag fields of a directory entry. The first field, *update-list*, consists of n bits, one for each of the n processors in the system. Each processor has a designated bit in this field which is set when it makes a read-update request, and reset when the cache line is replaced by the processor. Upon write backs to main memory, this field is checked and the updated cache line is sent to processor i if the i -th bit of the update-list is set. The *queue-tail* field is a pointer to a processor which was the last lock requester for the memory block. If queue-tail is *null* this memory block is currently unlocked.

For a read request, if the requested data is in the cache, the cache provides the data regardless of the state of the cache line. If the data is not in the cache, the request is forwarded to the main memory. Regardless of the state of the corresponding directory entry, the main memory sends the data to the requesting cache. A read-update request can be serviced by the cache if the update bit of the cache line is set. Otherwise, this request is forwarded to the main memory. The main memory provides the requested data and sets the appropriate bit in the update-list. Once this bit is set, the processor will be supplied with the new data whenever the memory block is written back by another processor. The write backs occur when a cache line is replaced, unlocked, or flushed.

Upon a lock request it is sent to the main memory and the directory entry is investigated. If the memory block is not locked, the address of the requester, say P_i , is stored in the queue-tail field of the directory entry. The memory block is sent to the requester and the cache sets the lock status of the cache line according to the type of the lock. If the memory block is locked, the lock request is forwarded to the processor addressed by the current queue-tail field and P_i becomes the new queue-tail field of the directory entry. The next pointer of the current queue-tail processor is now

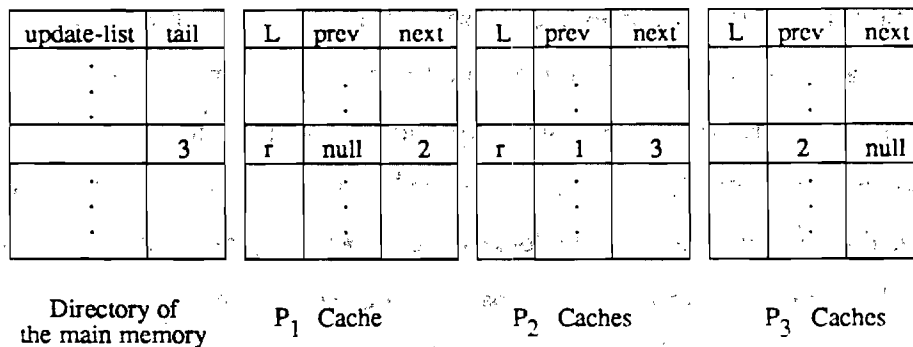


Figure 2: A waiting queue: doubly linked list

made to point to P_i . P_i 's lock request is granted if the cache line of the tail processor is currently holding a read-lock, and P_i 's request is also a read-lock. Figure 2 illustrates a queue for a series of lock requests, P_1 :read-lock, P_2 :read-lock, P_3 :write-lock. P_1 and P_2 currently hold read-locks while P_3 is waiting for a write-lock. For more details on this lock-based protocol the reader is referred to a companion paper [LR90].

Upon an unlock request, the cache controller releases the lock to the next waiting processor (if any), and writes the cache line to the main memory (if necessary). When a write-lock is released there could be more than one processor waiting for a read-lock. The lock release notification goes down the linked list until it meets a write-lock requester (or end of the list), and thus, allows granting of multiple read-locks. When a processor unlocks a read-lock and the processor is not the sole lock owner, the list is fixed up similar to deleting a node from a doubly-linked list. When a lock is released by a tail processor, the main memory marks the tail pointer null. Note that the unlocking processor is allowed to continue its computation immediately, and does not have to wait for the unlock operation to be performed globally.

Replacing a cache line that is a part of the list of lock requesters is a bit more complicated. The most simple and straightforward solution is to disallow replacement of such a cache line. However, this solution may be feasible only if a fully associative or a set-associative caching strategy is in use. If a direct mapped caching scheme is in use it is necessary to modify the tag handling mechanism of cache controllers. When a locked cache line is replaced, the cache controller preserves the lock

status and the two pointer fields in the tag memory of the cache line, i.e., only the data part and address tag part of the line is evicted. We assume that the compiler allocates at most one lock variable to a cache line. Since the new line to occupy this evicted cache line is for ordinary data, it does not interfere with the list structure for the lock (guaranteed by the compiler). Cache controllers do not perform address tag matching for lock update operations to allow the operations to be performed correctly even after the locked cache line is replaced. A subsequent access (read, write or unlock) to the lock variable will reload the replaced cache line.

However, restricting one locking process per processor at a time may be too restrictive. Since a processor holds (or waits for) only a small number of locks at a time, a small separate fully-associative cache for lock variables can be an efficient method to eliminate the restriction. This cache has tag fields as shown in Figure 1. Other ordinary shared or private data are stored in another cache which does not have tag fields for locks such as lock-status, previous, and next.

Another hardware requirement for our protocol is a write buffer. The flush operation is supposed to write back all the updated cache lines. However, detecting updated cache lines cannot be done by a compiler statically since shared data may be referenced through pointers. Searching all the cache lines by hardware is very inefficient. So, our protocol needs a write buffer which holds updated data until they are globally performed.

5 Performance Analysis

In this section, we present a simple performance analysis of our cache protocol. We compare ours with a write-back invalidation full-map directory (WBI) protocol that uses the sequential consistency model. First, we analyze the performance of each scheme for lock and ordinary memory accesses. Based on this analysis, cache schemes are compared for common structures of parallel programs such as task queue model, and static scheduling model.

5.1 Lock Operations

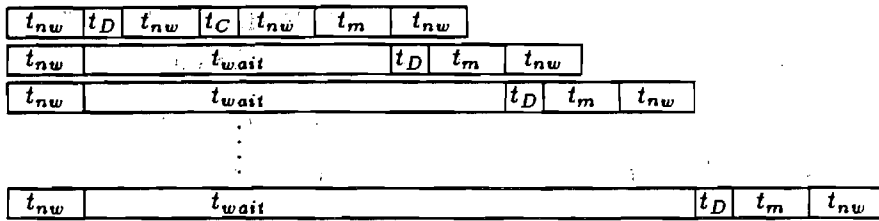
First, we derive expressions for time and network message complexity for acquiring a mutually exclusive lock to enter a critical section for the WBI protocol. Recall that with the sequential consistency model, read and write operations are sufficient to implement synchronization operations

correctly. An implementation of synchronization operations can be more efficient with an atomic test&set primitive which most architectures provide. However, implementation of the test&set on traditional cache schemes can create additional penalties due to the *ping-pong* effect [DSB88]. Since the 'set' part of the test&set primitive involves a write to a piece of shared data, a spin-lock may cause each contending processor to invalidate (or update) other caches continually with the sequential consistency model. The following test-test&set method avoids the ping-pong effect by busy-waiting on the cache memory without modification.

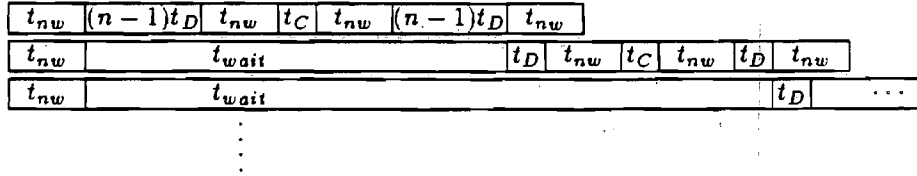
```
repeat
    while(LOAD(lock_variable) = 1) do nothing;
    /* spin without modification */
until(test&set(lock_variable) = 0);
```

But, this method still generates considerable network traffic when a lock is released since all the waiting processors try to modify the cache line, thus invalidating (or updating) the corresponding cache line of the other caches.

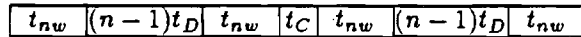
Figure 3 shows a timing diagram for the case when n processors execute test-test&set at the same time. The lock variable is assumed to be initially in the dirty state in a peer cache. t_{nw} denotes the message transit time on the network. For simplicity of analysis, we assume the same time for all types of network messages: request to the main memory, query from the main memory to a cache, acknowledgment, and block transfer. Let t_m be the memory access time; t_D and t_C denote directory checking time of the main memory and the cache memory, respectively. In this analysis, we assume requests to memory are queued by order of arrival if the memory is busy. Figure 3-a shows the parallel execution of load instruction of test-test&set by n processors. n read requests are issued at the same time (t_{nw}). For the first request (top time line in Figure 3-a), the main memory performs a directory checking (t_D) and sends a request for the block to the cache which has the dirty copy (t_{nw}). After checking the cache directory (t_C), the cache controller sends the block to the main memory (t_{nw}). Once the requested data arrives it is written to the main memory (t_m), and sent to the requester (t_{nw}). After these steps the main memory has a valid copy. So, subsequent read requests require just directory checking (t_D), wait for memory (t_{wait}), memory read (t_m), and block transfer (t_{nw}). Note that memory operations cannot be overlapped.



a. Time lines for the load part of test-test&set



b. Time lines for the test&set part of test-test&set



c. reset

Figure 3: Timing diagram for parallel exclusive locks for WBI protocol

Assuming that the lock-variable is 0, all the processors are going to execute test&set operations next. But these test&set requests can be serviced only after the last load request finishes. These requests can be serviced by the memory only after the last t_m request of Figure 3-a finishes. Referring to the top line in Figure 3-b, the first test&set operation (t_{nw}) invalidates valid copies which were acquired by the above load operations by $n - 1$ processors. Sending $n - 1$ invalidation messages is a sequential operation for the main memory $((n - 1)t_D + t_{nw})$. After invalidating local copies (t_C), all these $n - 1$ cache controllers send acknowledgments to the main memory. Receiving these acknowledgments is also a sequential operation $((n - 1)t_D)$. After that, memory acknowledges the test&set of the first processor (t_{nw}), allowing it to complete successfully. For the next test&set operation (second time line in Figure 3-b) the main memory requests the block from the cache that has a dirty copy after checking its directory ($t_D + t_{nw}$). The block is transferred to the main memory from the cache ($t_C + t_{nw}$), and then to the test&set requester ($t_D + t_{nw}$).

Among the n processors executing the test&set operation, only one completes it successfully. The remaining $n - 1$ processors re-execute the load operation and busy-wait on cached copies of the lock variable until the lock variable is reset by the lock owner. Since all the waiting processors have valid copies (in their caches), the reset operation invalidates $n - 1$ copies. Now, $n - 1$ processors compete for the lock simultaneously with one dirty copy in the resetting processor's cache, which is the initial condition that was assumed when n processors competed for the lock. This parallel lock contention is repeated n times, with one less contender in every round, until all the requesters get serviced.

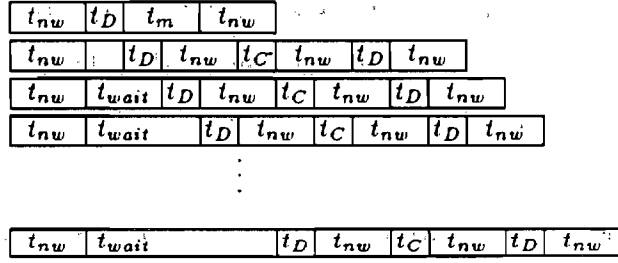
Since the performance of large-scale multiprocessors depends on the amount of traffic on the interconnection network, the number of messages generated by a cache protocol can be used as a metric for evaluating the performance of cache protocols. Let $f_m(n)$ be the number of messages generated for granting a lock to one processor when n processors are competing simultaneously. Then $f_m(n)$ is computed as the sum of (refer to Figure 3): $4 + 2(n - 1)$ for load operations, $2 + 2(n - 1)$ for the first test&set, $4(n - 1)$ for the next $n - 1$ test&sets, $4 + 2(n - 2)$ for $n - 1$ re-loads, and $2 + 2(n - 1)$ for reset. Therefore, $f_m(n) = 12n - 2$, and the total number of messages for servicing n lock requests is

$$\sum_{i=1}^n f_m(i) = 6n^2 + 4n$$

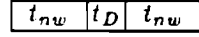
For measuring the execution time for the n parallel lock requests, we made the following assumptions: a) Lock contention of other processors do not delay the memory accesses of a processor holding the lock. b) The sojourn time inside the critical section is long enough so that the reset operation comes after all the test&sets and re-loads are completed. c) The network transit time is a constant independent of switch contentions. From Figure 3-a, load operations take $3t_{nw} + t_C + t_D + t_m + (n - 1)(t_D + t_m)$ until the last memory operation completes. The first test&set takes $2(n - 1)t_D + 3t_{nw} + t_C$ before successfully entering the critical section. Let t_{cs} be the processing time in the critical section, and t_1 be the time at which the first lock holder completes unlock operation. Then t_1 can be expressed as a function of n ,

$$t_1 = f_1(n) = 3t_{nw} + t_C + t_D + t_m + (n - 1)(t_D + t_m) + \quad (1)$$

$$2(n - 1)t_D + 3t_{nw} + t_C + \quad (2)$$



a. Time lines for n parallel locks



b. Time line for unlock

Figure 4: Timing diagram for parallel exclusive locks for the proposed lock-based scheme

$$t_{cs} + \tag{3}$$

$$2(n-1)t_D + 4t_{nw} + t_C \tag{4}$$

$$= t_{cs} + 10t_{nw} + 3t_C + nt_m + (5n-4)t_D \tag{5}$$

where the first line is for n loads, the second line is for test&set, the third line is for a critical section, and the fourth line is for unlock. Since $t_i = t_{i-1} + f_i(n-i+1)$ for $2 \leq i \leq n$, the completion time for n processors executing the critical section is

$$\begin{aligned}
t_n &= \sum_{i=1}^n f_i(i) \\
&= nt_{cs} + 10nt_{nw} + 3nt_C + \frac{n(n+1)}{2}t_m + \frac{(5n-4)(5n-3)}{2}t_D
\end{aligned}$$

For our scheme, the timing diagram for acquiring a mutually exclusive lock (write-lock) is shown in Figure 4. The first lock operation (top time line of Figure 4-a) requires one network message for lock request (t_{nw}), a directory check at the memory (t_D), memory read (t_m), and a network message for granting the lock (t_{nw}). For each subsequent request, the memory does a directory check (t_D), sends a message to the current tail (t_{nw}), which updates its directory (t_C), sends an acknowledgment (t_{nw}), which is received by the memory (t_D), and results in a message from the memory to the new tail (t_{nw}). For n lock operations, $2+4(n-1)$ messages are issued. Each unlock

operation generates 2 messages (See Figure 4, one for unlock, and one for waking up the next requester). So, for servicing n lock requests, total number of messages generated is

$$\begin{aligned} & 2 + 4(n - 1) + 2n - 1 \\ & = 6n - 3 \end{aligned}$$

And, for the completion time,

$$\begin{aligned} t_1 &= 2t_{nw} + t_D + t_m + t_{cs} + 2t_{nw} + t_D \\ t_i &= t_{i-1} + t_{cs} + 2t_{nw} + t_D \quad (2 \leq i \leq n) \end{aligned}$$

Therefore,

$$\begin{aligned} t_n &= t_1 + (n - 1)(t_{cs} + 2t_{nw} + t_D) - t_{nw} \\ &= nt_{cs} + (2n + 1)t_{nw} + (n + 1)t_D + t_m \end{aligned}$$

The subtraction of t_{nw} in the above equation accounts for the last unlock operation in which the main memory does not send a wake-up message.

The other extreme of lock competition to the parallel lock is when all the n processors are serialized, i.e., only one processor requests the lock at a time. For the WBI protocol, each lock requester generates 4 messages for load, 4 messages for test&set, and 0 message for reset since the cache has a dirty copy of the lock variable. Thus, n processors generate $8n$ messages. The time spent by each processor is $4t_{nw} + t_D + t_C + t_m$ for loading, $4t_{nw} + t_C + 2t_D$ for test&set, and t_{cs} for the critical section. Since lock requests are occurring serially, the time for executing the critical section with the WBI protocol for each processor is $8t_{nw} + t_{cs} + 2t_C + 3t_D$. With our scheme, the time for each processor to execute the critical section serially is $3t_{nw} + t_D + t_{cs}$; with 2 messages for acquiring the lock and one message for releasing the lock.

5.2 Barrier Synchronization

Barrier requires all participating processors to synchronize at a certain point (the barrier). Since the barrier counter can be a hot-spot several approaches [YTL87, Bro86] have been proposed for distributing the contention. However, we assume a traditional implementation using a counter with lock operations on the counter for the performance analysis. The cost for a barrier-request is the cost

for lock request plus the cost for accessing the counter. So, the performance of the barrier depends on the arrival pattern. For the WBI protocol, the barrier-notify step causes $n - 1$ simultaneous read requests. If we assume serial arrivals for a barrier-request, each processor generates 8 messages for a serial lock, 8 messages for reading and writing the counter variable, 2 messages for reading a flag variable. The time taken for a barrier-request is $8t_{nw} + 3t_D + 2t_C$ for a lock, $8t_{nw} + 3t_D + 2t_C$ for incrementing the counter, and $2t_{nw} + t_D$ for reading the flag variable, where the time for incrementing the counter and the time for checking conditions are not included. So, each barrier request takes $18t_{nw} + 7t_D + 5t_C$ time, and generates 18 messages.

The barrier-notify is a write to the flag variable on which all the waiting processors busy-wait. This write causes invalidations of all the $n - 1$ copies. So, messages generated by a barrier-notify are 1 write request, $n - 1$ invalidation signals to caches, $n - 1$ acknowledgments, 1 acknowledgment to the writer, $n - 1$ re-load requests, and $n - 1$ block transfers. So, a barrier-notify generates a total of $5n - 3$ messages. The time spent for a barrier-notify is t_{nw} for a write request, $(n - 1)t_D + t_{nw}$ for sending $n - 1$ invalidations, t_C for invalidation at the caches, $(n - 1)t_D + t_{nw}$ for receiving $n - 1$ acknowledgments, and t_{nw} for sending an acknowledgment to the barrier-notifier, resulting in a total time of $4t_{nw} + 2(n - 1)t_D + t_C$.

With our scheme, the barrier can be implemented by lock and read-update primitives. The counter is secured by lock operations and the busy-waiting is implemented by the read-update primitive. Processors arriving at the barrier increment the counter. If it is less than n , the processor issues a read-update for the flag variable. The processor that arrives last at the barrier sets the flag variable and notifies this update using the flush primitive. A barrier-request generates one message for write-lock and one for unlock. Both write-lock and unlock each take $t_{nw} + t_m$ time. When the flag is written and flushed by a processor, $n - 1$ processors waiting for updates will receive the update. So, it generates 1 flush request and $n - 1$ word transfers, and takes $t_{nw} + (n - 1)t_D + t_{nw}$ time.

Summary of the costs for each synchronization scenario with the WBI protocol and ours is presented in Table 1 and 2.

synchronization operation	WBI protocol	Our Scheme
parallel lock	$6n^2 + 4n$	$6n - 3$
serial lock	8	3
barrier request	18	2
barrier notify	$5n - 3$	n

Table 1: Number of messages generated by synchronization operations.

synchronization operation	WBI protocol	Our Scheme
parallel lock	$nt_{cs} + 10nt_{nw} + n(n+1)/2t_m + 5n(5n-1)/2t_D$	$nt_{cs} + (2n+1)t_{nw} + (n+1)t_D + t_m$
serial lock	$8t_{nw} + 5t_D + t_m + t_{cs}$	$3t_{nw} + t_D + t_{cs}$
barrier request	$18t_{nw} + 12t_D$	$2(t_{nw} + t_m)$
barrier notify	$4t_{nw} + (2n-1)t_D$	$2t_{nw} + (n-1)t_D$

Table 2: Time taken by synchronization operations. t_C is replaced by t_D for simplicity. Costs for serial lock and barrier request are for one processor.

5.3 Shared Variables

While the performance of the invalidation-based directory scheme is sensitive to the sharing pattern of memory accesses, our scheme is independent of the sharing pattern since strong coherency is enforced only for lock requests. In this subsection, we analyze the time taken for accessing private/shared data during an epoch of computation for the WBI protocol. The analysis is based on two basic assumptions: the mean time between shared accesses is exponentially distributed, and shared accesses are uniformly distributed over k shared variables. Following are the parameters used:

N : total number of memory accesses

k : the number of shared variables in words

h : cache-hit ratio

sh : ratio of shared accesses

r : ratio of read accesses

w : ratio of write accesses, i.e., $1 - r$
 t : mean time between memory requests

The total time for accessing data can be divided into two components: private accesses and shared accesses. For both types of accesses, we assume the same hit-ratio. However, the cache-hit ratio may be decreased for shared lines by interferences from other processors such as invalidations and external reads of a dirty cache line. So, we develop the following equations;

$$\begin{aligned}
 total - time &= N((1 - sh)t_{private} + sh * t_{shared}) \\
 t_{private} &= h * t_C + (1 - h)2t_{nw} \\
 t_{shared} &= r * t_{shared-read} + w * t_{shared-write} \\
 t_{shared-read} &= (h * P_{invalid} + 1 - h)(t_{read-miss}) + h(1 - P_{invalid})t_C \\
 t_{shared-write} &= (h * P_{not-dirty} + 1 - h)(t_{write-miss}) \\
 &\quad + h(1 - P_{not-dirty})t_C
 \end{aligned}$$

where t_X is a time needed for an X -type memory access. $P_{invalid}$ is the probability that a shared variable in the cache of a given processor has been invalidated since it was last accessed by the same processor. This invalidation(s) results in changing the state to invalid, which would have been valid otherwise. Likewise, $P_{not-dirty}$ is the probability that there has been an external access(s) to a shared variable after the last write to the variable by a given processor. This external access(es) results in changing the dirty state to valid or invalid, thus causing a write miss.

The time interval between two shared accesses from a given processor is t/sh and the probability that another processor issues a shared write during the time interval is $1 - e^{-\lambda t/sh}$ where λ is the mean arrival rate of shared writes from a given processor, i.e., $sh * w/t$. Since there are k shared variables which can be accessed by any processor, the probability that the write is to the same shared variable is $1/k$. Since the invalidation can be from any of $n - 1$ processors,

$$P_{invalid} = 1 - (1 - \frac{1}{k}(1 - e^{-w}))^{n-1}$$

Similarly, the time interval for two consecutive writes from a processor is $t/(sh + w)$ and the mean arrival rate of shared accesses is sh/t . Therefore,

$$P_{not-dirty} = 1 - \left(1 - \frac{1}{k}(1 - e^{-1/w})\right)^{n-1}$$

If false sharing is considered both $P_{invalid}$ and $P_{not-dirty}$ would be higher.

The time for servicing a read-miss, $t_{read-miss}$, is $2t_{nw} + t_D$ if the main memory has a valid block, and $4t_{nw} + 2t_D + t_C$ otherwise. The main memory has a valid copy if the last global operation on that block was a read. So,

$$t_{read-miss} = r(2t_{nw} + t_D) + w(4t_{nw} + 2t_D + t_C)$$

On the other hand, $t_{write-miss}$ is $4t_{nw} + 2t_D + t_C$ if the block is in dirty state in another cache, and $2t_{nw} + t_D + v(2t_{nw} + 2t_D + t_C)$ if there exist v valid copies. So,

$$t_{write-miss} = r(2t_{nw} + t_D + v(2t_{nw} + 2t_D + t_C)) + w(4t_{nw} + 2t_D + t_C)$$

Let m_X be a number of messages generated by a X -type memory access. Then we can get the following equations by the same way as for the timing analysis.

$$total - messages = N((1 - sh)m_{private} + sh * m_{shared})$$

$$m_{private} = 2(1 - h)$$

$$m_{shared} = r * m_{shared-read} + w * m_{shared-write}$$

$$m_{shared-read} = (hP_{invalid} + 1 - h)m_{read-miss}$$

$$m_{shared-write} = (hP_{not-dirty} + 1 - h)m_{write-miss}$$

$$m_{read-miss} = 2r + 4w$$

$$m_{write-miss} = r(2v + 2) + 4w$$

In the following subsections, we apply the cost functions developed thus far to several widely used structures of parallel programs.

