# Issues in the Design of Distributed Shared Memory Systems

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

By

Ajay Mohindra

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

Georgia Institute of Technology
May 1993

# Issues in the Design of Distributed Shared Memory Systems

Approved:

[illegible]

Dr. Umakishore Ramachandran, Chairman

[illegible]

Dr. Mustaque Ahamad

[illegible]

Dr. Richard LeBlanc Jr.

[illegible]

Dr. Karsten Schwan

[illegible]

Dr. Murthy Devarakonda (IBM)

Date Approved by Chairman 5/26/93

# Acknowledgements

First, I would like thank Dr. Kishore Ramachandran for enduring me for the past six years, and seeing me through the Ph.D. program. Without his guidance, I may not have seen the end of the tunnel. I would also like to thank all the members of my reading committee, who provided guidance and constructive criticisms, and ensured that what I did was worthwhile. Thank you Dr. Richard LeBlanc Jr., Dr. Mustaque Ahamad, Dr. Karsten Schwan, and Dr. Murthy Devarakonda. I would also like to thank Dr. Sudhakar Yalamanchili and Dr. H. Venkateswaran for their advice and guidance.

Special thanks go to folks in the CLOUDS Lab who kept the lab alive. Thanks to Sathis Menon, Mark Pearson, Gautam Shah, Ranjit John, M. Chelliah, Vibby Gottemukkala, Sreenivas Gukal, L. Gunaseelan, and Ross D'Souza. I would also like to thank Dr. Martin Davis Jr. for being a good friend. Special thanks go to Deeptendu Majumder, with whom I checked out numerous ethnic restaurants, and movies during my stay at Georgia Tech.

My Ph.D. would not have been possible without the love and support of my parents; I dedicate my dissertation to them. I would also like to thank my brother Vivek, and my sister Deepali, for their "long distance" support. And last, but not the least, I would like to thank my wife Reena, for her support and understanding, and for providing me with motivation to graduate.

# Contents

# List of Tables

# List of Figures

# Summary

This thesis examines the various system issues that arise in the design of distributed shared memory (DSM) systems. This work has been motivated by the observation that distributed systems will continue to become popular, and will be increasingly used for solving large computational problems. To this effect, shared memory paradigm is attractive for programming large distributed systems because it offers a natural transition for a programmer from the world of uniprocessors. The goal of this work is to identify a set of system issues, such as integration of DSM with virtual memory management, choice of memory model, choice of coherence protocol, and technology factors; and evaluate the effects of the design alternatives on the performance of DSM systems. The specific question that we are trying to answer is, "Can we determine a set of system design parameters that defines an efficient realization of a distributed shared memory system?". The design alternatives have been evaluated in three steps. First, we do a detailed performance study of a distributed shared memory implementation on the CLOUDS[1] distributed operating system. Second, we implement and analyze the performance of several applications on a distributed shared memory system. Third, the system issues that could not be evaluated via the experimental study, are evaluated using a simulation-based approach. The simulation model is developed from our experience with the CLOUDS distributed system. A new workload model that captures the salient features of

[1]CLOUDS is a distributed object-based operating system developed at Georgia Tech.

parallel and distributed programs is developed and used to drive the simulator. The key results of the thesis are: DSM mechanisms have to be integrated with the virtual memory management for providing high performance distributed shared memory systems; the choice of the memory model and coherence protocol does not significantly influence the system performance for applications exhibiting high computation granularity and low state-sharing; and an efficient implementation of DSM requires a careful design of miscellaneous system services (such as synchronization and data servers). The thesis also enumerates several questions related to future research directions.

# Chapter 1

# Introduction

Technological advances in recent years have spurred a trend towards workstation-oriented computing environments. Each workstation has computing power comparable to the mini-mainframes of the past. Availability of powerful computers connected via local (wide) area network has sparked interest in the area of distributed computing systems. Current research is targeting its efforts in utilizing the available computation power on the network in solving large problems through co-operative computing.

To facilitate programming of distributed systems, two basic paradigms exist: shared memory, and message-passing. These two paradigms have been used for interprocess communication and synchronization in multi-process computations. The duality between the two paradigms for structuring computations is well-known [21]. Nevertheless, shared memory has been an appealing paradigm from the point of view programming ease even in distributed systems. It is no surprise that several researchers [28, 30, 17, 7] have proposed system architectures that provide the abstraction of shared memory in a physically non-shared (distributed) architecture. We refer to this abstraction as *Distributed Shared Memory* (DSM). Figure 1 shows the conceptual representation of a distributed shared memory system. In the system, a set of nodes (computers) are connected via an interconnection network, and do not physically share memory. The DSM mechanisms allow an application to access shared data not physically resident at that node. These mechanisms are usually provided as a software layer either integrated with or on top of the operating system.

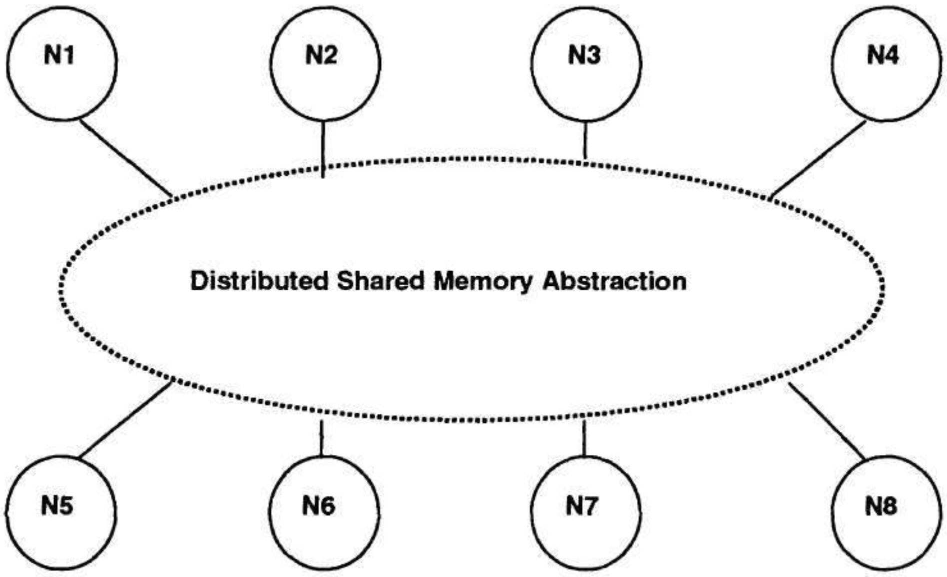

Figure 1: Distributed Shared Memory Abstraction

Another motivation for DSM arises from the structure of current distributed computing environments. A typical distributed computing environment consists of *compute servers*[1] and *data servers*[2] interconnected by a local area network. In such an environment, there are two tasks to be performed to execute a computation. The first task involves selecting a compute server, and the second task involves bringing the code and data from the data server to the selected compute server before executing the computation. The second task requires a remote paging facility. If

[1]Nodes where computation is performed.
[2]Nodes that serve as repositories for data.

sharing of data is coupled with this remote paging, it could be seen that DSM presents itself as a natural facility for combining the two.

Previous work in the area of distributed shared memory, as will be elaborated in Chapter 3, has been concerned with the design and implementation of distributed shared systems, and evaluation of algorithms for enforcing coherence of shared data. Some researchers have also focussed their efforts on designing fault-tolerant algorithms for distributed shared memory systems. Previous work, however, has ignored the study of system issues that need to be addressed in the design of distributed shared memory systems. The specific question that the research is trying to answer is:

*"Can we determine a set of system design parameters that defines an efficient realization of a distributed shared memory system?".*

In this thesis, we identify and evaluate the system issues (see Chapter 2) that need to be addressed for designing distributed shared memory systems. The issues relate to questions such as, whether to integrate DSM with virtual memory management, what type of memory model to provide, which coherence protocol to use for maintaining coherence of shared data, and what kind of impact do technology factors have on the DSM system performance. We evaluate these issues with respect to the available design alternatives. The evaluation is done in three steps. First, we do a performance study of an implementation of DSM (see Chapter 4). The performance study has provided us with an insight into the functioning of a distributed shared memory system. Second, we have implemented and studied the performance of several applications on a distributed shared memory system (see Chapter 5). Finally, aspects of the study that could not be evaluated via experimental studies are evaluated using simulation. In Chapter 6, we describe the design of a simulator that models a distributed shared memory system. The costs obtained from the performance study are used to assign costs to different components of the simulator. A new workload model is developed and used to drive the simulator. The workload model captures the salient features of distributed and parallel programs. In Chapter 7, we discuss the results of the research. The conclusions and contributions of the research are presented in Chapter 8.

The key contribution of this thesis is that it enumerates the systems issues and specifies the design parameters for addressing system issues for an efficient realization of a DSM system. The key results of the thesis are: DSM mechanisms have to be integrated with the virtual memory management for providing high performance distributed shared memory systems; the choice of the memory model and coherence protocol does not significantly influence the system performance for applications exhibiting high computation granularity and low state-sharing; and an efficient implementation of DSM requires a careful design of miscellaneous system services (such as synchronization and data servers).

# Chapter 2

# Issues in the design of DSM systems

As mentioned in Chapter 1, a designer needs to address several system issues during the design of a distributed shared memory system. These issues form the core of a DSM system design, and choice of solutions to these issues can significantly influence the overall system performance. In this chapter, we enumerate these issues and discuss the alternatives available for addressing these issues.

## 2.1 Virtual Memory and DSM

DSM is not true shared memory as is the case in shared memory multiprocessors (SMM). Thus remote memory accesses have to be reconciled with the memory management at each node. Of course, if the basic machine architecture does not support virtual memory then the solution could perhaps be simpler. However, if the basic architecture supports virtual memory then the DSM management and Virtual Memory (VM) management have to be integrated. In particular, the local memory at each node may be considered simply a cache of a global address space that spans the entire network. The DSM and VM management at each node would have to cooperate to ensure that the semantics implemented by the DSM manager and the VM manager are not compromised. The normal VM chores such as page replacement, swapping, and flushing have to be done in consideration with the DSM algorithms. Similarly, in satisfying a remote memory request, the DSM would have to consult the VM manager to get a page frame, etc. Upon release of a page, the DSM has to instruct the VM manager to invalidate page table entries and take other related actions.

The effectiveness of the DSM paradigm depends crucially on how quickly a remote memory access request is serviced, and the computation is allowed to continue, which in turn depends on several factors:

- the speed at which the VM system detects that a memory access fault (i.e. a page-fault) or a pre-fetching request entails a remote access
- the software overhead involved in the DSM protocol (i.e. coherence mechanism) for servicing a remote memory access request
- the software overhead involved in the communication subsystem (i.e. the basic transport protocol) for effecting the inter-node message communication to service the request
- the speed of the communication medium (i.e. hardware).

## 2.2 Granularity

There are two dimensions to granularity: *computation granularity* and *data granularity*. The former deals with the amount of computation a process has to do between synchronization or communication points in a multi-process computation. The latter deals with the amount of shared information processed during this computation phase.

Eggers and Katz [14] define "write-run" as a sequence of reads and writes by a given processor following an initial write executed by the same processor to a given shared memory location before an external read by a different processor to that shared memory location occurs. In a shared memory multiprocessor system, write-runs of representative applications may range from a few to a few tens of references. However, in a DSM system, write-runs of a few hundred instructions would be more appropriate given the latency for remote accesses.

Another distinction between the SMM and the DSM is in the data granularities of accesses that are practical in the two. In a uniprocessor memory hierarchy, the processor-to-cache transfer time is in the tens of nanoseconds, the cache-to-main memory transfer time is in the hundreds of nanoseconds, and the main memory-to-disk transfer time is in the order of milliseconds. Correspondingly, the granularity of transfer that makes feasible sense are: byte or word between the processor and the cache, a block of several bytes between the main memory and the cache, and a page ranging from 512 bytes to a several kilobytes between the main memory and the disk. DSM systems add a new dimension to the memory hierarchy, namely remote memory access across the network. The choice of the network plays a big role in determining the latency. Nevertheless, independent of this choice, there is a fixed software overhead to be incurred depending on the choice of the data transfer protocol on the network. Moreover, such remote memory accesses need to be integrated somehow with the memory management at each node. This requirement often forces the granularity of access to be an integral multiple of the fundamental unit of memory management (usually a page). However, it is possible to reduce the network latency by transferring the page partially. The key point to note is that the data granularity has to be sufficiently high to make the DSM paradigm viable.

## 2.3 Memory Model

In a uniprocessor, correctness of execution is ensured by preserving the order of memory references generated by a processor. Lamport [20] has proposed *sequential consistency* as a memory model for ordering shared memory accesses to ensure correct multiprocessor execution. In this model, the order of memory references generated by an individual processor is preserved, while the global order of memory references from all the processors is an arbitrary interleaving of the individual processors' reference streams that preserves the order of references emanating from each processor. Essentially, sequential consistency ensures that the view of the memory is consistent at all times from all the processors.

Insofar as correctness of multiprocessor execution is concerned, only the ordering of the shared memory references is of interest. Shared memory accesses may be categorized into three types:

- shared code,
- synchronization variables, and
- shared data variables.

Shared code is always read-only, and hence is always consistent. On the other hand, synchronization variables require that memory consistency be strictly preserved. Shared data variables normally require strict consistency as well. However, several applications exist wherein the program correctness would not be compromised even if there are temporary inconsistencies in the view of the shared memory as seen by distinct processes. Examples of such applications include asynchronous and iterative algorithms. Moreover, if the programs are written to obey some synchronization paradigm such as lock/unlock, and semaphore P/V, then ensuring a consistent view of shared data may be deferred to synchronization points in the program without compromising program correctness. Thus, sequential consistency is an overly restrictive memory model. This fact was first observed by Dubois, Scheurich and Briggs [13], who proposed *weak ordering* as an alternative to sequential consistency. Weak ordering requires that memory accesses from a process are performed in program order; synchronization operations are globally performed before allowing a process to continue; and all shared data accesses from a process are globally performed before issuing a synchronization operation. Several weaker memory models have been proposed. One such memory model is the *causal memory* model [1]. This model is based on the notion of *causality* [20], which is the fundamental event ordering mechanism in distributed systems. Similar to a message-passing system, the causal order is used to relate operations based on the *program* (local) order at processes and a *reads-from* order that is established between a write and its subsequent reads. This is similar to the *happens-before* relation defined in message-passing systems between an operation that sends a message and the operation that receives it. The causal memory model only guarantees that read operations do not return *causally overwritten* values. Another memory model weakens the ordering constraints by distinguishing between synchronization operations that acquire and release rights to access shared data [25]. Of course, all such weakening of the memory model assumes that it is possible to distinguish between two types of memory accesses: synchronization and read-write data. If such a distinction cannot be made then a conservative approach (such as sequential consistency) may be the only way to assure program correctness. Recently, there have been proposals for hardware support to make this distinction possible in SMMs [23, 33, 18]. DSM, on the other hand, is usually a software abstraction. Therefore, it is quite straightforward (with support from the compiler and/or operating system) to make this distinction possible and weaken the memory model.

## 2.4 Choice of Protocol

DSM assumes that all memory is globally shared. This assumption requires that independent computations started at different nodes see a consistent view of the shared memory. To facilitate this view would require a coherence scheme. Consistency maintenance of distributed shared memory is similar to cache coherence in multiprocessors. Shared memory multiprocessors such as Encore's Multimax, 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 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. The cache coherence algorithms that have been proposed for multiprocessors are viable 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, bus-based multiprocessors usually have the ability to invalidate (or update) all cached copies in one atomic bus cycle.

It is possible to devise distributed versions of cache coherence protocols to maintain the consistency of distributed shared memory. However, such implementations would suffer to some degree due to the mismatch in the capabilities of their intended environments and distributed systems. The definition of coherence that works well for shared memory multiprocessors is not appropriate in a distributed 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 expensive in a distributed system due to the cost of the invalidation messages. Invalidation involves at least sending a multi-cast message to all the nodes that have a read-copy of the data. Achieving reliable delivery of such multi-cast messages is expensive in a distributed system.

In reality, memory coherence and process synchronization are closely intertwined. A process acquires permission to read or write shared data invariably through some synchronization method. Absence of synchronization implies that any arbitrary ordering of simultaneous accesses to a shared location should yield valid results for a given computation. Therefore, in such cases there is no coherence requirement. In fact, it would be reasonable to argue that memory coherence should be defined in conjunction with process synchronization.

Not surprisingly, the solutions that have been suggested in the DSM environment are similar to the ones in the SMM caches. Broadly, these solutions may be classified into three categories:

1. **Write-invalidate policy**: In this protocol the writer acquires exclusive ownership by invalidating all copies before performing the write.

2. **Lock-based policy**: In this protocol lock requests (exclusive and shared) result in the data associated with the lock being sent to the requester along with the granting of the lock. Upon release of a lock, the associated data is sent back (if modified) to the server. Reads or writes to shared data without explicit locking follow single-copy semantics that does not allow multiple-readers or writers. A weaker form of read allows multiple-readers to shared data (without locking) but does not guarantee consistency.

3. **Write-update policy**: This protocol differentiates between two types of accesses: normal read/write and synchronization. Writes to shared data are buffered and consistency is enforced at synchronization points.

These three categories of cache protocols may be likened to deadlock prevention in an operating system, in that they prevent memory consistency violations from ever happening. It is possible to take a more liberal approach (similar to deadlock detection) and allow memory consistency violations to occur but have mechanisms in the system to detect such violations and take corrective action when they occur (see section 3.1).

In distributed systems, the number of messages is a measure of protocol performance. From this standpoint, the lock-based policy is expected to out-perform the other two, since coherence is maintained commensurate with the semantics of sharing in the computation. Moreover, since locking could be integrated with the data transfer, there is no need for any additional mechanisms for providing mutual exclusion for shared write accesses. In both write-update and write-invalidate policies there is a need to provide synchronization mechanisms on top of the coherence policy to assure mutual exclusion for multiple nodes requesting to write to the same page. However, lock-based policy has its drawbacks: In particular it does not have the generality of the other two policies. By decoupling memory coherence and synchronization, it is possible to devise synchronization mechanisms independent of the coherence policy. The lock-based policy requires explicit directives from the system software to know the semantics of sharing, while the other two do not require any such directives.

## 2.5 Synchronization

Another issue is the way interprocess synchronization is achieved in such systems. Extending the analogy of shared memory multiprocessors to DSM, it would seem that shared-memory style of synchronization would be expected in DSM systems as well. However, the granularity of accesses in DSM systems precludes using true shared memory style of synchronization such as Test-and-Set on arbitrary memory locations. One possibility is to combine synchronization with sharing as has been suggested in some multiprocessor cache protocols [23]. Another possibility is to have an orthogonal set of primitives to achieve synchronization. This latter approach is attractive since there could be situations where there may be very little sharing of data but independent computation may have to synchronize with one another. For example, in compute-intensive applications, such as the embarrassingly parallel kernel and matrix multiplication, interprocess synchronization is used only to indicate completion of computation. Some systems provide semaphore operations or lock operations in addition to the shared memory primitives.

## 2.6 Hardware Technology

There are two sources of overhead in a DSM system: the first is the communication overhead associated with the data transfer on the communication medium; and the second is the computational overhead associated with servicing remote memory requests. The choice of the communication medium (Ethernet, optical fiber, etc.) directly impacts the former, while the speed of the processor and any additional hardware support for DSM affects the latter.

In this chapter, we enumerated a set of issues that need to be addressed in the design of a distributed shared memory system. These issues form the basis of the work described in this thesis. In the next chapter, we present the work that has been previously done in the area of distributed shared memory.

# Chapter 3

# Related Work

Over the past decade, several systems have been developed and implemented that provide a shared memory abstraction in a physically distributed environment. However, the emphasis of the research has been restricted to the design, analysis, and implementation of algorithms for maintaining coherence of shared data. Some research has also been done to investigate the issues in providing reliable and recoverable distributed shared memory. In this chapter, we briefly summarize the work that has been done in the area of distributed shared memory, and qualitatively discuss these systems with respect to the issues outlined in chapter 2.

## 3.1 Apollo Domain

Apollo Domain [22] system is one of the earliest systems that employs DSM paradigm to assure consistency of shared objects in a local area network of personal workstations and data servers. It provides an integrated distributed environment with each node possessing a high degree of autonomy with additional system mechanisms that permit cooperation and sharing among the nodes. The Domain system allows users to name and access all objects in a transparent manner by having a distributed object storage system (OSS). The OSS is a flat address space of objects addressed by unique identifiers (UID's). The distributed OSS allows objects to be accessed from any node in the network. Processes could potentially access all objects by presenting their UID's and mapping the object into their respective address spaces.

The OSS is implemented in two layers. The first layer provides access to local objects that reside on the same node as the faulting process. The second layer provides access to remote objects in a transparent manner. On a page-fault, the OSS determines if the access is to a local object. If so then the object is read from the local disk and mapped into the process' address space. If the object is not locally available then it is located using an object locating service. After the object is located, the specific page is requested and mapped into the process' address space.

To assure consistency of replicated copies of an object a two-level approach is adopted. The lower level detects concurrency violations using a time-stamp based version number scheme for each object. The time-stamp corresponds to the time the object was last modified. Every node remembers the version number for all remote objects whose pages it has encached in its main memory. Every time an object is read from another node, its version number is returned with it. If it is the only page of the object encached in this node, its version number is remembered. If not, the returned version number must match the remembered version number for the object; otherwise a read concurrency violation has occurred. Every time a page of an object is written back to its home node, the current version number is sent with the write request and an updated version number is returned. The home node only accepts the page if the write occurs on a current version of the page; otherwise a write concurrency violation has occurred. A write request to a page updates both the home node's and the requesting node's time stamp for the object. The system also provides primitives to flush stale pages of cached object, inquiring current version number of an object, and sending back modified pages of a cached object.

The higher level provides an object locking mechanism. Several types of lock modes are provided including a multiple-readers/single-writer lock. Lock and unlock requests for remote objects are always sent to the home node. A lock request that is granted returns the current version number of the lock. This information is used to remove stale pages from the requesting node's main memory. The unlock operation forces modified pages back to the home node before the lock is released. In Domain, lock requests are not enqueued; if the lock is currently in use, then the requester is denied access to the lock and would have to retry later.

## 3.2 Ivy

Ivy [28] is a distributed shared memory system implemented on Apollo workstations interconnected by a token-ring network. It provides a shared virtual address space similar in concept to the Domain system with the difference that the granularity of access is a physical page in Ivy as opposed to an object in Domain.

In Ivy, a process address space is divided into two parts: a private part and a shared part. The private part is local to a process and cannot be accessed by any other process. The shared part is implemented using shared virtual memory. A process may access any memory location of the shared virtual memory through the shared part of its address space. To manage the shared virtual memory, each node has a memory mapping manager. The memory mapping manager implements the mapping between the physical memory of the processor and the shared virtual memory address space. The memory mapping manager at each node treats the physical memory as a cache for the shared virtual memory, and is responsible for maintaining the shared virtual memory coherent at all times. The shared virtual memory is implemented at the processor-level: Thus, once a page of the shared virtual memory is made available at a node by its memory mapping manager, it becomes accessible to all processes that execute at this node.