Parameters	value
h	0.95
r	0.70
t_{nw}	$\log_2 N$
t_D, t_C	1
t_m	4 (=B)
v	2

Table 3: Fixed parameters

5.4 Performance for Different Program Structures

In this subsection, we analyze the performance of cache schemes for programming paradigms which have been widely accepted in the parallel programming community. The first model represents a dynamic scheduling paradigm believed to be the kernel of several parallel programs [PolSS]. The basic granularity is a task. A large problem is divided into atomic tasks, and dependencies between tasks are checked. Tasks are inserted into a work queue of executable tasks honoring such dependencies. If a new task is generated as a result of the processing, it is inserted into the queue. All the processors execute the same code until the task queue is empty or a predefined finishing condition is met. A barrier is used to synchronize all the processors after executing a task. Correct queue operations require each queue access be atomic, and thus lock operations are needed for accessing the queue. Since all the processors are synchronized by the barrier, these lock requests are generated at the same time. The second model is the same as the first one except that the barrier synchronization is not used. The third model represents a static scheduling paradigm. In this model, the computation consists of several phases and each phase depends on the result of previous phase. Therefore, a barrier is used between phases. Table 3 shows values for fixed parameters.

Figure 5-8 show the effect of sh , k , N , and n on the completion time and the number of network messages for the WBI protocol and our scheme. In the legends, QB denotes the task queue with a barrier model, Q is the task queue without a barrier model, and S is the static scheduling case. Figure 5 shows that the completion times of both schemes are not affected much by sh , the degree of sharing. The WBI protocol is inferior to our scheme in all the tested cases. The completion time

for the WBI protocol increases as sh increases because it takes longer time for accessing shared variables than accessing private variables. But our scheme is not affected by sh since coherency maintenance is not necessary for shared variables. The huge performance gap between two cache schemes for the task queue with a barrier model comes from the overhead of parallel locks. The task queue alone (Q) or the barrier alone (S) does not hamper the performance that much.

Figure 6 shows the effect of different k values. As k , the number of shared variables, is decreases, the contention for share variables increase. The WBI protocol shows a slight decrease in the completion time and the number of messages generated as the value of k increase. For a fixed set of parameter values ($sh = 0.3$, $k = 30$, $N = 300$, and $n = 32$), time for accessing shared variables is 1315 and the total time for memory accesses is 1619.62 while the total execution time is 3294 for Q model. That shows almost half the execution time is spent for synchronization activities. Therefore, even the memory access time is affected considerably by k , the total execution time is not. The execution time of QB model is worse: more than 80% of the execution time is spent for synchronization.

The effect of the size of granularity in Figure 7 shows that the time is increased less than linearly and the number of messages increases slightly. That confirms the fact that the execution time is governed by synchronization overhead not by ordinary memory accesses. Otherwise the time should have increased linearly. The less increase in the number of messages can be explained by the fact that the ordinary memory accesses generate only a small portion of (usually, 3% for QB, 40% for Q and S) the total messages generated.

The question of scalability is answered in Figure 8. As the number of processors increases, the QB model shows sharp increases in both the completion time and the number of messages. That is because the QB model incurs $O(n^2)$ overhead for both the metrics (see Table 1 and 2. For our scheme the QB model generates $O(n^2)$ messages and $O(n)$ takes completion time. That explains the steep increase in the number of messages for our scheme.

6 Concluding Remarks

For multiprocessors to be scalable, they should be able to tolerate large memory latencies as well as hop-spot contention. This paper shows that the most scalability issues can be handled by coherent

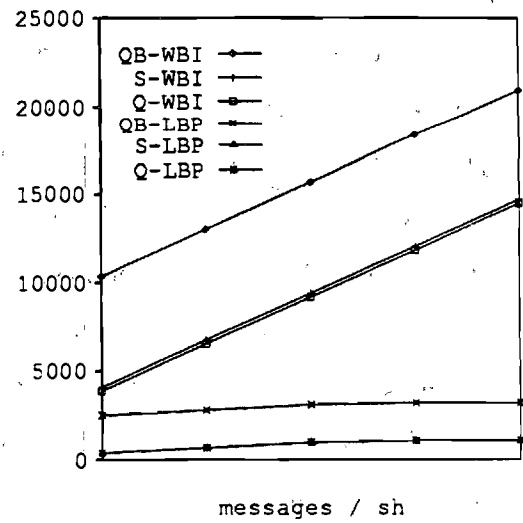
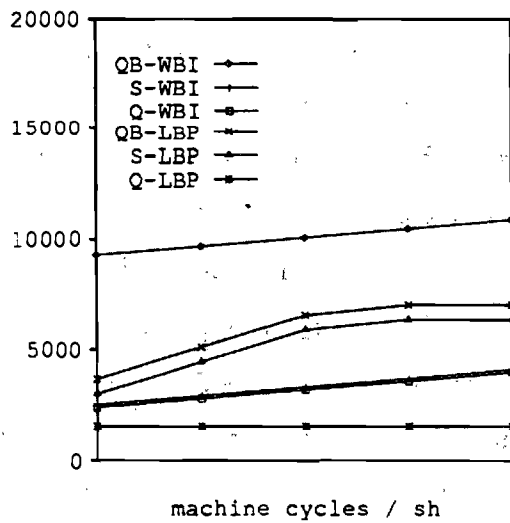


Figure 5: Effect of sh

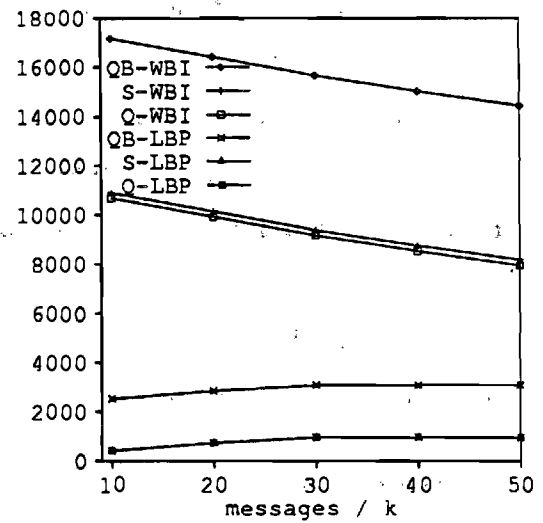
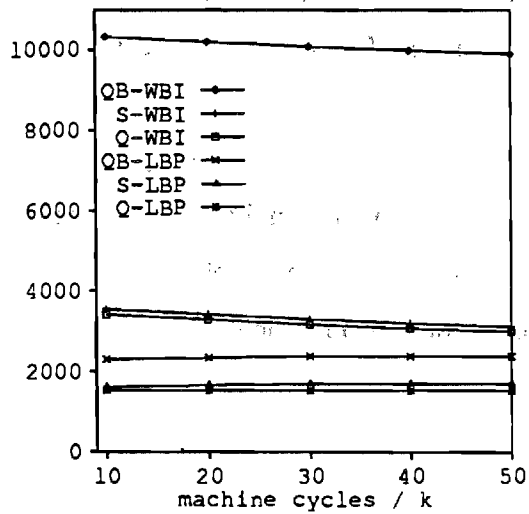


Figure 6: Effect of k

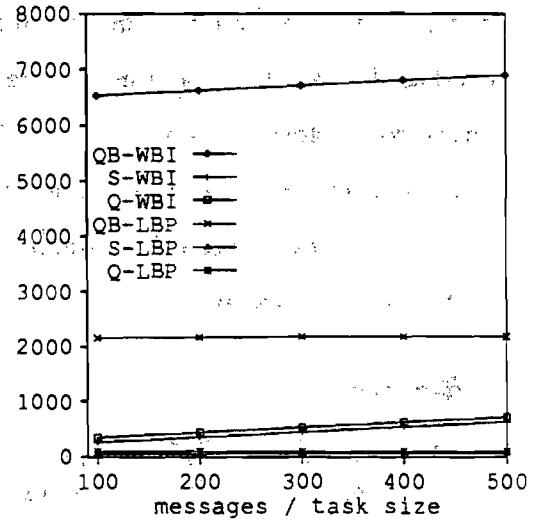
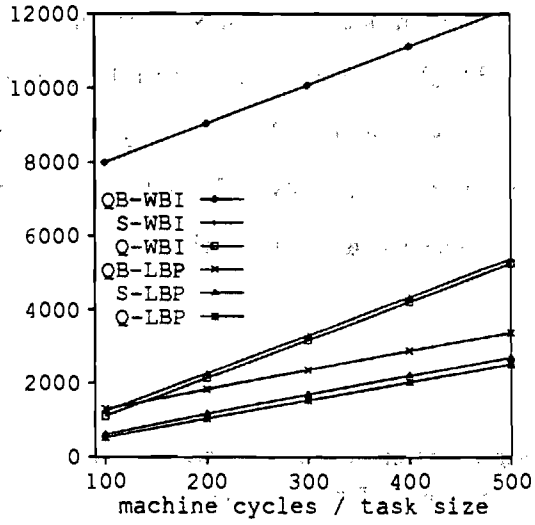


Figure 7: Effect of N

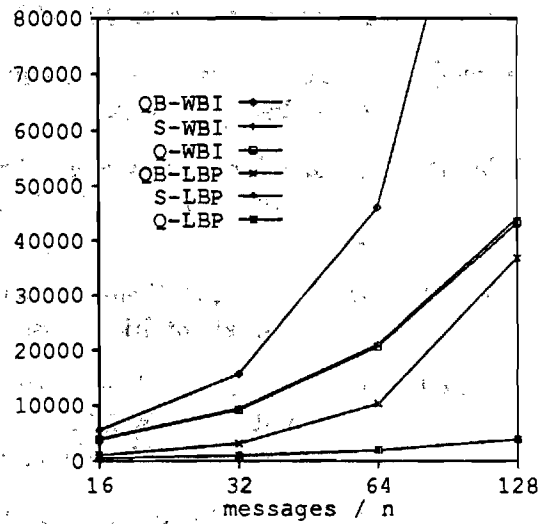
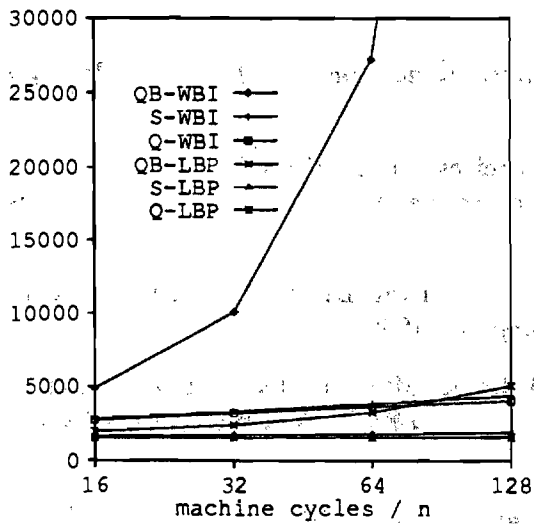


Figure 8: Effect of n

private caches. The weak coherence model reduces global accesses and makes fast the execution of critical sections. And, thus the overall memory latencies are reduced. The memory latencies are enlarged if the load on the interconnection network becomes large. Reader-initiated coherence minimizes the transactions required to maintain the cache consistency. Thus, coupled with the cache-based lock scheme, the new cache scheme will enable the system to scale to a large number of processors. Most of the new cache primitives of our cache scheme will be used by the compiler or the programmer. Thus, the future research includes the program analysis for detecting the need for our primitives. Trace-driven simulation is also being performed to verify the analytical model used for performance evaluation.

References

- [AB86] J. Archibald and J. Baer. Cache coherence protocols: evaluation using a multiprocessor model. *ACM Transactions on Computer Systems*, pages 278–298, Nov. 1986.
- [ABM89] Y. Afek, G. Brown, and M. Merritt. A lazy cache algorithm. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 209–223, June. 1989.
- [AH90] S. V. Adve and M. D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–11, May 1990.
- [BD86] P. Bitar and A. M. Despain. Multiprocessor cache synchronization : Issues, innovations, evolution. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 424–433, June 1986.
- [Bro86] E. D. Brooks. The Butterfly barrier. *International Journal of Parallel Programming*, pages 295–307, Aug. 1986.
- [BW87] J. Baer and W. Wang. Architectural choices for multi-level cache hierarchies. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 258–261, 1987.
- [CFKA90] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *IEEE Computer*, 1990.
- [CGB89] D. A. Cheriton, H. A. Goosen, and P. D. Boyle. Multi-level shared caching techniques for scalability in VMP-MC. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 16–24, June 1989.
- [CLLG89] K. Charachorloo, D. Lenoski, J. Laudon, and A. Gupta. Memory consistency and event ordering in scalable shared-memory multiprocessors. Technical Report CSL-TR-89-405, Stanford University, Computer Systems Laboratory, Nov. 1989.

- [DSB86] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434-442, June 1986.
- [DSB88] M. Dubois, C. Scheurich, and F. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, pages 9-21, Feb. 1988.
- [Goo83] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124-131, June 1983.
- [GW88] J. R. Goodman and P. J. Woest. The Wisconsin Multicube: a new large-scale cache-coherent multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 422-431, June 1988.
- [HA89] P. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memory. Technical Report GIT-ICS-89/39, Georgia Institute of Technology, Oct. 1989.
- [IEE89] IEEE P1596 - SCI Coherence Protocols. *Scalable Coherent Interface*, March 1989.
- [KEW⁺85] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 276-283, June 1985.
- [KMRS88] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:79-119, 1988.
- [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.
- [LR90] J. Lee and U. Ramachandran. Synchronization with multiprocessor caches. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 27-37, May 1990.
- [LS88] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, September 1988.
- [McC84] E. McCreight. *The Dragon Computer System: An early overview*. Xerox Corp., Sept. 1984.
- [MHW87] T. N. Mudge, J. P. Hayes, and D. C. Winsor. Multiple bus architectures. *Computer (USA)*, 20(6):42-48, June 1987.
- [Pol88] C. D. Polychronopoulos. *Parallel Programming and Compilers*, pages 113-158. Kluwer Academic Publishers, 1988.
- [PP84] M. Papamarcos and J. Patel. A low overhead solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348-354, June 1984.

- [SD87] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 234–243, June. 1987.
- [TS87] C. P. Thacker and L. C. Stewart. Firefly: A multiprocessor workstation. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, Oct. 1987.
- [Wil87] A. W. Wilson. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 244–252, June 1987.
- [YTL87] P. C. Yew, N. F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessor. *IEEE Transactions on Computers*, pages 388–395, April 1987.

Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer

Umakishore Ramachandran

Mustaque Ahamad

M. Yousef A. Khalidi

PAPER [RAK89]

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Ga 30332-0280 USA

Abstract

Clouds is a distributed operating systems research project at Georgia Tech. With threads and passive objects as the primary building blocks, *Clouds* provides a location-transparent protected procedure-call interface to system services. Mechanisms for synchronization within objects, and atomicity of computation are supported in *Clouds*. The primary kernel mechanism in object-based systems such as *Clouds* is the mapping of the object into the address space of the invoking thread. Hence the performance of such systems depends crucially on the efficiency of memory mapping. The problem gets exacerbated with distribution since now the object invoked by a thread may be located on a remote node. Since a thread can potentially invoke any object, the virtual address spaces of all objects can be viewed as constituting a "global distributed shared memory". Such a view is attractive from the perspective of software architecture since it suggests a uniform implementation of a system-wide memory-mapping mechanism. We present an organization and mechanisms for supporting this abstraction of a distributed shared memory. We propose a *distributed shared memory controller* that provides mechanisms for efficient access and consistency maintenance of the distributed shared memory. The novel feature of our approach is the exploitation of process synchronization to simplify consistency maintenance. The distributed shared memory mechanisms serve as the backbone for implementing object invocation, synchronization mechanisms, and network-wide memory management in the *Clouds* system.

1 Introduction

We are exploring hardware support to improve the performance of object-based distributed operating systems. The hardware environment consists of a collection of computing nodes interconnected by a local area network. There are one or more processors and a certain amount of memory in each node. Nodes do not share memory; message exchange across the network is the only mechanism for communication between them.

Many operating systems designs for such an environment [4,14,20,32] place a *message-passing kernel* on each node, supporting processes and communication between them via explicit messages. This kernel supports both *local communication*—communication between processes on the same node—and *non-local* or *remote* communication (sometimes implemented via a distinguished network manager process). Access to system services are requested via protected *procedure calls* in a traditional system, whereas in a message-based operating system they are requested via *message passing*. While a simple procedure call costs just a few instructions, and a protected procedure call (kernel call) costs a few hundred instructions, IPC costs a few

This work has been funded in part by NSF grants CCR-8619886 and MIP-8809268.

thousand instructions in several systems that we studied [28]. Message-based operating systems are attractive for structuring distributed systems due to the separation of policy (encoded in server processes) from mechanism (in the kernel).

Object-based distributed operating systems [7,2,39,34,26] view the resources of the system as a collection of objects. *Clouds* [7] is an object-based distributed operating system being developed at Georgia Tech. In *Clouds*, system services are encoded in *passive objects* (syntactic units that are similar in flavor to the server processes of message-based systems) that occupy distinct virtual address spaces in the system. Access to system services are requested by invocations (similar to the protected procedure calls of traditional systems) into the appropriate system object. The speed of object invocation is often used as a yardstick for measuring the performance of object-based systems. In passive object-based systems, invocation performance depends on the efficiency of object memory management (see §2). The problem gets exacerbated with distribution since now the invoked object may be located on a remote node.

In this paper we suggest mechanisms (that can be implemented in hardware/firmware) for supporting the abstraction of "globally distributed shared memory". In §2, we give the relevance of our work and motivate the need for supporting this abstraction. Related work in the area of memory coherence is presented in §3. With the *Clouds* operating system as our target application, we present our ideas on customizing the memory coherence requirements in §4. Our proposed hardware organization, the primitives provided by the distributed shared memory controller, and the algorithms for maintaining the consistency of the distributed shared memory are discussed §5. In §6 we describe a software implementation of the proposed primitives. A performance evaluation of our scheme is presented in §7. Finally, our conclusions are presented in §8.

2 Relevance

Clouds [7] is a distributed operating system that is intended to provide a unified environment over distributed hardware. Location independence for data as well as processing, atomicity of distributed computation, and fault-tolerance are some of the research goals of *Clouds*.

Objects and *threads* are the basic building blocks of *Clouds*. Objects are passive entities and specify a *distinct* and *disjoint* piece of the global virtual address space that spans the entire network. An object is the encapsulation of the *code* and *data* needed to implement the *entry points* in the object. Thus a *Clouds* object can be considered syntactically equivalent to an abstract data type in the programming language parlance. Access to entry points in the object are accomplished through a capability mechanism in the kernel.

Threads are the only active entities in the system. A thread is a unit of activity from the user's perspective. Upon creation, a thread starts executing in an object. A thread enters an ob-

ect. A thread enters an object by invoking an entry point in the object. It then executes the code in the entry point, and returns to the caller object. Binding the object invocations to the entry points in the object takes place at execution time. A thread in the course of its computation traverses the virtual address spaces of the objects that it invokes. A thread is comparable to a process (as defined in many conventional systems), with the exception that a thread may span machine boundaries. For the purposes of this paper, a process is synonymous to a thread.

The virtual address spaces of all objects can be viewed as constituting a "global distributed shared memory". Such a view is attractive from the perspective of software architecture since it suggests a uniform implementation of a system-wide memory-mapping mechanism. Local object invocation involves mapping the required memory segments of the object into the address space of the invoking thread. The current trend in structuring distributed systems is to use a collection of diskless computational servers (workstations) and a few data servers (file servers). In such an environment, the code and data for the (local) invocation has to be paged-in from the data server. Further, for remote object invocation we have one of two choices: The first choice is to perform the computation at the node where the object resides (remote procedure call). The second choice is to make the invocation appear local by bringing in the segments required for the invocation. While we have to support the former for immovable objects (such as an object that reads disks), we believe that the latter may be a better choice for movable objects. There are two reasons to support this belief:

• the principle of locality [15] that suggests an invocation (or other invocations in the same object) may be repeated

• the reduction in computational overhead due to the elimination of slave process management to support remote invocation at the node where the object resides [23].

The idea of using the abstraction of a globally distributed memory in an object-based system appears to be novel. Other researchers have proposed software architectures on the shared memory paradigm, in different settings:

• In a speech recognition application, Bisiani and Forin [9] use data structures that are shared by multiple language modules that are distributed on heterogeneous machines. They show that communication through shared memory is a viable alternative to message-passing even when the environment involves cooperation between multilingual program modules and heterogeneous machines.

• Processes in the programming language *Linda* [10,18] communicate via a globally-shared collection of ordered tuples.

• A logically shared bulletin-board is proposed by Birman, et al. [8] for structuring asynchronous interactions between processes in distributed systems.

• To exploit the multicast capability of local area networks, Mad and Bernstein [1] describe a bulletin-board based process communication mechanism that could just as well be implemented with a shared memory paradigm.

• Integrating the mechanisms for virtual memory management and local interprocess communication, Mach [32] gives efficient implementation of local interprocess communication. Currently, researchers at CMU are investigating the duality of shared memory and message passing in the context of network communication as well [38].

• [39] achieves substantial reduction in the cost of program migration by using copy-on-write techniques (Accent) and on-demand fetches during remote execution.

- Cheriton [11] advocates problem-oriented shared memory as the basic concept for structuring distributed systems.
- Emerald [21] is a distributed object-based language and system with support for object mobility.

We believe that software architectures based on the shared memory paradigm would benefit considerably (in both performance and ease of implementation) if the underlying hardware were to provide a transparent mechanism for efficient access and consistency maintenance of distributed shared memory. Thus the investigation of hardware support for providing the abstraction of a distributed shared memory is worthwhile.

3 Related Work

3.1 Shared Memory Multiprocessors

Consistency maintenance in distributed shared memory is similar to cache coherence in multiprocessors. Shared memory multiprocessors such as Encore's Multimax [16], consist of several processors connected to a common shared memory via a system bus. A main memory cache is associated with each processor to help reduce the traffic to the shared memory. Multiprocessor cache consistency protocols (such as [19,22]) ensure the following memory coherence constraint: a read operation performed by a processor returns the most recent value written into that location (by any processor). This criterion is appropriate in a shared memory multiprocessor since the system bus (a broadcast medium) serializes the memory operations of all the processors. This approach is a brute-force one to assure coherence since there is no semantic knowledge associated with the data being cached.

We make the following observations regarding these protocols: Multiprocessor cache coherence algorithms (see [3] for a survey) consider memory coherence problem in isolation. In reality, memory coherence and process synchronization are closely intertwined. The ability of a process to read or write shared data is invariably acquired through some synchronization method. Since the cached data has no synchronization information associated with it, these algorithms tend to be an overkill owing to their generality. However, these algorithms are a viable approach for solving the cache coherence problem in multiprocessors since the cost (measured in circuit complexity as well as time) of implementing them in hardware is a small fraction of the total system cost. Further, multiprocessors have the ability to invalidate all cached copies in one atomic bus cycle owing to the system bus.

3.2 Distributed Shared Memory

Some work has been done in extending the shared memory paradigm to a distributed system. Li [24] presents a slight variation of the Berkeley protocol for multiprocessor cache consistency [22] as a solution to maintain the consistency of distributed shared memory. The entire memory is considered potentially sharable for both reads and writes. Hence the current owner of a page (the node that has write access to the page) keeps a *copy-set*—the set of nodes that have a read-copy of the page. In Li's scheme, a write into a shared location results in an invalidation message to be sent to each of the nodes in the copy-set for the page containing the location. Note that the nodes that have a copy may no longer be interested in that page. These invalidation messages are a high price to pay in a distributed environment. Some of the pages may never be written into (for example text pages), but since the algorithm only deals with raw pages the overhead of

keeping the copy-set information is also incurred for such pages. Li's solution has the same drawback as the multiprocessor cache coherency algorithms—memory coherence problem is dealt with in isolation without considering process synchronization.

Agora [9] (see §2) supports shared data structures that span heterogeneous machine architectures and multiple languages. *Agora* adopts a mechanism similar to Li's with invalidation messages on writes to shared locations, and hence is inefficient for the same reason. However, since *Agora* is tailored for a very specific application it allows sharing at the level of individual data structures rather than raw pages.

Fleisch [17] proposes a distributed shared memory facility for the *Locus* [26] system that supports Unix System V shared memory semantics. The proposed coherence algorithm is similar to Li's [24]. Fleisch's work does not address such issues as locality and synchronization.

4 Customizing Coherence

We are interested in exploring hardware support for the abstraction of a distributed shared memory. The basic idea is the following: Each node has associated with it a "network cache"—a repository for recently accessed remote memory-segments and their owners (nodes). If a remote memory-segment is not in the cache, it is requested from the owner and cached for future reference. If the segment is "dirty", the network-cache becomes the supplier in the future for this segment. If and when the segment is replaced it is sent back to the owner of the segment.

How do we define memory coherence in this environment? The definition that works well for a shared memory multiprocessor is inappropriate for this environment since there is no "system bus" to impose a total order on the memory operations that are performed by all the processors. Further, while invalidation of cached copies of data is a viable approach in multiprocessors (with a system bus) it is infeasible (due to the cost of the invalidation messages) in a distributed system. Invalidation involves at least sending a multicast message to all the nodes that have a read-copy of the segment. Achieving reliable delivery of such multicast messages is prohibitively expensive in a distributed system [13].

Exploiting application specific semantic information would reduce the complexity of the problem and make a hardware solution viable. In the scheme that we propose we deal with process synchronization and memory coherence together. Since the application that we are trying to support is the Clouds operating system we use the structure of the objects to dictate the coherence requirement. However, since Clouds is a general purpose operating system we cannot make any more assumptions as *Agora* does in providing finer-grain sharing at the level of individual data structures. Read-only code and data areas can be distributed without the need for maintaining consistency. Only the read-write data area of an object requires maintenance of consistency.

5 Hardware Organization

The organization (Figure 1) we propose inside each node of the network is the following: a host that executes distributed applications; a distributed shared memory controller (DSMC) together with the network interface assists the host in mapping memory segments (local and remote) into the virtual address spaces of the application processes. There is a minimal *kernel* on the host that traps system calls, and virtual address translation

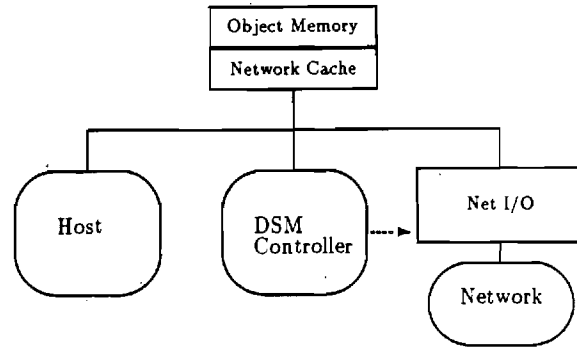


Figure 1: Hardware Implementation

faults. The DSMC is also in control of the network. The system memory is (logically) partitioned into two parts: One part (*object memory*) is for housing the segments of locally created objects; the other part (*network cache*) is for caching segments from remote objects. Conceptually there are two lists of process control blocks: the *host-list* that the host looks at to schedule runnable processes; the *DSMC-list* that the DSMC looks at to service memory segment requests (local and remote). The host enqueues processes that fault on virtual address translation in the *DSMC-list*. The DSMC enqueues processes that have become runnable again (after the fault service) in the *host-list*. The operations provided by the DSMC, and the algorithms for implementing these operations are the topics of interest in this paper. Although conceptually the DSMC is shown as a "co-processor", it can be implemented as a software module that coexists with the kernel (see §6).

5.1 Clouds Objects

Objects in Clouds consist of one or more of the following types of areas: read-only code, read-only data, shared read-write data. We refer to these areas as *segments*. Segments serve as containers of data and can be of variable size. The contents of a segment may only be accessed when the segment is attached to an object [6]. Segments persist until explicitly destroyed. We refer to the node where a segment is created as the *owner* of the segment. There is a well-known area of the system memory—*keeper segment*—maintained by the DSMC. We associate a *keeper location* with each read-write segment owned by the node. The keeper location for a segment points to the node currently having write access to the segment. *Owner* and *keeper* point to the same node at segment creation time.

5.2 Virtual Address

The virtual address generated by a process is interpreted as being composed of three fields: object name, segment name, and segment offset. Using the segment name, the DSMC does a table lookup to determine the location of the segment¹. The DSMC at each node keeps a *segment table*. This table contains the following information for segments currently mapped at the node: their sizes and types, and their mapping to physical segments or disk blocks. Further, for locally created segments the segment

¹The location table is maintained by the kernel. If the segment name does not appear in the table, the DSMC can request the kernel to locate the segment.

ble remembers the current location (*keeper*) of the segment. The segment-table entries also each have a *queue*—the list of processes waiting on the segment.

A segment fault occurs when the virtual address generated by a process is currently unmapped. In addition, every object invocation results in a fault to map in the segments needed for the invocation. A segment fault (or object invocation) manifests itself as a trap into the kernel on the host. As a result of the trap, the kernel calls an *object manager* that in turn invokes the DSMC primitives. The host enqueues the faulting process in the *DSMC-list* (for the DSMC), and schedules the next runnable process. The DSMC is responsible for mapping in the required segments, making the process runnable. A process waiting for a remote segment is queued on the appropriate entry in the map table. Upon receiving a remote segment in response to a previous request, the DSMC sets up the memory map of the processes waiting for the segment and enqueues these processes in the *host-list* (for the host).

Though the segment as used in this paper is a logical entity, it is mapped onto the underlying hardware into an integral number of pages. In Reference [6], we discuss an implementation of the DSMC as a software module that is part of the paging system. This implementation is summarized in §6.

DSMC Primitives

The DSMC provides data transfer and synchronization primitives supporting the abstraction of a global distributed shared memory. From the applications (Clouds kernel in our case) we entertain four types of requests:

get(segment): The DSMC is responsible for fetching the required segment, setting up the segment table of the faulting process, and enqueueing the process on the host-list for execution.

discard(segment): The DSMC frees the physical memory occupied by the segment by sending the segment back to the owner.

segment, semaphore: The DSMC performs an atomic semaphore *P* operation on the specified synchronization variable.

segment, semaphore: The DSMC performs an atomic semaphore *V* operation on the specified synchronization variable.

DSMCs exchange messages to satisfy these requests. The messages recognized by the DSMC are the following:

```

msg(segment, mode)
msg(segment)
msg(segment)
msg(segment)
msg(segment, semaphore index i)
msg(segment, semaphore index i)
msg(segment, semaphore index i)
msg(segment, error type)

```

In the next few subsections, we give the algorithms required for implementing the data transfer and synchronization primitives. In the case of understanding the algorithms, we show a single execution while processing these primitives. However, we note that the DSMC is multiprogrammed. For example, upon receiving a *msg_get* request

to its peer and then proceeds to the next request in its queue (DSMC-list). Eventually, when a reply arrives from its peer the local DSMC takes the appropriate action. In the algorithms to follow, we do not show this asynchrony. For simplicity, we show each request from a DSMC to its peer as a synchronous one.

5.3.1 Data Transfer

Get(segment) is the primitive for mapping a currently unmapped segment. On a segment fault the host enqueues the faulting process in the DSMC-list with a *get* request for the required segment. A *get* request for a read-only segment is trivially satisfied if the object is local; if the object is remote the DSMC requests its peer (at the owner node) for the read-only segment. On receiving the request, the peer DSMC sends the read-only segment to the requesting DSMC without performing any house-keeping work.

A *get* request for a read-write segment is implemented by the following algorithm:

```

if local(segment) then
  case keeper of
    self:
      if not memory(segment) then
        bring segment into object-memory from disk;
      endif;
    remote:
      send msg_get(segment) to remote DSMC;
      receive msg_segment(segment) from remote DSMC;
      place segment in object-memory;
      change keeper to self;
    endcase;
  else /* segment is remote */
    /* go ask the owner */
    send msg_get(segment) to remote DSMC;
    receive msg_segment(segment) from remote DSMC;
    place segment in network-cache;
  endif;
  /* the requested segment is now in memory;
  map it into the process' address space */
  map segment into process' address space;

```

When a DSMC receives *msg_get* request for a (read-write) segment it does the following:

```

case keeper of
  self:
    if not memory(segment) then
      bring segment into object-memory from disk;
    endif;
    change keeper to requesting DSMC;
    send msg_segment(segment) to requesting DSMC;
  remote:
    send msg_forward(segment) to keeper;
    /* note at most one forwarding */
    change keeper to requesting DSMC;
  endcase;

```

The owner may forward the request to the current keeper. A DSMC that receives a *msg_forward* request for a (read-write) segment does the following:

```

send msg_segment(segment) to requesting DSMC;
invalidate the segment entry in the segment-table;
return freed segment in the network-cache to free-list;

```

Placing a segment in the network-cache may involve freeing up segments from the network-cache. The DSMC sends *msg_discard* to the owner DSMC of the segment for this purpose. The algorithm for freeing up a segment from the network-cache is the following:

```

invalidate the segment entry in the segment-table;
if read-write(segment) then
  if dirty(segment) then
    send msg_discard(segment) enclosing segment to owner;
  else /* clean segment */
    send msg_discard(segment) to owner;
endif;
endif;

```

The owner DSMC upon receiving *msg_discard* request does the following:

```

if segment enclosed then
  if memory(segment) then
    write segment into memory;
  else
    write segment onto disk;
  endif;
endif;
change keeper to self;

```

Due to network delays it is possible that a node (the future keeper of a segment) may receive a *msg_forward* request before the segment arrives from the owner or the forwarder (the current keeper). However, recall that the keeper information with the owner of an object is absolute. Therefore the node can simply buffer the segment-request, and honor it when the segment arrives, without compromising the correct operation of the system.

5.3.2 Synchronization

Get, and *discard* are not enough to efficiently implement the synchronization primitives provided by the Clouds kernel. For example, consider semaphore operations (*P* and *V*) supported by the Clouds kernel for synchronization inside an object. With just *get* and *discard* operations supported by the DSMC, an obvious implementation of the semaphore operations may be the following: place the semaphore data structure (a value field, and a list of waiting processes) in a read-write segment; on every semaphore operation *get* this data structure from its keeper; and perform the operation atomically (disabling interrupts). By definition, semaphore operations order the execution of cooperating processes. Therefore, it seems wasteful to ship the semaphore data structure back and forth between these processes when they are on different nodes.

For now we consider the solution where the semaphore operations are performed at the owner node. The semaphore operations are provided as primitives understood by the DSMC. Synchronization variables are allocated semaphore segments. We note the fundamental difference between the roles played by the DSMC in the data transfer operations (*get*, *discard*) and the synchronization operations (*P*, *V*): In the former, the DSMC fetches the current copy from the keeper (it does not have to worry about the contents of the fetched segment); whereas in the latter the DSMC manipulates the contents of the synchronization variables at the owner node and reports success/failure to its requesting peer DSMC.

When a process performs an operation (*P* or *V*) on a synchronization variable, it results in a trap to the kernel. The

kernel in turn enqueues this process in the *DSMC-list*. The algorithm executed by the DSMC for implementing *P(segment, semaphore i)* or *V(segment, semaphore i)* operation is the following:

```

if local(segment) then
  case operation of
  P:
    decrement synchronization variable i;
    if variable value less than zero then
      enqueue process on queue(segment, i);
    else
      enqueue process on host-list; /* ready to resume execution */
    endif;
  V:
    increment synchronization variable i;
    if variable value less than or equal to zero then
      remove an entry from queue(segment, i);
      case entry of
      remote DSMC:
        send msg_ack(segment, i) to remote DSMC;
      process:
        enqueue process on host-list; /* removed entry */
      endcase;
    endif;
    enqueue process on host-list; /* V-ing process */
  endcase;
else /* remote segment */
  case operation of
  P:
    send msg_P(segment, i) to remote DSMC;
    enqueue process on queue(segment, i);
    receive msg_ack(segment, i) from remote DSMC;
    dequeue process from queue(segment, i);
    enqueue process on host-list; /* ready to resume execution */
  V:
    send msg_V(segment, i) to remote DSMC;
    enqueue process on host-list; /* V-ing process */
  endcase;
endif;

```

On receiving a *msg_P* or a *msg_V* request the DSMC does the following:

```

case operation of
P:
  decrement synchronization variable i;
  if variable value less than zero then
    enqueue (requesting DSMC, process) on queue(segment, i);
  else
    send msg_ack(segment, i) to requesting DSMC;
  endif;
V:
  increment synchronization variable;
  if variable value less than or equal to zero then
    remove an entry from queue(segment, i);
    case entry of
    remote DSMC:
      send msg_ack(segment, i) to remote DSMC;
    process:
      enqueue process on host-list; /* removed entry */
    endcase;
  endif;
endcase;

```

5.3.3 Merging Data Transfer and Locking

Our data transfer primitives eliminate invalidation messages by keeping exactly one copy of a read-write segment. However.

keeping just one copy reduces availability (for readers) in applications that can be modeled as a readers/writers problem. The fact that there is exactly one copy of a read-write segment is appropriate from the point of view of the writers while being a severe restriction for the readers. Since we expect such applications to be encountered quite frequently, we propose the following modification to the *get* primitive to increase the availability. The modification is to include *mode* information in the primitive: *get(object, segment, mode)*, where *mode* can be one of *read-only*, *read-write*, or *none*. The DSMC keeps two additional pieces of information in the segment table for read-write segments: *lock mode*, and *readers*. When the owner DSMC receives a request for a read-write segment it does the following:

```

if queue(segment) empty then
if (lock-mode = none) and (mode = none) then
  case keeper of
  self:
    if not memory(segment) then
      bring segment into object-memory from disk;
    endif;
    change keeper to requesting DSMC;
    send msg_segment(segment) to requesting DSMC;
  remote:
    send msg_forward(segment) to keeper;
    /* note at most one forwarding */
    change keeper to requesting DSMC;
  endcase;
return;
elseif (lock-mode <> none) and (mode = none) then
  enqueue requesting DSMC on queue(segment);
return;
else /* mode <> none */
  if keeper is remote then
    /*
     * the owner remains the keeper when get requests are issued
     * with mode = read-only or mode = read-write;
     * if keeper is remote, he acquired the segment in mode 'none';
     * so grab the segment back from him
     */
    send msg_get(segment, none) to keeper;
    receive msg_segment(segment) from keeper;
    change keeper to self;
  endif;
  if (lock-mode <> read-write) and (mode = read-only) then
    /* readers can enter if lock-mode is either read-only or none */
    set lock-mode to read-only;
    increment readers;
    end msg_segment(segment) to requesting DSMC;
  if (lock-mode = none) and (mode = read-write) then
    /* a writer can enter if lock-mode is none */
    set lock-mode to read-write;
    end msg_segment(segment) to requesting DSMC;
  else
    enqueue requesting DSMC on queue(segment);
  endif;
  /* if (lock-mode <> read-write)... */
  /* if (lock-mode = none)... */
  /* queue is non-empty */
  enqueue requesting DSMC on queue(segment);
  /* if queue(segment)... */

```

algorithm does not result in the generation of any invalid messages. The readers (and writers) have to explicitly *discard* message on the segment. The *msg_discard* message used by the owner to release the read-write lock on the segment. On receiving a *msg_discard* request the DSMC does the following:

```

if lock-mode = read-only then
  decrement readers;
  if readers = 0 then
    lock-mode = none;
  endif;
else
  lock-mode = none;
endif;
process_queue:
if queue(segment) non-empty then
  if lock-mode = none then
    if first-entry.mode = read-only then
      set lock-mode to read-only;
      foreach subsequent-entry in the queue
        send msg_segment(segment) to requesting DSMC;
      until (subsequent-entry.mode = read-write) or
        (queue(segment) is empty) or (subsequent-entry.mode = none);
    elseif (first-entry.mode = none) and
      (queue(segment) not empty) then
      send msg_error(segment, mode_conflict) to requesting DSMC;
      goto process_queue;
    elseif (first-entry.mode = none) and
      (queue(segment) is empty) then
      send msg_segment(segment) to requesting DSMC;
    else /* first-entry.mode = read-write */
      set lock-mode to read-write;
      send msg_segment(segment) to requesting DSMC;
    endif;
  endif;
endif;
endif;

```

Note that mode none is a programming anomaly amidst read-only and read-write requests, and is treated as such by responding with a *msg_error* message. Upon receiving the *msg_error* message, the initial requester may decide to re-issue the *get* request.

How does the kernel know when to discard a segment? In readers/writers problem the reader (or the writer) explicitly acquires the appropriate lock, reads (or writes), and releases the lock. In Clouds, *lock* and *unlock* are system operations. When the Clouds application programmer uses these primitives, the operating system translates these primitives to *get* and *discard* the appropriate segments, respectively. We note that there are possibilities of deadlock if the application fails to release a lock. If an application fails to perform an *unlock* operation, the algorithm is in error. We envision a deadlock detection mechanism in the operating system that is layered on top of the DSMC mechanisms. The deadlock detector would issue *unlock* requests on behalf of the deadlocked processes. Note that if the kernel fails to perform a *discard* operation upon an *unlock* request, it is tantamount to a kernel bug.

With this enhancement our DSMC provides as much as availability as Li's algorithms [24] without incurring the high cost of invalidation messages. Note that explicit discards from the readers are in lieu of the invalidation messages. However, we contend that these are exactly the minimum number of messages required to maintain a consistent shared data structure.

5.3.4 Weaker Semantics

The above DSMC primitives provide strong memory coherence, where a write to a location by a process is seen by other processes when they access the same location. While this criterion is appropriate for applications that rely on the DSMC to enforce memory coherence, other applications may find these primitives too restrictive. Some applications may need to provide for memory coherence as part of their algorithms, while other applica-

tions may not need strong memory coherence. For example, a system monitoring facility may need to inspect the contents of some segments without acquiring locks, while a distributed game that maintains the state of a graphics screen in shared memory may sacrifice strong memory coherence for better performance.

For such applications, we provide a simple mechanism to acquire a copy of a segment without enforcing memory coherence. We define *weak-read* mode for the *get* request. A *get(segment, weak-read)* request for a segment acquires a copy of the segment from the owner DSMC. Upon receiving a *get(segment, weak-read)* request, the owner DSMC sends a copy of the segment to the requester, regardless of the fact whether a copy of the segment exists in any other node.

5.3.5 Summary of Modes

In summary, using the *get* primitive a segment may be acquired in one of four modes: *read-only*, *read-write*, *weak-read*, or *none*. *Read-only* mode signifies non-exclusive access but guarantees that the segment will not change until the node explicitly discards the segment. *Read-write* mode signifies exclusive access (for the node) with a guarantee that the segment will not be thrown away until the node explicitly discards the segment. *Weak-read* mode signifies non-exclusive access with no guarantee whether the segment will change or not. *None* mode signifies exclusive access with no guarantee whether the segment will be taken away or not.

6 Implementation of DSMC

Ra [5,7] is an operating system kernel designed to be the nucleus of Clouds operating system. It is currently implemented on the Sun-3 architecture. Ra defines and manages three primitive abstractions: *segment*, *virtual space*, and *isiba*. The contents of a segment may only be accessed when that segment is mapped to a range of addresses in a virtual space. Virtual spaces abstract the notion of an addressing domain, and they are composed of segments. Ra *isibas* are an abstraction of the fundamental notion of computation or activity and can be thought of as lightweight processes.

The Ra kernel is responsible for mapping segments into virtual memory using the memory management hardware provided by the underlying architecture. The size of a segment is a multiple of the physical page size. Ra assumes the existence of *partitions* that are responsible for realizing, maintaining, and storing segments. Partitions are an example of *system objects*. System objects encapsulate necessary and/or useful operating system services and resource managers that have direct access to the Ra kernel, but are nonetheless outside the kernel. System objects are trusted software modules that are loaded dynamically in the system space. Other system objects include device drivers, resource managers, and user-level object support. Each partition provides (at least) the following calls for use by Ra: *activate/deactivate segment*, *create/destroy segment*, and *page-in/page-out* portions of segments. When Ra is instructed to service a segment request (e.g. to map a segment into a virtual space), it invokes the appropriate partition to fetch the segment into physical memory. Ra then manipulates the memory management hardware to map the physical pages appropriately.

We have implemented the DSMC as a software module that consists of approximately 3500 lines of C++ [34]. Figure 2 shows the organization of the DSMC implementation on Ra. The boxes in the figure denote system objects. The DSMC cooperates with remote DSMC's to implement the distributed shared memory primitives. DSM Partition is a Ra partition

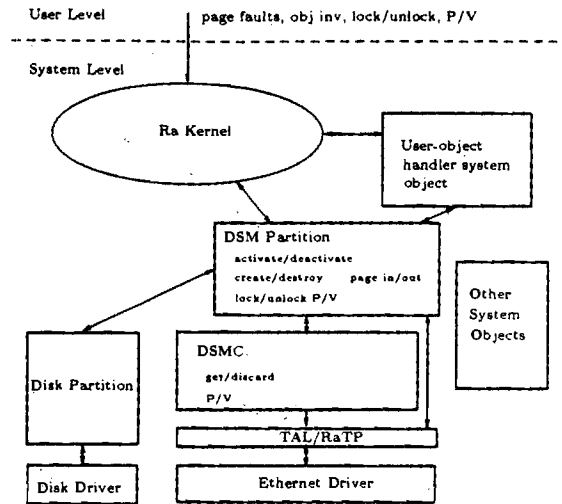


Figure 2: Organization of DSMC implementation under Ra

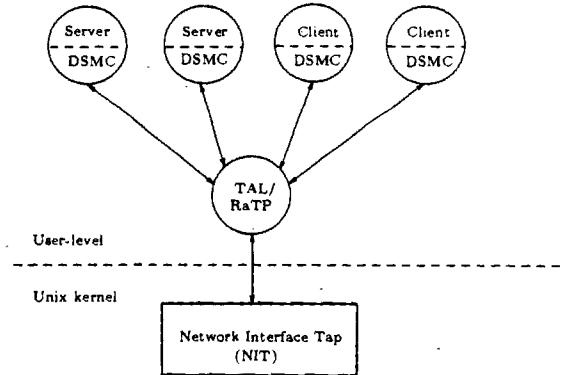


Figure 3: Organization of DSMC implementation on Unix

that provides the kernel with the ability to create/destroy and activate/deactivate segments, page-in/page-out portions of segments, and semaphore P/V operations. The DSMC decides if a segment is owned by the local node or a remote node. It uses the Disk Partition to access local segments, and cooperates with the DSMC to access remote segments. *Transaction abstraction layer* (TAL) is a simple transaction-oriented communication protocol that provides reliable multi-packet request/response messages. It is used by the DSMC to communicate with remote DSMC's. TAL protocol is similar to other transaction-oriented protocols such as VMTP [12]. However, it is much simpler than VMTP since it is tailored to our application domain. Because the DSMC algorithms require simple request/response messages only (possibly with message forwarding), it is possible to substitute other transaction-oriented protocols for TAL. The Disk Partition maintains segments owned by the local node on the local secondary storage (if any).