Ivy uses a write-invalidate type of coherence protocol to manage its shared virtual memory. The virtual memory is partitioned into pages. Individual pages can exist in read-only, write, or nil mode. Ivy uses multiple-readers/single-writer memory semantics. In this approach, all read-only copies of a page are invalidated when any processor attempts to write to the page. Three different flavors of the invalidation scheme have been implemented in Ivy. In the central scheme, a central manager maintains a table to keep track of the locations for all the pages. On a page-fault, the faulting processor asks the central manager for a copy of the page. The central manager then asks the *owner* of the page to send a copy of the page to the faulting processor. A node is said to be the owner of a page if it was the last node that modified the page. The second approach, fixed distribution scheme, is similar to the centralized scheme except that each node is assigned a pre-determined set of pages to manage. A mapping function is used to perform this distribution. This scheme avoids the single site bottleneck of the centralized scheme. The third scheme is the dynamic distributed manager algorithm that keeps track of the ownership for all the pages. This is done by adding a field called *probOwner* in each page table entry at all the processors. The probOwner field is used as a hint to locate the true owner of a page. A processor sends the request for a page to the node indicated in the probOwner field of the page. If the node that receives the page request is not the current owner of the page, it forwards the request to the node indicated in its page table. Initially, the probOwner field is set to some default value in all the processors. The probOwner field is updated whenever a processor receives an invalidation request, a processor relinquishes the ownership of a page on a read or write page-fault, or a processor forwards the page-fault request to another node.

On a read page-fault, the processor locates the owner of a page and sends a request to the owner of the page. The owner maintains a set of all nodes that have a read-only copy of the page in a *copyset*. The owner adds the faulting processor to the copyset of the page and sends a copy of the page to the faulting processor. On a write page-fault, the processor locates the owner of the page and sends a request to the owner of the page. The owner of the page sends the page and its copyset to the faulting processor. The faulting processor sends an invalidation message to all the processors in the copyset of the page. When all the invalidation requests have been acknowledged, the faulting processor restarts the blocked process.

Ivy provides synchronization mechanisms based on the primitives (*event-counts*) provided by the underlying operating system. An event-count supports four operations: *init*(count), *read*(count), *await*(count, value), and *advance*(count). Init primitive initializes an event-count. Read primitive returns the value of the event-count. Await primitive suspends the calling process until the event-count value reaches a specified value. Advance primitive increments the value of the event-count by one and wakes up any sleeping processes. Any process may use an event-count after the event-count has been initialized.

An extension of Ivy's memory coherence protocol has been implemented in the Mirage system at UCLA [17]. It allows a reader or a writer of a page to retain access to the page for a fixed duration of time regardless of the pending requests. This is done to guarantee forward progress of the computation by reducing thrashing of heavily shared data pages.

## 3.3 CLOUDS

CLOUDS [9] is a distributed operating system developed at Georgia Tech. One of the distinctive features of CLOUDS is its separation of two notions that have been traditionally inter-twined in most operating systems, namely, address space and computation. The former is specified by objects and the latter by threads in CLOUDS. An object is a passive entity (i.e. there is no process associated with it) that is part of a global name space. It specifies a distinct virtual space that is unique in the entire distributed system. The object encapsulates data that can be manipulated only from within the object. There are *entry points* in the object that are invocable from other objects. The entry points contain code for manipulating the data in the object, and may themselves invoke entry points in other objects. To allow concurrent execution of more than one computation in the same object, shared-memory style synchronization primitives are provided by the operating system.

A thread is an active entity that provides the notion of a computation. It executes in the context of an object. During the course of execution, a thread may invoke entry points in other objects. Thus, a thread is not associated with a single address space. Further, since these objects may not all be at the same node, a thread may span machine boundaries during the course of execution.

The collection of objects in CLOUDS represents a distributed shared virtual space. A thread traverses the address spaces of the objects that it invokes during its execution. Objects are composed of segments that form the basic unit of sharing. Each segment may be composed of one or more pages. Pages are the units of distribution. There is an entity, Distributed Shared Memory Controller (DSMC) [30] at each node that owns and maintains the segments that are created in the node. The DSMC provides a set of primitives for segment access and transport, and is responsible for preserving the consistency of the segments that it owns. DSMC uses a lock-based protocol for coherence maintenance that unifies synchronization and transport of data. It supports both exclusive (read-write) locks as well as shared (read-only) locks for segment access. Upon a lock request, the owner DSMC encloses the requested segment (parts thereof) in the message that grants the lock request, thus providing synchronization for free. A segment may be requested by a thread in one of the four modes: *read-only*, *read-write*, *weak-read*, and *none*. Read-only mode provides a non-exclusive lock on the segment while read-write mode provides an exclusive lock on the segment. Mode *none* gives exclusive access to the segment without locking the segment, i.e., any new request would result in the segment being yanked away to service the request.

These three modes provide sequentially consistent memory semantics for the nodes accessing the segments. However, there are situations where such strong memory coherence may not be required (e.g. a monitoring thread that wants to "inspect" the contents of a segment). For such purposes, the weak-read mode is provided. In this mode a current copy of the segment is sent to the requester. The thread would continue to receive updates to the segment if it had specified the *update* option on the weak-read request. A thread explicitly relinquishes a lock that it has acquired for a segment by using the *discard* primitive.

## 3.4 Mach and Agora

Mach [32] is a multiprocessor operating system kernel developed at CMU. It provides five basic abstractions: *task*, *thread*, *port*, *message* and *memory object*. A task is an execution environment that includes a virtual address space and an access list to system resources. A thread is a basic unit of CPU scheduling and it executes in the environment provided by a task; a port is a communication channel; and a message is a typed collection of data objects. Inter-thread communication is effected using messages on the ports. A memory object is a collection of data that may be mapped into the address space of any task. It is a structured mechanism for managing virtual memory independent of the underlying architecture.

An address space consists of a collection of memory mappings between a task and memory objects. A task may modify its address space by allocating and de-allocating a region of virtual memory. A task may also set protection attributes and specify inheritance of a region of virtual memory. It could create and manage a memory object that maybe mapped into the address space of another task. There are two ways of sharing memory between tasks in Mach: *copy-on-write* and *read-write*. In copy-on-write sharing, unrelated tasks share an address space without the actual data being copied. The first task that tries to write gets a copy of the shared address space. The copies become distinct and different from this point on. Read-write shared memory is created by allocating a memory region and setting its inheritance attributes to *shared*, *copy*, or *none*. Subsequent child task creation obeys the inheritance attributes specified by the parent. Pages specified as shared are physically shared between the parent and the children, i.e., there is exactly one copy of the page in the multiprocessor system. A child gets a logical copy of a page that is marked copy. Pages marked as none are not shared between the parent and the children.

Note that memory objects may be shared across the network. In this case, the physical memory is considered to be a cache for the memory objects. Associated with each memory object is a server called *pager* that manages the memory object. The pager handles any request for the memory object through a communication port. On a page-fault, the Mach page-fault handler checks whether the faulting thread has the access permissions for the page. If the permissions are correct then the page request is sent to the pager for the memory object. The pager services the request and sends back the page from the memory object. Mach provides sequentially consistent memory coherence semantics using a write-invalidate approach for sharing of pages across the network.

Although Mach's shared memory semantics are geared towards managing shared memory in a tightly-coupled multiprocessor, there is nothing in the design that precludes its use in a loosely-coupled system. In fact, Agora [5] is a system that is built on top of Mach with the specific intent of providing shared memory semantics in a loosely-coupled system. The Agora system allows processes to share structured data, e.g., abstract data types across heterogeneous architectures over a local area network. Agora uses the shared memory abstraction of the Mach operating system to share data structures among processes on the same machine. It also provides simple locks to synchronize access to shared data. To provide sharing across the network, the shared data structures are stored in the shared memory of the process that created the data structure. This copy of data is called the *master copy* while a copy of the data at another node is called a *cache copy*. Data is shared using copy-on-write semantics with updates to the cache copies. Writes to shared data are done on the master copy while data is read from the cache copy. A server process running on the node with the master copy is responsible for updating the copies of the data at other nodes. A read may potentially return stale data if the read occurs after the write is complete on the master copy and before the updates are propagated to the cache copies. The system expects that synchronization is implemented orthogonally using semaphores to guard against such stale accesses.

## 3.5 Memnet

Memnet [12] is a shared local area token-ring network being developed at the University of Delaware. It provides close coupling to the processors of a distributed multiprocessor system. There are three distinctive features of this project: first, it allows a granularity of access (32-byte chunks) finer than a page; second, it employs

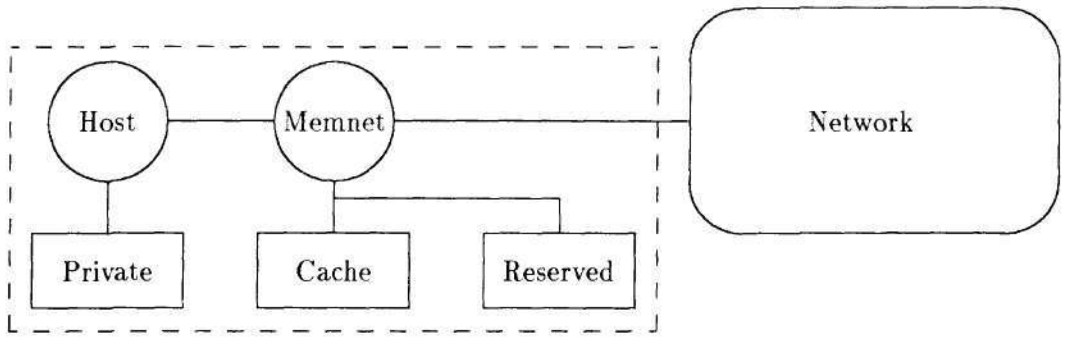

Figure 2: Memnet Architecture

dedicated hardware (Memnet device) to service remote memory accesses; third, it exploits the features of a special-purpose token ring network to implement a write-invalidate style of cache protocol. Given that there is an appreciable software overhead for remote access, dedicated hardware is almost a necessity to assure acceptable performance in DSM systems. The Memnet system does not support virtual memory. Therefore, the way DSM is managed on Memnet is very similar to cache management in a shared memory multiprocessor.

Each node in the system consists of a host and a Memnet device (Figure 2). The host has access to its private memory, which is inaccessible to other nodes. There is a large shared memory that is accessible from any node in the system. This shared memory is divided into 32-byte chunks and distributed among all the Memnet devices. The hardware address space seen by each host has two parts: private and shared. References to the shared part are passed to the associated Memnet device, which coordinates with other devices to resolve the references.

The physical memory associated with each Memnet is divided into two parts: *reserved* and *cache*. The reserved part is the permanent residence for the portion of the global shared memory that is managed by this Memnet device, while the cache is a temporary store for chunks that have been brought from remote Memnet devices. There is a chunk status table with each Memnet device that contains an entry for the chunks that are present in its physical memory. Memnet uses a write-invalidate style of cache protocol. A chunk may be in one of three states: valid (readable but not writable), exclusive (readable and writable), and invalid.

When a reference is generated for a chunk that is not locally available, then this request is sent around the token-ring. The first node that has valid copy of this chunk responds to this request. In case the reference is a "write", all other valid copies are invalidated before the chunk is written. In this sense, Memnet treats the token-ring as a logical broadcast bus. When a chunk has to be replaced from the cache, it is written back to its designated reserved area in the appropriate Memnet device. Since the system does not explicitly provide synchronization mechanisms, they have to be implemented at the user level to protect shared accesses.

## 3.6 Choices

Choices [34] is an operating system architecture developed at the University of Illinois at Urbana-Champaign. It uses class hierarchies and object-oriented programming to support the building of customized operating systems for shared memory and networked multiprocessors.

The virtual memory management system of Choices is similar to that of Mach. Choices uses the idea of a memory object that is cached in physical memory.

The memory object abstraction is provided by the *MemoryObject* class while the virtual memory abstraction is provided by the *Domain* class. The Domain class maintains information about the association between the virtual memory of an application and the memory objects. It provides methods to bind and release a virtual address to a memory location. Sharing of memory is achieved by mapping the same memory object into multiple Domains. Sharing across the network is achieved via the *DistributedMemoryObjectCache* class. This class is responsible for servicing page-faults on a node for shared data residing on a remote node. It communicates with its peers to maintain the consistency of shared data using a write-invalidate style of protocol similar to the distributed manager protocol of Ivy. The granularity of remote access is a page. Choices also provides for locking a page similar to the read-write mode of CLOUDS (see section 3.3) to guarantee atomic updates to a memory object by denying access until the lock is removed. Another variant (similar to Mirage) that Choices allows is retaining access to a page for a fixed duration of time regardless of other pending requests.

## 3.7 Mether

Mether [29] provides a set of mechanisms for sharing memory across the network on top of SunOS 4.0. Mether differs from most other distributed shared memory systems in that it does not provide sequentially consistent memory coherence. A process can continue to write on a page without the changes being reflected in other copies of the page. The other copies of the page may be updated in one of the following three ways: The process with the consistent copy of the page may initiate the updates to be propagated to all the other copies; a process holding an inconsistent copy of the page may invalidate its copy, resulting in a page-fault the next time it tries to access that page; a process holding an inconsistent copy of the page may explicitly request a consistent copy of the page. As should be evident, the user is responsible for tailoring the consistency requirements commensurate with the needs of the application. Mether provides a set of system calls to facilitate customizing the coherence requirements.

Mether provides for data driven page-faults. In a data driven page-fault, the process that caused the page-fault is blocked. No request for servicing the page-fault is sent by the server across the network. The page-fault is serviced when another process actively sends out an update for the page that caused the fault. Thus, the page-faults are completely passive. Mether defines two types of pages: a short page (32-bytes) and a full page (8192 bytes). A short page, referred to as a *subset*, corresponds to the first 32-bytes of a full page, while a full page is referred to as a *superset*. A process is ready to resume execution following a page-fault, as soon as the subset of the page is made available to this node.

## 3.8 Munin

Munin [7] is a distributed shared memory system that allows shared memory parallel programs to be executed on distributed memory multiprocessors. It differs from other distributed shared memory systems in that it uses multiple consistency protocols, and its use of a weaker memory model based on release consistency[1].

[1]In release consistency, memory consistency is enforced at a release synchronization point.

In Munin, shared program variables are annotated with their expected access pattern, and these annotations are then used by the runtime system to choose a consistency protocol best suited to that access pattern. At present, Munin supports seven different types of annotations: read-only, migratory, write-shared, producer-consumer, reduction, result, and conventional. These annotations are then used by the runtime system to select appropriate consistency protocol for sharing. Munin uses weaker sharing semantics (using release consistency) to mask network latency and reduce the number of messages required to keep memory consistent.

## 3.9 Hardware assisted distributed shared memory systems

In recent years, several systems have been proposed that implement the distributed shared memory abstraction in hardware. Two examples are the DASH multiprocessor [26], and KSR-1 [33].

The DASH architecture consists of processing nodes connected to an interconnection network. It uses a distributed directory-based cache coherence protocol. Each processing node consists of a small number of processors, called a cluster; a small portion of the shared memory; and a directory controller interfacing the cluster to the network. The memory hierarchy consists of two levels: cluster memory, and global memory. The cluster memory consists of the memory available with the processors of the cluster, while the global memory consists of memory available in all the clusters. On an access miss, an attempt is made to service the data request by the processors within a cluster. If the request cannot be serviced then it is sent to other processors outside the cluster. Each processing node has a directory memory corresponding to its portion of the shared physical memory. For each memory block, the directory memory stores the identities of all remote nodes caching that block. Using the directory memory, a node can send either invalidation or update messages to those processors that are caching the block. DASH uses an invalidation-based ownership protocol for maintaining consistency of shared data. Data consistency is maintained at the granularity of a cache line within a cluster, and at the granularity of a memory block between clusters.

KSR-1 is a 64-bit cache only memory architecture based on an interconnection of a hierarchy of rings. It implements a system virtual address (SVA) space that is global to the entire system. The SVA consists of the union of all the memory available with the individual processor caches. Each cache is subdivided in 16-Kbyte pages, which are further divided into 128-byte sub-pages. A data item on KSR-1 does not have any home associated with it. The data item moves from one cache to another cache as dictated by the memory access pattern of the application. An invalidation-based cache coherence protocol is used to maintain consistency of shared data. The unit of cache consistency is a sub-page. Access miss on a data item is sent on the local ring. If another cache is able to service the data request, it does so by sending the sub-page to the requesting cache. If no cache on the local ring has the data item, the request is propagated to the next level of the ring.

## 3.10 Discussion

In this section we compare the features of the DSM systems surveyed in this chapter with respect to the issues enumerated in chapter 2 (see Table 1). All the DSM systems presented in this chapter, with the exception of Agora and Munin have integrated the DSM management with the VM management, i.e., the DSM manager co-operates with the VM manager to service page-faults. Accesses to remote memory are referred to the DSM manager by the VM manager, which in turn satisfies the request using its own coherence protocol. Thus, a page-fault to local memory is indistinguishable from a page-fault to remote memory, insofar as a process is concerned. The difference may only be in the latency of service. Memnet does not support virtual memory. The shared memory in Memnet is at the physical address level and its management is similar to private caches in an SMM. References to shared memory are serviced by a Memnet device without the software overhead associated with the VM management. A similar approach is used in DASH and KSR-1 systems. CLOUDS provides a tighter integration of the VM system and the DSM system than any of the other systems by maintaining sharing information at the thread (process) level as opposed to processor level. In fact, such an integration is essential for a system that uses a lock-based coherence protocol to assure mutual exclusion (when needed) for a thread from all other threads, including ones that execute on the same processor. On the other hand, Agora and Munin use the library-approach. Both systems require the user to specify which data structures in the program are shared. This information is used by the DSM at runtime for maintaining coherence for shared data. Due to the additional overhead for processing shared memory requests in the library-approach, the library approach is expected to perform poorly as compared to the integrated-approach.

Data granularity (see section 2.2) has two aspects: *unit of transfer* and *unit of access and locking.* Unit of transfer refers to the amount of information shipped across the network to satisfy a remote request, while the unit of access and locking is self explanatory. One disadvantage of integrating DSM management with VM management is that the data granularity of shared memory may be dictated by the underlying VM architecture. Most VM architectures provide address mapping and protection attributes at the level of a page or multiples of a page. This feature could constrain the unit of transfer between the DSM managers to be a page, and could lead to inefficiencies if the size of the shared data structure is less than a page. For example, suppose the size of a data object is 512 bytes. In a DSM system with a page size of 8192 bytes, an access to this data object would result in the transfer of 8192 bytes, an unnecessary overhead of 1500%. This overhead could be reduced with some simple optimizations. One such optimization is implemented in Mether, where each access to shared data results in the transfer of the first 32-bytes of the page. A process may decide to fetch the entire page if it so desires by examining the first 32-bytes of the page.

Another disadvantage of such an integration (between VM and DSM) could be that the unit of access and locking may be constrained to be a page (or multiples thereof). If multiple data structures are allocated on the same page, then this constraint could lead to *false sharing*, wherein distinct private data structures appear shared due to co-location on the same page. However, the advantage of such an integration is that the hardware memory management facilities in the underlying architecture could be exploited to efficiently implement access control and locking of the shared data structures.

Most DSM systems follow a strict memory model for data accesses: a read to a memory location always returns the most recent write to that location. However, as was mentioned earlier (see section 2.3) such a strict memory model is not required to ensure program correctness. In fact, weakening the memory model could result in significant performance advantage since the DSM manager would not have to incur the overhead associated with the strict memory model. CLOUDS, Mether, Agora, Munin, and DASH are examples of DSM systems that have implemented weaker models of memory coherence. The weak-read mode of segment access in CLOUDS allows an application to acquire a segment without the overhead for consistency maintenance. The application may choose to receive updates to this segment asynchronously. In Mether, a process may continue to read a stale copy of a page that has since been written by another process. It is up to the application to either force updates to be propagated on writes, or request for updates to existing copies prior to reading a page. Similarly in Agora, a server process (that has the master copy) is responsible for propagating updates to cache copies at other nodes. Munin implements several types of memory coherence protocols including a weaker memory model based on release consistency. A user needs to specify the type of memory consistency that should be used for a particular memory object. DASH also implements the release consistency memory model, and uses invalidates and updates for maintaining coherence of shared data.

Ivy, Mach, Memnet, Choices and KSR-1 implement a write-invalidation based memory coherence protocol. Agora uses a write-update style protocol, while CLOUDS uses a lock-based protocol. All these systems prevent memory inconsistencies from happening (*a la* deadlock prevention in operating systems). Domain, on the other hand, uses a version-based protocol to detect inconsistencies after they have happened (*a la* deadlock detection in operating systems), and takes corrective action.

The lock-based protocol of CLOUDS avoids having to send invalidation messages upon writes as is the case with write-invalidation style protocols. However, in lieu of these messages, the requesting thread has to explicitly relinquish a lock that it has acquired using the discard primitive. While in terms of the number of messages (invalidations or discards) there may not be a significant difference between the write-invalidation approach and CLOUDS, the burden of generating these messages is spread out among the readers and the writers in CLOUDS while it rests completely with the writers in the write-invalidation approach. Explicit locking of segments has the advantage of reducing the thrashing effect across the network that is possible in the write-invalidation protocols for highly shared data. Choices and Mirage attempt to reduce the extent of this thrashing by allowing a processor to retain control of a page for a fixed duration of time during which it may disregard remote requests for the same page.

In systems that do not use the lock-based approach, there is need for mechanisms to synchronize access to shared data. These mechanisms are orthogonal to the DSM and are usually built at the application level. This approach has the advantage of flexibility in the choice of primitives for synchronization. CLOUDS, Domain, and Choices provide some form of synchronization integrated with the

DSM system.

There are two aspects to the overhead associated with a DSM system: processing (software), and communication (hardware). The former deals with maintaining the state information in software for the shared memory pages as well as the overhead for the message-exchange protocol on the communication medium. The latter relates to the cost of shipping a unit of data across the network. The processing overhead could be reduced by providing some form of hardware (firmware) support. Memnet, DASH and KSR-1 are examples of such a system.

## 3.11 Other Work

There has been very little in published literature in the the area of providing fault-tolerance and recovery in DSM systems. Wu and Fuchs [39] have examined the problem of rollback recovery in a DSM system. They have proposed a user-transparent check-pointing recovery scheme and a twin-page disk storage management technique to implement recoverable DSM. The proposed check-pointing scheme is integrated with the memory coherence protocol used to manage DSM. The twin-paged disk design allows check-pointing to proceed in an incremental fashion without an explicit undo at the time of recovery.

Brett Fleisch [16] has examined the issues concerning reliability of DSM systems in the event of site failures, specifically of sites that store the request queues. Reliability of a site is improved by storing shadow copies of stored requests and replica of data pages on backup sites. Stumm and Zhou [37] have proposed extensions for basic DSM algorithms to make the algorithms tolerate single host failures. Fault tolerance is achieved by replicating state information onto physically separated hosts.

In this chapter, we presented the work that has been previously done in the area of distributed shared memory. We described the work in context to the systems issues identified in chapter 2. In the next chapter, we present a detailed performance study of an implementation of DSM to better understand the interaction between various subsystems associated with DSM design.

| | Domain | Ivy | Clouds | Mach | Agora | Memnet | Choices | Mether | Munin | DASH | KSR-1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Virtual Memory | Yes | Yes | Yes | Yes | No | No | Yes | Yes | No | No | No |
| Granu-larity | Page | Page | Page | Page | Data struct. | 32-bytes | Page | Page | Data struct. | 16-bytes | 128-bytes |
| Memory Model | SC | SC | SC, UD | SC | UD | SC | SC | UD | SC, UD | RC | SC |
| Coherence Protocol | Ver. Num. based | WI | LBS | WI | WU | WI | WI and LBX | WU | WI, WU, and LBX | WI, WU | WI |
| Synch. | Yes | No | Yes | No | No | No | Yes | No | No | Yes | Yes |
| Dedicated Hardware | No | No | No | No | No | Yes | No | No | No | Yes | Yes |

*Legend:*

| | |
|---|---|
| SC : Sequential Consistency | |
| RC : Release Consistency | UD : User-defined |
| WI : Write Invalidate | WU : Write Update |
| LBX : Lock-based (exclusive) | LBS : Lock-based (exclusive and shared) |

Table 1: Comparison of DSM systems

# Chapter 4

# Distributed Shared Memory in CLOUDS: A Case Study

As mentioned earlier, the process of evaluating the impact of the system issues on the performance of DSM systems consists of three steps. As a first step, we would like to study the performance of an implementation of DSM. Such a study would primarily serve two purposes: First, it would help us better understand the interaction between various issues in a real environment; Second, the performance measurements obtained from such a study can be used for assigning costs to different components of the simulator, which is used for the simulation study (see Chapter 6). We use the CLOUDS distributed operating system as our target system for evaluation. The CLOUDS operating system provides a shared memory model of computation in a distributed setting. DSM is used as the primary vehicle for supporting this model of computation. For more details about the CLOUDS operating, system the reader is referred to [9]. The DSM primitives and the algorithms used for maintaining coherence of shared data are due to Ramachandran *et al.* [30]. For completeness, we present a summary of the DSM primitives in the next subsection.

## 4.1 DSM Primitives

In CLOUDS, associated with each segment, which is the unit of sharing, is a node called the *owner* node where the segment resides on stable storage. The DSMServer[1] at the owner node is responsible for maintaining the consistency of the segment. The DSM subsystem in CLOUDS uses a lock-based scheme to provide coherence of shared data. It supports two primitives for acquiring and releasing data: `get` and `discard`. The get primitive can be used to acquire a segment in one of the following four modes: read-write, read-only, none, and weak-read. *Read-write* mode signifies exclusive access to a segment guaranteeing that the segment will not be thrown away until the node explicitly discards the segment. *Read-only* mode indicates non-exclusive access with the guarantee that the segment will not change until the node explicitly discards the segment. *None* mode (the default) indicates exclusive access with no guarantee whether the segment will be thrown away or not. *Weak-read* mode signifies non-exclusive access with no guarantee whether the segment will change or not. A node can obtain a segment in weak-read mode with an option to receive updates of the segment implicitly.

The DSMServers implement a First-Come-First-Served queue discipline for processing remote segment requests. If the request cannot be honored immediately, it is queued at the server until the relevant lock is released.

When a get primitive is issued in mode read-write or read-only the local DSMServer sends a request to the remote DSMServer requesting the segment. The remote DSMServer locks the segment (if it is currently unlocked) with a read-lock

[1]Process/thread that handles DSM related requests.

(for read-only) or a write-lock (for read-write) and returns the segment to the requesting DSMServer. The owner node keeps a count of the number of readers associated with a segment. Thus concurrent access of a segment is allowed for reading, and exclusive access for writing. The segment is kept write-locked until an explicit discard is received, and read-locked until all the readers have discarded the segment.

Upon receiving a get request in weak-read mode, the owner DSMServer sends a copy of the segment to the requesting DSMServer. Note that this may not be an up-to-date copy if the segment is currently write-locked by some other node. If the option to receive updates is set, then the owner automatically sends updates of the segment when the write-lock is released. On receiving a get in mode none, the owner DSMServer does one of the following: the none mode request is queued if the segment is currently locked in either read-only or read-write mode; if the segment is available, it is sent to the requesting DSMServer who now becomes the *keeper* of the segment; a subsequent request for the same segment (in modes read-only, read-write, or none) is forwarded to the *current keeper* who forwards the segment to the requesting DSMServer. A segment held in mode none can be returned to the owner by using a discard primitive, or it can be taken away by its owner when the keeper DSMServer is instructed to forward the segment to another node. The algorithms for coherence maintenance are in [30]. The low-level communication protocol used in the CLOUDS operating system to support DSM is called `RaTP` [38]. It provides reliable transfer of data between nodes.

## 4.2 Methodology

This section briefly describes the methodology used for taking the performance measurements reported in the next section. The CLOUDS operating system is implemented on a configuration of Sun 3/60s connected by a 10Mbit/sec ETHERNET. Taking performance measurements posed two problems for us. First, the operating system did not provide any means of taking timing measurements from within the kernel. Second, the Sun 3/60 workstation does not provide any hardware timer that can be used to take timing measurements. We have installed a microsecond timer [8] to the Sun 3/60 workstations, and have added calls to the CLOUDS operating system to read the timer. The timing measurements are done by reading the timer before executing the code and after executing the code. The difference between these two readings gives the time (in microseconds) to execute the piece of code. Each call to read the timer has an overhead of 20 microseconds. The times reported in the next section are an average of number of such readings. The measured total time reported in the Tables 2, 3, 4, and 5 gives the high level time for a particular operation. The breakdown times have been obtained by inserting timer read calls around major function calls (e.g. context switching, transmit processing, reply processing, sundry overhead). A page refers to 8 Kbytes. We report the performance results for three categories of experiments.

## 4.3 Performance Measurements

The first category of experiments is basic system timings, summarized in Tables 2, 3 and 4. The tables also show a breakdown of the measured total time in terms of the low-level chores that constitute a system function. All the processing times in the table are measured numbers (by instrumenting the kernel) while network latency (wire overhead) is computed from the amount of data transmitted and the network bandwidth. The context switch time of 0.15 milliseconds involves switching between two different isibas.[2] This switching entails saving registers, and other state information associated with an isiba and installing the new isiba on the processor. There is very little MMU overhead associated with this switching since all the kernel isibas exist in the same machine address space. All the network communication times shown in Tables 2, 3 and 4 are between two compute servers. A null round-trip message (64 bytes) between two isibas making use of the Ethernet system object takes 1.59 milliseconds. The RaTP level null round-trip time is 3.56 milliseconds. Given this null round-trip time, a page transfer takes 12.3 milliseconds at the RaTP level as it breaks up an 8 Kbytes message into 6 packets (Table 4). Note that Ethernet allows a maximum packet size of 1532 bytes [36].

The second category of experiments exercises the DSM subsystem, that builds on the basic timings. Table 5 summarizes the results and gives a breakdown of costs for the `none` mode requests. Other modes would incur the same cost if there were no queueing delays at the servers. A `get` from a data server takes 15.5 milliseconds. Comparing the DSM and RaTP timings (Tables 4 and 5) for

[2]An isiba is similar to the concept of a thread or process in UNIX.

a page transfer, it can be seen that the DSM protocol has an overhead of 3.2 milliseconds. This overhead includes updating state information for the shared segment and coherence maintenance. A `get` with forwarding incurs an additional overhead of 3 milliseconds over the simple DSM transfer due to an extra message being exchanged between the owner and the keeper of the segment.

The third category of experiments, shown in Table 6, deals with the servicing of page-faults. In the case of remote page-faults, there is no disk access involved (i.e. page is in memory at the remote server).

- A segment is currently with the data server that owns it. A DSMServer on a compute server requests a page from that segment while servicing a page-fault. The average time for servicing such a page-fault is 16.3 milliseconds.
- A segment is currently in use at a compute server. To service a page-fault for this segment at another compute server, one level of forwarding (from the data server that owns the segment) of the request is involved. This three way exchange of messages results in a page-fault servicing time of 19.3 milliseconds.

It should be noted that the VM overhead of installing a page once a DSM `get` completes is only 0.800 milliseconds, difference between Tables 5 and 6. The last two entries in Table 6 show the time for servicing a page-fault on a segment owned by the local partition. Such faults do not require network messages, resulting in a time of 1.52 milliseconds for a zero-filled page and a time of 0.65 milliseconds for a non zero-filled page.

| Basic system performance (All times are in milliseconds) | Breakdown | Measured Total Time |
|---|---|---|
| Basic context switching | | 0.150 |
| Null Round trip time using Ethernet system object | | 1.590 |
| - Transmit processing by sender | 0.450 | |
| - Wire overhead (64 bytes, computed) | 0.051 | |
| - 1 Context switch at receiver | 0.150 | |
| - Transmit processing by receiver | 0.450 | |
| - Wire overhead (64 bytes, computed) | 0.051 | |
| - 1 Context switch at sender | 0.150 | |
| TOTAL TIME | 1.302 | |

Table 2: Basic system timings on CLOUDS

## 4.4 Analysis

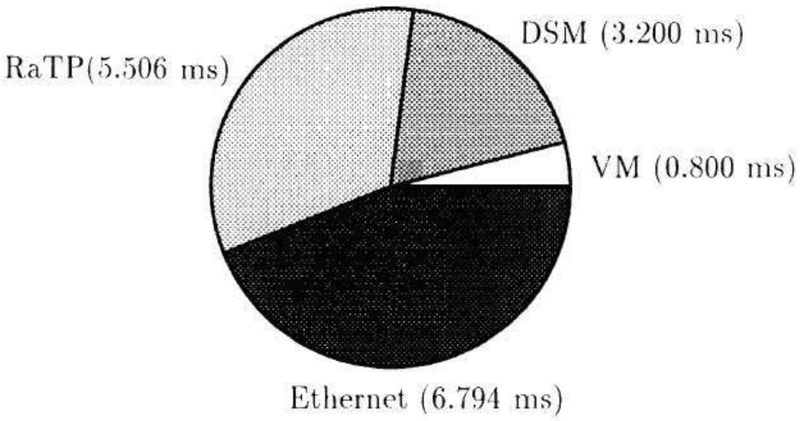

Figure 3: Cost associated with each subsystem in servicing a DSM page-fault. Total = 16.3 ms

Based on the performance measurements presented earlier, figure 3 shows the breakdown of the total time spent in each subsystem associated with servicing a DSM page-fault on CLOUDS. The total page-fault servicing time can be expressed as a sum of two types of costs: fixed cost and variable cost. The fixed cost consists

| Basic system performance (contd.) (All times are in milliseconds) | Breakdown | Measured Total Time |
|---|---|---|
| Null Round-trip time at RaTP level | | 3.560 |
| - Initiating request transaction (including a 32-byte copy of RaTP header into an Ethernet buffer) | 0.800 | |
| - Wire overhead (64 bytes, computed) | 0.051 | |
| - RaTP processing at server before wakeup of server thread | 0.265 | |
| - 2 Context switches at the receiver | 0.300 | |
| - Server processing of the request | 0.150 | |
| - Initiating reply transaction (including a 32-byte copy of RaTP header into an Ethernet buffer) | 0.600 | |
| - Wire overhead (64 bytes, computed) | 0.051 | |
| - RaTP processing before wakeup of the client thread | 0.265 | |
| - 2 Context switches at the sender | 0.300 | |
| - Client processing of the reply | 0.100 | |
| TOTAL TIME | 2.882 | |

Table 3: Basic system timings on CLOUDS (contd.)

of the overhead associated with the VM subsystem and the cost of sending a data request to the data server while the variable cost consists of the cost of sending the data back to the requester. The variable cost controls the latency of data as seen by an application process because the application process cannot start processing the data until the entire data page has been transferred. Ideally, in a DSM system, one would like to keep the *fixed cost per byte* (see equation 1) and *latency per byte* (see equation 2) low.

$$fixed\ cost\ per\ byte \ = \ \frac{VM\ overhead\ + data\ request\ cost}{PageSize} \tag{1}$$

| Basic system performance (contd.) (All times are in milliseconds) | Breakdown | Measured Total Time |
|---|---|---|
| Transfer time at RaTP level (64-byte request one-way, 8 Kbytes other-way) | | 12.300 |
| - Initiating request transaction (including a 32-byte copy of the RaTP header into a Ethernet buffer) | 0.800 | |
| - Wire overhead (64 bytes, computed) | 0.051 | |
| - RaTP processing at the server before wakeup of the server thread | 0.265 | |
| - 2 Context switches at the server node | 0.300 | |
| - Server processing of the request | 0.150 | |
| - Initiating reply transaction (8 Kbytes, 6 packets, including one copy of 8 Kbytes plus headers into 6 Ethernet buffers) | 2.724 | |
| - Wire overhead (8 Kbytes + headers, computed) | 6.794 | |
| - RaTP processing before wakeup of client thread | 0.737 | |
| - 2 Context switches at the client node | 0.300 | |
| - Client processing of reply (accepts data in a buffer, no copying involved) | 0.100 | |
| TOTAL TIME | 12.221 | |

Table 4: Basic system timings on CLOUDS (contd.)

$$latency\ per\ byte = (server\ proc.\ cost) * PageSize + \frac{PageSize}{Media\ bandwidth} \quad (2)$$

$$Total\ overhead\ per\ byte = fixed\ cost\ per\ byte + latency\ per\ byte \quad (3)$$

Equation 1 implies that systems that incur high VM overhead (such as the library-approach), and high cost for sending a request can minimize *fixed cost per byte* by increasing the page-size. However, equation 2 dictates that the page-size should be kept small for keeping the *latency per byte* low. Ideally, one would like minimize the *total overhead per byte* as given in equation 3. We will use these

| DSM operations (segments in memory) (All times are in milliseconds) | Breakdown | Measured Total Time |
|---|---|---|
| Get from a data server (no forwarding) | | 15.500 |
| - Basic RaTP 8 Kbytes transfer | 12.300 | |
| - 1 Context switch at the server | 0.150 | |
| - DSM processing at the server (updating state information) | 1.600 | |
| - One 8 Kbyte copy from Ethernet buffers into a client buffer | 1.450 | |
| TOTAL TIME | 15.500 | |
| Get from a data server (with forwarding) | | 18.500 |
| - Basic RaTP 8 Kbyte transfer | 12.300 | |
| - 1 Context switch at the server on the owner node | 0.150 | |
| - DSM processing at the server (updating state information) | 0.800 | |
| - Sending a forwarding request to the current keeper | 1.780 | |
| - 1 Context switch at the keeper node | 0.150 | |
| - DSM processing at the keeper (updating state information) | 1.600 | |
| - One 8 Kbyte copy from Ethernet buffers into a client buffer | 1.450 | |
| TOTAL TIME | 18.230 | |

Table 5: DSM timings on CLOUDS

equations in chapter 7 for deriving values for the page-size parameter for different system configurations.

Another point to note from figure 3 is that the majority of the total time is

| Page Fault Service | Time (in milliseconds) |
|---|---|
| Segment owned by a remote data server (no forwarding) | 16.30 |
| Segment owned by a remote data server (with forwarding) | 19.30 |
| For a perishable segment (with zero-fill) | 1.52 |
| For a perishable segment (without zero-fill) | 0.65 |

Table 6: Page-fault service times on CLOUDS

spent in the communication subsystem (communication protocol and data transmission). This observations indicates that for an efficient implementation of distributed shared memory, the cost of data transfer has to be reduced. Some techniques to bring this cost down is through using an improved communication protocol that has a relatively low overhead; using a faster communication medium to cut down the time spent on raw data transfer; data compression techniques for faster data transfer; and using additional hardware to improve processing overhead associated with the DSMServer.

In this chapter, we presented a detailed performance study of a distributed shared memory implementation on the CLOUDS operating system. The study provided us with breakdown of costs associated with various components of distributed shared memory system. The values are later used to assign costs to individual components of the distributed system simulator that has been designed to evaluate the system issues.

# Chapter 5

# Implementation and Analysis of Benchmarks

The second step in our evaluation process is to study the performance of several applications on top of a distributed shared memory system. Such a study would provide us with insights into the performance of DSM with respect to the various design alternatives available for addressing the system issues. For this purpose, we selected a set of six applications. These applications range from highly parallel computation kernels to asynchronous algorithms. Each application exhibits different characteristics with respect to the memory access patterns, amount of computation granularity, amount of data granularity, and amount of synchronization. These applications are implemented on the CLOUDS distributed system. In the following sections, we first present the system architecture that is used in the study, followed by the the performance of each of the applications.

## 5.1 System Architecture

The system[1] used in our study consists of a set of Sun 3/60 workstations connected via a 10 Mbit/sec Ethernet. Logically, the nodes can be classified into three categories. Compute servers are the nodes where processes comprising the distributed application execute. Processes running at different compute servers can share data, caching the shared state in their local memories. Data server nodes store the shared state when it is not cached at compute servers and also maintain information needed for coherence activities. Finally, we have synchronization servers which implement synchronization constructs used to coordinate access to data shared by processes. The three types of servers capture the functionality provided by the system. A given node may act interchangeably as a compute server, a data server and a synchronization server.

We studied these applications with respect to two coherence protocols:[2] write-invalidate and write-update. These protocols are implemented in the operating system.

• `Write-invalidate`: The implementation of the write-invalidate protocol uses a static owner to provide sequentially consistent memory. Upon a page-fault, the page is requested from the data server (owner). On writes, read copies are invalidated. The data server keeps information about the current writer and the readers that have cached copies.

• `Write-update`: The write-update protocol provides a weak memory

[1]Due to resource constraints, we studied the performance of these applications on only one system architecture.

[2]The lock-based scheme was not studied as its implementation was flawed.

model. It is based on the assumption that all program are written with some synchronization model in mind. Therefore, it is possible to defer consistency actions to certain synchronization points. One of the key problems in DSM systems is the potential for *false-sharing* that exists in a page-oriented implementation. With an invalidation-based protocol it is impossible to avoid this false sharing even if synchronization information is used to defer the consistency actions. Therefore, we have implemented an update-based protocol. The basic idea is the following. A node records all modifications to a page in a shadow copy (transparent to the program). Prior to exiting a synchronization region, an XOR of the original page and its shadow is generated for each dirty page (similar to the *diff* in [7]). The modifications to the data are sent to the data server (owner). The data server merges the modifications to its copy of the page, and sends the modified page to all nodes that are interested in receiving the updates for the page. Upon receiving the modified page, the program is allowed to exit the synchronization region. Thus, the write-update protocol allows multiple nodes to actively write-share a page, thereby avoiding the penalties due to false-sharing which are inherent in invalidation based systems. On the debit side, this memory system does incur the overhead of making a shadow copy for modified pages, generating the XOR-pages, and applying an XOR-page to the original data page. Further, there is still a potential for false-sharing in the form of updates to a page (or parts thereof) that a node is no longer interested in. This could be avoided if the memory system allows a way for a node to "unregister" itself from active sharing. The correctness of allowing concurrent writes to the same page is guaranteed by the assumption that the program itself obeys a synchronization model.

We use page-faults, access violation, and synchronization events to perform coherence activities. For example, when a reference is made to a page that is not cached, the fault handler requests the page from the appropriate data server (there is a one-to-one correspondence between a given page and a data server). When the page is received, it is mapped and the faulting process continues. The implementation of the synchronization constructs (locks, semaphores and barriers) is centralized. For a given synchronization variable, a single server maintains its state and the queue of processes blocked on it.

In the following sections, we present the results of the performance of the applications for the two coherence protocols. The timing measurements are done using a microsecond timer. Prior to the start of the measurements, all shared data is prefetched by the compute servers; therefore, the times do not include times for any disk I/O.

## 5.2 Embarrassingly Parallel Benchmark

The embarrassingly parallel (EP) problem is typical of many Monte-Carlo simulation applications. The problem requires generation of Gaussian random deviates according to a prescribed scheme and tabulation of the number of pairs in successive square annuli. This kernel exhibits a high degree of computation granularity with the only requirement for communication being the combination of the 10 sums from the individual processors at the end. As a result, this kernel is expected to perform well regardless of the underlying coherence protocol.

### Implementation of Clouds

We ran the problem on Clouds for $\mathcal{N} = 2^{16}$ iterations. The $\mathcal{N}$ iterations are equally divided among the available number of processors. The number of processors is varied from 1 to 6. Each processor computes the successive square annuli, and at the end updates the global table. The kernel performs equally well for the two coherence protocols. Table 7 shows the completion times and achieved speedups for this application.

| # of | write-update | | write-invalidate | |
|---|---|---|---|---|
| Proc. | Time (sec) | Speedup | Time (sec) | Speedup |
| 1 | 178.66 | - | 176.00 | - |
| 2 | 89.23 | 1.99 | 88.48 | 1.99 |
| 3 | 62.10 | 2.88 | 59.06 | 2.98 |
| 4 | 47.35 | 3.77 | 46.57 | 3.78 |
| 5 | 36.84 | 4.85 | 36.67 | 4.80 |
| 6 | 30.30 | 5.90 | 30.53 | 5.76 |

Table 7: Completion times and speedups for the Embarrassingly Parallel benchmark, $\mathcal{N} = 2^{16}$ iterations

## 5.3 Integer Sort Benchmark

Integer sort is used in "particle-in-cell" applications. The problem statement for the integer sort benchmark requires that $\mathcal{N}$ keys be sorted in parallel. The keys are generated by a prescribed sequential key generation algorithm, and are stored contiguously in shared memory. The benchmark requires computing the rank for each key in the input sequence.

## Implementation on CLOUDS

We have implemented two versions of the integer sort benchmark on CLOUDS. Both versions use the bucket sort algorithm. However, the task graphs for the two version are quite different. In version $\mathcal{V}1$, the generation of ranks for the keys is done in parallel, while in version $\mathcal{V}2$, the generation of ranks for the keys is done sequentially. Figures 4 and 5 show the task graphs for the two versions. In $\mathcal{V}1$, each key is read and count of the bucket to which it belongs is incremented. A prefix sum operation is performed on the bucket counts. Finally, the keys are read again and assigned ranks using the prefix sums. The algorithm has been shown to perform well on KSR-1, a tightly-coupled shared memory machine and has been adapted from [31]. In $\mathcal{V}2$, each processor reads a portion of the keys and updates the count of the bucket to which the individual keys belong. The final rank assignment for the keys is done by one processor using a shared data structure that contains the sum of all bucket counts computed by individual processors. The key difference in the two versions is the number of synchronization operations (barriers) that are performed. Version $\mathcal{V}1$ performs 7 synchronization operations (all barriers), while version $\mathcal{V}2$ performs only one barrier and a set of semaphore operations. The motivation for two different implementations is to study the effect of the structure of the task graph on the application's performance in a distributed system. Each version sorts $\mathcal{N} = 2^{20}$ keys. For the two versions, one would expect that as the number of processors are increased, the completion time for the benchmark decreases.