The DSMC implementation executes on top of Unix (as a user process) or Ra. The operating system dependencies are isolated in a few C++ classes. The organization of the DSMC implementation on Unix is shown in Figure 3. On Unix, TAL runs as a user process that uses SUN's Network Interface Tap (NIT) [35] to receive packets from the net and to route them

among a set of clients and servers. The DSMC code is linked-in with client code. DSMC code is also linked-in with server code that uses the Unix file system to store segments. Implementation of DSMC on Unix and Ra serves three purposes:

1. The Unix environment makes it easy to test and verify the DSMC and TAL protocols.
2. The Unix file system is available for use as permanent store for segments. Ra executes on diskless Sun-3 workstations with backing store provided by Unix machines.
3. The strength of Unix is the rich program development environment that it provides. The strength of Clouds is transparent management of distributed data and computation. Providing inter-operability between Unix and Clouds is one of our design goals. DSMC implementation on Unix and Ra serves this purpose. System and user objects are developed on Unix and demand-paged to Ra via DSMC mechanisms.

The Unix implementation of the DSMC and TAL is complete, and we report on its performance in §7. The Ra implementation of the DSMC is awaiting the completion of an ethernet interface for Ra.

Performance Results

We measured the performance of the DSMC implementation on Unix. All measurements are done on Sun-3/60 workstations with 4M bytes of memory, connected through a 10M bits/sec ethernet. We mask out the cost of secondary storage access by paging segments in memory before measuring the costs of the DSMC primitives. Table 1 summarizes the results.

Get or discard (8K bytes)	
without forwarding:	43.4 ms
throughput:	185 Kbytes/s
with forwarding:	63.7 ms
throughput:	126 Kbytes/s
V operation:	16.5 ms
P operation:	31.5 ms
Activate segment:	23.3 ms

Table 1: Measurements of DSMC operations on Unix

Table shows that on an average fetching a segment (with forwarding) of size 8K bytes (the page size on the Sun-3) costs 63.7 ms. Van Renesse et al. report a transfer rate of 43.4 ms. Van Renesse et al. report a transfer rate of 40 Kbytes between two user processes on different nodes using an RPC on a 10M bits/sec ethernet [36]. Our implementation uses two user processes per node and still compares favorably with the figures reported by van Renesse et al. A null from one DSMC to another costs roughly 20 ms, a large portion of which is spent context switching between the kernel and TAL, and between TAL and DSMC. Moving TAL into the kernel would eliminate the additional context switching we are currently investigating such an implementation. The above V operation costs only 16.5 ms since it is non-blocking, i.e., the issuing process continues without waiting for acknowledgment from the remote DSMC.

We have also evaluated our DSMC scheme and compared it with a simulation [28]. In the simulation, the DSMC is implemented as a software module as described in §6. The performance shows that:

- Over a range of object invocation locality, DSMC has significant advantages over RPC.
- Use of distributed shared memory achieves automatic distribution of processing load, by performing the invocation locally instead of on the owner nodes.
- Instead of addressing memory coherency and synchronization separately, applications that can be modeled as readers/writers problems benefit considerably when locking and segment access primitives are combined.

As mentioned before, the DSMC primitives are provided to the operating system as a set of mechanisms for managing memory in the network. Policy decisions, such as when to use RPC and when to use the DSMC primitives, are in the operating system. The operating system decides how and when to use the DSMC primitives. For example, during periods of high concurrent access to an object, the operating system may use the DSMC primitives to locate the object at one node, and then use RPCs to invoke the object thereby reducing data movement across the network.

Comparison with Li's Scheme

To illustrate the advantage of combining mutual exclusion and consistency maintenance, we show how a readers/writers problem can be implemented using our scheme and Li's scheme, and compare the number of messages generated in each case. Li's basic scheme can be summarized as follows:

- On a read fault:
 1. Ask manager for page.
 2. Manager forwards request to owner.
 3. Owner sends copy of page to requester.
- On a write fault:
 1. Ask manager for page.
 2. Manager forwards request to owner.
 3. Owner sends page and copy-set to requester.
 4. Requester invalidates all copies in copy-set by sending a message to each node holding a read-copy of the page.

Suppose we want to program the following readers/writers problem: A segment that is accessed by a set of readers and writers resides on the manager node M, and each of the readers/writers runs on a different node (for simplicity, we assume that no reader/writer runs on the manager node). Each reader/writer computes for a while, then accesses the shared segment. It is clear that some mechanism to synchronize access to the segment is needed.

In our scheme, locks (see §5.3.3) can be used to solve this problem as follows:

```

Reader:
loop
  compute for a while
  lock segment in read-only mode
  /* lock generates one message to manager */
  /* manager eventually sends back segment */
  access segment
  unlock segment
  /* unlock generates one message to manager */
endloop
    
```

Similarly for a writer:

Writer:

```

loop
  compute for a while
  lock segment in read-write mode
  /* lock generates one message to manager */
  /* manager eventually sends back segment */
  access segment
  unlock segment sending modified segment back to manager
  /* unlock generates one message to manager */
endloop

```

It is not clear how one can program this simple readers/writers problem using Li's primitives, for Li does not address the issue of process synchronization. Process synchronization has to be addressed separately, because there is no way within Li's scheme for a user to lock a page while accessing it. Therefore, we assume that a lock operation is implemented by sending a message to a distinguished server, requesting access to the shared segment in the required mode. When the reply is received from the server, the shared segment is accessed. When the reader/writer is finished accessing the segment, it sends an unlock message to the server.

Using our scheme, the lock operation generates 2 messages: one message to the manager, and another from the manager to the requester (the second message includes the segment). The unlock operation generates one message to the manager (a *discard* message that includes the segment in the case of unlock by a writer). Accessing the segment generates no messages, because the segment is made available locally as a result of the lock operation. Using Li's scheme, the lock operation generates 2 messages: one message to the server, and another from the server back to the requester (indicating that the requester can now access the segment). The unlock operation generates one message to the server. However, accessing the segment may generate several messages. On a read-fault, 2 messages are generated: one to the current holder of the segment, and another from the holder to the requester (the second message includes the segment). In addition to the 2 messages required to bring the segment from the current holder to the requester, a write-fault also generates $O(r)$ invalidation messages, where r is the number of readers. The invalidation messages are required because process synchronization is separated from consistency maintenance—the fact that the lock server gave permission for access to the segment is unknown to the consistency maintenance algorithm. The table below summarizes the comparison in terms of the number of messages:

	Number of messages	
	Reader	Writer
Our Scheme	3	3
Li's Scheme	5	$5+O(r)$

8 Conclusion

The abstraction of distributed shared memory is attractive from the point of view of object-based systems such as Clouds as well as other software architectures that use the shared memory paradigm for process communication. Such architectures would benefit considerably if the underlying hardware were to provide some transparent mechanism for efficient access and consistency maintenance of the distributed shared memory. We presented an organization and mechanisms for supporting this abstraction. The novel features of our approach are the use of distributed

shared memory as an alternative to RPC, and the exploitation of process synchronization to simplify consistency maintenance.

References

- [1] M. Ahamad and A. J. Bernstein. An application of name based addressing to low level distributed algorithms. *IEEE Transactions on Software Engineering*, SE-11(1):59–67, January 1985.
- [2] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden system: a technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–58, January 1985.
- [3] J. Archibald and J. Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.
- [4] F. Baskett, J. H. Howard, and J. T. Montague. Task communication in Demos. In *Proceedings of the 6th Symposium on Operating Systems Principles*, pages 23–31, November 1977.
- [5] J. M. Bernabéu Aubán, P. W. Hutto, and M. Y. A. Khalidi. *The Architecture of the Ra Kernel*. Technical Report GIT-ICS-87/35, School of Information and Computer Science, Georgia Institute of Technology, December 1987.
- [6] J. M. Bernabéu Aubán, P. W. Hutto, M. Y. A. Khalidi, M. Ahamad, W. F. Appelbe, P. Dasgupta, R. J. LeBlanc, and U. Ramachandran. Clouds — a distributed, object-based operating system: architecture and kernel implementation. In *European UNIX systems User Group Autumn Conference*, EUUG, October 1988.
- [7] J. M. Bernabéu Aubán, P. W. Hutto, M. Y. A. Khalidi, M. Ahamad, W. F. Appelbe, P. Dasgupta, R. J. LeBlanc, and U. Ramachandran. The architecture of *Rai*: a kernel for *Clouds*. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, January 1989.
- [8] K. P. Birman, T. A. Joseph, F. Schmuck, and P. Stephenson. *Programming with Shared Bulletin Boards in Asynchronous Distributed Systems*. Technical Report 86-772, Cornell University, Dept. Of Computer Science, August 1986.
- [9] R. Bisiani and A. Forin. Architectural support for multi-language parallel programming on heterogeneous systems. In *Proceedings 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOSII)*, pages 21–30, Palo Alto, CA, October 1987.
- [10] N. Carriero and D. Gelernter. The S/Net's Linda kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.
- [11] D. R. Cheriton. Problem-oriented shared memory: a decentralized approach to distributed systems design. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 190–197, May 1986.
- [12] D. R. Cheriton. VMTP: a transport protocol for the next generation of communication systems. In *Proceedings of SIGCOMM '86*, August 1986.

- D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77-107, May 1985.
- D. R. Cheriton and W. Zwaenepoel. The distributed V kernel and its performance for diskless workstations. *Operating Systems Review*, 17(5):128-140, October 1983.
- P. J. Denning. On modeling program behavior. In *Proceedings of the Spring Joint Computer Conference, Volume 40*, pages 937-944, AFIPS Press, Arlington, Va., 1972.
- Encore Computer Corporation. *Multimax Technical Summary*. Cedar Hill St., Marlboro, MA 01752, 1986.
- B. D. Fleisch. Distributed shared memory in a loosely coupled distributed system. In *Spring COMPCON '88*, 1988.
- D. Gelernter. Generative communications in Linda. *ACM Transactions on Programming Languages and Systems*, (1):80-112, January 1985.
- R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 124-131, June 1983.
- Haskin, Y. Malachi, W. Sawdon, and G. Chan. Recovery management in QuickSilver. In *Proceedings of the 8th Symposium on Operating Systems Principles*, November 1987.
- Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained ability in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109-133, February 1988.
- Katz, S. Eggers, D. A. Wood, C. Perkins, and R. G. Shelton. Implementing a cache consistency protocol. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 276-283, June 1985.
- Y. A. Khalidi. *Hardware Support for Distributed Object Systems*. PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1989. In preparation.
- and P. Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, pages 229-239, August 1986.
- Northcutt. *Mechanisms for Reliable Distributed Real-Time Operating Systems*. Volume 16 of *Perspectives in Computing*, Academic Press, 1987.
- Popek et al. LOCUS: a network transparent, highly distributed system. In *Proceedings of the 8th Symposium on Operating System Principles*, pages 169-177, October 1981.
- Ramachandran. *Hardware Support for Interprocess Communication*. PhD thesis, Computer Sciences Department, University of Wisconsin - Madison, September 1986.
- Ramachandran, M. Ahmad, and M. Y. A. Khalidi. *Unsynchronization and Data Transfer in Maintaining Consistency of Distributed Shared Memory*. Technical Report GIT-ICS-88/23, School of Information and Computer Science, Georgia Institute of Technology, June 1988.
- Ramachandran and M. Y. A. Khalidi. A measurement study of hardware support for object invocation. *IEEE Computer Graphics and Applications*, 1989. To appear. Available as Technical Report GIT-ICS-88/21.
- [30] U. Ramachandran and M. Y. A. Khalidi. *Programming with Distributed Shared Memory*. Technical Report GIT-ICS-88/38, School of Information and Computer Science, Georgia Institute of Technology, October 1988.
- [31] R. Rashid and G. Robertson. Accent: a communication oriented network operating system kernel. In *Proceedings of the 8th Symposium on Operating Systems Principles*, pages 64-75, December 1981.
- [32] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOSII)*, pages 31-39, Palo Alto, CA, October 1987.
- [33] R. E. Schantz, R. H. Thomas, and G. Bono. The architecture of the Cronus distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986.
- [34] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 1986.
- [35] Sun Microsystems. *Unix Interface Reference Manual*. Mountain View, California, 1986.
- [36] R. van Renesse, H. van Staveren, and A. S. Tanenbaum. Performance of the world's fastest distributed operating system. *Operating Systems Review*, 22(4):25-34, October 1988.
- [37] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Peterson, and F. Pollack. Hydra: the kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337-345, June 1974.
- [38] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th Symposium on Operating Systems Principles*, November 1987.
- [39] E. Zayas. Attacking the process migration bottleneck. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 13-24, November 1987.

PAPER [RKS92]

A Measurement-based Study of Hardware Support for Object Invocation

UMAKISHORE RAMACHANDRAN AND M. YOUSEF AMIN KHALIDI
School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA
30332-0280, U.S.A.

SUMMARY

The object invocation paradigm is attractive for structuring distributed systems. Distributed object-based operating systems view the resources of the system as a collection of objects. Object invocation is the primary mechanism in such systems, and is often used as a yardstick for measuring the system performance. However, existing systems of this flavour exhibit poor performance due to the mismatch between the requirements of the object invocation mechanism and the machine architecture. Through measurements of an existing object-based kernel, we present a breakdown of the costs involved in implementing the object invocation mechanism. The measurements suggest architectural solutions to improve the performance of such systems. We present our preliminary studies towards providing hardware support for the object invocation mechanism.

KEY WORDS Object invocation Measurements Hardware support Distributed operating systems

INTRODUCTION

This paper explores architectural support to improve the performance of object-based distributed operating systems. The hardware environment consists of a collection of computing nodes interconnected by a local area network. There are one or more processors and a certain amount of memory in each node. Nodes do not share memory; message exchange across the network is the only mechanism for communication between them.

Operating system structures for a distributed environment follow one of two paradigms: *message passing* or *object invocation*. Message-based operating systems¹⁻⁴ place a *message-passing kernel* on each node, supporting processes and communication between them via explicit messages. This kernel supports both *local communication*—communication between processes on the same node—and *non-local* or *remote* communication, sometimes implemented via a distinguished network manager process. In a traditional system such as Unix, access to system services is requested via *protected procedure calls*, whereas in a message-based operating system it is requested via *message passing*. Message-based operating systems are attractive for structuring distributed systems due to the separation of policy, encoded in server processes, from mechanism in the kernel.

Object-based distributed operating systems⁵⁻⁸ view the resources of the system as a collection of objects. Objects are similar to abstract data types, and are written as individual modules composed of the specific operations that define their interface.⁸ Access to system services is requested by invocation on the appropriate system object. In this sense, object-based distributed operating systems combine the advantages of both the traditional systems and the message-based systems. The invocation mechanism is similar to a protected procedure call, and objects encapsulate functionality similar to the server processes of message-based systems. Object invocation is the fundamental facility in object-based systems, and the speed of object invocation is often used as a yardstick for measuring the performance of such systems. The objective of this research is to understand the costs incurred in object invocation, and to propose hardware and software mechanisms to reduce these costs, and hence improve the performance of object-based systems.

*Clouds*⁵ is used as the test bed for evaluating our research ideas. Although the issues investigated in this research are presented in the context of *Clouds*, they are general and apply to other object-based systems.

Clouds is an object-based distributed operating system being developed at Georgia Tech. *Objects* and *threads* are the basic building blocks in *Clouds*. Objects are passive and persistent entities in the system. An object is the encapsulation of the *code* and *data* structures needed to implement the *entry points* in the object. These entry points provide the procedural interface for an activity to execute the code in an object. This code may itself call entry points in the same or other objects. Objects are *disjoint* partitions of a global virtual space that spans the entire network. Objects consist of one or more of the following types of areas: read-only code, read-only data, shared read-write data. These areas are referred to as *segments*.

Threads are the only active entities in the system. A thread is a unit of activity from the user's perspective. Upon creation, a thread starts executing in an object. A thread enters an object by invoking an entry point in the object. It then executes the code in the entry point, and returns to the caller object. The binding of an object invocation to an entry point in an object takes place at execution time, and more than one invocation may execute in the same object concurrently. Figure 1 shows the model of computation in *Clouds*. A thread is comparable to a process as defined in many conventional systems, with the exception that a thread may span machine boundaries. For the purposes of this paper, a process is synonymous with a thread.

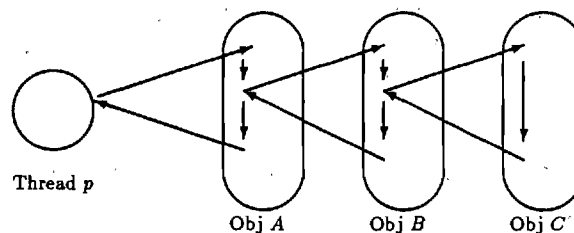


Figure 1. Model of computation

MEASUREMENTS

A performance study of several object-based operating systems is desirable to ensure that we are not discovering coding inefficiencies of one operating system, but instead see a trend that is common to all systems. Unfortunately, there are very few systems, implemented on bare hardware, that can be helpful in such a study. Alpha⁸ and Clouds⁹ are appropriate systems to study, but we only have access to the Clouds kernel. Therefore, the Clouds kernel is used as the system to study, but necessary care must be taken so as not to draw the wrong conclusions when analysing the Clouds implementation. The implementation may hide the real costs of object invocation. Therefore, a two-step strategy is followed:

1. Extensive profiling of the current implementation is done. Implementation inefficiencies are identified, and modifications to the kernel are made to remove these inefficiencies.
2. The modified kernel is then analysed, and costs intrinsic to object-based systems implemented on conventional machines are identified.

To identify and eliminate all inefficiencies amounts to rewriting the kernel. Therefore, an effort is made to remove glaring inefficiencies and to conduct the experiments in such a manner as to exercise only the portions of the kernel that we are interested in. For example, by bringing objects into main memory before starting an experiment, the performance of secondary storage is masked out. It is to be noted that a new implementation is currently under way based on our experience with the existing system. The new implementation is discussed in a subsequent section.

Overview of the invocation mechanism in Clouds

Local invocation in passive-object systems involves mapping the required memory segments of the object into the address space of the invoking thread. Remote invocation is implemented as a local invocation at the remote node where the invoked object resides, similar to a remote procedure call.¹⁰

The Clouds kernel is implemented on the Vax-11 architecture. The Vax architecture provides paged virtual memory, with the system kernel mapped into one half of the memory space (*system space*) and the currently executing process mapped into the other half (*process space*). The process space is further divided into the Vax-designated P0 and P1 spaces. Program images and most of their data reside in P0 space, whereas P1 space contains the process stacks and other data.¹¹ A page table that resides in physical memory describes the system space, and two separate page tables describe the currently mapped process space. The process page tables reside in system space virtual memory. The hardware provides a *translation look-aside buffer* (TLB) for caching recent virtual-to-physical memory translations. The TLB is split into two halves: one half caches translations from the system space, whereas the other half caches translations from the process space. The latter part is invalidated on each process context switch.

Local object invocation is considered first. When a thread t executing in object O_1 invokes an entry point in object O_2 , the following steps are taken: the calling object O_1 constructs two argument lists (*arglists*), one for transferring arguments out, and the other to receive output parameters from object O_2 . Each arglist consists of a count field, an overflow indicator field, and an array of argument descriptor records. Each

data descriptor record specifies one argument or a pointer to an argument (see Reference 12 for more details). After constructing the arglists, thread t enters the kernel through a protected system call (trap). The parameters to the system call include the starting addresses of two arglists, a capability to object O_2 , and the number of the entry point to invoke in O_2 . The kernel then calls the ObjInvoke routine that uses the capability to search an in-memory system table for O_2 . If a descriptor for O_2 is not found, local secondary storage is searched. When O_2 is found, its descriptor is read into memory, and the information contained in the object's descriptor is used to construct a P0 page table for the object. The kernel then copies the arguments onto the process stack, allocates an ObjRec structure that holds information about O_1 , and links it to the *process control block* (PCB) in a LIFO manner to be used later during object return. A new mapping of the P0 space is set up, and a process context switch that actually maps O_2 into P0 is executed. A side-effect of the context switch is the invalidation of the process's TLB. Execution continues in the ObjInvoke routine which returns to the trap handling routine, which in turn 'returns' to object O_2 to start the invocation.

Before O_2 returns to O_1 , an arglist of return parameters is constructed. The thread t then enters the kernel through the protected system call interface, passing back the starting address of the return arglist and a success/error indication. The kernel calls ObjReturn to perform the following steps: the first ObjRec is dequeued from the PCB, and the information contained in it is used to locate O_1 . The return parameters are copied into a temporary area, a new mapping of the P0 space is set up, and a context switch is executed. The return parameters are then copied into the locations specified by the 'out' arglist. ObjReturn then returns to O_1 , via the trap handling code.

If O_2 is not found locally when thread t enters the kernel during object call, O_2 is assumed to be remote and an invocation request is broadcast to other nodes. The RPC server on the node that has O_2 acknowledges the invocation request and creates a local slave process to invoke O_2 on behalf of t . The slave process proceeds to invoke O_2 locally, and when the invocation completes, it sends the return arglist back to the invoking node. When the return arglist is received, it is acknowledged, and ObjReturn is called to return to O_1 . The two important differences between local and remote invocation are the necessity to send messages through the local area network, and the need to employ a slave process at the remote node to perform the invocation.

Measurements and improvements

We added a profiling facility to the Clouds kernel. The mechanism has two components:

1. The program counter (PC) is sampled every clock tick (10ms). Each time the PC is sampled, a counter corresponding to the value of the PC is incremented. Each 4 bytes of kernel code has a corresponding counter, and a separate counter is incremented when the value of the PC falls in user space.
2. For each procedure call, the caller is noted, plus a count is maintained of the number of times the procedure has been called by this caller.

The PC sampling and the procedure call counts are collected during run-time. This information is then used to construct a graph of the dynamic call sequence. The dynamic call graph along with the symbol table and the static graph of the program are then used to produce a detailed analysis of the kernel during the time when profiling

is enabled. For each procedure, the information produced includes the percentage of the total run time spent in it, the number of times it is called, and the percentage of the time spent in each of its children. The algorithms used to build the dynamic and static graphs, and to calculate the time spent in each procedure are based on the Unix utility gprof.

In the following experiments, the objects involved are brought in memory to factor out the secondary storage access, and the profiling is done on an otherwise idle system. The costs of occasional reception of broadcast messages from the Ethernet plus the cost of the profiling code itself are factored out of the results.

Local invocation

To measure the costs involved in object invocation, two objects Caller and Callee are used in the following experiments. In each experiment, five separate runs are made, with object Caller calling an entry point in object Callee 10,000 times, transferring-in 0 to 4096 bytes of data. Object Callee transfers-out 0 to 2048 bytes. Table I lists the cost of each object invocation—the sum of kernel and user times. Tables II and III

Table I. Local object invocation costs

Bytes (in + out)	Time (ms)
0	3.4910
32 + 16	3.6040
64 + 32	3.6430
128 + 64	3.6710
256 + 128	3.7670
512 + 256	3.8770
1024 + 512	4.3020
2048 + 1024	5.1580
4096 + 2048	6.8220

Table II. Local object invocation, 0 bytes

	Time, ms	Percentage
Total kernel mode time	2671	100.0
Trap handling	160	6.0
ObjInvoke		
Locating object in memory	97	3.6
Checking rights + context switch + misc	265	9.9
Processing args	99	3.7
Allocating the ObjRec structure	769	28.8
Disabling and enabling interrupts	287	10.7
ObjReturn		
Context switch + misc	215	8.0
Processing Args	176	6.6
Deallocating ObjRec structure	285	10.7
Disabling and enabling interrupts	287	10.7
Other	31	1.2
User mode time	820	

Table III. Local object invocation, 1024 bytes

	Time, ms	Percentage
Total kernel mode time	3374	100.0
Trap handling	160	4.7
ObjInvoke		
Locating object in memory	97	2.9
Checking rights + context switch + misc	265	7.9
Processing args	463	13.7
Allocating the ObjRec structure	760	22.5
Disabling and enabling interrupts	297	8.8
ObjReturn		
Context switch + misc	224	6.6
Processing args	477	14.1
Deallocating ObjRec structure	290	8.6
Disabling and enabling interrupts	297	8.8
Other	44	1.3
User mode time	928	

give a breakdown of the total object call time for the two cases of 0 and 1024 bytes of transferred-in data, respectively. (In Tables II and III, 'misc' refers to housekeeping activities performed by the kernel, such as initializing the contents of the ObjRec structure.)