**Version $\mathcal{V}1$**

As one increases the number of processors to solve the problem, one would expect that the completion times to decrease because the total work is equally divided among participating processors. However, the results of the implementation of version $\mathcal{V}1$ are quite surprising as almost no speedups observed for any of the two coherence protocols (see Figure 6). The implementation using the write-invalidate protocol performs poorly due to the effects of *false-sharing.* False-sharing causes data pages that contain the data structures for computing the key densities to thrash between processors, thereby negating any performance gains achieved due to increased parallelism. On the other hand, the problem of false-sharing is absent in the write-update protocol. One would, therefore, expect it to perform better than the write-invalidate protocol. The write-update protocol performs poorly because of the high overhead it incurs at synchronization points. As one would recall, version $\mathcal{V}1$ performs 7 barriers operations. Prior to each barrier operation, the write-update protocol performs global writes to shared data. As it turns out, this operation is quite expensive to perform in a distributed system because all modifications need to be propagated to all participating processors, a potential $O(n^2)$ messages. This is an artifact of the structure of algorithm wherein during different phases, each processor accesses different portions oi the shared data. A different algorithm that enforces localized accesses to shared memory by individual processors would perform better as each processor can specify which updates to global memory it would like to receive, thereby eliminating the potential for $O(n^2)$ messages at synchronization points. For the implementation of version $\mathcal{V}1$ on the write-update coherence protocol, all gains due to the absence of false-sharing are negated by the need to perform global writes at synchronization points.

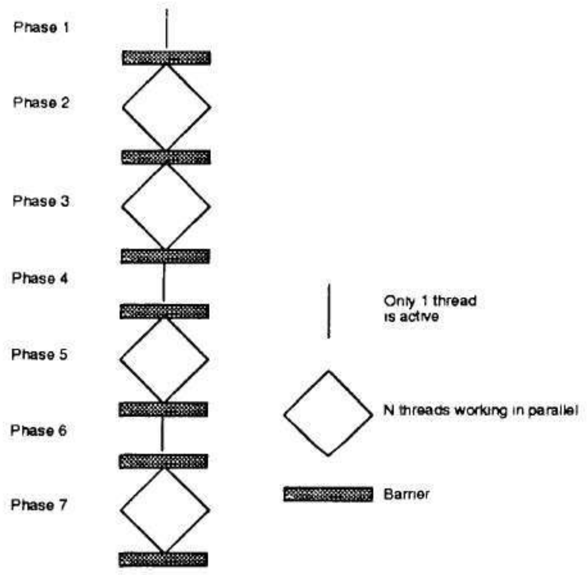

Figure 4: High level structure of the Integer sort benchmark, Version $\mathcal{V}1$

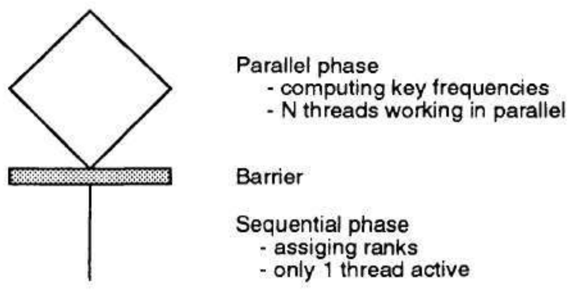

Figure 5: High level structure of the Integer sort benchmark, Version $\mathcal{V}2$

## Version $\mathcal{V}2$

The second version of the integer sort benchmark has two distinct phases: a parallel phase, and a sequential phase. In the parallel phase, all processors read in the key values from their respective portions and increment the bucket counts. At the

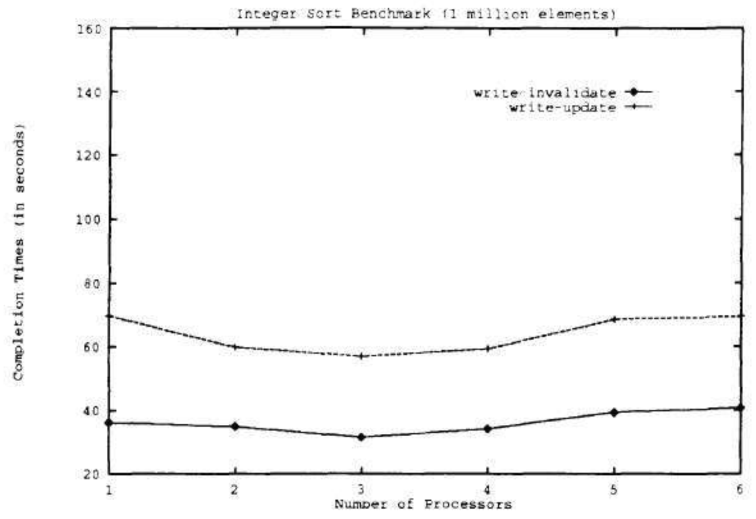

Figure 6: Completion times for the Integer Sort benchmark, Version $\mathcal{V}1$, $\mathcal{N} = 2^{20}$ keys

end of the parallel phase (before executing the barrier), each processor updates the global bucket counts for the keys. After the barrier, one processor reads in the global bucket counts and assigns the ranks to individual keys. The sequential component of the algorithm constitutes approximately 60% percent of the total execution time on one processor. This version, therefore, has very limited parallelism. Both the write-update and write-invalidate perform equally well. However, the completion times on these protocols starts to saturate beyond 4 processors, as the sequential component of the algorithm starts to dominate. Table 8 shows the breakdown for the sequential and parallel phase of the algorithm for the two coherence protocols.

| # of | write-update | | | write-invalidate | | |
|---|---|---|---|---|---|---|
| Proc. | Tot | Seq | Par | Tot | Seq | Par |
| 1 | 13.13 | 7.76 | 5.37 | 13.18 | 7.77 | 5.44 |
| 2 | 10.59 | 7.88 | 2.71 | 10.69 | 7.77 | 2.91 |
| 3 | 9.70 | 7.86 | 1.84 | 9.93 | 7.85 | 2.08 |
| 4 | 9.46 | 7.03 | 1.43 | 9.65 | 7.85 | 1.79 |
| 5 | 9.18 | 7.87 | 1.21 | 9.41 | 7.85 | 1.55 |
| 6 | 9.03 | 8.00 | 1.03 | 9.27 | 7.86 | 1.40 |

Table 8: Breakdown of times for the Integer Sort algorithm version $\mathcal{V}2$.

## 5.4 Matrix Multiplication Benchmark

In matrix multiplication, the product of two $\mathcal{N} \times \mathcal{N}$ matrices is assigned to a third resultant matrix. This algorithm is highly parallelizable as no interprocess communication is needed during the computation of the individual rows of the resultant matrix.

### Implementation on Clouds

On Clouds, we ran this benchmark for $\mathcal{N} = 256$ rows. Each participating processor computes a set of rows of the resultant matrix. At the end, a barrier is executed to indicate the end of the computation. The completion times and speedups are summarized in Table 9. As no interdependencies exist between computation of any two rows, one would expect the completion times to be independent of the underlying coherence protocol. This is indeed the case.

| # of | write-update | | write-invalidate | |
|---|---|---|---|---|
| Proc. | Time (sec) | Speedup | Time (sec) | Speedup |
| 1 | 449.48 | - | 446.13 | - |
| 2 | 226.17 | 1.99 | 227.12 | 1.96 |
| 3 | 152.47 | 2.95 | 155.15 | 2.88 |
| 4 | 114.79 | 3.92 | 121.11 | 3.68 |
| 5 | 93.13 | 4.83 | 95.78 | 4.66 |
| 6 | 76.36 | 5.89 | 80.27 | 5.56 |

Table 9: Completion times and speedups for the Matrix Multiplication benchmark for $\mathcal{N} = 256$ rows

## 5.5 Conjugate Gradient Benchmark

In the conjugate gradient (CG) benchmark, the power method is used to find an estimate of the smallest eigenvalue of a symmetric positive definite sparse matrix with random pattern of non-zeros. The algorithm for CG is adapted from [31].

### Implementation on Clouds

We have implemented CG with an input sparse matrix of size $\mathcal{N} = 1400$ rows, and 100300 non zero entries. About 90% of the time for this benchmark is spent in performing sparse matrix multiplication. Therefore, in the parallel version of the benchmark, only the sparse matrix multiplication portion is parallelized. The implementation consists of alternating sequential and parallel phases. The parallel phase corresponds to the sparse matrix computation. As matrix multiplication forms the core of the benchmark, one would expect speedups similar to ones obtained for matrix multiplication benchmark (see section 5.4), i.e., the underlying coherence protocol would not impact the performance of the benchmark. The results for the implementation of the benchmark are quite surprising. An implementation with no data partitioning shows significant performance degradation beyond 2 processors for the write-invalidate protocol. This degradation is due to the false-sharing of the vector containing the result of the sparse matrix multiplication. Unlike the matrix multiplication benchmark, where the resultant matrix is distributed among the processors, the CG benchmark requires the result of the sparse matrix multiplication be placed in a single output vector. The output vector is then used during the sequential phase of the computation. In our implementation the output vector is an array of 1400 floating point numbers that can fit in one 8192 byte physical page. Although individual processors write to disjoint rows of the output vector, the fact that the vector resides on one physical page causes it to thrash between processors, thereby degrading the performance. On the other hand, there is no problem of false-sharing in the write-update protocol as multiple writers to a physical page are allowed to coexist. However, another problem which causes performance degradation for the write-update protocol is performing global writes prior to a synchronization point. The global writes causes the generation of $O(n^2)$ messages as each processor sends its writes to all other processors. This causes severe performance degradation especially for large number of processors. One can see in Figure 7, the curve for write-update with global updates starts to show degradation beyond 3 processors. In CG, the output vector is used only by 1 processor that performs the sequential phase. Eliminating the generation of $O(n^2)$ messages by allowing a processor to *turn-off* receipt of those updates that it is not interested in, causes the performance to continuously improve for the write-update protocol upto 5 processors (curve marked write-update with controlled updates). Similarly, doing careful data partitioning of the output vector for the write-invalidate protocol causes the performance to improve upto 5 processors. However, beyond 5 processors, due to reduced computation granularity, the cost of moving data to the node that performs the sequential component starts to dominate, resulting in loss of performance for the two protocols.

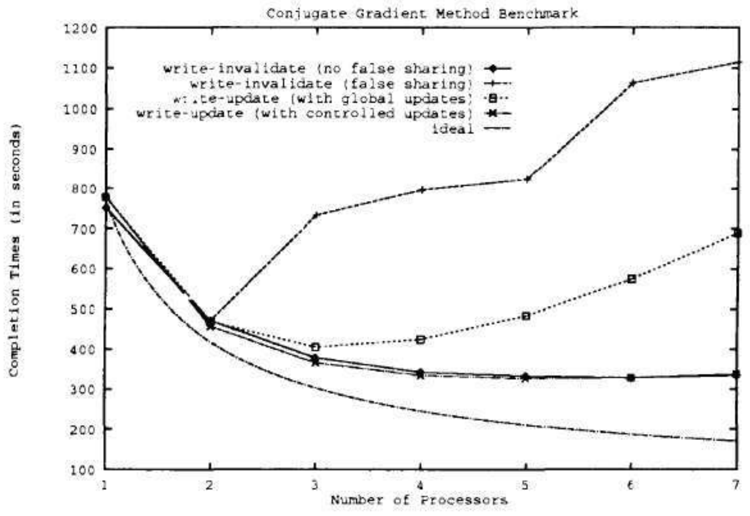

Figure 7: Completion times for the Conjugate Gradient benchmark

## 5.6 SCAN Benchmark

The transaction processing benchmarks [15] consists of three basic benchmarks: sort, scan, and debit-credit. Out of these three, we studied the implementation of the SCAN benchmark on CLOUDS. The SCAN benchmark specifies a sequential scan of a file, reading and updating records. The high level structure of the SCAN benchmark is shown in figure 8. A duplexed transaction log is automatically maintained for transaction recovery. Such scans are typical of end-of-day processing in on-line transaction processing systems. The benchmark requires that each record be locked, read, modified, updated, and unlocked. In the parallel implementation of the algorithm, the data is partitioned among available processors, and each processor performs a sequential scan of its portion of the database. In our implementation, we did not implement the transaction recovery log.

```
for i in 0 to TotalNumberOfRecords do
      WriteLock(record i);
      Read record i;
      Change record i;
      Rewrite record i;
      UnLock(record i);
endfor ;
```

Figure 8: Pseudo code for the SCAN benchmark

## Implementation on CLOUDS

We ran the SCAN benchmark on CLOUDS for $\mathcal{N} = 10000$ records. Each processor performs a sequential scan of its portion of data. Exclusive access to records is controlled by assigning one lock per 100 records. The benchmark has very little communication overhead as there is no contention for data. The benchmark shows good performance for the two coherence protocols. Figure 9 shows the completion time for the benchmark for the two protocols.

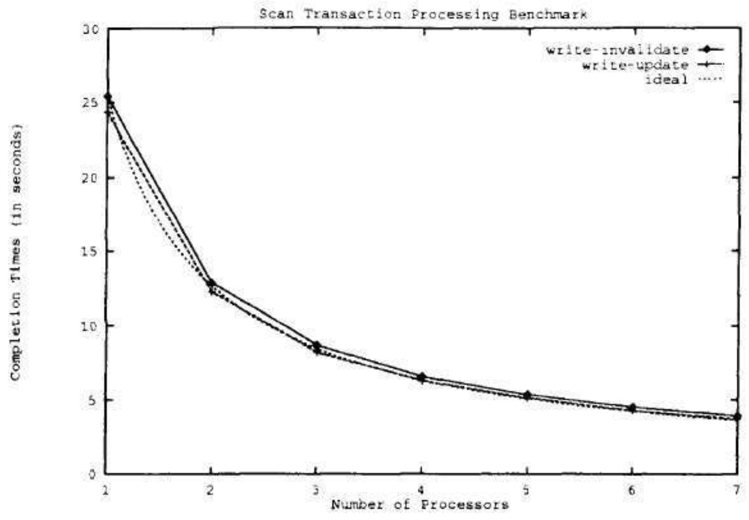

Figure 9: Completion times for the SCAN benchmark

## 5.7 Traveling Salesman Problem

The traveling salesman problem (TSP) requires the computation of the shortest tour that visits all the cities exactly once. A set of cities, along with a starting city, and distances between cities is specified as input. TSP can be solved with a branch-and-bound algorithm. The algorithm constructs a tree of possible solutions with the root of the tree being the starting city. The path from the root to a leaf node represents a tour that visits all the cities *en route* exactly once. The goal is to find the path from the root to the leaf node with the minimum weight. The sequential implementation of the algorithm uses a depth-first heuristic.

### Implementation on Clouds

The parallel implementation of TSP on Clouds is similar to the one in [27]. A master processor generates a number of jobs consisting of the partial paths for the top two levels of the search tree. The jobs are kept in a globally shared queue. Each participating processor picks up a job from the central queue, and using the depth-first search heuristic, computes the possible paths to cover all cities in the tour. The value of the best computed path so far is kept in a globally shared variable. At each level of the tree, a processor compares its current tour value with the global minimum. A path is abandoned (or pruned) if the current tour value exceeds the global minimum. A processor can update the global minimum only if the tour value of the path computed by it is smaller than the global minimum. The problem completes when all jobs in the queue have been processed.

| # of | write-update | | write-invalidate | |
|---|---|---|---|---|
| Proc. | Time (sec) | Speedup | Time (sec) | Speedup |
| 1 | 58.73 | - | 54.48 | - |
| 2 | 35.98 | 1.63 | 37.96 | 1.44 |
| 3 | 27.57 | 2.13 | 31.37 | 1.74 |
| 4 | 25.52 | 2.30 | 27.95 | 1.95 |
| 5 | 25.92 | 2.27 | 27.69 | 1.97 |
| 6 | 27.29 | 2.15 | 27.93 | 2.01 |

Table 10: Completion times and speedups for the Traveling Salesman Problem, 11 city tour

The only actively shared piece of data in TSP is the current best tour value. To prevent false-sharing, we stored the current best tour value on a separate page. The frequency of updates to the current best tour value depends on the weights assigned to the input city matrix, and how soon the global minimum is found in the computation. In our implementation, we use a $11 \times 11$ distance matrix with the distances uniformly distributed over the interval $[0, 100]$ units. We observe that the speedups achieved for the algorithm depend upon how soon is the global minimum found in the computation. If the minimum for the tour is found earlier in the computation then close to linear speedups are observed. Table 10 shows the completion times and speedups for the write-update and write-invalidate coherence scheme. One key point to note is that the speedup saturates at about 2 for four or more processors. The reason for this saturation is that the cost of propagation of the best tour value starts to dominate. In both these protocols, any change to the current best tour value results in the propagation of an entire page (8192 bytes in CLOUDS) of data. This is a high cost of sharing a value which is just an integer.

In this chapter, we presented the results of the experimental study of six applications on the CLOUDS distributed system. These applications range from highly parallel computation kernel to asynchronous algorithms. The experimental study provided us with the comparison of the two coherence protocols: write-update and write-invalidate, for different application workloads. The study also highlighted other factors such as false-sharing and synchronization cost that can significantly impact the performance of an application. In the next chapter, we evaluate the issues, which could not be evaluated via experimental studies, using a simulation-based approach.

# Chapter 6

# Simulation Studies

The third step in our evaluation process is to study the effects of the system issues that could not be evaluated via experimental study using simulation. As mentioned earlier, the goal of this thesis is to evaluate the design alternatives that are available for addressing the system issues. We use a simulation-based approach rather than an experimental approach because the latter approach places constraints on the study by limiting the choice of alternatives that can be studied. The constraints are primarily placed on the study of the effect of the technology factors (such as processor speed, speed of the communication medium, physical page size, and additional support for distributed shared memory) on the performance of the distributed shared memory system. If one would like to study the effects of different alternatives using an experimental approach then he/she has to build all possible system configurations that need to be studied. Such an approach is very expensive, both resource-wise and time-wise, to realize if a large number of system configurations need to be studied. On the other hand, a simulation-based approach offers the flexibility to easily model different system configurations by tweaking the parameters of the simulator. One drawback of a simulation-based approach is that the results obtained via a simulation study are not quite exact.

To remedy this drawback, we do two things: First, the costs assigned to the different components of the simulator are obtained from the performance study of an implementation of DSM (see Chapter 4). Second, we validate the workload model, used to drive the simulator, using some of the applications that are implemented on the CLOUDS distributed system (see Chapter 5).

In this chapter, we present the results of the simulation study. We first describe the design of the simulator and the workload model used in the study. This is followed by the validation of the workload model. We conclude the chapter by presenting some of the results of the study.

## 6.1 Simulator

We have constructed a simulator to evaluate the design alternatives presented earlier. The simulator is written in CSIM [35], a process oriented simulation language. The distributed system modeled by the simulator consists of a collection of nodes interconnected by a local area network. Each node in the network has a processor, a DSM coprocessor, and local memory. The local memory acts as a cache for the portion of the distributed shared memory currently residing at the node. The processor generates memory references according to a specified workload model. The workload model is described in detail in section 6.2. Remote memory references are serviced by the DSM coprocessor at each node in concert with other DSM coprocessors. In the simulator, each node is modeled as a set of three CSIM processes: a compute engine, a DSM server, and a media server. The interconnection network is modeled as a CSIM facility. The compute engine models a processor with associated local memory. Shared references which are not currently encached in the local memory are communicated to the DSM server by the compute engine. The DSM server simulates the appropriate coherence protocol. The media server models the communication subsystem of a node. It differentiates between two types of messages: CONTROL and DATA. Each control message is 64 bytes long while the size of the data message is determined by the *page-size* parameter used in the simulation. The media server models the bandwidth characteristics of Ethernet and an optical fiber. It models the contention aspects of using a shared broadcast medium without modeling the collision and back-off aspects that are inherent in an Ethernet type of protocol. In addition to these three per node CSIM processes, a CSIM process serves as a centralized lock server. Figure 10 shows the conceptual picture of the simulator.

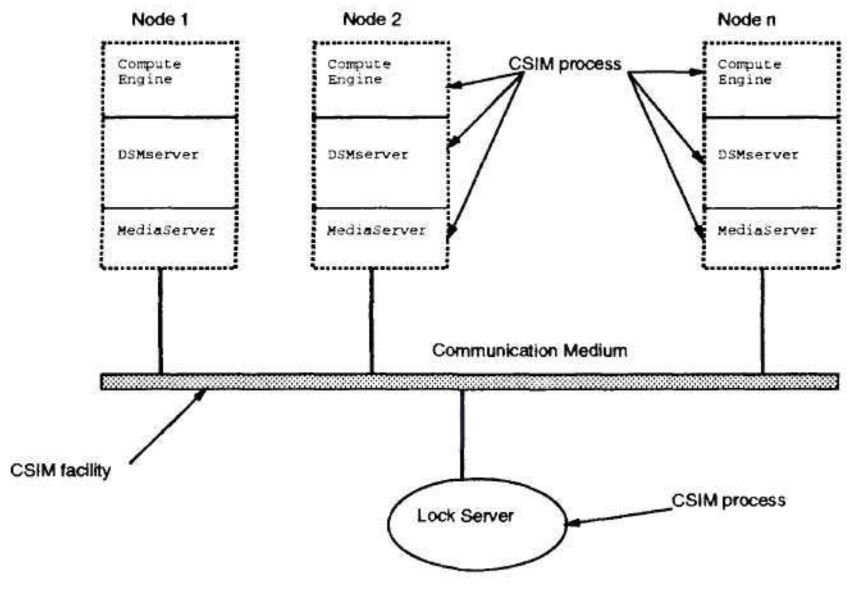

Figure 10: Conceptual picture of the simulator

## 6.2 The Workload Model

Each simulator needs to be driven by a workload model. There are three types of workload models that can be used: execution driven, trace driven, or a probabilistic workload model. In execution driven simulations [11], the application programs are allowed to execute on native hardware, and only interesting events are captured and executed on a simulator. The advantage of using an execution driven simulation is that the simulation time is considerably reduced as most of the application code executes at the speed of the host processor. However, the disadvantage of this approach is that the simulation is closely tied to the hardware architecture of the native machine, thereby limiting the set of alternatives that can be studied. In trace driven simulations, traces from a application are captured and used to drive a simulator; while in a probabilistic workload model, the memory reference stream of individual processors is generated using some probability distribution. The advantage of using a trace-driven simulation is that the workload model accurately models an application. However, trace driven simulation has the limitation that only a few number of applications, which have available traces, can be used. These set of applications may not be representative of all application workloads. Therefore, in view of these limitations, we chose to use the probabilistic workload model to drive our simulations. The probabilistic workload model has the flexibility of modeling a wide variety of workloads by tweaking the probabilities associated with the workload model, thereby allowing us to study a wide range of workload models. The disadvantage of the probabilistic workload model is that the workload may not correspond to any real application. To remedy this defect, we validated our workload model with some of the distributed applications that have been implemented on the CLOUDS distributed system.

Archibald and Baer [3] have proposed a simple memory reference generator based on a probabilistic approach to evaluate cache coherence schemes in a shared memory multiprocessor. In their model, each processor generates a memory reference stream. A memory reference (read or write) could either be to private or shared blocks; locality of references to shared blocks is modeled by increasing the probability for accesses to recently used shared blocks. The interaction between the memory reference streams of the different processors is simulated for different coherence protocols. A synthetic reference generator is used by Kessler and Livny [19] to evaluate distributed shared memory algorithms, in which the main difference from Archibald and Baer's model is that the memory reference stream of each processor is a sequence of shared and private phases. During a private phase the accesses are strictly to private memory, while both shared and private memory may be referenced during a shared phase. Each phase is characterized by length, placement, locality, read to write ratio, and type (private or shared).

Synchronization is an important aspect of any parallel program design, and the memory reference streams of processors executing a parallel program will consist of synchronization accesses and normal read/write accesses. By exploiting synchronization related information of a program, it is possible to weaken the memory consistency requirements, thereby improving overall system performance (see section 2.3). The workload model, described in the next section, captures synchronization aspects of a program; a feature absent in other probabilistic workload models.

### 6.2.1 Structure of the Workload Model

The workload model described in this section captures salient features of parallel and distributed programs. Specifically, it models class of applications that belong to the single-program-multiple-data (SPMD) style of programming. In a SPMD program, individual processors execute the same piece of code, albeit on possibly disjoint sets of data items. Processors synchronize with each other using semaphores, locks (shared or exclusive), or barriers. Semaphores and locks are used for protecting pieces of shared data, while barriers are typically used to indicate the end of a computation phase, or the computation itself.

As with any program, a parallel program in our workload model is represented as a collection of tasks. The inter-relationship between these tasks is captured by a task dependency graph, that suggests a partial execution order for the tasks that constitute the parallel program. A task is ready for execution when all tasks that precede it in the dependency graph have been completed. A work queue is maintained that contains the set of tasks that are ready for execution. Tasks are inserted into this queue honoring the dependencies in the task graph. A processor accesses the work queue to acquire a task to be executed next. When the work queue becomes empty and all the tasks have terminated, the parallel program is said to have completed.

Each task is a memory reference stream of finite length (specified by a parameter) and is composed of a sequence of *compute* and *synchronization* phases. During a compute phase, the processor generates references (reads or writes) to private and shared data. A compute phase is characterized by the following parameters: the number of memory references, read to write ratio, probability for shared and private data accesses, and the degree of locality within the phase. The compute phase is similar to the shared phase as defined by Kessler and Livny [19]. A synchronization phase consists of read/write data accesses (both private and shared), with a percentage of the shared data accesses being done under the control of explicit synchronization. Thus, a compute phase corresponds to a phase in a SPMD program in which computation is performed, while a synchronization phase corresponds to a phase in which shared data is manipulated under the control of some synchronization variable. Figure 11 shows the composition of the the two phases within a task (the associated parameters are given in parentheses).

The degree of locality within a phase defines the spatial locality for references within a page. In addition to this, the workload model allows designating distinct and disjoint regions of the shared address space to each task; and there is a parameter, called `InterTaskRefProb`, that governs the fraction of shared references of a task that are directed to other tasks' as opposed to its own region. This feature of the workload model captures the SPMD style of programming, wherein individual processors primarily operate on distinct portions of shared data, with occasional references to other portions of shared data. To capture effects of false-sharing, we provide the `FalseSharingRefProb` parameter. Another parameter, called `SynchReferenceProb`, controls the percentage of accesses to shared regions that are performed under the control of explicit synchronization (shared or exclusive). This parameter models the number of critical sections in the SPMD program.

| Type of model | Parameter variable | Default values |
|---|---|---|
| Transaction Model | `PvtProb` | 0.70 |
| | `OtherRegion` | 0.20 |
| | `SynchRef` | 1.00 |
| | `InterTaskRefProb` | 0.00 |
| Iterative Model | `PvtProb` | 0.70 |
| | `OtherRegion` | 0.20 |
| | `SynchRef` | 0.20 |
| | `InterTaskRefProb` | 0.05 |
| | `ReadInterTaskRefProb` | 1.00 |
| Asynchronous Model | `PvtProb` | 0.70 |
| | `OtherRegion` | 0.20 |
| | `SynchRef` | 0.20 |
| | `InterTaskRefProb` | 0.05 |
| | `ReadInterTaskRefProb` | 0.80 |

Table 11: List of parameters for domain specific workload models

2. **Iterative Model**: Iterative algorithms such as linear equation solvers, have the characteristic that shared data is not modified except at well defined synchronization points (such as a barrier). Such a data access pattern would allow a task to access the shared data without acquiring any locks for the purposes of reading. The iterative model captures this characteristic by allowing some percentage of the shared references (only reads) to be directed to other tasks' regions. (`OtherRegion` $\neq 0$, 0.05 in our experiments).

3. **Asynchronous Model**: In this model, tasks that comprise a computation do not synchronize with one another explicitly. In terms of the workload model this feature would translate to tasks reading and writing to shared memory without explicit synchronization. However, an implementation of this model in a shared memory environment may involve the use of locks

to govern access to mailboxes that may be used for asynchronous communication among the tasks. This workload model is similar to the data access patterns of asynchronous algorithms that rely on some other property such as convergence for correctness and termination [4]. In terms of task parameters, some percentage of the shared references (both reads and writes) are directed to other tasks' regions. (`OtherRegion` $\neq 0$, 0.05 in our experiments).

Table 11 summarizes the default values for the parameters that define the three domain specific workload models. Table 12 shows the default values for the other parameters used in the simulator.

### 6.2.3 Validation of the Workload Model

As mentioned earlier, we chose to validate our workload model using some of the applications that have been implemented on the CLOUDS distributed system. The validation will illustrate that by properly tuning the parameters of the workload model, one can model any SPMD style application. For the purpose of validation, we use two applications: integer sort, and **scan** – a transaction processing benchmark. The approach we take is as follows. The performance of these two applications is measured from the implementation on the CLOUDS distributed system. Using the program code the basis, we determine the values for the key parameters of the workload model. Ideally, executing the resulting workload model on top of the simulator should yield results identical to the measurements on the real system. Any variations in the results should be easily explained given some simplifying assumptions that are made in the simulator.

## Integer sort

The algorithm for the integer sort benchmark is taken from [31]. The algorithm uses bucket sort algorithm for generating the ranks for the input keys. The algorithm consists of seven distinct phases with barrier synchronization between consecutive phases. For more details on the algorithm for version $\mathcal{V}1$, see section 5.3.

The structure of the application task graph can be easily modeled using our workload model. The task graph for the application consists of 7 levels. At the end of each level, there is an implicit barrier; modeled in our workload model by the fact that the processor has to acquire a task from the central work queue. Number of tasks at each level is equal to the number of threads that are active during that phase. Using the listing for the program, we determine the number of references made during each level of the task graph. The shared address space is computed using the size of the shared data structures in the application. In our example, size of the total shared data space is equal to approximately 2150 Kbytes for $\mathcal{N} = 2^{18}$ elements. Similarly, other parameters are determined and assigned. To account for false-sharing aspects of the program, we set the `FalseSharingRefProb` parameter. See appendix A.1 for details about computation of these parameters.

After determining the values for various parameters, we ran the simulation for the write-invalidate coherence protocol. The results of completion times generated by an actual run and the simulation experiments are shown in Table 13. The values within parentheses are reported at 90% confidence level. As can be seen from the table, the simulation results agree quite well with the real results. Comparing the results using the `t-test` indicates no difference between the results obtained via the two techniques (The confidence intervals contain zero). This validation shows the workload model can capture the salient application characteristics with a careful choice of parameters.

## `SCAN` — A transaction processing benchmark

SCAN is one of the three TPS benchmarks used for evaluating transaction systems. The details about the benchmark can be found in section 5.6. This benchmark is an example of the transaction workload model described earlier. The task dependency graph for the SCAN benchmark consists of only 1 level, with the number of tasks equal to the number of processors participating in the computation. The memory reference stream for each processor consists of only synchronization phases. Each synchronization phase corresponds to accessing a 100 records of a file under the control of an exclusive lock. The number of references made during each synchronization phase is determined from the set of operations performed on each record (approximately 500). The total shared memory requirement for this benchmark is 130000 bytes as all records are stored in shared memory. See appendix A.2 for listing of the SCAN benchmark. The results obtained via the simulation study and measurements for the write-invalidate coherence are summarized in Table 14. The results obtained via simulation are similar to the those obtained via actual measurements.

| Parameter variable | Parameter meaning | Default values |
|---|---|---|
| PvtProbInSynch | Probability of access to private data during a synch. phase | 0.70 |
| PvtProbInCompute | Probability of access to private data during a compute phase | 0.70 |
| SynchReferenceProb | Probability that the next phase is a synch. phase | 0.20 |
| AvgTaskLength | Average task length (number of references) | 100,000 |
| MaxLockRefRange | Granularity of synch. phase (number of references) | 3000 |
| MaxShdRefRange | Granularity of the compute phase (number of references) | 3000 |
| ReadLockProb | Probability of acquiring a read lock for a synch. phase | 0.80 |
| WriteLockProb | Probability of acquiring a write lock for a synch. phase | 0.20 |
| ReadProb | Probability that access to a memory location is a read | 0.80 |
| InASynchPhaseSynchRefProb | Probability that the shared reference is to locked data | 0.50 |
| LocalityProb | Probability that the next shared reference would be in the neighborhood of the current reference | 0.80 |
| LocalityDistribution | The function that specifies the probability distribution for shared data access | +/- 80 bytes |
| InterTaskRefProb | Probability that a task accesses shared data outside its domain | 0.05 |
| ReadInterTaskRefProb | Probability of a read access for inter-task references | 0.80 |
| FalseSharingRefProb | Models degree of false-sharing | 0.00 |
| SharedAddressSpace | Total size of the shared address space | 1 Mbytes |
| NumberOfTasks | Total number of tasks in the parallel program | 50 |
| NumberOfNodes | Number of processors in the system | 8 |
| BlockSize | Amount of data transferred upon request for shared memory | 8192 |
| MediaSpeed | Speed of the network | 10 Mbps |

Table 12: List of parameters for the simulator along with the default values

| Number of | Measured | | Simulated | |
|---|---|---|---|---|
| Processors | Time | Conf. Interval | Time | Conf. Interval |
| 1 | 6.51 | (6.44, 6.57) | 6.56 | (6.56, 6.57) |
| 2 | 6.83 | (6.76, 6.89) | 6.85 | (6.73, 6.98) |
| 3 | 8.41 | (8.01, 8.82) | 8.06 | (7.93, 8.20) |
| 4 | 13.02 | (12.35, 13.69) | 13.23 | (12.85, 13.61) |

Table 13: Comparison of results obtained via simulation with actual measurements for the Integer Sort benchmark for $2^{18}$ elements

| Number of | Measured | | Simulated | |
|---|---|---|---|---|
| Processors | Time | Conf. Interval | Time | Conf. Interval |
| 1 | 23.91 | (23.90, 23.92) | 23.18 | (23.08, 23.21) |
| 2 | 12.20 | (12.16, 12.25) | 11.59 | (11.58, 11.60) |
| 3 | 8.45 | (8.28, 8.62) | 8.80 | (8.80, 8.81) |
| 4 | 6.43 | (6.36, 6.50) | 6.47 | (6.47, 6.48) |

Table 14: Comparison of results obtained via simulation with actual measurements for the SCAN benchmark for 10000 records

| Issues | Available alternatives |
|---|---|
| Data granularity (page size) | 512, 1024, 2048, 4096, or 8192 bytes |
| Coherence protocol | write-invalidate, lock-based, or buffered-update |
| Communication medium | 10 Mbps (Ethernet-like), or 1 Gbps (Fiber-like ) |
| Processor Speeds | 3 MIPS, 25 MIPS |
| Number of nodes | 4, 8, 16 |

Table 15: List of alternatives evaluated using simulation

### 6.2.4 Parameters for the Simulation

We have designed a set of experiments to study the effects of the various design alternatives presented earlier. The approach we take is as follows: we use a set of compute nodes (3 MIPS CPU) connected by 10 Mbps Ethernet as the baseline system. We then designed our experiments to evaluate the effects of each issue on the performance of the overall system as compared to the baseline system. A list of issues that are studied is summarized in Table 15.

The experiments have been conducted for the three workload models described in section 6.2.2. An application is modeled as a 4-level deep task dependence graph, with 16 tasks at each level, yielding a total of 64 tasks. A task on level $i+1$ is not executed until all tasks at level $i$ have been completed. Each task generates $100,000$ references. The lengths of the compute and the synchronization phases are specified as input parameters. The shared address space is 1 Mbytes divided into 128 logical segments of 8 Kbytes each. The logical segment is made up of physical pages; the page-size is specified as an input parameter. Segment is a unit of locking assumed in the lock-based protocol while page is a unit of data transfer on remote memory request for all three protocols. For the purposes of this study it is assumed that the program level locks generated by the workload model map exactly to the segment level locks. This assumption essentially removes the effects (due to lock granularity and data transfer granularity mismatch) of false-sharing for the write-invalidate protocol, and the effects of limited concurrency for the lock-based protocol.

In all our experiments, we fix the following parameters to be unchanged:

70% private data references, 80% reads, and 20% of shared references performed under explicit synchronization in the iterative and asynchronous workload models. The parameters that are varied in the experiments are summarized in Table 15. We use completion time as the metric for comparison.

## 6.3 Simulation Results and Discussion

We present the results in two parts: First, we discuss the effects of granularity of data transfer, and choice of coherence protocol with respect to the three workload models. Second, we present the impact of the hardware technology on performance.

### 6.3.1 Transaction Model

One would expect that larger data granularity would reduce the number of messages in the system as fewer data requests are generated, and would increase spatial locality. However, larger data granularity also increases the potential for contention of shared data due to false sharing, thereby degrading system performance. Figures 12, 13 and 14 show the performance for a 4-, 8-, and 16-node system connected via a 1Gbps communication medium. In the transaction workload model (see Figure 12), we observe that the performance improves as the data granularity is increased for all three coherence protocols. False sharing is not an issue for this workload since all shared data references are performed under the control of a lock and since we assume lock granularity is a segment.

The lock-based scheme is expected to incur a lesser number of messages

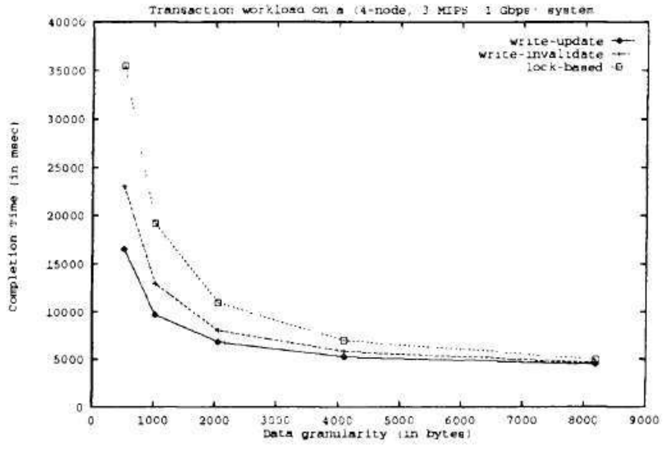

Figure 12: Transaction workload model's performance on 4 nodes on fiber

on synchronized data accesses since it combines data transfer with synchronization. However, in this protocol the data pages associated with the lock are always shipped to the requester along with the granting of the lock irrespective of whether the requester has a valid copy or not. As can be seen in Figure 12, the lock-based protocol performs poorly at low data granularity compared to the other two. The reason is because at low data granularity more number of messages are required to bring in the entire segment associated with the lock. The write-invalidate and write-update schemes may not have to incur this message overhead if the data is valid at the requester. However at higher data granularity the lock-based scheme performs better since the number of messages per lock request reduces significantly. Overall the write-update scheme performs better than either the write-invalidate or lock-based scheme (see Figure 12), although at large data granularity, the difference between the write-invalidate and write-update scheme is statistically insignificant. In the write-update scheme, only the updates are sent to the server at synchronization points, and further the protocol does not incur the overhead

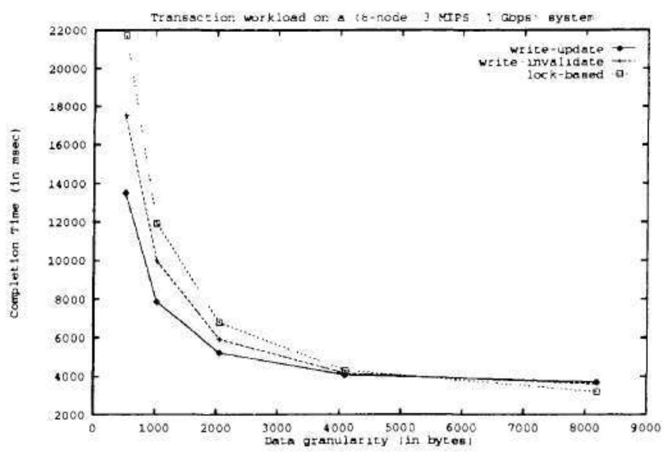

Figure 13: Transaction workload model's performance on 8 nodes on fiber

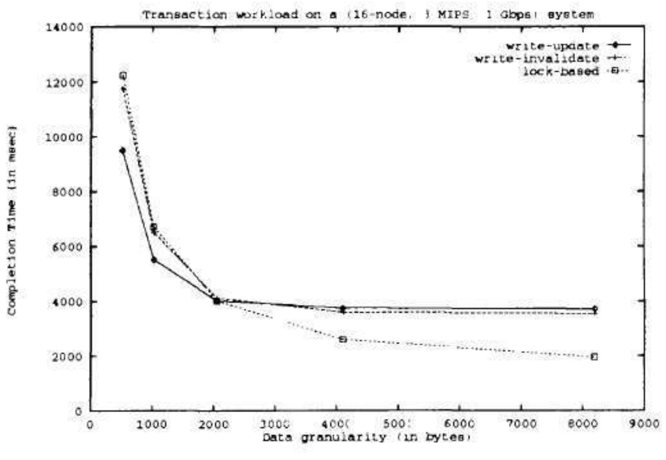

Figure 14: Transaction workload model's performance on 16 nodes on fiber

of invalidation messages. It is interesting to note for larger systems (8 and 16 nodes) the lock-based scheme performs much better than the other two schemes for large data granularity (see Figures 13 and 14). For write-invalidation scheme, the probability of the data associated with a lock being valid decreases due to the increased concurrent activity over the same number of shared segments. Similarly, for write-update scheme the updates are sent to the current set of potential readers and all of them may not actually use it in the future. On the other hand, the lock-based protocol incurs exactly the minimum number of messages required to get the lock and data. As we increase the number of nodes in the system, the number of messages becomes an important factor (due to contention for the communication medium) in determining the system performance.

In contrast, if one considers a system with Ethernet as the communication medium then the results for the transaction workload model are completely different. Figures 15, 16, and 17 show the results for the transaction workload model on a 4-node, 8-node, and 16-node system connected via a 10Mbps Ethernet. Unlike the earlier results, the lock-based scheme performs considerably worse than the write-update and the write-invalidate schemes. This is because the lock-based scheme sends a copy of the data-page to the requester regardless of the data-page being valid at the requester. The cost associated with the transmission of this data becomes dominant with the slower Ethernet medium, resulting in poor performance for the lock-based protocol.

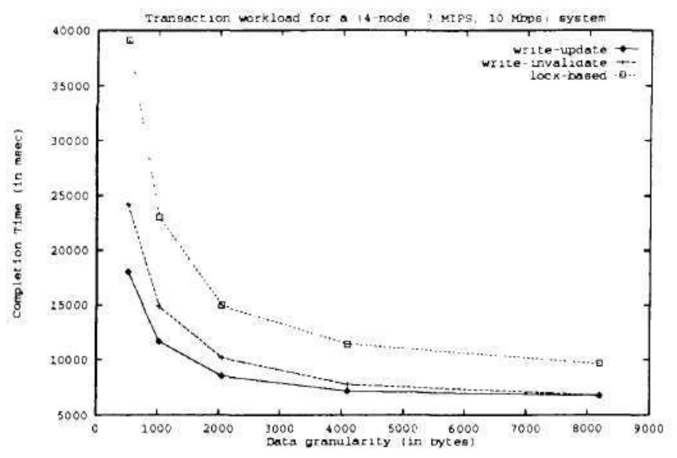

Figure 15: Transaction workload model's performance on 4 nodes on Ethernet

### 6.3.2 Iterative Model

Recall that the iterative workload model (see section 6.2.2) allows a task to access shared data for reading without explicitly acquiring read-locks. For this model, increasing data transfer granularity improves system performance for the lock-based and write-update schemes (see Figure 18). However, for the write-invalidate scheme, the performance benefit due to the reduced number of messages (at larger data granularity) is offset by an increase in false sharing, thus resulting in system performance degradation. Since read-shared copies are invalidated upon a write, the cost of re-reading a new valid copy increases with increasing data granularity for a given sharing pattern. The problem becomes even more acute when more nodes are added to the system, as now it is more likely that read-shared data pages may become invalid (see Figure 19). Since false sharing is not an issue with either the lock-based or write-update protocols, we do not see a similar performance degradation with either of these protocols.