Allocating/deallocating the ObjRec structure and enabling/disabling interrupts account for 60.9 per cent of the total kernel time (Table II). This high cost is an artefact of two glaring inefficiencies in the current implementation:

1. On each object call the system memory heap is accessed to allocate an ObjRec structure. On object return, the heap is called to release this structure.
2. Each of the disable and enable routines, coded in assembly language, are called twice during object invocation. The objective of the disable routine is to raise the processor level while saving state. The enable routine lowers the processor level only if the previous level is lower than the current level. A counter and a variable that holds the saved processor level are used to keep track of the nesting depth of enable/disable.

These inefficiencies can be eliminated with the following modifications:

- (a) Allocating a free list of empty ObjRec structures, and accessing the heap only when the list is exhausted is expected to save most of the time spent in accessing the heap.
- (b) The enable routine can be replaced with four in-line instructions that exchange the processor level with a new level and save the old level on the current stack. Similarly, the disable routine can be replaced with three in-line instructions that set the processor level to the previous level stored on the stack.

Table IV shows the cost of object invocation after these two modifications to the kernel. Note that the times shown in the table are the average cost per invocation, and include kernel and user mode times. Tables V and VI show the corresponding breakdown of the object invocation time for the two cases of 0 and 1024 bytes of transferred-in data,

Table IV. Modified local object invocation costs

Bytes (in + out)	Time (ms)
0	2.2610
32 + 16	2.2490
64 + 32	2.2900
128 + 64	2.3240
256 + 128	2.4040
512 + 256	2.5990
1024 + 512	3.0220
2048 + 1024	3.7190
4096 + 2048	5.5950

Table V. Modified local object invocation, 0 bytes

	Time, ms	Percentage
Total kernel mode time	1441	100.0
Trap handling	160	11.1
ObjInvoke		
Locating object in memory	97	6.7
Checking rights + context switch + misc	265	18.4
Processing args	99	6.9
Allocating the ObjRec structure	199	13.8
ObjReturn		
Context switch + misc	215	14.9
Processing args	176	12.2
Deallocating ObjRec structure	199	13.8
Other	31	2.15
User mode time	820	

Table VI. Modified local object invocation, 1024 bytes

	Time, ms	Percentage
Total kernel mode time	2094	100.0
Trap handling	160	7.6
ObjInvoke		
Locating object in memory	97	4.6
Checking rights + context switch + misc	265	12.7
Processing args	463	22.0
Allocating the ObjRec structure	182	8.7
ObjReturn		
Context switch + misc	224	10.7
Processing args	477	22.8
Deallocating ObjRec structure	182	8.7
Other	44	2.1
User mode time	928	

respectively. The time to enable/disable interrupts is now an insignificant portion of the object invocation time and is included as part of allocating the ObjRec structure in Tables V and VI. Comparing Tables II and V, the total kernel mode time spent on object invocation is reduced by 45 per cent from an average of 2671 μ s to 1441 μ s.

Though more tuning seems possible, the decompositions of costs in Tables V and VI are representative of a typical object invocation mechanism implemented on a conventional machine. Some of the costs shown in the Tables cannot be eliminated. For example, trap handling, checking rights, and initializing the ObjRec structure consist of straight-line code that cannot be eliminated. However, the measurements point to a mismatch between the invocation mechanism and the machine architecture. This mismatch is discussed next.

TLB flushing costs

As mentioned before, two context-switch operations are needed per object invocation. Each context-switch operation costs around 10–18 per cent of kernel mode object invocation time (see Tables V and VI). Most of this cost is eliminated using a protected procedure call to effect object invocation (see next section). However, there is still a hidden cost that results from switching address spaces on invocation. This hidden cost is due to flushing the TLB, and it manifests itself mostly in user mode time. We now investigate this point in more detail.

Conventional *memory management units* (MMU) provide a process with one virtual address space. These MMUs typically have the following three components:

1. An address-space pointer that points to the address-space mapping table, also referred to as the page table. Address-space switching is usually accomplished with a hardware pointer update in most MMUs.
2. A page table that holds mapping and protection information. In most MMUs the page tables are kept in physical memory. The page table may be organized as a linear table (e.g. Vax-11), a tree (e.g. Motorola 68851,¹³ Sun-3 MMU¹⁴) or a hash table (e.g. IBM PC/RT,¹⁵ HP Precision Architecture¹⁶). The latter two organizations are aimed at minimizing the amount of physical memory space that has to be reserved for holding the page tables when the virtual memory space is large and sparse. Organizations as a linear table necessitates the use of other mechanisms, such as paging the page tables themselves, to limit the size of the page tables in physical memory.
3. A TLB that holds recent virtual to physical address translations. A TLB is a high-speed associative cache and its purpose is to eliminate the need to access page tables that are often kept in main memory on each virtual-to-physical translation. In many MMUs, TLB entries are invalidated on every address space switch.

Object-based systems such as Clouds present a problem that is not very well handled by conventional MMUs. A thread in the course of its computation traverses the virtual address spaces of several objects. The operating system assigns a unique address space for each object. When a thread invokes an entry point in an object two events occur:

1. The memory management hardware switches the address space for the currently executing thread such that memory accesses henceforth will use the address space of the invoked object.

2. The TLB is flushed due to the switch in address spaces.

When a thread returns to the caller object, these two steps are repeated to switch back to the address space of the caller object.

It is known that the frequency of context switches can affect the TLB performance significantly.^{17,18} In conventional systems such as Unix, if the TLB size is large enough, adequate hit-rate performance can be achieved by choosing an appropriate organization of the TLB.^{19,18} However, our simulation studies^{20,21} show that in object-based systems increasing the TLB beyond a certain size does not help in reducing the hit rate.

To illustrate the effects of flushing the TLB on each object call and return, an experiment is conducted on a modified version of the Clouds kernel. An object, named Same, with two entry points SameCaller and SameCallee is used. SameCaller calls SameCallee, accesses n pages of memory and repeats this sequence 10,000 times. SameCallee accesses n different pages of memory and returns to its caller. Each page in memory is 512 bytes in size. The Clouds kernel is modified such that an intra-object invocation does not result in a context switch. All the kernel steps involved in object invocation are performed, except that the actual context switch instructions are replaced with no-op instructions. Figure 2 compares the average costs of object invocation with and without TLB flushing. In the figure, t_T refers to the average total object invocation time, and t_u refers to the average time spent in user mode per object invocation. The total number of pages accessed in SameCaller or SameCallee per invocation is more than n because a number of code and stack pages are additionally accessed during each invocation.

On a context switch, the process TLB is flushed but the kernel TLB is left intact. After the context switch, almost all accesses to the process space (i.e. to thread data

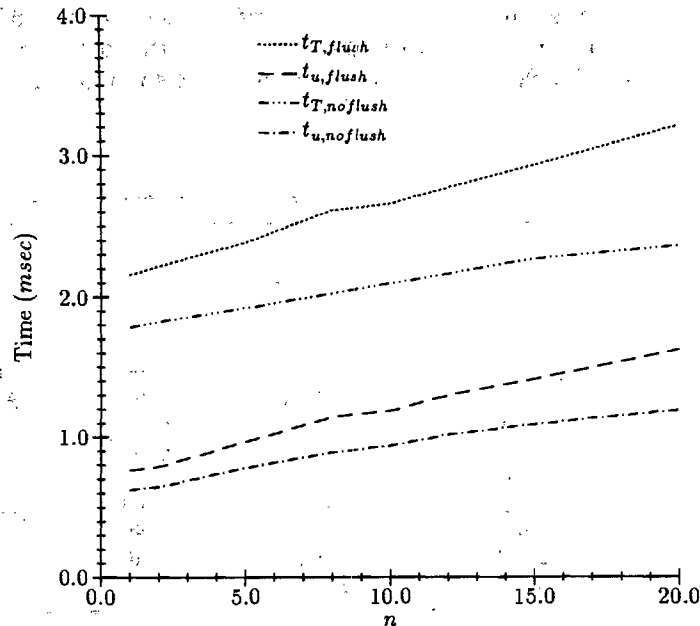


Figure 2. Comparison of TLB flushing and no flushing

and to the newly installed object) are made in user mode. Therefore, the cost of TLB flushing manifests itself mostly in user mode time t_u . There are, however, few pages in the process space that are accessed by the kernel *after* the context switch and *before* starting execution in the newly mapped object. On object call, some parameters (around 50 bytes) are pushed on the user stack (thus accessing at most two pages), and on object return the output parameters are copied into the user object. Therefore, TLB flushing affects mostly user mode time.

In Figure 2, comparing $t_{T, flush}$ to $t_{T, no flush}$ for the case of $n = 2$; shows that $t_{T, no flush}$ is 17.9 per cent smaller than $T_{T, no flush}$. For $n = 5$, the difference is 19.4 per cent. In the next section, we discuss a TLB design that supports caching of recent address translations across object invocations.

Remote invocation

To measure the cost of remote object invocation, we use the Caller and Callee objects again, but in a distributed setting, with each object loaded on a different machine. In these experiments, no data is transferred back from Callee to Caller. Caller invokes an entry point in Callee, the processor on which Caller executes remains idle until the invocation returns, and the invocation is repeated 1000 times. Column R1 in Table VII shows the total cost per object invocation for a range of transferred data. Tables VIII and IX provide a rough breakdown of the costs after eliminating CPU idle time. The following observations can be made based on these measurements:

- (a) As mentioned before, a slave process is created on the remote node to perform the invocation locally on behalf of the invoking thread. Table VIII shows that around 43 per cent of the remote node CPU time is used in reclaiming slave processes and returning them to the pool of available slaves. Though no other activity is running on the machines during the experiment, the CPU idles only 23 per cent of the total experiment time.
- (b) The general mechanism of the heap is used by the RPC and networking code to allocate memory buffers. We replaced the kernel heap with local free lists of memory blocks of the required sizes. We also modified the enable/disable routines

Table VII. RPC costs in milliseconds

Bytes (in)	RPC method		
	R1	R2	R3
0	52.69	29.90	21.90
32	52.71	27.20	22.10
64	52.74	27.40	22.60
128	52.78	28.00	23.00
256	53.00	29.00	24.10
512	54.52	32.60	27.50
1024	64.92	37.90	32.80
2048	85.08	57.50	57.50
4096	125.15	89.90	89.90

Table VIII. Remote object invocation (R1), 1024 bytes, remote node

	Time, ms	Percentage
Total time (including idle time)	64.92	
Total CPU time used	49.94	100.0
User mode time	0.60	1.2
Kernel mode time	49.34	98.8
Slave reclamation	21.31	42.7
RPC server total time		
CPU time sending/receiving messages	9.96	19.9
Starting slave	3.67	7.3
Locating object in memory	0.94	1.9
RPC return (includes CPU time to send results back)	10.04	20.1
Local object invocation (object call + return)	2.20	4.4
Miscellaneous overhead	1.22	2.4

Table IX. Remote object invocation (R1), 1024 bytes, local node

	Time, ms	Percentage
Total time (including idle time)	64.92	
Total CPU time used	13.21	100.0
User mode time	0.33	2.5
Kernel mode time	12.88	97.5
ObjInvoke	1.67	12.6
RPC_Invok		
Allocating network ports & housekeeping overhead	3.00	22.7
RPC_Send		
Sending invocation request	3.10	23.5
Receiving acknowledgement	2.01	15.2
RPC_GetResults		
Receiving results	2.05	15.5
Sending acknowledgement	0.93	7.0
Miscellaneous overhead	0.13	1.0

as previously described in local invocation. These modifications are included in the following two experiments, shown in columns R2 and R3 in Table VII, and they reduce remote invocation time by approximately 9 per cent.

To illustrate how slave reclamation affects system behaviour, the same remote invocation experiment is repeated *without* reclaiming slaves, i.e. assuming that process reclamation costs zero time. The results are shown in column R2 in Table VII. The objective is to illustrate the effect of slave reclamation on RPC cost. Note that the difference between R1 and R2, when transferring 1024 bytes, is 27.02 ms. Of this time, 21.31 ms are due to slave reclamation costs and the rest are due to the modifications to buffer allocation and the enable/disable routines. The results of eliminating the two acknowledgements²² from the RPC protocol are shown in column R3 of Table VII.

From these experiments, the following observations are made concerning remote object invocation:

- (a) Slave start/reclamation is very important and must be done fast. In the R1 case above, reclaiming slave processes is a bottle-neck that slows both the local and remote nodes. While servicing remote invocation requests, the Clouds kernel is also trying to prepare the 'dead' slaves for more work. Though the Clouds implementation of slave reclamation could be improved, there is always some cost for assigning cohort processes to perform remote invocations.
- (b) The Clouds kernel passes little additional data on an object invocation, though a sophisticated operating system is expected to pass more thread-specific data. This information may include the thread identifier name, accounting information, controlling terminal information, etc. The operating system would then use this thread-specific information to create an environment similar to the invoking thread's environment. Passing additional data and creating a new thread environment on the remote node is expected to add more time to remote procedure calls.

Thus, the additional cost in a remote (non-local) invocation involves (a) sending messages across the network, (b) setting up an environment similar to the invoking thread's environment, and (c) assigning a cohort process to do the actual work. An alternative to sending the computation to a remote node is to bring the required data to the local node. This alternative, which we call *distributed shared memory*, has been explored in other systems, such as Emerald²³ and Apollo Domain.²⁴ In a subsequent section, the mechanisms needed to support distributed shared memory are presented, and a hardware module to assist in this function is proposed.

MEMORY MANAGEMENT SUPPORT

Based on our measurements, we believe that object-based systems would benefit considerably if the machine architecture were to provide support for a process to traverse and cache recent address translations in multiple address spaces. In many MMUs, a change in address-space pointer also results in flushing the TLB. For example, the Vax *load process context* instruction changes the page table pointers of process space and flushes the process space half of the TLB. Some MMUs, such as the Motorola 68851, the Amdahl 470V/7 and the IBM 3033, associate a virtual address tag with the TLB entries, allowing entries for more than one process to be in the TLB at the same time. In the Motorola 68851,¹³ a process identifier is stored in the TLB as part of the tag. Therefore, apart from the usual replacement of entries when the TLB is full, only when a process identifier is re-used the entries corresponding to that process identifier need to be flushed from the TLB. However, these MMUs do not meet the requirements of object-based systems, as will be evident from the discussion at the end of this section.

To better support object invocation, a scheme is presented in which the address space switch is effected by a protected procedure call mechanism, and virtual-to-physical address translations are cached across object invocations.²⁰ This scheme, shown in Figure 3, is an extension of the one used in the Motorola 68851. The machine virtual address space is partitioned into three regions: the K space or kernel space, the P space or process space, where the currently running process has its stack and other

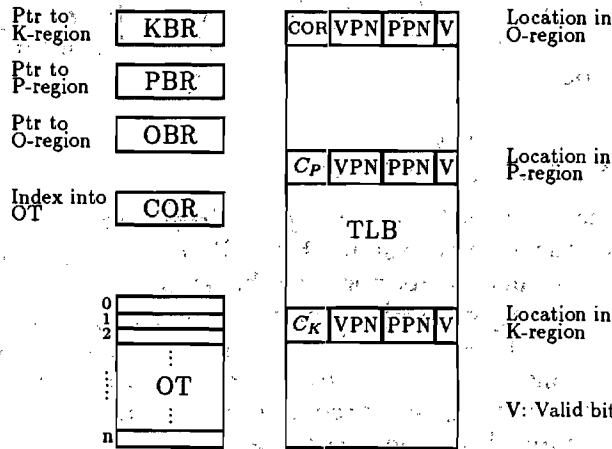


Figure 3. Proposed MMU structure

process-related data; and the O space or object space, where the currently invoked object resides. Virtual memory is organized in pages. Our proposed MMU uses a page table in physical memory to translate virtual page numbers, VPN, to physical page numbers, PPN. In the description to follow, 'address space pointer' refers to the base of the page table for a particular address space.

Three base registers, KBR, PBR and OBR, point to address spaces K, P and O, respectively. An *object table*, OT, holds the address space pointers of recently invoked objects. Every invocation results in the following:

1. OBR is set to point to the address space of the invoked object by the operating system.
2. The MMU searches OT for an entry with the same value as OBR. If such an entry is found, the index of this entry is used as an *object identifier*, and a *current object register*, COR in the MMU is set to this value. If no matching entry is found in the OT, an entry is re-used, and all TLB entries with the same object identifier as the index of the chosen entry are flushed.

A return from an object invocation is similar, with the operating system setting the OBR to the address space of the caller object. As is evident from this description, with our MMU the invocation is effected by a protected procedure call instead of a context switch.

The TLB entries are tagged with (object identifier, VPN) for O space translations. Translations in the P and K spaces are cached in the TLB with (C_P, VPN) and (C_K, VPN) as the tags respectively, where C_P and C_K are MMU constants outside the range of OT indices. The P space entries (with C_P tag) are flushed on every process switch.

Virtual addresses generated by the CPU are translated as follows: an address from the O space is prefixed with the contents of COR, and an associative search of the TLB is performed. A hit occurs when the (COR, VPN) pair matches the TLB tag. An address from the P or K space is prefixed with the appropriate MMU constant, either C_P or C_K, for TLB look-up. A hit occurs when the (Const, VPN) pair matches the TLB tag.

When an address-space pointer is re-used (e.g. when an object is destroyed), system software instructs the MMU to flush any matching entry in the OT with the same contents as the address space pointer. The MMU, in turn, flushes any corresponding entries in the TLB.

Evaluation

We evaluated our scheme using trace-driven simulation, and compared its performance to the usual case of flushing the TLB on each object call and return (see Reference 20 for more details). In performing this study, we were faced with the lack of traces of programs running on object-based systems. The current Clouds implementation runs only a few 'toy' programs, and does not support instruction tracing.

The following approach was used to generate the input for our simulator. A number of programs written in the C programming language, each of which consisted of several separately compiled modules, were executed under Unix on a Vax-11, and an address trace was generated for each program. The address traces were in turn used to drive a simulator that treated inter-module calls as object calls. Whenever a call (return) crossed from one module to another, it was treated as an object call (return). Intra-module calls were treated as normal procedure calls. Calls to library routines, as well as data accessed while executing inside an object were treated as part of that object's virtual address space.

Modular programs tend to group related functions and data in the same module, and to pass data by value across modules. This behaviour is very similar to what we expect to have on Clouds, with separate objects replacing program modules. The test programs used in the simulation follow this style of programming, with an average size of 500 lines of code per module, which is about the size of a Clouds object. We, therefore, believe that traces generated as explained above are appropriate for evaluating our scheme. The traces, however, lack kernel mode addresses, and therefore our simulation does not consider translation of K space addresses.

The results of our performance study are summarized below, where T1 refers to our scheme, and T0 refers to the usual case of flushing the TLB on object invocations:

1. For a given TLB size, the miss ratio of T1 is consistently less than the miss ratio T0. For a TLB size of 128, the miss ratio of T1 is 100 times less than the miss ratio of T0.
2. The T1 miss ratio decreases as the TLB size increases beyond the point when the T0 scheme miss ratio levels off. T1 uses the additional TLB entries to cache more address translations that may be used in the future, unlike T0 which flushes cached translations on each object call and return.
3. Depending on ratio T_h/T_m (where T_h is the hit cycle time and T_m is the miss cycle time), the overall percentage improvement of T1 over T0 ranges from around 15 to 30 per cent.

The memory management unit that we presented is tailored to support object invocation. It is an engineering solution combining the features available in commercial MMUs. The strengths and weaknesses of commercial MMUs *vis-à-vis* the features required for an efficient implementation of Clouds are discussed below:

- (a) *The ability to cache TLB information across object invocation.* The cost of

flushing TLB entries at object call/return is an implicit cost that affects mostly user mode time. Systems with one large virtual address space do not require flushing the TLB across object invocations. Examples of such systems include the IBM RT/PC, HP Precision Architecture and SPUR machines.²⁵

- (b) *The ability to implement an object call/return by performing a small number of operations on the MMU.* The cost of required MMU operations is an explicit part of object invocation. This cost is quantified by the number of MMU registers, either in software or hardware, that need to be modified on each object call/return. In our proposed MMU, changing the OBR is the only operation that is required to effect an object call/return. A paper design of object invocation implementation on segmented single virtual address space systems, such as IBM RT/PC and HP Precision Architecture, reveals that only a small number of MMU operations is required. However, our experience in implementing the new Clouds kernel on the Sun-3 MMU⁹ reveals that many MMU operations are needed on this machine.
- (c) *The ability to represent sparse address spaces efficiently.* Inefficient representation of sparse address spaces results in page tables that have large memory requirements and lengthy initialization time, relative to the size of address space actually allocated. As mentioned before, page tables organized as trees or inverted tables are suitable for representing sparse address spaces, whereas linear page tables are not.
- (d) *The ability to share memory among address spaces.* This feature is required to support sharing of segments among virtual spaces. Sharing segments in our proposed scheme and other MMUs with page-tables organized as trees is easy. A segment is shared between virtual spaces by sharing a page table subtree between the different page tables. MMUs with inverted page tables (e.g. IBM RT/PC) and virtually-addressed data caches (e.g. SPUR) cannot have two different virtual addresses mapping into the same physical address (i.e. no *aliasing*). Sharing of memory between different spaces is possible only through hardware segment sharing, which imposes limitations on the number of segments per object and the way they can be shared between virtual spaces. Clouds advocates a model of programming with a large number of possibly small-sized segments. Any segment is potentially sharable among virtual spaces. Because sharing can only be at the hardware segment level, a hardware segment maps exactly one software segment. For example, in the IBM RT/PC the CPU generates 32-bit virtual addresses with the high-order four bits of the virtual address selecting one of sixteen segment registers. In a paper design of object space implementation on the IBM PC/RT, the sixteen segments are allocated as follows: seven each for O and P spaces and two for K-space. Each hardware segment maps at most one software segment.

The HP Precision Architecture organizes its virtual space differently. Each virtual address is composed of a segment register and an offset. User software can change some of the segment registers without kernel assistance. Using such an organization, the number of software segments per virtual space is not constrained by the number of hardware registers. However, user software is responsible for loading the segment registers with hardware segment numbers before accessing a segment that is not already mapped. This organization complicates user software and may result in performance degradation if the object is

composed of many segments. Moreover, the segment registers are loaded with hardware segment numbers that the kernel has to set up. The kernel has to communicate these number to user software when objects are loaded in memory, and has to make sure that no object still uses a hardware segment number that is about to be reused.

- (e) *The ability to allocate/deallocate ranges of addresses easily.* The ability to allocate/deallocate ranges of addresses is related to sharing of memory segments among spaces, because ranges of addresses are allocated/deallocated at the segment level. The number of page table and MMU operations required to allocate/deallocate a range of addresses affects the cost of attaching and detaching⁵ segments to virtual spaces. Some systems (e.g. Vax and Sun-3 MMU) require traversing a potentially large number of page-table entries to allocate or deallocate a range of addresses. Both IBM RT/PC and HP Precision Architecture adequately support this feature.

In summary, MMUs with a single large virtual space come closest to meeting the requirements of maintaining an object space. They do not require TLB flushing across object invocation, and may not require a large number of operations to implement object call/return. However, such MMUs either restrict the amount of sharing possible between virtual spaces, or introduce software complexities when sharing segments.

The requirements discussed so far are basic to managing object space. Systems such as Clouds⁹ and Argus,⁷ support the notions of atomicity of computation and recoverability of data, sometimes referred to as transactions. This notion requires that memory segments be *recoverable* if a computation needs to be aborted. Software techniques such as shadowing and logging are usually employed to implement transactions.²⁶ However, it is possible to reduce the burden on the software by providing some mechanisms in the MMU for supporting transactions. IBM RT/PC implements one such mechanism called transaction locking in its MMU.¹⁵ In Reference 27, Chang and Mergen report on implementing a transaction system using these mechanisms. We are currently evaluating the efficacy of incorporating transaction support in our MMU design. It should be noted that such a mechanism can be easily added to our present design.

DISTRIBUTED SHARED MEMORY

In a distributed object-based system, the virtual address spaces of all objects can be viewed as constituting a *global distributed shared memory*. Such a view is attractive from the perspective of software architecture since it suggests a uniform implementation of a system-wide memory-mapping mechanism. Local object invocation involves mapping the required memory segments of the object into the address space of the invoking process by installing the object as the current O space with the process's P space. The current trend in structuring distributed systems is to use a collection of diskless computational servers or workstations, and a few data servers or file servers. In such an environment, the code and data for the local invocation has to be paged-in from the data server. Further, for remote object invocation we have one of two choices: the first choice is to perform the computation at the node where the object resides through remote procedure call. The second choice is to make the invocation appear local by bringing in the segments required for the invocation. Whereas the former has to be

supported for immovable objects such as an object that reads disk blocks, the latter may be a better choice for movable objects. There are two reasons to support this belief:

- (a) the principle of locality²⁸ that suggests an invocation or other invocations in the same object may be repeated.
- (b) the reduction in computational overhead due to the elimination of slave processes management to support remote invocation at the node where the object resides.²¹

In References 29 and 21 the concept of distributed shared memory as an alternative to RPC is proposed, the algorithms to maintain *strong* and *weak* memory consistency are presented, and the algorithms are evaluated using simulation. Each object is owned by one node, and segments of an object can be at other nodes temporarily. Each node has associated with it a 'network cache'—a repository for recently accessed remote memory segments and their owners (nodes). A node that caches a segment from a remote owner is called a *keeper* of the segment.

The organization (Figure 4) proposed inside each node of the network is the following: a host that executes distributed applications; a distributed shared memory controller (DSMC) together with the network interface assists the host in mapping memory segments, both local and remote, into the virtual address spaces of the application processes. There is a minimal *kernel* on the host that traps system calls and virtual address translation faults. The DSMC is also in control of the network. The system memory logically is partitioned into two parts: one part, *object memory*, is for housing the segments of locally created objects; the other part, *network cache*, is for caching segments from remote objects. Conceptually there are two lists of process control blocks: the *hostlist* that the host looks at to schedule runnable processes and the *DSMC list* that the DSMC looks at to service memory segment requests. The host enqueues processes that fault on virtual address translation in the DSMC list. The DSMC enqueues processes that have become runnable again after the fault service in the host list.

The basic operations provided by the DSMC are get and discard. The get operation is used to fetch a segment (or a part thereof) from its owner, whereas discard is used to return a segment to its owner. The DSMC provides synchronization primitives as separate P and V semaphore operations, or as combined access and lock operations using the get and discard primitives.

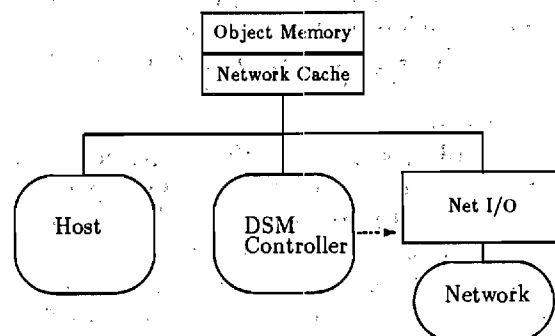


Figure 4. Hardware implementation

Using the get primitive a segment may be acquired in one of four modes: read-only, read-write, weak-read or none. Read-only mode signifies non-exclusive access but guarantees that the segment will not change until the node explicitly discards the segment. Read-write mode signifies exclusive access (for the node) with a guarantee that the segment will not be thrown away until the node explicitly discards the segment. Weak-read mode signifies non-exclusive access with no guarantee whether the segment will change or not. None mode signifies exclusive access with no guarantee whether the segment will be thrown away or not.

We evaluated our DSMC scheme and compared it to RPC using simulation.²⁹ In the simulation, the DSMC is treated as a software module that is part of the kernel. The performance study shows that

1. Over a range of object invocation locality, DSMC has significant advantages over RPC.
2. Use of distributed shared memory achieves automatic distribution of processing load, by performing the invocation locally instead of on the owner nodes.
3. Instead of addressing memory coherency and synchronization separately, applications that can be modelled as readers/writers problems benefit considerably when locking and segment access primitives are combined.

In Reference 5 the design and implementation of a native kernel that integrates distributed shared memory for network-wide memory management is discussed. The DSMC primitives are provided to the operating system as a set of mechanisms for managing memory in the network. Policy decisions, such as when to use the RPC mechanism and when to use the DSMC primitives, are in the operating system. For example, during periods of high concurrent access to an object, the operating system uses the DSMC primitives to locate the object at one node and uses RPCs to invoke the object, thereby reducing data movement across the network.

Our work is built on the large body of work that exists in maintaining cache coherence in multiprocessors.³⁰ The DSMC algorithms deal with providing the mechanisms needed to maintain consistency of shared data in a non-shared memory architecture. Kai Li³¹ addresses the same problem, wherein the entire memory is composed of untyped pages that are potentially sharable for both reads and writes. The novelty in our work is in exploiting application semantics to type the segments as read-only or read-write, and using the type specifiers in the DSMC primitives as hints for simplifying consistency maintenance. By merging process synchronization with data transfer, the DSMC primitives provide mutual exclusion for free.

Other researchers have proposed the use of shared memory as a distributed systems structuring concept.³² The Apollo Domain system²⁴ is a loosely coupled network of computers that provide the user with a view of a single-level store. This view allows programs to share files, specifying the semantics of sharing, such as exclusive/non-exclusive, at the time of opening the file. On opening the file, it is mapped into the virtual address space of the program, and henceforth reads and writes to the file are no different from simple memory reads and writes. These system architectures assume the existence underneath of a consistency preserving mechanism similar to Li's and ours. Distributed shared memory in an object-based environment plays a role similar to the one played by network file systems, such as NFS³³ and Sprite,³⁴ in conventional systems.

CONCLUSIONS AND FUTURE WORK

The Clouds kernel served well as a prototype, and enabled us to gain important insight into the requirements of the object-based model of computation. However, it has several drawbacks. It is a monolithic kernel that is hard to modify and maintain, and the implementation is very machine-dependent. In addition, message transmission in Clouds is slow. Sending messages from one process running in a local kernel to a process running in a remote kernel takes roughly 10 ms. Most of this cost can be attributed to a poor network interface.³⁵ Other systems, such as QuickSilver³ and V,¹ report numbers in the range of 6 to 1.5 ms on faster hardware.

Our notions on structuring object-based operating systems has matured since the prototype design was begun. A new kernel for Clouds, called the *Ra* kernel, has been designed and implemented.^{5,9} *Ra* runs on the Sun-3 architecture, and incorporates support for distributed shared memory. The DSMC has been implemented in software and we are currently in the process of evaluating the implementation.^{36,37} Further refinement and implementation of an MMU tailored to object-based systems that incorporates our TLB scheme, and a hardware module that implements the DSMC protocol are some of the work we have identified for future research.

ACKNOWLEDGEMENT

This work has been funded in part by NSF grants CCR-8619886 and MIP-8809268.

REFERENCES

1. D.R. Cheriton and W. Zwaenepoel, 'The distributed V kernel and its performance for diskless workstations', *Operating Systems Review*, **17**,(5) 128-140 (1983).
2. Mike Accetta, Roberta Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian and Michael Young, 'Mach: a new kernel foundation for Unix development', *Proceedings Summer Usenix*, July 1986.
3. R. Haskin, Y. Malachi, W. Sawdon and G. Chan, 'Recovery management in QuickSilver', *Proceedings of the 11th Symposium on Operating Systems Principles*, November 1987.
4. R. Rashid and G. Robertson, 'Accent: a communication oriented network operating system kernel', *Proceedings of the 8th Symposium on Operating Systems Principles*, December 1981, pp. 64-75.
5. José M. Bernabéu Aubán, Phillip W. Hutto, M. Yousef A. Khalidi, Mustaque Ahamad, William F. Appelbe, Partha Dasgupta, Richard J. LeBlanc, and Umakishore Ramachandran, 'The architecture of Ra: a kernel for Clouds', *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, January 1989.
6. Guy T. Almes, Andrew P. Black, Edward D. Lazowska and Jerre D. Noe, 'The Eden system: a technical review', *IEEE Trans. Software Engineering*, **SE-11**, (1), 43-58 (1985).
7. B. Liskov and R. Scheifler, 'Guardians and actions: linguistic support for robust, distributed programs', *Ninth Conference on Principles of Programming Languages*, 1982.
8. J. Duane Northcutt, *Mechanisms for Reliable Distributed Real-Time Operating Systems*, Volume 16 of *Perspectives in Computing*, Academic Press, 1987.
9. José M. Bernabéu Aubán, Phillip W. Hutto, M. Yousef A. Khalidi, Mustaque Ahamad, William F. Appelbe, Partha Dasgupta, Richard J. LeBlanc and Umakishore Ramachandran, 'Clouds—a distributed, object-based operating system: architecture and kernel implementation', *European UNIX systems User Group Autumn Conference*, EUUG, October 1988.
10. Andrew D. Birrell and Bruce Jay Nelson, 'Implementing remote procedure calls', *ACM Transactions on Computer Systems*, **2**, (1), 39-59 (1984).
11. Digital Equipment Corporation, *VAX Architecture Handbook*, 1986.
12. E.H. Spafford, 'Kernel structures for a distributed operating system', *Ph.D. Thesis*, School of Information and Computer Science, Georgia Institute of Technology, 1986. Available as *Technical Report GIT-ICS-86/16*.

13. Motorola MC68851 Paged Memory Management Unit User's Manual, Prentice-Hall, 1986.
14. Sun Microsystems Inc., *Writing Device Drivers for the Sun workstation*, 1986, Manual No. 800-1780-10.
15. International Business Machines, *IBM RT/PC Hardware Technical Reference Manual, Volume I*, 1986, Manual number 74X9961.
16. Michael J. Mahon, Ruby Bei-Loh Lee, Terrence C. Miller, Jerome C. Huck and William R. Bryg, 'Hewlett-Packard precision architecture: the processor', *Hewlett-Packard Journal*, August 1986, pp. 4-21.
17. M. Satyanarayanan and Dileep P. Bhandarkar, 'Design trade-offs in VAX-11 translation buffer organization', *IEEE Computer*, December 1981, p. 103-111.
18. Alan Jay Smith, 'Cache memories', *Computing Surveys*, **14**, (3), 473-530 (1982).
19. Cedell A. Alexander, William M. Keshlear and Faye Briggs, 'Translation buffer performance in a Unix environment', *Computer Architecture News*, **13**, (5), 2-14 (1985).
20. Umakishore Ramachandran and M. Yousef A. Khalidi, 'Memory management support for object invocation', *Technical Report GIT-ICS-88/3*, School of Information and Computer Science, Georgia Institute of Technology, January 1988.
21. M. Yousef A. Khalidi, 'Hardware support for distributed object-based systems', *Ph.D. Research Proposal*, School of Information and Computer Science, Georgia Institute of Technology, 1988.
22. J.H. Saltzer, D.P. Reed and D.D. Clark, 'End-to-end arguments in system design', *ACM Trans. Computer Systems*, **2**, (4), 277-288 (1984).
23. Eric Jul, Henry Levy, Norman Hutchinson and Andrew Black, 'Fine-grained mobility in the Emerald system', *ACM Trans. Computer Systems*, **6**, (1), 109-133 (1988).
24. P.J. Leach, P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson and B.J. Stumpf, 'The architecture of an integrated local network', *IEEE J. Selected Areas in Communications*, **SAC-1**, (5), 842-856 (1983).
25. David A. Wood, Susan Eggers and Garth Gibson, 'SPUR memory system architecture', *Technical Report UCB/CSD 87/394*, Computer Science Division, University of California, Berkeley, December 1987.
26. Philip A. Bernstein, Vassos Hadzilacos and Nathan Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.
27. Albert Chang and Mark F. Mergen, '801 storage: architecture and programming', *ACM Trans. Computer Systems*, **6**, (1), 28-50 (1988).
28. P.J. Denning, 'On modeling program behavior', *Proceedings of the Spring Joint Computer Conference*, AFIPS Press, Arlington, Va., 1972, Vol. 40, pp. 937-944.
29. Umakishore Ramachandran, Mustafaq Ahmad and M. Yousef A. Khalidi, 'Unifying synchronization and data transfer in maintaining coherence of distributed shared memory', *Technical Report GIT-ICS-88/23*, School of Information and Computer Science, Georgia Institute of Technology, June 1988.
30. J. Archibald and J. Baer, 'Cache coherence protocols: evaluation using a multiprocessor simulation model', *ACM Trans. Computer Systems*, **4**, (4), 273-298 (1986).
31. Kai Li and Paul Hudak, 'Memory coherence in shared virtual memory systems', *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, ACM, August 1986, pp. 229-239.
32. D.R. Cheriton, 'Problem-oriented shared memory: a decentralized approach to distributed systems design', *Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986, pp. 190-197.
33. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh and B. Lyon, 'Design and implementation of the Sun network filesystem', *Proceedings of the USENIX 1985 Summer Conference*, June 1985, pp. 119-130.
34. M.N. Nelson, B.B. Welch and J.K. Ousterhout, 'Caching in the Sprite network file system', *ACM Trans. Computer Systems*, **6**, (1), 134-154 (1988).
35. Digital Equipment Corporation, *DEUNA User's Guide*, 1983. Manual No. EK-DEUNA-UG-001.
36. Umakishore Ramachandran and M. Yousef Amin Khalidi, 'An implementation of distributed shared memory', *Technical Report GIT-ICS-88/50*, School of Information and Computer Science, Georgia Institute of Technology, December 1988.
37. Umakishore Ramachandran and M. Yousef Amin Khalidi, 'Programming with distributed shared memory', *Technical Report GIT-ICS-88/38*, School of Information and Computer Science, Georgia Institute of Technology, October 1988.

Hardware Support for Interprocess Communication

UMAKISHORE RAMACHANDRAN, MARVIN SOLOMON, AND MARY K. VERNON, MEMBER, IEEE

PAPER [RSV90]

Abstract—In recent years there has been increasing interest in message-based operating systems, particularly in distributed environments. Such systems consist of a small message-passing kernel supporting a collection of *system server* processes that provide such services as resource management, file service, and global communications. For such an architecture to be practical, it is essential that basic message exchanges be fast, since they often replace what would be a simple procedure call or “kernel call” in a more traditional system. Careful study of several operating systems shows that the limiting factor, especially for small messages, is typically not network bandwidth but processing overhead. Therefore, we propose using a special-purpose coprocessor to support message passing. Our research has two parts. First, we partitioned an actual message-based operating system into *computation* and *communication* parts, executing, respectively, on a *host* and a *message coprocessor* interacting through shared queues, and measured its performance on a multiprocessor. Second, we designed hardware support in the form of a special-purpose *smart bus* and *smart shared memory* and demonstrated the benefits of these components through analytical modeling using *Generalized Timed Petri Nets*. Our analysis shows good agreement with the experimental results and indicates that substantial benefits may be obtained from both the partitioning of the software between the host and the message coprocessor and the addition of a small amount of special-purpose hardware.

Index Terms—Architecture support, bus protocol, distributed systems, measurements, message-based operating systems, performance Petri nets, system architecture.

I. INTRODUCTION

WE are interested in providing hardware support to improve the performance of distributed systems. The hardware environment consists of a collection of computing nodes interconnected by a local area network (LAN). There are one or more processors and a certain amount of memory in each node. The nodes do not share memory; message exchange across the network is the only mechanism for communication between them. Many recent operating systems designs for such an environment [6], [25]–[27] place a *message-passing kernel* on each node, supporting processes and communication between them via explicit messages. This kernel supports both *local communication*—communication between processes on the same node—and *nonlocal or remote communication* (sometimes implemented via a distinguished network manager process).

Manuscript received April 21, 1989; revised March 17, 1990. This work was supported in part by the Defense Advanced Research Projects Agency, the National Science Foundation, and International Business Machines. An earlier version was presented at the 14th International Symposium on Computer Architecture.

U. Ramachandran is with the School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332.

M. Solomon and M. K. Vernon are with the Department of Computer Science, University of Wisconsin-Madison, Madison, WI 53706.

IEEE Log Number 9036217.

Access to system services is requested via protected *procedure calls* in a traditional system, whereas in a message-based operating system it is requested via *message passing*. While a simple procedure call costs just a few instructions, and a protected procedure call (kernel call) costs a few hundred instructions, IPC costs a few thousand instructions in several systems that we studied [20]. Since message exchange is the basic kernel mechanism in message-based operating systems, the performance of the system depends crucially on the rate of message exchange. Our measurements (see Section II and [24]) as well as the measurements of others [6], [11] indicate that for small messages (which make up the vast majority of all messages sent [6]), the limiting factor is the high processing overhead that is incurred in message passing rather than limited network bandwidth or the time to copy messages from buffer to buffer.

There are two important figures of merit in this environment: *roundtrip time*, and *message throughput*. Roundtrip time is the elapsed time seen by an application between sending a message and receiving a reply from the intended receiving process. This figure of merit affects an individual application's performance. Message throughput is a global figure of merit that determines the performance of the entire system. Informally, it is the number of messages that the system handles per unit time. We show that a modest amount of additional hardware can significantly improve message throughput and average roundtrip time in a multiprogramming environment. We also show that additional hardware support in the form of high-level bus primitives affords even greater improvement in communication subsystem performance.

A. Background and Related Work

Available hardware support [1], [14] and previous modeling studies [12], [29] address the issue of off-loading communication protocols onto front-end processors, and provide evidence that this approach can have a significant performance payoff. However, these and other previous proposals of hardware support for interprocess communication (see survey in [21]) are more limited than the study reported in this paper in several respects.

First, previous work generally assumes “communication” to be the work that is performed by the operating system to satisfy nonlocal requests. However, for message-based operating systems, measurements by us (see Section II and [24]) and others [6] show that there is a high processing overhead for local communication as well. Existing hardware support for interprocess communication takes the form of

1) operations on structured data types in the instruction set architecture of the processor (often through microcode) such

send and receive in Intel's iAPX 432 [8]

- 2) network interfaces with direct access to host memory (e.g., Interlan Ethernet [14])
- 3) protocol processors that implement some form of transport level protocol for off-machine communication (e.g., TP interface board from ABLE [1])
- 4) multiprocessor architectures with one processor performing message passing functions, generally for a group of processors (e.g., Kmap of Cm* [10])
- 5) bus architectures that support higher level operations (e.g., VMEbus [9]).

Second, many proposals include only limited, low-level support for communication, leaving out support for operations such as address translation, control block manipulation, and kernel buffering, which account for a substantial portion of message passing overhead.

Finally, a front-end processor for a specific network protocol (such as TCP/IP [19]) may not mesh well with the operating system primitives, and therefore may incur higher overhead than necessary. The problem needs to be addressed at a much higher level.

Proposed Organization

The organization inside each node is shown in Fig. 1. There is a *host* in each node that executes the message-based operating system and the applications. The *shared bus*, the *shared memory*, the *message coprocessor*, and the *network interfaces* together function as a single unit in assisting the host in message-passing activities. The host, the message coprocessor, and the network interfaces interact and synchronize via shared memory. This organization is similar to the ones assumed in the studies of network front-ends such as [12], and in commercial products such as the ABLE Easyway Port [1]. [7], the authors report on a similar hardware organization implementing the run-time support for an extended version of Hoare's CSP. However, what distinguishes our work from their work is the level of message-passing support. In this research, we provide hardware support for message-passing at the level of the operating system primitives. Our solution to the problem suggests a system architecture that has a software aspect and a hardware aspect.

Our organization is aimed at improving the message throughput and reducing the roundtrip time by providing concurrent processing support for message passing. This organization raises two main questions.

- 1) How should the message-based operating system be *partitioned* between the host and the message coprocessor?
- 2) What kind of hardware assist is appropriate to support interaction between the host, the message coprocessor, and network devices?

Our objectives in this research are a) to answer these questions, and b) to quantify the performance improvement due to our system architecture over an organization that does not use such hardware assistance.

For a given semantics of interprocess communication, roundtrip time reflects the processing overhead that is incurred before the message transfer between the sender and the receiver. If there are only two processes communicating with

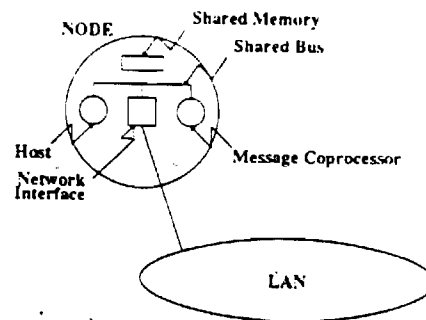


Fig. 1. Node architecture.

each other in the entire system, shifting communication tasks to a coprocessor will lead to an *increase* in processing overhead due to the interaction between the host and the message coprocessor. However, we show that this increase can be kept very small by a careful partitioning of the message-based operating system. Moreover, through our performance analysis, we show that the per-process roundtrip time improves as a result of improving the message throughput when there are several processes communicating with one another.

The rest of the paper is organized as follows. We summarize our findings from a study of four distributed systems in Section II. The software partition that we implemented is discussed in Section III, and Section IV describes the hardware organization. Performance analysis is presented in Section V, and concluding remarks are given in Section VI.

II. MEASUREMENTS OF DISTRIBUTED SYSTEMS

We studied the design and implementation of four operating systems in detail: *Charlotte* [3], *Jasmin* [15], *Quicksilver*¹ [26], and *Unix*. We profiled them (see [24] for more details) to ensure that we are not discovering coding inefficiencies of one operating system but see a trend that is common to all these systems. Our model of a distributed system assumes that processes communicate via explicit messages and that system services are provided by trusted server processes (as opposed to a monolithic kernel). *Charlotte*, *Jasmin*, and *925* belong to this model. However, all of these systems are experimental research projects. Therefore, we also studied *Unix*, which is not a message-based operating system, to see whether operating systems in extensive use suffer from similar problems.

To summarize the results from the studies, there are two important characteristics of distributed message-based operating systems.

Structure: System services in a distributed system are accomplished by a combination of *computation* and *communication*. By computation we mean processing done on behalf of *servers*. By communication we mean the system code that has to be executed to process a communication request.

Communication Overhead: There is a fixed overhead incurred in communication (independent of the message size) that can be decomposed into components such as checking the validity of an IPC call, addressing and manipulating control blocks, and short-term scheduling. There is a variable overhead due to kernel buffering that is dependent on the size of the

¹ The system we studied was an earlier version of Quicksilver called 925.

message and the number of times the message is copied from source to destination. This combined overhead is present for both local and nonlocal communication. For nonlocal communication there is additional overhead such as sending network packets, processing network interrupts, checksum calculation, and retransmission. For small messages, i.e., message size smaller than 100 bytes, copy-time is less than 20% of the total roundtrip time. For large messages, i.e., message size larger than 1000 bytes, copy-time begins to dominate the total roundtrip. Some systems (such as Accent [25] and Mach [2]) integrate virtual memory management with interprocess communication to reduce copy overhead for local message passing. However, given that copy-time accounts for less than 20% of the roundtrip time for short messages, communication overhead remains a significant performance bottleneck even in such systems.