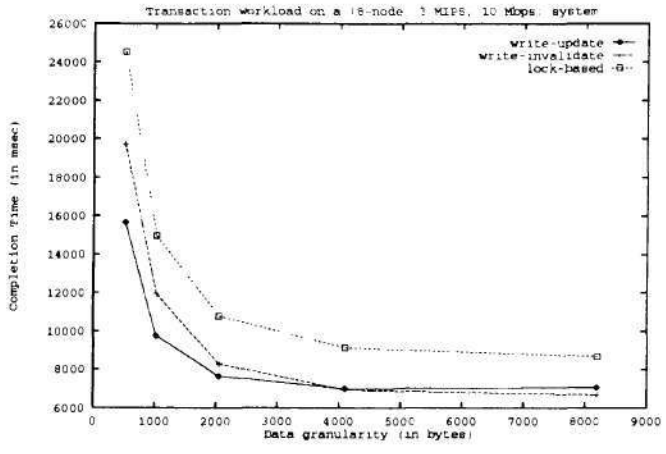

Figure 16: Transaction workload model's performance on 8 nodes on Ethernet

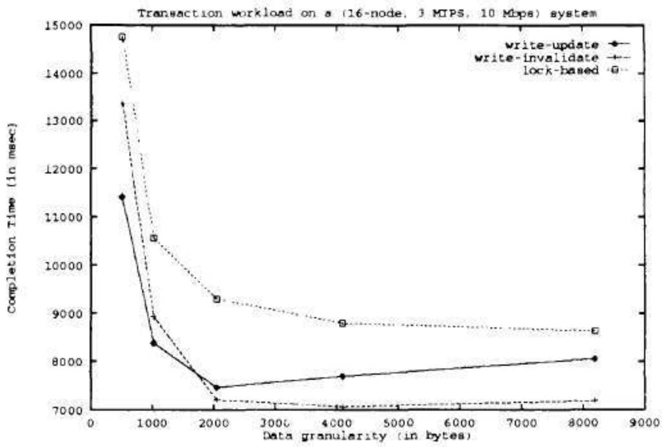

Figure 17: Transaction workload model's performance on 16 nodes on Ethernet

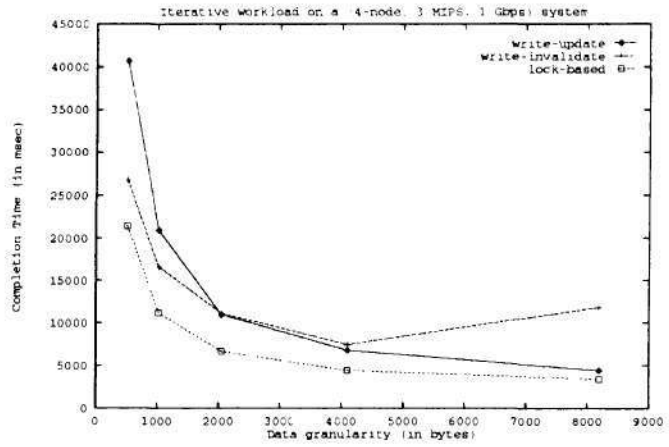

Figure 18: Iterative workload model's performance on 4 nodes on fiber

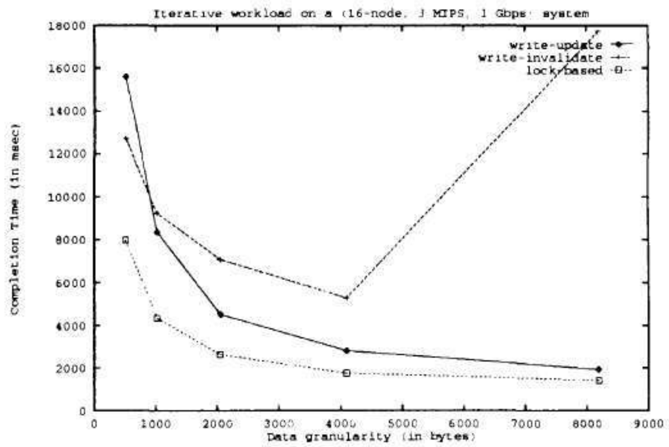

Figure 19: Iterative workload model's performance on 16 nodes on fiber

Since both the lock-based and write-update schemes allow the copies of shared data to remain inconsistent between synchronization points, these two are expected to perform better than write-invalidate scheme for the iterative workload model. Figures 18 and 19 confirm this hypothesis. However it is surprising that write-update scheme does not do as well as lock-based scheme. In the write-update scheme, updates for all modified pages are sent at the end of each synchronization epoch. This set of pages could potentially include ones that are unrelated to this particular epoch. As a result this scheme could incur more overhead than entirely called for in the iterative workload model. The lock-based scheme by associating locks with segments does not have to incur this unnecessary overhead. This effect is more apparent at low data granularities (small page sizes). In fact, as can be seen even write-invalidate scheme performs better than write-update scheme at sufficiently small data granularity since the need for unnecessary updates in the latter over-shadows the ill-effect of false-sharing in the former. At higher data granularities the distinction between lock-based and write-update schemes is lesser.

The results for the iterative workload model do not change if the communication medium is replaced by a 10Mbps Ethernet because only 20% of the data accesses are made under the control of a lock. Hence, the performance degradation as a result of shipping data with the lock is not very significant for the lock-based protocol.

### 6.3.3 Asynchronous Model

In this model, unsynchronized write-sharing of data is allowed (see section 3.2). Further the domain of write-shared data is the entire shared data space. Thus the model itself has a high built-in overhead (as compared to the iterative model) for both the write-invalidate or write-update style protocols. In the former, invalidations may have to be sent to all the nodes while in the latter updates may have to be sent to all the nodes. This is evident by comparing absolute completion times for the same amount of total work (in terms of number of memory references) for the two workload models (see Figures 18 and 21). As can be seen from Figure 21 increasing the data granularity helps both the protocols. The positive effect of reducing the number of messages at larger data granularities seems to dominate the negative effect of false-sharing for the write-invalidate protocol. The lock-based protocol (owing to its assumption that computations obey a synchronization model) is basically incompatible with this asynchronous workload model. Owing to the protocol allowing exactly one-copy of a segment (regardless of the data granularity) for such asynchronous accesses the lock-based protocol performs consistently worse than the other two for all data granularity (see Figure 7). However due to lesser number of messages at larger granularities the performance of lock-based protocol approaches that of the other two.

The results for the asynchronous workload model do not change if the communication medium is replaced by a 10Mbps Ethernet because only 20% of the data accesses are made under the control of a lock. Hence, the performance degradation due to shipping of data with the lock is not very significant for the lock-based

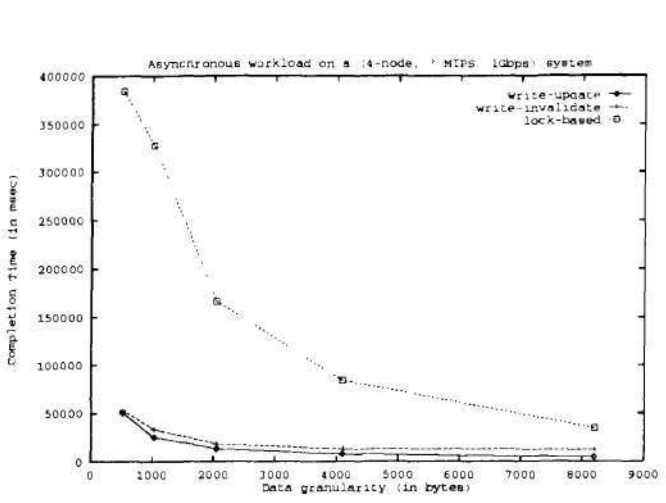

Figure 20: Asynchronous workload model's performance on 4 nodes on fiber protocol.

### 6.3.4 Hardware Technology

We conducted several experiments to determine the effects of new technology on the overall performance. When the processors in the baseline system are replaced with faster processors (see Figure 22), the overall system performance improves, although the percentage improvement due to reduce computation times is not uniform across all data transfer sizes. The non-uniform improvement across the range of data granularity can be explained as follows: For low data granularity, more number of data requests are generated, thereby increasing the computation requirements associated with page-fault handling and DSM related state maintenance; as a result, the processor speed has a significant impact on the performance than for large data granularity. Similar performance improvement is observed when

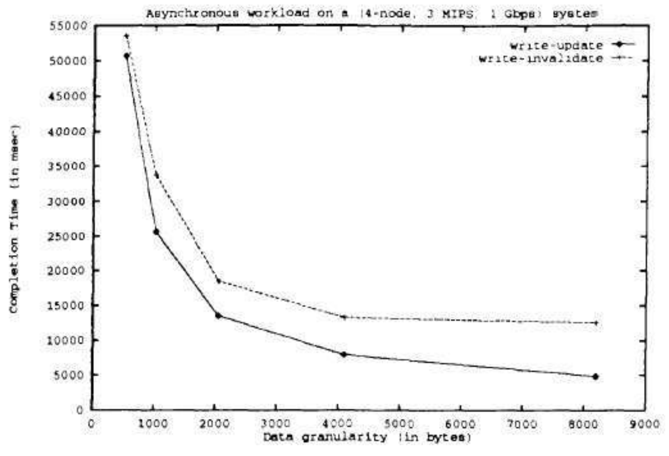

Figure 21: Asynchronous workload model's performance on 4 nodes on fiber. Comparing write-update and write-invalidate

the communication medium in the baseline system is replaced with a faster communication medium (see Figure 23). The reason for the improvement is reduced transmission times. The impact of communication speed on the performance becomes more significant as the data granularity is increased because at low data granularity the access to the medium is the primary source bottleneck (due to large number of messages).

## Effects of technology factors on system scalability

We define scalability of a distributed system as the effect of increased number of nodes in the system on the performance of the problem being solved, e.g., if a problem of size $\mathcal{N}$ is solved in time $t_1$ by $\mathcal{P}$ processors then it should be solved by $k\mathcal{P}$ $(k > 1)$ processors in time $t_2$ where $t_2 \leq t_1$. Ideally, we would like $t_1/t_2 = k$. In this case, the problem is said to achieve linear speedup.

To study the effects of technology factors on system scalability, we examined

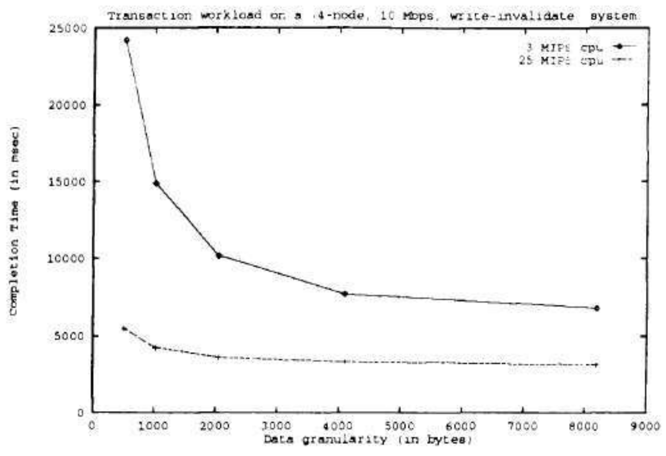

Figure 22: Effect of processor speed on performance

a distributed system using the transaction model and the write-update scheme. Figures 24, 26, 25, and 27 show the results for four possible system configurations: 3 MIPS CPU with 10 Mbps and 1 Gbps medium, and 25 MIPS CPU with 10 Mbps and 1 Gbps medium.

Figure 24 shows the system performance as a function of data transfer granularity for 4-, 8-, and 16-node configurations. Although for a given number of processors the system performance improves as the data transfer granularity is increased, the performance degrades as the number of processors is increased for a given data granularity. This is because the increase in computational power through the additional processors is not matched by the available communication bandwidth. The scalability problem becomes even more acute when faster processors are used in the system without changing the communication medium (see Figure 25). Similar behavior is observed for the iterative and the asynchronous models also.

The degradation in system scalability can be eliminated by using a faster

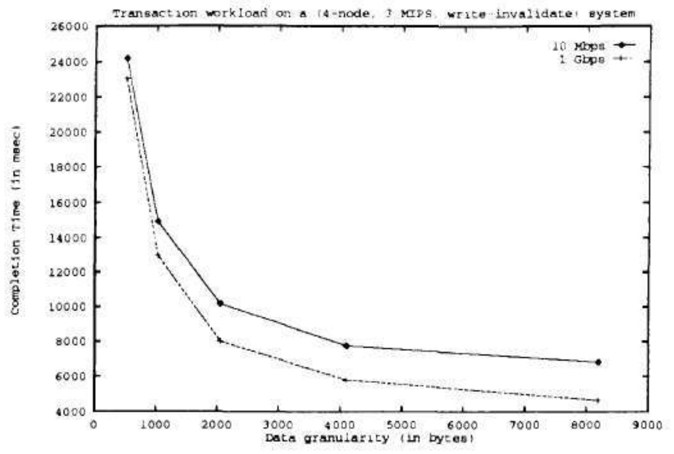

Figure 23: Effect of communication speed on performance

communication medium (see Figures 26 and 27). Note however that for 16 processors there is very little improvement in performance with increased data granularity. In the transaction workload model the processors compete for the same fixed number of locks. There are two sources of overhead in this model: one corresponding to data transfer and the other corresponding to the waiting time for lock service. The data transfer overhead is dependent on the number of messages and the size of each message. Since for the configurations studied the processing overhead per message dominates the actual transmission time on the wire, the number of messages is the more critical factor in determining the data transfer overhead. At low data granularity the number of messages is higher but the waiting time is higher; while at high data granularity the number of messages is smaller but due to the increased service time for each lock request (owing to the larger data granularity) the waiting time for locks is higher. Thus the two sources of overhead balance each other out resulting in no net gain in performance for large data granularities when the size of the system is scaled up.

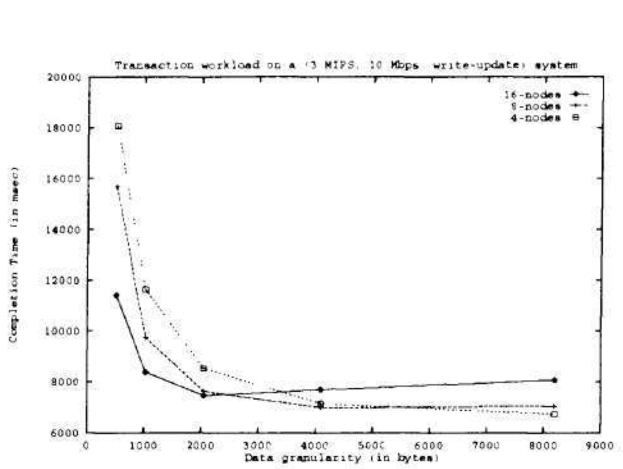

Figure 24: System scalability with 3 MIPS CPU and 10 Mbps network

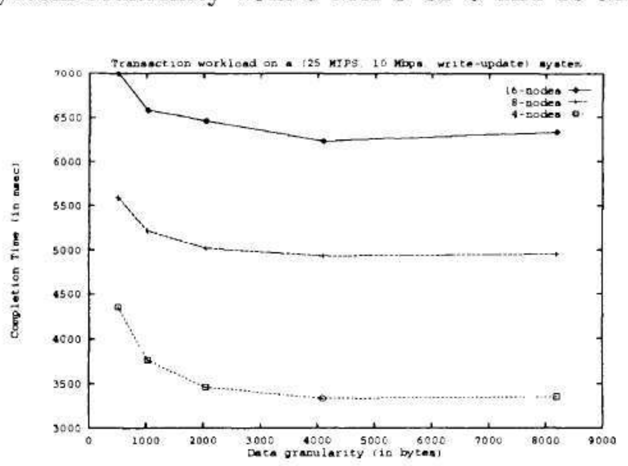

Figure 25: System scalability with 25 MIPS CPU and 10 Mbps network

In this chapter, we have presented a simulation-based study of issues that could not be evaluated via experimental studies. A detailed discussion of the results based on the experimental and simulation study is presented in the next chapter.

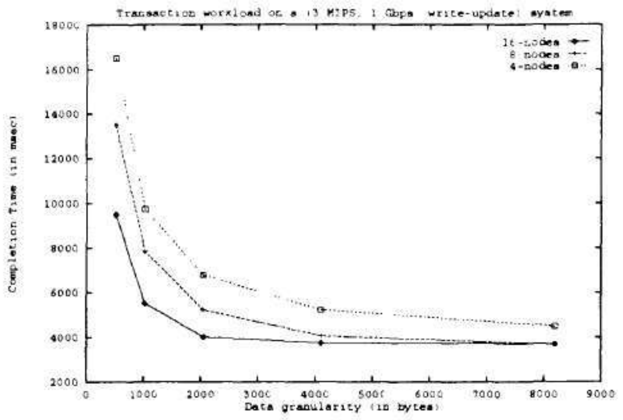

Figure 26: System scalability with 3 MIPS CPU and 1 Gbps network

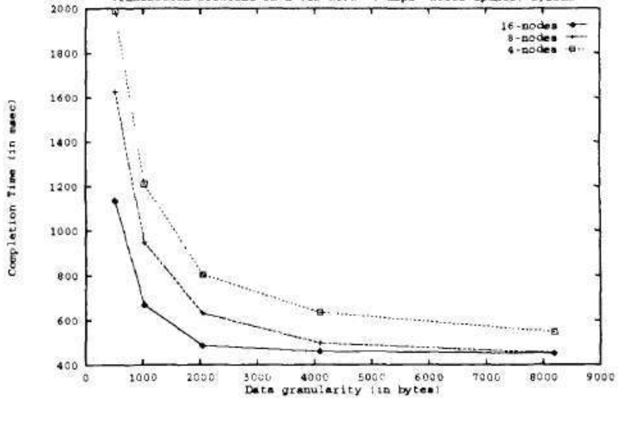

Figure 27: System scalability with 25 MIPS CPU and 1 Gbps network

# Chapter 7

# Discussion

We started out this research with the goal of evaluating the system issues in the design of distributed shared memory systems. We first identified a set of system issues along with the possible design alternatives available for addressing these issues. The evaluation was done in three steps: First, we studied the performance of an implementation of distributed shared memory. Second, we analyzed the performance of several applications on CLOUDS, a distributed shared memory system. Finally, we evaluated the issues using a simulation-based approach. Based on the results of the simulation studies, and implementation and analysis of the applications, we present here some observations on the design and performance of distributed shared memory systems. These observations are made with respect to the system issues identified in chapter 2.

## 7.1 Virtual Memory and DSM

There are two ways in which the distributed shared memory abstraction can be provided in a system: One, integrate the distributed shared memory mechanisms with the operating system; Second, provide the abstraction as a set of library functions accessible from the user-level. We call the first approach as the integrated-approach to DSM, and the second approach as the library-approach to DSM. The implementation of DSM considered in our study uses the integrated-approach. The advantage of using this approach is that the overheads associated with servicing DSM page-faults are very low, as all DSM related processing is done inside the operating system. In CLOUDS, the integrated-approach incurs an overhead of approximately 800 $\mu sec$ per page-fault. As a result, the overall performance of DSM is very good. On the downside, the integrated-approach is quite inflexible as any minor change to the distributed shared memory system requires modifications to the operating system. The library approach to DSM, on the other hand, is quite flexible to deal with, as only the library needs to be modified. However, it would perform quite poorly due to the overheads associated with context-switching, crossing user-to-kernel address boundaries, etc. As DSM deals with physical pages as units of data, a system designer implementing the library-approach would also have to modify the operating system to provide hooks for manipulating data pages (such as installing and invalidating) from the user-level. Some operating systems, such as MACH, do provide such hooks (via *external pagers*), thereby simplifying the implementation of the library-approach. If a system designer needs to provide a high performance DSM system then we would recommend the integrated-approach to DSM. Table 16 summarizes the advantages and disadvantages of the two approaches.

## 7.2 Granularity

There are two aspects to the issue of granularity: computation granularity, and data granularity. As mentioned earlier, computation granularity is the amount

|  | Approach |  |
|---|---|---|
|  | Integrated | Library |
| 1 | low overheads, $O(\mu sec)$ | high overheads, $O(msec)$ |
| 2 | inflexible | flexible |
| 3 | transparent to the user | Provide hooks in the operating system for installing, invalidating pages |

Table 16: Integrated vs Library: Comparison of the two approaches

of computation a process has to do between synchronization and communication points in a multi-process computation, while data granularity deals with the amount of shared information processed during a computation phase.

• *Effects of computation granularity:* In distributed systems connected via a local area network, network latencies are high. Therefore, any problem that has to be solved in a distributed environment (through cooperative computing) should have sufficiently high computation granularity to justify the added communication costs. The goal is to have a high *CGRatio* in equation 4.

$$CGRatio = \frac{Time\ spent\ in\ the\ computation}{Time\ spent\ in\ requesting\ data\ for\ the\ computation} \qquad (4)$$

Figure 28 shows the plot for a curve with *CGRatio* = 1. In order to achieve good speedups, the *CGRatio* for an application should fall in the shaded region for a given DSM implementation (*CGRatio* > 1). The vertical lines on the chart indicate the minimum time that is spent in transferring a unit of data between two nodes in a particular DSM implementation. For example, in CLOUDS, at least 16 msec is spent in transferring data between two nodes. This is because during each transfer a minimum of 8-Kbytes is transferred. Values for other systems differ depending on the size of the unit of data transfer, speed of the communication medium, and other

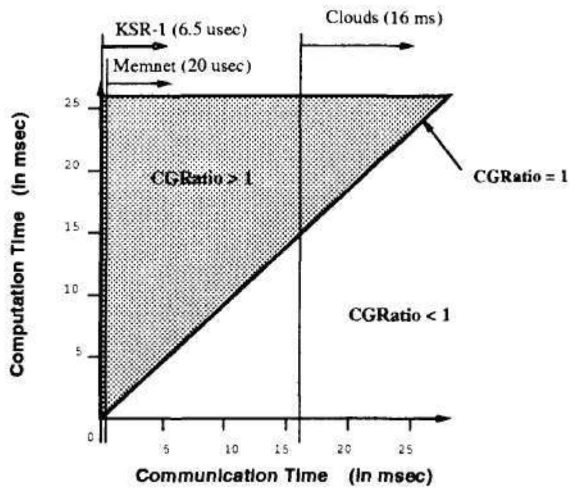

Figure 28: Computation to communication ratio requirements

overheads associated with the transfer. To achieve good speedups on a particular implementation, the *CGRatio* for an application should fall in the shaded region to the right of the vertical line for that system. Table 17 classifies the systems surveyed in chapter 3, based on the relative grain of computation needed to achieve good performance. A system designer can calculate the computation requirements for his/her DSM design by matching the minimum communication time for the system with those shown in the chart (see figure 28).

Based on the computation and communication requirements, we can classify the seven applications that we studied into three categories: high, medium, and low *CGRatios* (see Table 18). In our studies, applications which exhibit very large computation granularity and very little state-sharing, such as EP, matrix multiplication, and SCAN benchmark, perform quite well (show good speedups) for all processor configurations. Other applications with medium *CGRatios* show reasonable speedups for up to 4 processors. Beyond 4 processors, the completion times do not decrease further with the addition of more processors, mainly due to lack of sufficient computation granularity available at each processor. As a result, for large system configurations, the communication costs start dominating the completion times (small *CGRatio*) thereby degrading the overall performance. Better speedups can be achieved by increasing the computation granularity for each processor either by scaling the problem size for large number of processors, or using a small number of processors for a given problem size. Applications with low *CGRatios*, such as integer sort $\mathcal{V}1$. did not perform well on CLOUDS because the computation granularity is quite small to yield good speedups. Such applications will perform well on systems, such as KSR-1, that efficiently support fine-grained parallelism (see [31]).

| Computation Granularity | | |
|---|---|---|
| Large | Medium | Small |
| Domain, Ivy, CLOUDS, Mach, Agora, Choices, Mether, Munin | Memnet | DASH, KSR-1 |

Table 17: Computation granularity requirements

| *CGRatio* | | |
|---|---|---|
| High | Medium | Low |
| EP, Matrix Multiplication, SCAN, TSP | Integer Sort $\mathcal{V}2$, CG | Integer Sort $\mathcal{V}1$ |

Table 18: Classification of the applications based on the *CGRatio*

• *Effects of data granularity*: The issue of data granularity can be related to the amount of data exchanged between nodes at the end of a computation phase because it is this data that will be processed in the next computation phase.