III. SOFTWARE PARTITION

In a distributed system, users request system services by *communicating* with the *servers*. These servers *compute* to satisfy the requests possibly communicating with other servers. Through our profiling studies, we showed that a large percentage of the roundtrip time can be attributed to short-term process scheduling and control block manipulation functions. These kernel functions are performed for both local and nonlocal communication. Therefore, it is clear that message-passing support should be provided transparently for both local and nonlocal communication. Further, timing measurements for performing typical services on Unix (see [24] for more details) suggest that server computation times are comparable to communication times incurred in the message kernel. The structure of the distributed system and the results of our studies suggest the following partition of the message-based operating system between the host and the message coprocessor: *computation* on the host and *communication* on the message coprocessor (Fig. 1).

The servers execute on the host and the message-passing kernel executes on the message coprocessor. The code and private data structures of the message-passing kernel are in the local memory of the message coprocessor. The code and data structures for the servers reside in the local memory of the host. The shared memory contains the task control blocks and kernel buffers. The message coprocessor is in control of the network interface. There are two lists of task control blocks in the shared memory: the computation list and the communication list, representing work to do for the host and message coprocessor, respectively. The lists are ordered by task scheduling priority. The host interrupts and informs the message coprocessor that the communication list is nonempty. Similarly, the message coprocessor informs the host that the computation list is nonempty via interrupt. When the computation list is nonempty, the host gets the first task from the list and executes it until the task makes a communication request. At that point, the host enqueues the task on the communication list and starts executing the next task in the computation list. The task control block contains the information needed to process the communication request. When the communication list is nonempty, the message coprocessor gets the first task

from the list and executes the communication processing code associated with that particular request. This processing will involve such chores as checking the validity of the IPC call, addressing and manipulating control blocks, kernel buffering, short-term scheduling decisions, sending network packets for nonlocal messages, and responding to network interrupts. As a result of this processing, a task that was waiting for a message may become ready to execute on the host. Network interrupts are serviced by the message coprocessor on a priority basis and lead to similar short-term scheduling decisions.

To verify the feasibility of such a software partition and to gather actual timing information, we implemented such a partition on an experimental system. 925 [26] is an early version of an experimental operating system (since renamed *Quick-silver*) for a network of workstations. For the purposes of this discussion, 925 is quite similar to the Stanford V Kernel [6]. The version we modified ran on a multiprocessor workstation with three Motorola 68000 processors [16], each with local memory, connected by a Versabus [17] to each other, a shared memory, and an early experimental version of the IBM token ring [5]. We emulated the message coprocessor with one of the processors, and measured the implementation to obtain the processing times for the kernel activities involved in message passing.

Through the implementation, we established the feasibility of partitioning² the message-based operating system between the host and the message coprocessor. Another important fruit of this exercise was insight into the kinds of system data structures that are used in communication processing, the operations that are done on them, and the overhead for these operations. Buffers and lists of control blocks are the data structures that are extensively manipulated in communication processing. Operations on these data structures include copying and atomic queue manipulation. With the Motorola 68000 implementation it takes 220 μ s of processing time to copy 40 bytes, and 74 μ s of processing time to perform an *atomic* queueing operation (which involves obtaining a mutual exclusion lock, performing the queueing operation, and releasing the lock). There are four copy operations and 16 queueing operations (see [20] for details) in one roundtrip (nonlocal communication). Hence, these times are important since they constitute a significant portion of the total roundtrip time. Drawing on our implementation experience, we describe an improved hardware organization in the next section.

IV. HARDWARE ORGANIZATION

The system data structures in *shared memory* are manipulated by all the units inside each node. The operations on these data structures can be grouped into three categories: movement of blocks of data, queue manipulation, and simple read/write. The above groups of operations are general and applicable for implementing the semantics of interprocess communication of any operating system. Hence, it is appropriate to support these operations on the shared memory at the bus level.

² The host to message coprocessor interaction results in an additional overhead for IPC, compared to a uniprocessor implementation of a message-based system. However, this overhead was less than 10% of the total roundtrip time.

Several recent bus proposals support block transfer primitives [4]. However, these bus proposals are intended for a versatile environment with multiple memory modules, processor modules, and device modules. In our environment, there is a limited shared memory holding task control blocks and kernel buffers. The units that access this memory are the message coprocessor, the host, and the network interfaces. This memory does not contain either "kernel programs" or "user programs." On the contrary, it contains only protected kernel data structures that are manipulated by trusted kernel code executing in the message coprocessor and the host. Each unit that accesses this memory has exactly one outstanding request.

In a *limited controlled environment* it would be more cost effective for the memory to handle multiplexed block transfers. Moreover, none of the existing bus proposals support atomic queue manipulation primitives.

We have designed a *smart bus* for message-passing support. To support the high-level primitives in this bus, we have designed a *smart shared memory* that implements these primitives. To put our bus design in the proper perspective, we would point out that the intent is not to invent a standard for system buses. In fact, we view the bus, the message coprocessor, the shared memory, and the network interfaces together as a single unit that provides message-passing support to the rest at the level of the operating system primitives. This unit exists with the rest of the node architecture that includes the bus on which the host, the devices, and the host memory reside.

Smart Bus Overview

Smart bus connects the host, the message coprocessor, and the network interfaces to the shared memory. Multiplexed block transfer and atomic queue manipulation are the transactions supported on the smart bus. Smart bus decouples requests for block transfers from the actual data transfers. The shared memory caches information regarding block transfer requests (address and size) in an internal table, so that it can start a lower priority request after servicing a higher priority one. The bus is never locked for arbitrary amounts of time, thus guaranteeing access for higher priority requests in finite time. A unit can have exactly one outstanding block transfer request. Therefore, the shared memory does not have to handle any flow control problems. Prioritized arbitration among competing units is supported on the bus. The arbitration scheme we use in our bus design is inspired by Futurebus but is simpler owing to the limited environment. Bus transfer rate is scalable with device technology due to the asynchronous protocol.

Physically, the bus includes 16 multiplexed address/data lines, four-lines for commands, and four-lines for a tag. In addition, there are arbitration lines for access control, protocol lines to complete the asynchronous handshake, and a system reset line for startup. We refer to a transition on a protocol line as a *clock-edge*.

The number of multiplexed address/data lines in our design is 16, stemming from the fact that our experimental results were obtained from a 16-bit Versabus [17] implementation. We maintain compatibility with our experimental results we

TABLE I
SMART BUS COMMANDS

Commands	Handshake Time
Simple Read	Eight-edge
Block transfer	Four-edge
Block read data	Two-edge
Block write data	Two-edge
Enqueue control block	Four-edge
Dequeue control block	Four-edge
First control block	Eight-edge
Write two bytes	Four-edge
Write byte	Four-edge

used 16-bit address/data lines. However, there is no inherent assumption in our design that would preclude extension to a wider bus.

The shared bus supports three categories of transactions: *block requests*, *queue manipulation*, *simple read/write*. Table I gives a summary of the smart bus commands and the time (measured in clock-edges) for performing the operations.

1) *Block Requests*: There are three transactions provided in this category: **block transfer**, **block read data**, and **block write data**. These primitives allow movement of blocks of contiguous data between the shared memory and other units in each node. They allow the shared memory to be multiplexed for handling simultaneous requests. **Block transfer** and **block write data** are initiated by the CPU's and network devices. Henceforth, we refer to either a CPU or a network device as a *processor*. The processor that initiates **block transfer** specifies whether it is a read or a write. **Block read data** is initiated by the shared memory. While **block transfer** is the primitive used by the processor to convey the intent to the shared memory, **block read data** and **block write data** are the primitives used to effect the actual data movement.

In **block transfer**, the processor sends the starting address of the block and a count indicating the number of contiguous bytes of information to be transferred. The command (read or write) is specified on the command bus. Shared memory stores them in its internal table and responds by returning a *tag* that uniquely identifies the transaction. **Block read data** and **block write data** are primitives that are issued following the block transfer request. Both these primitives result in data transfer. Shared memory executes **block read data** to send the data along with the tag that uniquely identifies the processor of the **block transfer** request. The processor monitors the tag bus. When there is a tag match, the processor receives the data from the bus. Information transfer is in the opposite direction for **block write data**. Following a request to write a block of data, the processor executes **block write data** sending the data along with the tag to the shared memory. Shared memory receives them and uses the tag as an index into its internal table to get the address where the data are to be stored.

2) *Queue Manipulation*: There are three primitives provided in this category: **enqueue control block**, **first control block**, and **dequeue control block**. By presenting the memory as a singly-linked circular list of control blocks, these primitives allow atomic queueing operations to be performed on these lists. The data structure in memory looks as shown in Fig. 2. When presented with a list address, the memory unit

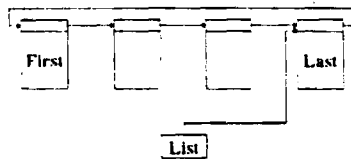


Fig. 2. Queue data structure.

views it as the address of the location in memory ("List" in Fig. 2) that points to the tail of the list. Definitions of these primitives are given below. In each case, "list" refers to the location in memory that points to the *tail* (last element) of the list (Fig. 2).

1) **Enqueue**(*element*, *list*): Add *element* to the tail of *list* and update *list* to point to the newly added item.

2) **First**(*list*): Dequeue the first item in *list* and return a pointer to it. Return a distinguished value if the list is empty.

3) **Dequeue**(*element*, *list*): Search for *element* in *list* and remove it. If *element* is not found, do nothing. This operation can be expensive if the list is long, but our experience with the 925 implementation shows that it is used only in exceptional circumstances.

3) **Read/Write**: In addition to the above transactions, the bus supports simple read/write primitives at byte granularity.

B. Smart Shared Memory

We designed a bus architecture that is appropriate within the functional unit composed of the message coprocessor and the network interfaces. However, this design implicitly assumes that the shared memory has the necessary "intelligence" to handle the high-level requests of the smart bus. The design also assumes that the processors either themselves or within their bus interfaces have the necessary intelligence to generate these requests. Fortunately, even though the bus transactions are high-level, the nature of the environment makes these transactions feasible from the point of view of hardware implementation. Moreover, the nature of the environment makes it possible to provide these facilities at a reasonable "cost".³ We demonstrate this feasibility through a detailed design of a *smart shared memory* [20]. The controller for the smart shared memory is microprogrammed, and has under 3000 bits of microcode. Based on the complexity of the design, we also show that the entire design can be packaged in two chips. The data path (without the memory system) can be implemented as a single chip with roughly 6000 active components. The sequencer can be implemented as a single chip with roughly 1000 active components.

V. PERFORMANCE ANALYSIS

Our solution to the message-passing problem in distributed systems had two parts: *software partition* and *hardware organization*. While we implemented the software partition, limitations on the time and money we had available for fabricating and testing the hardware lead us to model rather than build the *smart bus* and the *smart shared memory*. Moreover, modeling allowed us to parameterize the design, thus enabling individual features to be evaluated. The results of

³ We measure "cost" by the complexity (component count) of the design.

our performance analysis were sufficiently encouraging that we are now considering an experimental implementation of the hardware.

We modeled our design using *Generalized Timed Petri Nets* (GTPN) [13], an extension of Petri nets [18] that allows assignment of firing durations to transitions and relative probabilities to alternative paths through the net. We then used a tool that builds the set of reachable states for the GTPN model and solves the resulting Markov chain to determine steady-state performance measures. Aggregate performance measures specified by the user (e.g., system throughput) are also computed automatically by the tool. This approach provides more precise results than simulation, but the formulation of the model requires some care lest the number of states become excessive. Some of the techniques we used to avoid state explosion are interesting in their own right. The interested reader is referred to [23] for details.

A. Architectures

We compare three architectures.

Architecture I (Fig. 3) is a uniprocessor implementation of a distributed system. The message-based operating system executes on the host. The host is in control of the network interface.

Architecture II (Fig. 4) is the organization we implemented in 925 (see Section III).

Architecture III is similar to architecture II, with the difference that a smart bus interconnects the different units within each node and a smart memory serves as shared memory.

One important fruit of the implementation is that it gave us the timing values needed for driving the different models. These timing values are the processing times for the different components of message passing. In the architectures we are comparing, we assume the processors to be identical. Hence, the processing times we obtained from our implementation are applicable to all of them.

B. Workload Description

In this section, we describe the workload that we used as the basis for comparing the different architectures. While this is not the only possible workload, it is a typical workload in a distributed system. We plan to study other workloads in future.

A client loops making blocking *send* requests:

```
loop
  send;
end;
```

A server loops posting *receive* system calls:

```
loop
  receive;
  compute;
  reply;
end;
```

When the send and the receive match, a rendezvous takes place between the client and the server. The server then computes for a while processing the request in the message from the client. At the end of the computation phase, the server

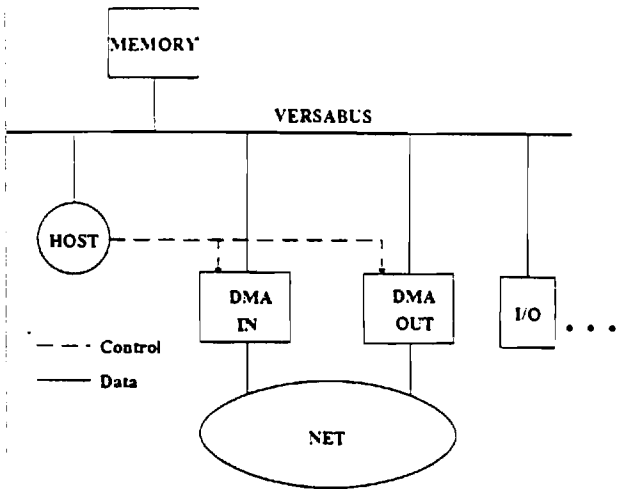


Fig. 3. Architecture I: Uniprocessor.

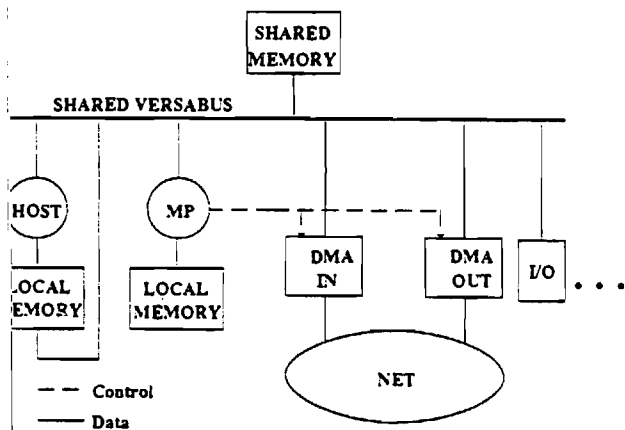


Fig. 4. Architecture II: Message coprocessor.

of communication time and compute time. As we mentioned earlier, by communication we mean the system code that has to be executed to process a communication request. Intuitively, a compute-bound conversation is characterized by an offered load near zero, while a communication-bound conversation has a load near unity.

C. Processing Times

In our implementation, the Motorola 68000 CPU was driven by an 8 MHz clock, yielding an instruction execution rate of roughly 0.3 MIPS [16]. Versabus [17] memory cycle time is on an average 1 μ s. In our models, we assume an instruction execution time of 3 μ s and a Versabus memory cycle time of 1 μ s. We also assume that the four-edge handshake of smart bus equals Versabus memory cycle time and that the two-edge handshake equals half the Versabus memory cycle time. We should point out that a much higher speed is achievable for the smart bus with current technology. However, these conservative times for smart bus primitives give a more realistic basis for comparing the different architectures. Table II shows a comparison of implementing queue manipulation and block transfer operations for architectures II and III. For architecture II, each of *enqueue*, *dequeue*, and *first* involves the following steps to be performed by the message coprocessor: lock a semaphore, execute the queue manipulation algorithm (see Section IV-A2), and release the semaphore. The message coprocessor executes a program loop for reading or writing a block in architecture II. The processing time for this loop execution is shown in Table II. The message coprocessor in architecture III executes three instructions to initiate any of the smart bus primitives. For example, to initiate block transfer the message coprocessor writes the starting address, count, and command to its bus interface.

Tables III-VIII contain a breakdown of the communication time for one roundtrip conversation into component message-passing activities. The breakdown shows the processing time and the time spent in accessing shared data structures for both local and nonlocal conversations. The times for architecture II were obtained directly from our implementation. The times for architecture I were obtained from architecture II by eliminating the overhead for synchronization between the host and the message coprocessor. The times for architecture III were derived from architecture II after factoring in the primitives of the smart bus.

The following table shows the cumulative processing and shared memory access times for architectures II and III, and the percentage reduction in these times due to the smart bus primitives.

letes the request from the client with a *reply*, complete rendezvous between the client and the server. We call extended request-reply sequence between the client and server a *conversation*. Our workload contains both local nonlocal conversations. The number of simultaneous conversations and the amount of computation specified in each conversation are the two parameters we vary in the workload. true that in a real system, clients compute as well. However, we designed our workload to stress the performance of message-based operating system composed of the message client and the servers. Therefore, for these experiments we do not consider client-computation in our workload.

Offered load is a measure of the communication load that is presented to the system by each conversation, defined as the ratio of communication time (in a roundtrip) to the sum

Communication	Activity	Cumulative Time (Microseconds)			
		Arch II	Arch III	Difference	Reduction
Local	Processing	4770	3300	1470	30%
	Shared Memory Access	828	612	216	26%
Nonlocal	Processing	6770	5200	1570	23%
	Shared Memory Access	988	692	296	29%

TABLE II
COMPARISON OF PROCESSING TIMES

Operation	Architecture II		Architecture III		Handshake
	Processing Time	Time Spent In Memory Cycles	Processing Time	Time Spent In Memory Cycles	
	micro-seconds				
Enqueue	60	14	9	1	Four-edge
Dequeue	60	14	9	1	Four-edge
First	60	14	9	2	Eight-edge
Block Read (40 Bytes)	180	20	9	11	One four-edge followed by twenty two-edge
Block Write (40 Bytes)	180	20	9	11	One four-edge followed by twenty two-edge

TABLE III
ARCHITECTURE I: LOCAL CONVERSATION

Processor	Initiator	Action	Time (microseconds)	
			Processing	Shared memory access
Host	Client	Syscall Send	1040	150
Host	Server	Syscall Receive	650	120
Host		Match client with server	1240	140
Host	Server	Compute	Workload Parameter	
Host	Server	Syscall Reply	1020	210
Host		Restart Server	140	60
Host		Restart Client	140	60

TABLE IV
ARCHITECTURE I: NONLOCAL CONVERSATION

Processor	Initiator	Action	Time (microseconds)	
			Processing	Shared memory access
Host	Client	Syscall Send	1140	150
DMA	Client	DMA out	200	30
Host	Server	Syscall Receive	650	120
DMA	Network interrupt	DMA in	200	30
Host	Network interrupt	Match client with server	1790	210
Host	Server	Compute	Workload Parameter	
Host	Server	Syscall Reply	1060	220
DMA	Server	DMA out	200	30
DMA	Network interrupt	DMA in	200	30
Host	Network interrupt	Cleanup and Restart Client	830	130

D. Validation

Our experimental implementation on the 925 system differed from architecture II in two ways.

- 1) There were two hosts in each node instead of one.
- 2) The network interfaces required an additional copy from

TABLE V
ARCHITECTURE II: LOCAL CONVERSATION

Processor	Initiator	Action	Time (microseconds)	
			Processing	Shared memory access
Host	Client	Syscall Send	320	78
MP	Client	Process Send	900	104
Host	Server	Syscall Receive	320	78
MP	Server	Process Receive	510	74
MP		Match client with server	1160	64
Host	Server	Restart Server	60	50
Host	Server	Compute	Workload Parameter	
Host	Server	Syscall Reply	320	78
MP	Server	Process Reply	1060	182
Host		Restart Server	60	50
Host		Restart Client	60	50

the kernel buffers to the memory-mapped network buffers in shared memory.

We used the workload described in Section V-B for performance measurements of the implementation. We validated a model for nonlocal conversations of our experimental implementation against these performance measures. Fig. 5(a), (b), and (c), shows the agreement between the experimental and model results. We note that for one and two conversations [Fig. 5(a)] the agreement is very good (within 3% for one and 10% for two). For three and four conversations [Fig. 5(b) and (c)], the model results are within 10% of the experimental results at high offered loads, while at low offered loads the deviation is within 25%. The optimistic prediction in the case of low offered load (high computation) is partly due to a load-leveling effect in the model not present in the experimental implementation: in the implementation, a process is bound to a particular host, whereas in the model, a request can be serviced on any available host. When the load is less communication-intensive, server processes spend a larger fraction of time on the host and as a result the throughput

TABLE VI
ARCHITECTURE II: NONLOCAL CONVERSATION

Processor	Initiator	Action	Time (microseconds)	
			Processing	Shared memory access
Host	Client	Syscall Send	320	78
MP	Client	Process Send	1000	104
DMA	Client	DMA out	200	30
Host	Server	Syscall Receive	320	78
MP	Server	Process Receive	510	74
DMA	Network interrupt	DMA in	200	30
MP	Network interrupt	Match client with server	1650	104
Host	Server	Restart Server	60	50
Host	Server	Compute	Workload Parameter	
Host	Server	Syscall Reply	320	78
MP	Server	Process Reply	920	128
DMA	Server	DMA out	200	30
Host		Restart Server	60	50
DMA	Network interrupt	DMA in	200	30
MP	Network interrupt	Cleanup client	750	74
Host		Restart Client	60	50

TABLE VII
ARCHITECTURE III: LOCAL CONVERSATION

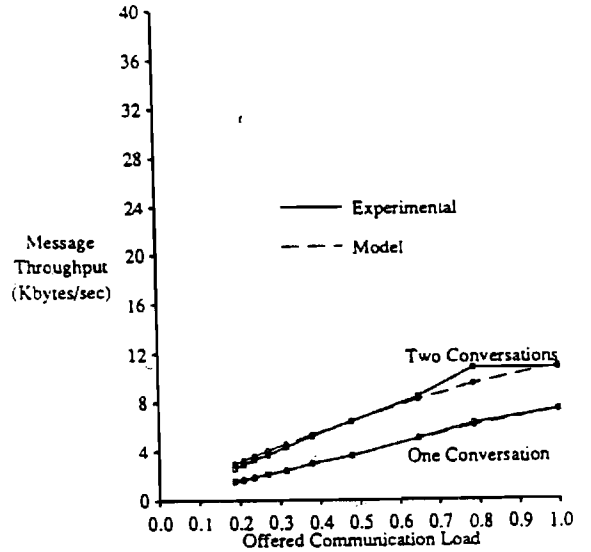
Processor	Initiator	Action	Time (microseconds)	
			Processing	Shared memory access
Host	Client	Syscall Send	220	52
MP	Client	Process Send	612	71
Host	Server	Syscall Receive	220	52
MP	Server	Process Receive	451	61
MP		Match client with server	922	61
Host	Server	Restart Server	60	50
Host	Server	Compute	Workload Parameter	
Host	Server	Syscall Reply	220	52
MP	Server	Process Reply	475	113
Host		Restart Server	60	50
Host		Restart Client	60	50

redicted by the model is higher. However, despite this effect, the model results show good overall agreement with the experimental results.

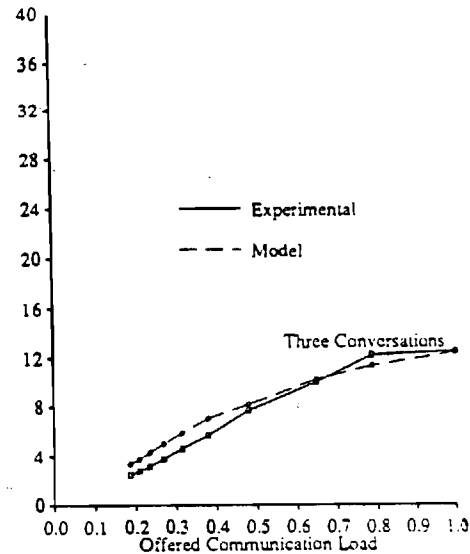
Results

In this section, we present and compare the results of solving the models for the three architectures for the workload we described earlier.