On page-based systems, regardless of the amount of sharing, the amount of data exchanged between nodes is usually a multiple of the physical page-size of the underlying architecture. In our study, all applications exhibited very small data granularity, while the underlying system supported very large physical pages (8 Kbytes). If the shared data is stored in contiguous memory locations then most data can be stored in few physical pages. This strategy often gives rise to the problem of *false-sharing* wherein disjoint pieces of shared data, operated upon by distinct processors, reside on the same physical page. As a result, the system performance degrades as the common physical page thrashes between different processors. The problem further exacerbates as more nodes are used for solving the problem. Such behavior is observed for the CG, and the integer sort (version $\mathcal{V}1$) benchmarks. One way to reduce the problem of false-sharing is by partitioning the shared data structures on to disjoint physical pages. For systems with a large physical page-size, such partitioning of data can result in significant wastage of the virtual address space. Such wastage can be reduced if the distributed shared memory system is implemented on architectures which support a smaller physical page-size.

Another factor that affects the value for the page-size is the *total overhead per byte* associated with fetching a data-page. Recall, in chapter 4 we computed the value for *total overhead per byte* as the sum of the *fixed cost per byte* and *latency per byte* (see equations 1, 2, and 3).

$$\begin{aligned} Total\ overhead\ per\ byte \ &= \ \frac{VM\ overhead\ + data\ request\ cost}{PageSize} \\ &+ \ (server\ proc.\ cost) * PageSize + \frac{PageSize}{Media\ bandwidth} \end{aligned}$$

Using the values for different components of the distributed shared memory system, one can compute the effect of increase in page-size on the *total overhead per byte* for a particular system. Figure 29 shows the expected *overhead per byte* for the CLOUDS implementation of DSM using a 10 Mbps Ethernet. In the plot, we assume that the VM overhead is 0.800 msec, cost of sending a data request is 3 msec, and server processing cost is 0.200 msec/Kbyte of data. As can be seen from the figure, the minimum occurs somewhere between 1 - 2 Kbytes. For page-size values larger than 2 Kbytes, the *latency per byte* dominates the *total overhead per byte* while for values less than 1 Kbytes, the *fixed cost per byte* dominates the *total overhead per byte.* Table 19 lists values for the page-size parameter for different values of the VM overhead, server processing overhead, and data transmission cost. The values listed in Table 19 indicate the minimum value of page-size; and is obtained by differentiating the *total overhead per byte* with page-size parameter and solving for page-size (see equation 5).

$$PageSize = \sqrt{\frac{Media\ bandwidth(VM\ overhead\ + data\ request\ cost)}{(Media\ bandwidth) * (Server\ proc.\ cost) + 1024}} \quad (5)$$

| | VM ovhd | Data req cost | Server proc. | Media Speed | Page-size |
|---|---|---|---|---|---|
| 1 | 0.80 msec | 3.00 msec | 0.20 msec/K | 10 Mbps | 1 - 2 Kbytes |
| 2 | 10.0 msec | 3.00 msec | 0.20 msec/K | 10 Mbps | 3 - 4 Kbytes |
| 3 | 0.80 msec | 1.00 msec | 0.20 msec/K | 1 Gbps | 2 - 3 Kbytes |
| 4 | 10.0 msec | 1.00 msec | 0.20 msec/K | 1 Gbps | 7 - 8 Kbytes |
| 5 | 0.005 msec | 0.02 msec | 0.05 msec/K | 8 Gbps | 0.7 Kbytes |

Table 19: Optimal value of page-size for different system configurations

Table 19 indicates that a single value of the page-size parameter is not appropriate for all types of DSM system designs. The value should be decided

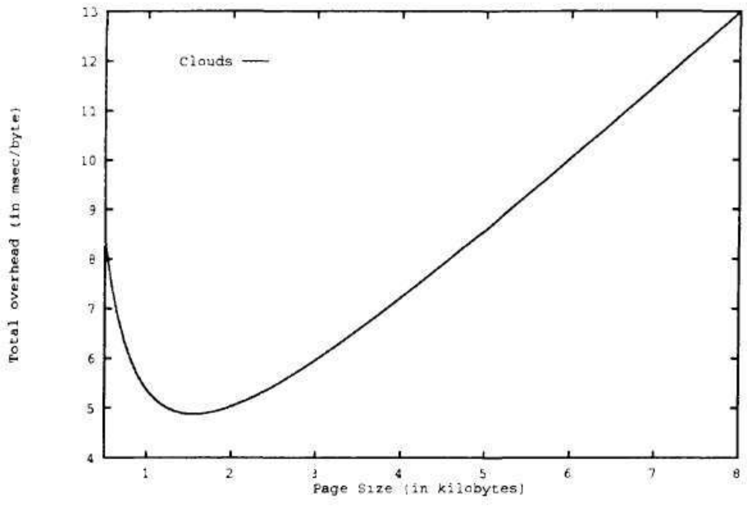

Figure 29: Total overhead per byte for DSM on CLOUDS

based on the other design decisions, such as approach to DSM, expected server processing overheads, and cost of data transmission. For example, a page-size of 1 - 2 Kbytes is appropriate for a software implementation of DSM using the integrated-approach and Ethernet-like communication medium. Systems such as CLOUDS that have similar characteristics but are implemented on architectures with 8-Kbyte page-size pay a high penalty for *latency per byte*. On the other hand, systems that provide hardware support for DSM (indicated by small VM overheads, server processing overheads), and faster communication medium can utilize smaller page-sizes (see entry 5 in Table 19). KSR-1 is an example of such a system that uses a 128-byte sub-page as the unit of data transfer and coherence maintenance. As the value for the page-size is usually tied to the system architecture used for the implementation, a system designer should carefully analyze his design decisions before selecting the architecture for implementation of DSM.

## 7.3 Memory Model and Coherence Protocol

Programming on any system requires that the users be offered a programming model that they can use for writing programs. Distributed systems are no different. The choice of a memory model is closely tied to the type of coherence protocol that is used for maintaining coherence of shared data. The experimental results presented in this thesis are for two kinds of memory models: one weak, and one strict. Corresponding to these two memory models, we considered two coherence protocols – write-update, and write-invalidate. The write-update protocol is based on the buffered consistency (BC) memory model while write-invalidate is based on sequential consistency (SC). In our simulation studies, we also studied the lock-based protocol. The lock-based protocol restores sequential consistency at well-defined points governed by locks, with hooks for weaker semantics (see Chapter 2). In the following discussion, we refer to the memory model implemented by the lock-based protocol as the SCsynch model.

Table 20 ranks the performance of the three memory models for the seven applications and three workload models that we studied. Interestingly, for applications that exhibited high *CGRatios* (EP, Matrix Multiplication, SCAN, TSP), the choice of memory model does not make a significant difference on the performance of the application. The main reason is that the application's communication requirements are very low such that it does not matter which memory model is used. For medium-grained applications such as integer sort $\mathcal{V}2$ and CG, the BC memory model performs well because it supports concurrent writes to heavily shared data pages. The SC memory model performs poorly because it pays a high overhead

| Application | Rank |
|---|---|
| EP, Matrix Multiplication, SCAN, TSP | (1) BC, SC |
| Integer Sort $\mathcal{V}2$ | (1) BC<br>(2) SC |
| CG | (1) BC<br>(2) SC |
| Integer Sort $\mathcal{V}1$ | (1) SC<br>(2) BC |
| Transaction Workload | (1) SCsynch<br>(2) BC<br>(3) SC |
| Iterative Workload | (1) SCsynch<br>(2) BC<br>(3) SC |
| Asynchronous Workload | (1) BC<br>(2) SC<br>(3) SCsynch |

Table 20: Ranking of the three memory models

for maintaining consistency of heavily shared data pages. For small-grained applications such as integer sort $\mathcal{V}1$, the BC memory model performs poorly compared to SC because the former incurs high overheads at synchronization points. These overheads negate any gains of using a weaker memory model. For our simulation studies, we considered a wide range of workload models, and weaker memory models perform well for configurations with large number of processors (SCsynch for iterative, BC for asynchronous; see section 6.3). The sequentially consistent models did not perform well due to the increase in overhead for maintaining coherence of data in large configurations.

## Programming Ease

Other factors that can influence the choice of the memory model, and coherence protocol are the aspect of programming ease, and system scalability. By ease of programming, we mean how much work the programmer has to do in writing an efficient distributed application. For programming ease, stricter memory models are better suited because these are well understood by the programming community. On the downside, to achieve good performance, the programmer (or the compiler) has to do good job at data placement to avoid false-sharing. As mentioned earlier, performance degradation due to false-sharing magnifies in systems with large page-sizes. On the other hand, weak memory models are new to the programming community, and are not understood enough by the users to exploit the weakness of memory in the applications. False-sharing may need to be addressed in models which use invalidation-based approach to provide weaker semantics. Programming effort is less for protocols which eliminate false-sharing by using an update-based scheme.

Another aspect to programming ease is the question where should one focus his/her efforts in writing efficient distributed shared memory programs: at the application level, or at the system level. The advantage of focussing efforts at the system level is that an application programmer (naive or advanced) needs to do very little work in writing efficient programs because the underlying system has been tuned to provide good performance. Such is the case with using the integrated-approach to DSM wherein a programmer is oblivious of the structure of the underlying system. All data that is needed by an application is transparently and efficiently fetched to the node where the execution takes place. On the other hand, if the focus is at the application level then a programmer needs to be aware of the complexities of the underlying system for writing efficient programs. A naive implementation of an application may result in a poor performance as the underlying system may not be tuned to support DSM very efficiently. Such is the case with using the library-approach to DSM wherein a programmer has to specify the data structures that are shared globally, and the kinds of sharing patterns expected of the shared data. Munin is an example of such a system.

## System Scalability

By system scalability, we mean how many nodes can efficiently implement DSM without incurring significant performance degradation. One measure of system scalability is the number of messages required for maintaining coherence of shared data. Table 21 shows the number of messages generated in the three coherence protocols (with and without multi-cast). If no multi-casting is used then one can see that both the write-update and the write-invalidate schemes can potentially generate number of messages proportional to the number of nodes participating in the computation ($r \rightarrow \mathcal{N}$). On the other hand, lock-based protocol is insensitive to the number of nodes participating in the computation. However, in both the lock-based and write-invalidate protocol, the number of messages increases as the the degree of sharing is increased (number of messages is a function of the degree of coherence, $c$).

Table 22 rates the scalability of the three protocols based on different parameters values, assuming no multi-cast. We analyze each of the four cases below.

| Protocol | Number Of Messages | |
|---|---|---|
| | Without multi-cast | With multi-cast ($r$=$1$) |
| write-update | $\mathcal{S}(5 + 2rw) + 2\mathcal{P}(1\text{-}h)$ | $\mathcal{S}(5 + 2w) + 2\mathcal{P}(1\text{-}h)$ |
| write-invalidate | $\mathcal{S}(5 + 2rwc + c(1\text{-}w)) +$ $\mathcal{P}(1\text{-}h)(2+ c(w(5 + 2r) + 1))$ | $\mathcal{S}(5 + 2wc + c(1\text{-}w)) +$ $\mathcal{P}(1\text{-}h)(2+ c(7w + 1))$ |
| lock-based | $3\mathcal{S} + \mathcal{P}(1\text{-}h)(2 + c)$ | $3\mathcal{S} + \mathcal{P}(1\text{-}h)(2 + c)$ |

| | |
|---|---|
| $\mathcal{S}$ | Number of synchronization phases |
| M | Amount of memory operated by a processor during a computation phase |
| $w$ | Probability that an access is a write operation |
| $h$ | Hit ratio |
| $c$ | Probability that an access read/write will cause coherence messages to be sent to other nodes |
| $\mathcal{N}$ | Number of nodes participating in the computation |
| $r$ | Number of nodes involved in receiving coherence messages. $r \leq \mathcal{N}$ |
| $\mathcal{G}$ | Unit of data transfer |
| $\mathcal{P}$ | Number of messages needed to bring in $\mathcal{M}$ bytes of memory. $\mathcal{P} = \frac{\mathcal{M}}{\mathcal{G}}$ |

See appendix D for details.

Table 21: Number of messages generated in the three coherence protocols

1. If an application does not require any coherence to be enforced ($c$ = $0$) then the lock-based scheme will generate a fewer number of messages because it combines data transfer with synchronization. One example of such an application is an implementation of TSP that allows the nodes to use their local copies of the best tour-value. Only when a processor needs to update the global best tour-value, it does so under the control of a lock. This application does not need any coherence activity to be performed during computation of the best tour-value. The other two protocols will generate equal number of messages, albeit more than lock-based, because separate messages are needed for acquiring/releasing locks during the computation.

|  | Condition | Order |
|---|---|---|
| 1 | No coherence needed, c = 0 | (1) lock-based<br>(2) write-invalidate, write-update[a] |
| 2 | No computation phase, $\mathcal{M} = 0$<br>$\Rightarrow \mathcal{P} = 0$ | (1) lock-based<br>(2) write-update<br>(3) write-invalidate |
| 3 | $r \rightarrow \mathcal{N}$ | (1) lock-based<br>(2) write-update<br>(3) write-invalidate |
| 4 | Number of synchronization phases tend to 0, $\mathcal{S} \rightarrow 0$ | (1) write-update<br>(2) lock-based<br>(3) write-invalidate |

[a]Provided the reader turns off receipt of updates

Table 22: Scalability of the three coherence protocols without multi-cast

2. For applications that access data under the control of a synchronization, the lock-based scheme generates fewer number of messages than the other two protocols because it combines data access with synchronization. The write-update protocol generates fewer messages than write-invalidate because the former supports concurrent writers to the same physical page while the latter does not. The SCAN benchmark is one example of such an application.

3. If the number of nodes for which memory consistency needs to be enforced reaches $\mathcal{N}$ then the number of messages generated for the write-invalidate scheme increases more rapidly than the write-update scheme because the former enforces memory consistency during the synchronization and computation phase while the latter enforces memory consistency only at the end of the synchronization phases. The lock-based scheme scales better than the other two because the number of messages is independent of the number of

nodes participating in the computation.

4. If an application has very few synchronization phases then the benefits of the lock-based scheme in combining data access and synchronization become negligible. As a result, the write-update scheme scales better than the other two because it does not generate messages to enforce memory coherence during computation phases.

A system designer can analyze the target set of applications that will run on the DSM system to see which type of applications will be more often used. The designer should then select the memory coherence protocol accordingly by comparing the number of messages using table 21.

## 7.4 Synchronization

We discuss the issue of providing synchronization with DSM under a broader category of *miscellaneous system services.* Simulation studies (see Chapter 6) have been performed assuming miscellaneous system services (such as acquiring/releasing locks, barriers, and disk I/O) incur negligible cost; therefore the results of the studies do not show significant effect of these services on the performance. In our implementation studies, however, we observed that these services play a key role in determining the overall performance of the application. Most applications that we consider belong to the class of SPMD programs with approximately equal amounts of computation being performed at each node. As a result, the processors have a tendency to reach a synchronization point in the program at about the same time, causing bursts of synchronization activity. Such bursts of activity cause the central synchronization server to become overloaded, resulting in severe performance degradation especially for large number of processors. Similar performance degradation due to the data server becoming a bottleneck is observed for the write-update protocol. In the write-update protocol, all processors perform a `cp_synch()` operation prior to a synchronization point. In `cp_synch()`, all modifications made to shared memory are identified and sent to the data server. As all processors reach the synchronization point at approximately the same time, the data server becomes the bottleneck while servicing the `cp_synch()` requests. The performance deteriorates further as more nodes are added to the system. One technique to eliminate the problem would be to reduce the number of messages that are generated by individual processors at synchronization points. In the write-invalidate and write-update protocols two separate messages are generated at a synchronization point: one for doing the synchronization operation; and one for requesting the data associated with the synchronization operation. Combining these two messages, as is done in the lock-based protocol, would significantly improve the system performance. This point is supported by the simulation studies for the transaction workload model (see section 6.3.1). Using distributed servers for providing miscellaneous services may also alleviate the server bottleneck problem. It is essential, therefore, that the system designer pay equal attention to the design of miscellaneous system services for scalable distributed shared memory systems.

## 7.5 Hardware Technology

Performance of any distributed system is closely tied to the hardware technology the system is built around. With the advent of faster microprocessors, it is possible to build more powerful systems. However, one area of the design that is seldom given much thought to is the type of communication medium. Use of slower communication medium such as Ethernet, with faster processors causes the communication to become a bottleneck. This is confirmed by the results of the simulation studies (see section 6.3.4), which indicate that for building scalable DSM systems a faster interconnection network is a must. System designers interested in building new distributed systems should pay close attention to alleviating the communication bottleneck by considering new communication technologies, such as fiber optics, and ATM networks, for the design. Providing additional hardware support for DSM can also improve system performance by off-loading all shared memory related activity from the host. There are two types of overheads associated with a software implementation of DSM systems. One, the time spent in performing housekeeping chores during servicing of DSM requests (approximately, 20% on CLOUDS); Second, time spent in processing control messages such as invalidation, updates, for maintaining coherence of shared data. These activities can reduce the number of cycles available for a host for performing useful work. To reduce this overhead, hardware support for DSM is essential especially in large distributed systems where distributed shared memory traffic could be high. See appendix C for more details on the design of hardware support for DSM.

## 7.6 Conclusions

| System | Architecture | | |
|---|---|---|---|
| Parameters | Large | Medium | Small |
| Processor Speed | 12 mips | 20 mips | 40 mips |
| Communication Speed | 10 Mbps | 16 Mbps | 1 Gbps |
| VM Overhead | 800 $\mu$sec | 400 $\mu$sec | 10 $\mu$sec |
| Memory Model | BC | BC | BC |
| Coherence Protocol | write-update | write-update | write-update |
| Page-size | 1 - 2 KB | 512 bytes | 128 bytes |

Table 23: Characteristics of three types of DSM systems

| | CGRatio | | |
|---|---|---|---|
| Architecture | High | Medium | Low |
| Large | $O(1000)$ | $O(100)$ | $O(10)$ |
| Medium | $O(1000)$ | $O(1000)$ | $O(100)$ |
| Small | $O(1000)$ | $O(1000)$ | $O(1000)$ |

Table 24: Number of nodes that can efficiently execute an application based on the CGRatio

Based on the experimental and simulation results, table 23 lists the characteristics of three types of DSM systems that support large-grain, medium-grain, and small-grain parallelism. Note that the cost of building a DSM system increases as one moves from large-grain to small-grain systems because former are usually built with slower processors and slower communication mediums, while the latter needs additional hardware support for achieving good performance. However, the DSM design does not preclude use of faster communication medium with slower processors, though use of faster processors with slower communication medium does not scale well (see section 6.3.4). For these three types of DSM systems, table 24 lists the number of nodes that can efficiently execute the three different classes of applications (based on the *CGRatios*). Based on our experience with the seven applications on the CLOUDS distributed system, we would categorize any application with a *CGRatio* less than 10 as small-grain, between 10 and 1000 as medium-grain, and greater than 1000 as large-grain. Note that this classification uses the CLOUDS system as the point of reference. As can be seen from table 24, small-grain DSM systems can scale to thousands of nodes for all three classes of applications provided the number of messages generated in the system for maintaining memory consistency is kept to a minimum using weaker memory models and multi-cast techniques.

In this chapter, we discussed the issues related to the design of distributed shared memory systems based on the results of obtained from our simulation and experimental study, and provided guidelines to designers who are interested in the design of scalable distributed shared memory systems.

# Chapter 8

# Conclusions and Future Work

## 8.1 Concluding Remarks

The thesis starts with the premise that distributed shared memory is a viable programming paradigm for programming large distributed systems. Based on this premise, we have investigated several issues that arise in the design of such systems, and tried to answer the question whether we can identify a set of issues, along with the design parameters, that define an efficient implementation of distributed shared memory systems. The answer to this question has provided several contributions.

First, we have identified a set of system issues that form the core of a distributed shared memory system design. These issues include integration of distributed shared memory with virtual memory management, granularity of computation and data, choice of memory model, choice of the coherence protocol, and technology factors. We have also identified a set of possible design alternatives that are available for addressing each of these issues.

Second, we have analyzed the performance of an implementation of distributed shared memory on the CLOUDS distributed operating system. The study provided us with an insight into the functioning of a distributed shared memory system. The performance study helped us in identifying performance bottlenecks, and provided us with timings measurements associated with individual components of the DSM subsystem. These times were later used to assign costs to the different components of the simulator.

Third, to evaluate the various design alternatives, we have implemented and analyzed the performance of several applications on the CLOUDS distributed system. Issues that could not be studied via experimental studies have been studied using a simulation model. To drive the simulator, we designed a workload model that captures the salient features of programming parallel and distributed systems. The simulator is used to analyze system performance with respect to data granularity, types of coherence protocols, effect of communications media, and any additional hardware support. We state conditions to determine appropriate values for addressing the issues enumerated earlier. A system designer can use these conditions to decide the alternatives that are appropriate for the distributed shared memory system he/she is designing. Some of the key results of the study indicate that the choice of coherence protocol does not matter for applications that exhibit high computation granularity and low state sharing; coherence protocols based on weaker memory models are suitable for use in large distributed shared memory systems; the unit of data granularity (page-size) depends on the overhead associated with servicing data requests and cost of data transmission; miscellaneous system services, such as the synchronization server, and the data server, play a significant role in influencing the performance of an application; and the application performance can be improved by providing additional hardware support for distributed shared memory.

## 8.2 Future Work

The research in this thesis has answered some questions related to distributed shared memory systems. However, several questions remain unanswered. We briefly discuss some of these questions in this section.

The benchmarks used in the study were drawn from a set of applications that exhibit fine-grain parallelism. Given the high network latencies in some distributed systems such applications may not be appropriate for benchmarking such systems. It would be interesting to identify applications that are more appropriate for distributed systems.

The research in this thesis examined the effects of the operating system issues on the performance of DSM systems. We did not consider other issues namely, object and process migration, reliability, availability, and fault-tolerance that are equally important. We would like to study the impact of these issues on the design of DSM systems, and see how these issues impact the results presented in this thesis.

To date, many researchers have designed and implemented several experimental distributed shared memory systems. However, none of these systems have left the research laboratories and made it into systems for daily use. One reason for this is researchers have been unable to identify applications except numerically intensive computations that can efficiently make use of the distributed shared memory abstraction. A few fertile areas for research include the use of distributed shared memory abstraction in the design of multi-media servers, parallel/distributed file servers, and main memory database systems. With the advent of high speed networks, it is likely that the communication bottleneck found in current (1993) implementations of distributed shared memory systems will be eliminated. This opens new possibilities for using distributed shared memory in daily life.

In this research, we presented a preliminary design for the type of hardware support that would benefit DSM systems. However, to completely understand the performance implications of such support, it is essential that a detailed performance study be carried out. As part of our future work, we plan to implement and do a detailed performance study of the controller. We would also like to explore the performance implications of using one processor of a multiprocessor machine as a dedicated DSM processor, and study its performance in comparison to the dedicated controller approach.