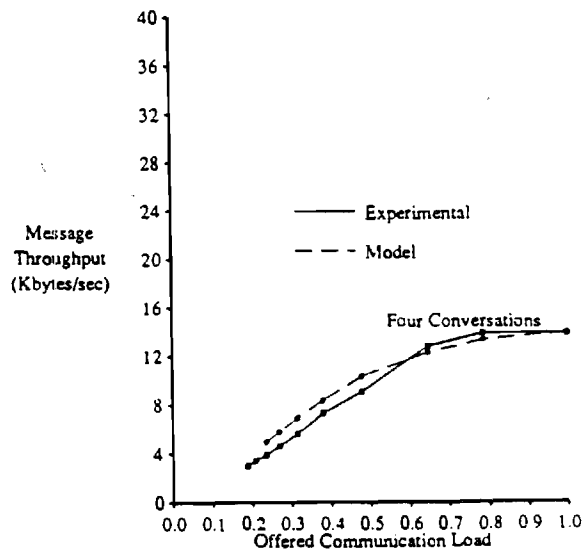
1) *Maximum Communication Load:* Fig. 6(a) and (b) compares the throughput of architectures I, II, and III, under conditions of maximum communication load for local conversations and nonlocal conversations. The throughput results shown in Figs. 6 and 7 are based on a size of 40 bytes for each send or reply message. For architecture I, the throughput for local conversations is the same irrespective of the number of conversations, a fairly intuitive result. For architecture II, the throughput for one conversation is slightly less than that for



(a)



(b)



(c)

Fig. 5. Model validation.

TABLE VIII
ARCHITECTURE III: NONLOCAL CONVERSATION

Processor	Initiator	Action	Time (microseconds)	
			Processing	Shared memory access
Host	Client	Syscall Send	220	52
MP	Client	Process Send	712	71
DMA	Client	DMA out	200	15
Host	Server	Syscall Receive	220	52
MP	Server	Process Receive	451	61
DMA	Network interrupt	DMA in	200	15
MP	Network interrupt	Match client with server	1362	71
Host	Server	Restart Server	60	50
Host	Server	Compute	Workload Parameter	
Host	Server	Syscall Reply	220	52
MP	Server	Process Reply	573	82
DMA	Server	DMA out	200	15
Host		Restart Server	60	50
DMA	Network interrupt	DMA in	200	15
MP	Network interrupt	Cleanup client	462	41
Host		Restart Client	60	50

architecture I. The loss represents the overhead involved in the information transfer between the host and the message coprocessor. However, note that this loss is very small ($\approx 10\%$). Increase in throughput with the number of conversations is less than linear due to the finite bandwidth of the message coprocessor. Note that architecture III is significantly better than both architectures I and II. The smart bus reduces the overhead in communication processing by providing high-level bus transactions. These transactions are significantly faster than a software implementation (see Section V-C).

Fig. 6(b) illustrates the results for nonlocal conversations. The tendency to saturate with number of conversations is less pronounced for nonlocal conversations when compared to local conversations, since the processing load is spread across two nodes. Once again we note that architecture III performs significantly better than architectures I and II.

We note that architecture II does not do significantly better than architecture I (both local and nonlocal conversations). However, these graphs are for maximum communication load. Under these conditions the host is idle most of the time since there is no computation in any conversation. The premise behind partitioning the software is that load in a distributed system consists of a good mix of computation and communication. In the next section, we will discuss our results under such typical load.

2) *Varying Workload*: In this section, we compare architectures I, II, and III under the assumption that the server does a certain nonzero amount of computation before replying to the client. As we mentioned earlier (see Section V-B), offered load is defined as $L = C/(C + S)$, where C is the communication processing requirement in one roundtrip and S is the server computation time. C is dependent on the architecture while S is a workload parameter. Tables IX and X give the offered loads for different server-computation times in

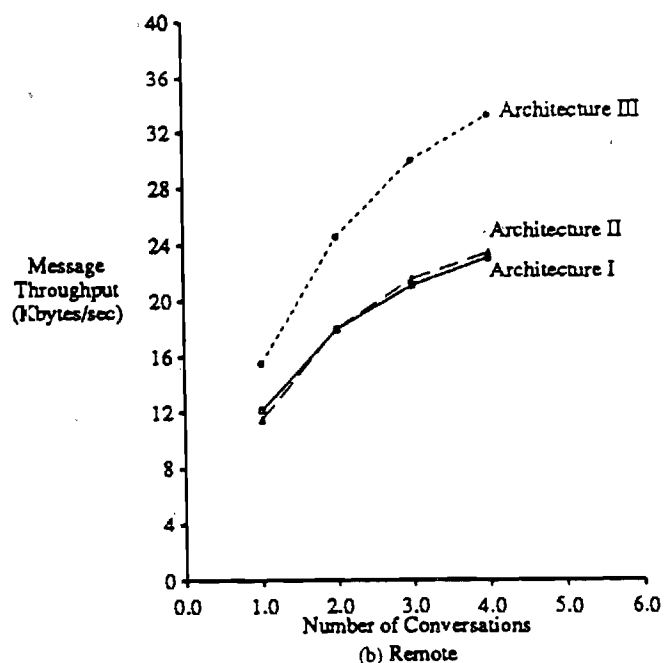
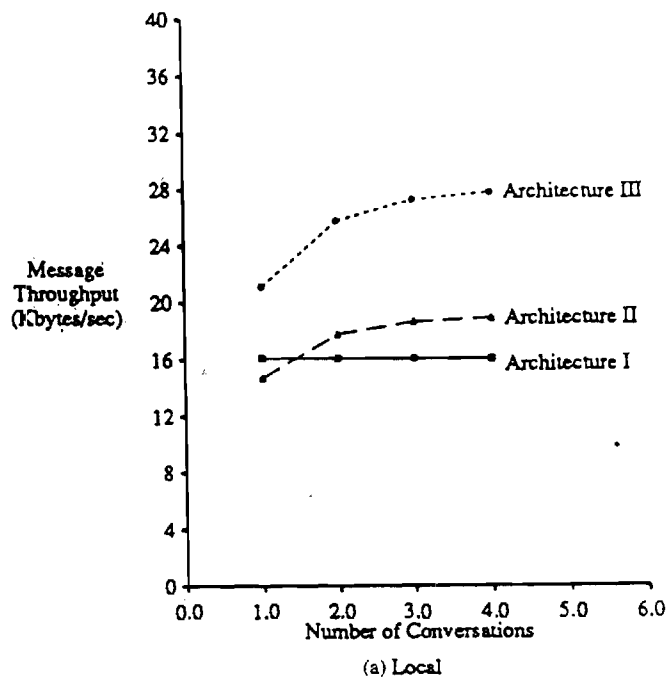


Fig. 6. Maximum communication load (40-byte messages).

the three architectures for local and nonlocal conversations, respectively. Note that the offered load for a given server-computation time is the least for architecture III since it has the least C , and slightly higher for architecture II. The value of S for a given service is the same for each of the three architectures. For example, our measurements of Unix on a processor that is about two to three times the speed of the modeled architecture show service times ranging from 0.2 to 6.1 ms (see [24]). Using Tables IX and X, we can read off the offered loads for each architecture given the server-computation time.

We want to be able to compare the performance of the three architectures for various servers. Fig. 7 illustrates how message throughput depends on offered load, as determined by

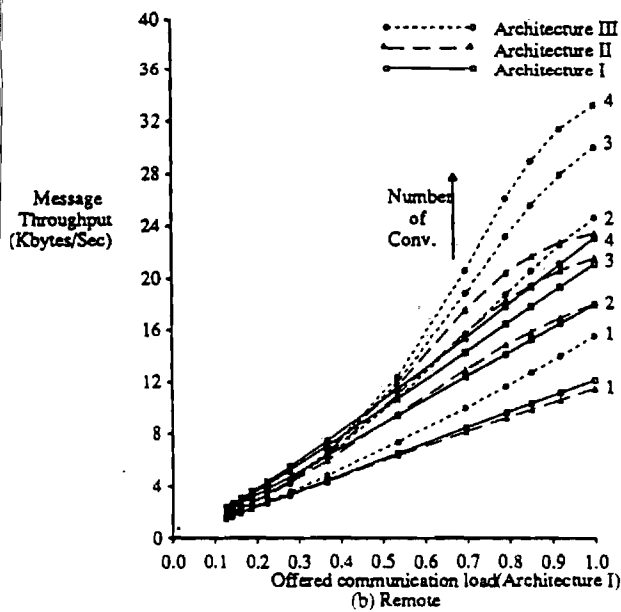
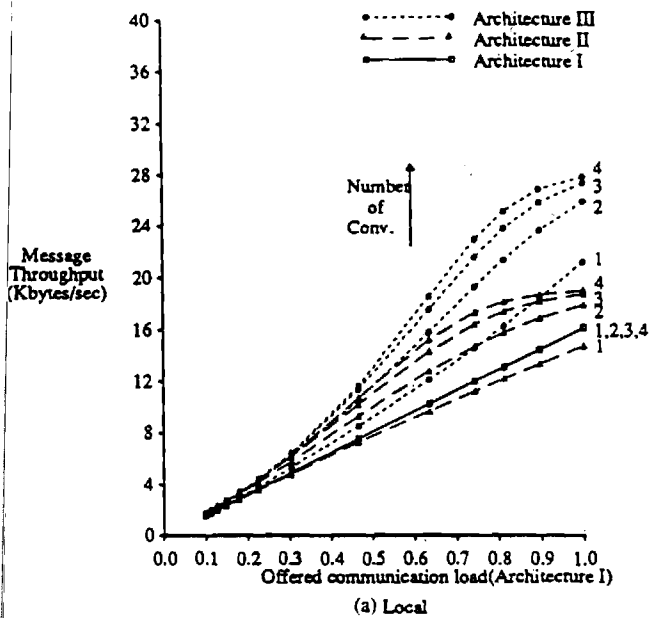


Fig. 7. Varying workload (40-byte messages).

TABLE IX
OFFERED LOADS (LOCAL)

Server Time (milli-seconds)	Offered Load in Architecture		
	I	II	III
0	1.0	1.0	1.0
0.57	0.897	0.905	0.867
1.14	0.813	0.827	0.769
1.71	0.744	0.761	0.689
2.85	0.635	0.656	0.571
5.7	0.466	0.488	0.399
11.4	0.304	0.323	0.249
17.1	0.225	0.241	0.181
22.8	0.179	0.193	0.142
28.5	0.148	0.160	0.117
34.2	0.127	0.137	0.100
39.9	0.111	0.120	0.087
45.6	0.098	0.107	0.077

TABLE X
OFFERED LOADS (NONLOCAL)

Server Time (milli-seconds)	Offered Load in Architecture		
	I	II	III
0	1.0	1.0	1.0
0.57	0.920	0.924	0.900
1.14	0.852	0.859	0.818
1.71	0.793	0.802	0.750
2.85	0.697	0.709	0.643
5.7	0.536	0.549	0.474
11.4	0.366	0.379	0.311
17.1	0.278	0.289	0.231
22.8	0.224	0.233	0.184
28.5	0.187	0.196	0.153
34.2	0.161	0.169	0.130
39.9	0.141	0.148	0.114
45.6	0.126	0.132	0.101

the amount of computation done by a server for each request. Since the offered load L depends on C , which is a function of the particular architecture, we normalized the results by plotting throughput for each architecture as a function of the offered load a given server would produce on architecture I. Fig. 7(a) illustrates local conversations while Fig. 7(b) illustrates nonlocal conversations.

For architecture I, with local conversation, the results are independent of the number of conversations. Architecture II does slightly worse than architecture I for one conversation due to the overhead in passing information between the host and the message coprocessor. However, as the number of conversations is increased, the throughput improves considerably over architecture I. With a message coprocessor equal in processing speed to the host, the upper bound for throughput improvement (with no overhead between the host and the message coprocessor) is a factor of two. Architecture II approaches this limit over a range (0.5-0.9) of values for offered load. When the load is more computation intensive there is no significant gain in partitioning the software. The graph defines a region of operation of the distributed system in terms of mixture of computation and communication for which the message coprocessor is viable. By providing high-level bus primitives, architecture III does better than both architecture I & II and over a wider range (0.4-0.95) of offered load. The tendency to saturate for three and four conversations is also less pronounced for architecture III.

Fig. 7(b) shows a comparison of results for nonlocal conversation. For architecture II, the improvement in throughput with offered load over architecture I is less pronounced for the number of conversations that we have modeled. However, note that for four conversations we see an improvement ($\cong 20\%$) over architecture I in the range of offered loads 0.7-0.9. Thus, the graphs do show a trend in predicting the improvement that is attainable for much larger systems. Unfortunately, given the limitations of existing modeling tools, we were unable to model larger systems. We note once again that architecture III shows a marked performance improvement over the first two architectures. Over the range of offered loads 0.6 and 1.0, architecture III does significantly better than both architectures I and II. The graph suggests that smart bus primitives are as

important for improving the performance of the system for nonlocal conversations as software partitioning.

3) *Partitioned Smart Bus*: We analyzed a fourth architecture that was motivated by the observation that task control blocks are a shared data structure between the host and the message coprocessor, whereas kernel buffers are a shared data structure between the message coprocessor and the network interfaces. We partition the smart shared memory and the smart bus as follows. The task control blocks are on a partition that interconnects the host with the message coprocessor and the kernel buffers are on a partition that interconnects the message coprocessor with the network interfaces.

We found in all cases that the partitioned organization did not perform significantly better than architecture III. We would have expected such an improvement in performance if there was a considerable contention for the shared memory. These performance results indicate that access to the shared memory is not the bottleneck in limiting the performance. For the same reason, for a given architecture, we do not expect a multiported shared memory to perform better than a single-ported shared memory for any of the four architectures that we analyzed.

F. Summary

In summary, the graphs show the following.

1) Over ranges of offered loads (0.4–1.0 for local and 0.6–1.0 for nonlocal), partitioning the message-based operating system and providing high-level bus primitives result in improvement in performance over a uniprocessor implementation. Thus, there is a range of mixes of computation and communication in which a message coprocessor is appropriate for improving the performance of the system. We observed that the times for typical system services (measured on a Microvax II running Unix) [24] such as timer, and reading/writing files, range from 0.2 to 6.1 ms. With a local-message communication time of 4.57 ms on Unix, these service times represent an offered load ranging from 0.96 to 0.43; with a nonlocal communication time of 6.8 ms the corresponding offered loads range from 0.97 to 0.53.

2) For one conversation there is a loss in performance due to software partitioning, but the loss is very small. Improvement in performance with the number of conversations is less than linear due to the finite bandwidth of the message coprocessor.

3) Smart bus primitives improve the performance of the system significantly for both local and nonlocal conversations.

4) Software partitioning, and high-level bus transactions (mirroring operating system functions), are a promising approach to solving the message-passing problem in distributed systems. Multiported memories do not help significantly since it is processing-time and not access to shared memory that is the limiting bottleneck.

VI. CONCLUDING REMARKS

Local area networking has enhanced the interest of researchers in experimenting with distributed message-based operating systems. Current research [6], [11] and our own measurements of several operating systems show that interprocess

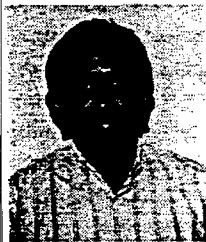
communication (message passing) is roughly two orders of magnitude slower than a simple procedure call. Since system services are requested via message passing, the performance of message-based operating systems depends crucially on the rate of message passing. Our goal in this research was to study the problem of interprocess communication in a distributed system, and suggest a system architecture that improves the performance in this environment. In this paper, we suggested a system architecture that was composed of a software aspect and a hardware organization. Using GTPN as a modeling tool, we showed that software partitioning and high-level bus transactions (mirroring operating system functions) are a promising approach to solving the message-passing problem in distributed systems. Future research directions include: extending the performance studies to different communication scenarios, exploring the instruction-set architecture of the message coprocessor, implementing in VLSI the hardware assists identified in this research, and extending our studies to the domain of multiprocessors connected by local area networks.

REFERENCES

- [1] ABLE. *Easyway Ethernet Port*, ABLE Computer, Irvine, CA 92714, 1984.
- [2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevianian, and M. Young, "Mach: A new kernel foundation for Unix development," in *Proc. Summer Usenix*, July 1986.
- [3] Y. Artsy, H. Chang, and R. A. Finkel, "Interprocess communication in Charlotte," *IEEE Software*, vol. 4, no. 1, pp. 22–28, Jan. 1987.
- [4] P. L. Borrill, "Microstandards special feature: A comparison of 32-bit buses," *IEEE Micro*, vol. 5, no. 6, Dec. 1985.
- [5] W. Bux, F. Closs, P. Janson, K. Kummerle, and H. R. Muller, "Architecture and design of a reliable token-ring network," *IEEE J. Select. Areas Commun.*, vol. SAC-1, no. 5, pp. 756–765, Nov. 1983.
- [6] D. R. Cheriton and W. Zwaenepoel, "The distributed V kernel and its performance for diskless workstations," *Oper. Syst. Rev.*, vol. 17, no. 5, pp. 128–140, Oct. 1983.
- [7] P. Corsini and C. A. Prete, "Architecture of the muteam system," *IEE Proc., Part E: Comput. and Digital Techniques*, vol. 134, no. 5, pp. 217–227, Sept. 1987.
- [8] G. Cox and W. Corwin, "A unified model and implementation for interprocess communication in a multiprocessor environment," in *Proc. Eighth Symp. Oper. Syst. Principles*, Dec. 1981.
- [9] W. Fischer, "IEEE P1014—A standard for the high-performance VME bus," *IEEE Micro*, vol. 5, no. 1, Feb. 1985.
- [10] S. H. Fuller, "Multi-microprocessors: An overview and working example," *Proc. IEEE*, vol. 66, no. 2, pp. 216–228, Feb. 1978.
- [11] R. D. Gaglianella and H. P. Katseff, "Meglos: An operating system for a multiprocessor environment," in *Proc. 5th Int. Conf. Distributed Comput. Syst.*, Denver, CO, May 1985, pp. 35–42.
- [12] T. Goradia and M. K. Vernon, "A model for quantitative analysis of network interface units," in *Proc. 6th Annu. Joint Conf. IEEE Comput. Commun. Societies (INFOCOM '87)*, San Francisco, CA, Mar. 1987.
- [13] M. A. Holliday and M. K. Vernon, "A generalized timed Petri net model for performance analysis," *IEEE Trans. Software Eng.*, Dec. 1987.
- [14] Interlan. N3010A Multibus Ethernet Communications Controller, 1983.
- [15] H. Lee and U. V. Premkumar, "The architecture and implementation of distributed Jasmin kernel," Tech. Memo. TM-ARH-000324, Bellcore, Morristown, NJ, Oct. 1984.
- [16] Motorola, *MC 68000 16-Bit Microprocessor User's Manual*. Englewood Cliffs, NJ: Prentice-Hall, 1982.
- [17] Motorola, *VERSAmodule Monoboard Microcomputer User's Guide*, Motorola Inc., 1982.
- [18] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
- [19] J. Postel, "Transmission control protocol," ARPA Network RFC 793, USC/Information Sciences Institute, 1981.
- [20] U. Ramachandran, "Hardware support for interprocess communi-

tion." Ph.D. dissertation, Tech. Rep. 667, Comput. Sci. Dep., Univ. Wisconsin—Madison, Sept. 1986.

- [21] —, "A survey of hardware support for interprocess communication." Tech. Rep. GIT-ICS-87/09, School of ICS, Georgia Instit. Technol., Jan. 1987.
- [22] U. Ramachandran, M. Solomon, and M. Vernon. "Hardware support for interprocess communication," in *Proc. 14th Int. Symp. Comput. Architecture*, June 1987, pp. 178-188. Also available as Tech. Rep. GIT-ICS-87/13, School of ICS, Georgia Instit. Technol., Jan. 1987.
- [23] —, "Techniques for reducing the complexity of large system models," in *Proc. 16th Int. Conf. Parallel Processing*, Aug. 1987, pp. 692-694. (detailed version available as Tech. Rep. GIT-ICS-87/07, School of ICS, Georgia Institute Technol., Jan. 1987).
- [24] U. Ramachandran, "Measurements of distributed operating systems," in *Proc. Twenty-First Annu. Hawaii Int. Conf. Syst. Sci.*, Jan. 1988. Also available as Tech. Rep. GIT-ICS-87/02, School of ICS, Georgia Institute Technol., Jan. 1987.
- [25] R. F. Rashid and G. G. Robertson, "Accent: A communication oriented network operating system kernel," in *Proc. Eighth Symp. Oper. Syst. Principles*, Pittsburgh, PA, Dec. 1981, pp. 64-75.
- [26] W. Sawdon, R. Haskin, Y. Malachi, and G. Chan, "Recovery management in Quicksilver," *ACM Trans. Comput. Syst.*, vol. 6, no. 1, pp. 82-108, Feb. 1988.
- [27] A. S. Tanenbaum and S. J. Mullender, "An overview of the Amoeba distributed operating system," *Oper. Syst. Rev.*, vol. 13, no. 3, pp. 51-64, July 1981.
- [28] D. M. Taub, "Arbitration and control acquisition in the proposed IEEE 896 futurebus," *IEEE Micro*, vol. 4, no. 4, pp. 28-41, Aug. 1984.
- [29] C. M. Woodside, "Optimal allocation of protocol processing between host and a front-end processor," in *Proc. IFIP WG7.3/TC 6 2nd Int. Symp. Perform. Comput. Commun. Syst.*, North Holland, 1984.



Umakishore Ramachandran received the Ph.D. degree in computer science from the University of Wisconsin—Madison in 1986.

Since then he has been an Assistant Professor in the School of Information and Computer Science at the Georgia Institute of Technology. His primary research interests are in computer architecture and distributed operating systems. He has participated in the design of several distributed systems: Charlotte at UW—Madison, Jasmin at Bell-Core, QuickSilver at IBM Almaden Research Center, and Clouds at Georgia Tech. He was also part of the PIPE computer architecture design team at UW—Madison. The research projects he is currently working on include distributed shared memory, memory management

structures for object-based systems, issues in the design of large-scale multiprocessors (such as caches and optical interconnects), reconfigurable super-computer architectures, and architecture support for message-based systems.

Dr. Ramachandran is a member of the Association for Computing Machinery and the IEEE Computer Society.



Marvin Solomon received the B.S. degree in mathematics from the University of Chicago and the M.S. and Ph.D. degrees in computer science from Cornell University, Ithaca, NY.

In 1976, he joined the Department of Computer Sciences at the University of Wisconsin—Madison, where he is currently Professor. He was a Visiting Lecturer at Aarhus (Denmark) University during 1975-1976 and Visiting Scientist at IBM Research, San Jose, CA, during 1984-85. His research interests include programming languages, program development environments, operating systems, and computer networks.

Dr. Solomon is a member of the IEEE Computer Society and the Association for Computing Machinery.



Mary K. Vernon (S'82-M'82) was born in Havre de Grace, MD, in 1953. She received the B.S. degree with Departmental Honors in chemistry in 1975, and the M.S. and Ph.D. degrees in computer science in 1979 and 1983, from the University of California, Los Angeles.

In August 1983, she joined the Department of Computer Science at the University of Wisconsin—Madison, where she is currently an Associate Professor. Her research interests include techniques for performance modeling of parallel

systems, performance Petri nets, and analysis of multiprocessor design issues.

In 1985, Dr. Vernon received the Presidential Young Investigator Award from the National Science Foundation. She is a member of the C.S. Advisory Board of the Computer Measurement Group, and the board of the ACM SIGMETRICS.