# Appendix A

# Integer Sort and SCAN benchmarks

## A.1 Integer Sort (version $\mathcal{V}1$) Benchmark

This section describes the computation of various parameters for the simulation for the integer sort program. All parameters except the amount of shared address space, number of references made by a task, and the parameter capturing false-sharing aspects of the program are assigned default values. We compute the shared memory requirements for the application for sorting $\mathcal{N} = 2^{18}$ elements (see Table 25) based on the source listing shown in section A.1. Table 26 shows the formulae used for computing the number of references made by a task at each level of the task graph. This is an approximate representation derived from the loops in the program. Note that we are only interested in the approximate number of references and not the exact number. The parameter `NumberOfNodes` is an input parameter to the simulator indicating the number of nodes participating in the computation. Parameter `FalseSharingRefProb`, which models the false sharing aspect of the application, is tuned to approximate the behavior of the implementation by comparing the results of the simulation and the implementation. One could use curve fitting techniques to extrapolate this parameter for a larger number of processors.

| Data Structure | Size | Actual size (in bytes) |
|---|---|---|
| short key[N] | 2*$\mathcal{N}$ | 524288 |
| short S[N] | 2*$\mathcal{N}$ | 524288 |
| int rank[N] | 4*$\mathcal{N}$ | 1048576 |
| int keyden[Bmax] | 4*$Bmax$ | 262144 |
| int keyden_t[MAX_PROCS][Bmax] | 4*MAX_PROCS*$Bmax$ | 8192 |
| int work_size_n | 4 | 4 |
| int work_size_k | 4 | 4 |
| int extra_n | 4 | 4 |
| int extra_k | 4 | 4 |
| int my_strt_n[MAX_PROCS] | 4*MAX_PROCS | 24 |
| int my_end_n[MAX_PROCS] | 4*MAX_PROCS | 24 |
| int my_strt_k[MAX_PROCS] | 4*MAX_PROCS | 24 |
| int my_end_k[MAX_PROCS] | 4*MAX_PROCS | 24 |
| int my_extra_k[MAX_PROCS] | 4*MAX_PROCS | 24 |
| int my_extra_n[MAX_PROCS] | 4*MAX_PROCS | 24 |
| int tmp_sum_k[MAX_PROCS] | 4*MAX_PROCS | 24 |
| int tkt[MAX_PROCS][128] | 4*MAX_PROCS*128 | 3072 |
|  | TOTAL SIZE | 2157752 |

Table 25: Approximate shared memory requirements for the integer sort benchmark for $\mathcal{N} = 2^{18}$ elements, Bmax = 2048, MAX_PROCS=6

### A.1.1 Source Listing of the Integer Sort Benchmark

```
short           key[N] = {0};

short           S[N] = {0};

int             rank[N] = {0};

int             keyden[Bmax] = {0};

int             keyden_t[MAX_PROCS][Bmax] = {{0, 0}};

long            work_size_n = 0;

long            work_size_k = 0;

long            extra_n = 0;
```

| Level | Number of References per task |
|---|---|
| 1 | 8 |
| 2 | $(11 + Bmax + 22 + Bmax) + (Bmax + Bmax)/NumberOfNodes + (4 * \mathcal{N} + 4 * \mathcal{N})/NumberOfNodes$ |
| 3 | $(NumberOfNodes + 6 * NumberOfNodes) + (Bmax + 3 * Bmax) + (2 * Bmax + 4 * Bmax)/NumberOfNodes$ |
| 4 | $(NumberOfNodes + 6 * NumberOfNodes)$ |
| 5 | $(2 * Bmax + 4 * Bmax)/NumberOfNodes$ |
| 6 | $(3 * Bmax + 6 * Bmax) * NumberOfNodes$ |
| 7 | $(6 * \mathcal{N} + 5 * \mathcal{N})/NumberOfNodes$ |

Table 26: Number of references made by each task for the integer sort benchmark for $\mathcal{N} = 2^{18}$ elements, Bmax = 2048

```
    long            extra_k = 0;

    long            my_strt_n[MAX_PROCS] = {0};

    long            my_end_n[MAX_PROCS] = {0};

    long            my_strt_k[MAX_PROCS] = {0};

    long            my_end_k[MAX_PROCS] = {0};

    long            my_extra_k[MAX_PROCS] = {0};

    long            my_extra_n[MAX_PROCS] = {0};

    int             tmp_sum[6] = {0};

    int             tkt[6][128] = {{0, 0}};

int is::start(int nprocs, int seq, int chunk) {

    int             i, j;

    C_printf("Kernel IS, chunk=%d n", chunk);

    barrier.ReadAssociate(0, 0);

    /*********************** PHASE 1 **********************/

    if (seq == 0) {
        resettimer();
        work_size_n = N / nprocs;
        work_size_k = Bmax / nprocs;
        extra_n = N % nprocs;
        extra_k = Bmax % nprocs;
    }
    cp_synch();
    barrier.barrier();
    /************************  PHASE 2 **********************/
    if (seq < extra_n)
        my_extra_n[seq] = seq;
    else
        my_extra_n[seq] = extra_n;
    if (seq < extra_k)
        my_extra_k[seq] = seq;
    else
        my_extra_k[seq] = extra_k;
    my_strt_n[seq] = (seq * work_size_n) + my_extra_n[seq];
    my_end_n[seq] = my_strt_n[seq] + work_size_n - 1;
    if (seq < extra_n)
        my_end_n[seq] = my_end_n[seq] + 1;
    my_strt_k[seq] = (seq * work_size_k) + my_extra_k[seq];
    my_end_k[seq] = my_strt_k[seq] + work_size_k - 1;
    if (seq < extra_k)
```

```
        my_end_k[seq] = my_end_k[seq] + 1;
    for (i = 1; i <= 1; i++) {
        bucksort(seq, chunk, nprocs);
    }
    cp_synch();
    barrier.barrier();
    if (seq == 0) {
        long            t = readtimer();
        C_printf("Time = %dn", t);
    }
}
int is::bucksort(int seq, int chunk, int nprocs) {
    int             i, j, it, chu, k;
    C_printf("[%d] Start = %d  end = %dn",
            seq, my_strt_k[seq], my_end_k[seq]);
    /* Zero the keyden array */
    for (i=my_strt_k[seq]; i<=my_end_k[seq]; i++)
        keyden[i] = 0;
    for (i=0; i<Bmax; i++)
        keyden_t[seq][i] = 0;
    /* Count occurrences of each key (the 'key density') */
    for (i = my_strt_n[seq]; i <= my_end_n[seq]; i++) {
        k = key[i];
        keyden_t[seq][k] = keyden_t[seq][k] + 1;
}
cp_synch();
barrier.barrier();
/************************ PHASE 3 **********************/
for (j = 0; j < nprocs; j++) {
    keyden[my_strt_k[seq]] = keyden[my_strt_k[seq]]
        + keyden_t[j][my_strt_k[seq]]; }
for (j = 0; j < nprocs; j++) {
    for (i = my_strt_k[seq] + 1; i <= my_end_k[seq]; i++) {
        keyden[i] = keyden[i] + keyden_t[j][i]; } }
for (i = my_strt_k[seq] + 1; i <= my_end_k[seq]; i++)
    keyden[i] = keyden[i] + keyden[i - 1];
cp_synch();
barrier.barrier();
/*********************** PHASE 4 **********************/
if (seq == 0) {
    tkt[seq][0] = 1;
    tmp_sum[0] = 0;
    for (i = 1; i < nprocs; i++)
        tmp_sum[i] = tmp_sum[i - 1] + keyden[my_end_k[i - 1]];
}
cp_synch();
barrier.barrier();
/*********************** PHASE 5 **********************/
```

```
    for (i = my_strt_k[seq]; i <= my_end_k[seq]; i++)
        keyden[i] = keyden[i] + tmp_sum[seq];
    cp_synch();
    barrier.barrier();
    /*********************** PHASE 6 **********************/
    if (seq == 0) {
        for (i = 0; i < Bmax; i++)
            for (j = 0; j < nprocs; j++) {
                int             tmp_den = keyden_t[j][i];
                keyden_t[j][i] = keyden[i];
                keyden[i] -= tmp_den;
            }
        cp_synch();
    }
    barrier.barrier();
    /********************** PHASE 7 **********************/
    for (i = my_strt_n[seq]; i <= my_end_n[seq]; i++) {
        k = key[i];
        keyden_t[seq][k] = keyden_t[seq][k] - 1;
        rank[i] = keyden_t[seq][k];
    }
}
void is::init(int nprocs) {
    int             i;
    barrier.Initialize(nprocs);

    for (i = 0; i < N; i += 8192)

        S[i] = 0;

    for (i = 0; i < N; i++)

        rank[i] = 0;

    for (i = 0; i < Bmax; i++)

        keyden[i] = 0;

    resettimer();

    cp_synch();

    i = readtimer();

    C_printf("IS :: Initialization done in %d usec\n", i);

}
```

## A.2 SCAN Benchmark

This section describes the computation of various parameters for the simulation of the SCAN benchmark. All parameters except the amount of shared address space, number of references made by a task, and the parameter capturing false sharing aspects of the program are assigned default values. We computed the various parameters based on the source listing shown in section A.2.1. False-sharing is not a significant factor in this benchmark as all processors operate on disjoint pieces of data. Table 27 summarizes these parameters.

| Parameter | Formula |
|---|---|
| SharedAddressSpace | sizeof( struct record ) * MAXRECORDS |
| Number of References per task | (500*MAXRECORDS)/NumberOfNodes |
| FalseSharingRefProb | 0 |

Table 27: Approximate shared memory requirements for the SCAN benchmark for MAXRECORDS=10000 records

### A.2.1 Source Listing of the SCAN Benchmark

```
    #define        MAXLOCKS    100

    #define        MAXRECORDS    10000

clouds_class scan uses {}1

    C_rwlock        Alock[MAXLOCKS];

    Barrier     barrier;

public :

    entry void init(int);

    entry void readinputs(int, int, int, int);

    entry int  start(int myid, int procs,

             int start, int end, int delay);

end_class

struct record {

    char    name[45];

    char    address[45];

    int     age[2];

    int     ssno[2];

    int     wage[2];
    int     desig[2];
    int     super[2];
};
struct record file[MAXRECORDS]={0};
void scan :: readinputs(int myid, int nprocs,
                        int start, int end)
{
    int i,j;
    /* Prefetch your portion of data */
    for (i=start; i < 200; i++) {
        j = file[i].age[1];
    }
}
void scan:: init(int nprocs)
{
    int             i;
    for (i=0; i < MAXLOCKS; i++)
        Alock[i].Create();
    barrier.Initialize(nprocs);
    cp_synch();
    C_printf("Initialized\n");
}
scan:: start(int myid, int nprocs, int start,
             int end, int debug)
```

```
{
    int rec_no,k;
    unsigned long t2, t;
    struct record buf;
    barrier.ReadAssociate(0, 0);
    /******************** PHASE 1 ***********************/
    if (myid == 0) {
        resettimer();
    }
    int i=0;
    int lock;
    for (rec_no= start; rec_no < end; rec_no++) {
        if (rec_no == start) {
            lock = start/MAXLOCKS;
            if (debug)
                C_printf("[%d] Acquiring %d....\n", myid, lock);
            Alock[lock].wlock();
        }
        else
        if ((rec_no%MAXLOCKS) == 0) {
            if (debug)
                C_printf("[%d] Releasing %d....\n", myid, lock);
            Alock[lock].unlock();
            lock = rec_no/MAXLOCKS;
            if (debug)
                C_printf("[%d] Acquiring %d....\n", myid, lock);
            Alock[lock].wlock();
        }
        bcopy((char *) &file[rec_no],
               (char *) &buf, sizeof(struct record));
        for (k=0; k < 45; k=k+1) {
            buf.name[k] = buf.name[k] + 1;
            buf.address[k] = buf.address[k] + 1;
        }
        for (k=0; k < 2; k++) {
            buf.age[k] += 1;
            buf.ssno[k] += 1;
            buf.desig[k] += 1;
            buf.super[k] += 1;
            buf.wage[k] += 1;
        }
        for (i=0; i < 500; i++);
        bcopy((char *) &buf,
               (char *) &file[rec_no], sizeof(struct record));
    }
    if (debug)
        C_printf("[%d] Releasing %d....\n", myid, lock);
    Alock[lock].unlock();
    barrier.barrier();

    if (myid == 0) {

        t= readtimer();

        C_printf("Total Time = %u\n", t);

    }

    C_printf("%d done.........\n", myid);

}
```

# Appendix B

# Design and Implementation of Buffered Consistency based DSM on CLOUDS

This appendix presents the design and implementation of distributed shared memory based on the buffered consistency memory model (BC-DSM). The implementation is done on top of the CLOUDS distributed operating system. First, we briefly describe the buffered consistency memory model.

## B.1 The Buffered Consistency Memory Model

Buffered consistency (BC) has been proposed by Lee and Ramachandran [24] as a weak memory model for shared memory multiprocessors. The BC model recognizes two types of accesses: data and synchronization. Data accesses can be reads or writes to either private or shared data, whereas synchronization accesses are accesses to synchronization variables. The BC model distinguishes between two types of synchronization: non-consistency preserving (NP-Synch), and consistency preserving (CP-Synch). The BC model requires that synchronization accesses done by an application should be globally performed in the order of issue. Interleaving of synchronization accesses of different applications need not be sequentially consistent. The BC model places following restrictions on synchronization operations:

- The issue of an NP-Synch access does not require the preceding data accesses to be performed globally.

- Shared accesses following an NP-Synch access by an application cannot be issued until the NP-Synch access is performed.

- A CP-Synch access is not issued until all preceding writes to shared data have been globally performed.

BC implements reader-initiated memory coherence, i.e., if an application is interested in receiving modification for shared data then it would indicate so to the data server. As a result, any modification made to shared data will be propagated to the application until the request has been explicitly revoked.

In a nutshell, an implementation of buffered consistency in a distributed system requires three features from the underlying system:

- Ability to perform local reads and writes.

- Ability to perform global writes.

- Ability to suspend a process, a thread, or an application until all global writes prior to a CP-Synch operation have been globally performed.

## B.2 Implementation of BC-DSM on CLOUDS

On CLOUDS, the buffered consistency memory model has been integrated with the operating system. This has been done by modifying the DSM subsystem, which is responsible for maintaining coherence of shared data, to model the buffered consistency memory model. As a result, any application executing on CLOUDS can transparently use the buffered consistency memory model. To implement the coherence protocol, each DSMC needs to maintain some state information about the shared data pages that are resident at that node. The state information is maintained in state table, an entry of which is shown in Figure 30. Some fields of the state information are valid only at the *owner* node. An *owner* of a page is the node responsible for managing the consistency of the data page.

The following set of primitives are provided by the DSM subsystem for implementing buffered consistency.

**msg_get** *(interface to the kernel)* When a process needs a data page belonging to distributed shared memory, the kernel makes a request to the DSM subsystem on behalf of the process. During this time, the requesting process is blocked. If the data page is not locally available, the DSM subsystem sends a `msg_get` request to the *owner* node for that data page. The *owner* services the data request by returning a copy of the data page to the requesting node. Upon receipt of the data page, the kernel installs the page in the process' address space and resumes execution of the suspended process.

**cp_synch** *(interface to the kernel)* To globally perform all modifications to shared data before a CP-Synch point, the `cp_synch` primitive is provided. When a CP-Synch point is reached during the execution of an application, the kernel forces all changes made to the shared data pages to be flushed to the respective *owners* of the data pages. This is done by identifying all the dirty pages in the process' address space and issuing the `cp_synch` call for the page to the DSM subsystem. The DSM subsystem computes the difference page for the data page by *XORing* the contents of the data page with the unmodified (original) copy of the page. The difference page is then compressed[1] and sent to the *owner*. The *owner* uncompresses the difference page, applies it to its version of the data page. If any other node is interested in hearing about the modifications, the *owner* sends a copy of the difference page (via `msg_update` request) to all these nodes. Figure 31 shows the control flow for the `cp_synch()` system call.

**request_update** (*accessed via a system call)* If a process is interested in receiving updates to a page the it indicates this to the kernel via a system call. The kernel notifies the DSM subsystem. The DSM subsystem sends a `request_update` message to the owner for the segment. The owner will send future updates for a page via the `msg_update`.

**reset_update** (*accessed via a system call)* To stop receiving previously requested updates, for specific pages of a segment, the DSM subsystem sends a `reset_update` message to the owner of the segment. Again, this is done under user-direction.

**msg_update** *(interface to another DSM subsystem)* To propagate changes globally before the completion of the `cp_synch` operation, the DSM subsystem sends the `msg_update` message to other DSM subsystems that are interested in hearing about these changes. Upon receipt of a `msg_update` message, the

[1]Due to the large software overhead associated with doing compression, we do not do data compression in the current implementation. However, provision for data compression has been made in the hardware realization of the DSM subsystem.

DSM applies the modifications, and installs the new copy of the data in the address spaces of the affected processes. One way to to achieve this is to invalidate all copies of the data page in any process' address spaces. Subsequent page-faults on the data will result in the DSM subsystem supplying the latest copy of the page to the requesting process.

```
class DtableEntry : public BasicQueueElem {
public :
    SysName SegmentName;  //Name of the segment
    u_int block_number;   //Number of the block within the segment
    struct {
        boolean valid;                // Is the data valid?
        FrameHandle phys_frame;       // Where is the data located?
        stable *map;                  /* The phys_frame above belongs to
                                       * this map in the segment. map will be
                                       * used in invalidating the stable
                                       * entries.
                                       */
        FrameHandle orig_frame;       //Copy of the data item before being
                                      //installed in process' address space
        boolean get_read_pending;     // Are we still reading the data?
    } data;
    boolean owner_flag;   //Is this node the owner
    u_long copyset;       //List of nodes that have requested
                          //updates. Bitmask for now.
};
```

Figure 30: Structure of the *dtable* entry

## B.3 Performance of BC-DSM on CLOUDS

We have implemented buffered consistency as part of the distributed shared memory subsystem on CLOUDS. The basic performance for buffered consistency is

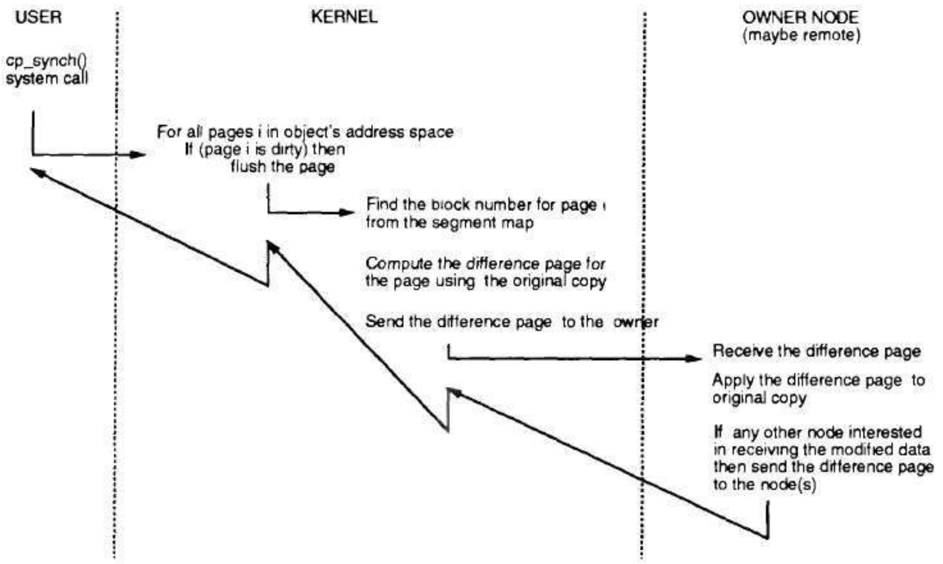
USER
KERNEL
OWNER NODE (maybe remote)
cp_synch() system call
For all pages i in object's address space
If (page i is dirty) then
flush the page
Find the block number for page i from the segment map
Compute the difference page for the page using the original copy
Send the difference page to the owner
Receive the difference page
Apply the difference page to original copy
If any other node interested in receiving the modified data then send the difference page to the node(s)

Figure 31: Flow of control for the `cp_synch()` system call

slightly poorer than other coherence protocols that provide sequentially consistent view of memory due to the cost of maintaining an extra copy of data page. This copy is used at the time of the `cp_synch()` call to generate the difference page. Tables 28 and 29 show the breakdown of times for page-fault servicing and the `cp_synch()` system call. These timing measurements are done on a Sun 3/60 with a microsecond timer. Approximately 25 ms per page is spent during a `cp_synch()` call. Bulk of the time is spent transmission of data to the owner, and any additional housekeeping done at the owner. Additional time would be needed if the data needs to be sent to other nodes, which are interested in receiving updates for that data page. A receipt of `msg_update` incurs a cost of approximately 3 ms, bulk of the times is spent in updating old copies of the data page.

| Operations<br>(All times are in milliseconds) | Breakdown | Measured<br>Total Time |
|---|---|---|
| Get from a data server | | 17.550 |
| - Basic RaTP 8K bytes transfer | 13.500 | |
| - 1 Context switch at the server | 0.150 | |
| - DSM processing at the server (updating state information) | 2.000 | |
| - One copy of data from Ethernet buffers into client buffers | 1.450 | |
| TOTAL TIME | 17.100 | |
| Write page fault servicing from another CLOUDS server | | 20.540 |
| - DSM message get call | 17.550 | |
| - Client side state maintenance | 0.800 | |
| - One copy for maintaining original copy of data | 1.450 | |
| TOTAL TIME | 19.800 | |

Table 28: Basic system timings for BC-DSM on CLOUDS

| Operations<br>(All times are in milliseconds) | Breakdown | Measured<br>Total Time |
|---|---|---|
| CP synch system call (if data page has been modified) | | 25.000 |
| - Trap into the kernel from user level | 0.131 | |
| - State maintenance | 0.800 | |
| - Computing the Log (in software for 8K data) | 4.700 | |
| - Cost of sending the flush request to the owner site (8 Kbytes data transfer, applying log at the server, state maintenance) | 17.700 | |
| - Cost of copying modified data for future use | 1.450 | |
| TOTAL TIME | 24.780 | |

Table 29: Service times for a `cp_synch()` system call for BC-DSM on CLOUDS

# Appendix C

# Hardware Support for Distributed Shared Memory

Performance studies of the distributed shared memory system on CLOUDS (see chapter 4) have shown that DSM related activity incurs additional processing overhead on a host node. This overhead usually consists of sending and receiving data, and processing control messages (such as invalidations, updates) for maintaining coherence of shared data. The amount of overhead is a function of the coherence protocol, the application characteristics, degree of sharing between nodes, and the number of nodes in the distributed system. Measurements show that approximately 20% of the processing done during servicing a remote page-fault is due to DSM related activity. Such overhead can substantially degrade system performance by reducing the available number of useful cycles. To alleviate this problem, we propose the design of a *Distributed Shared Memory Controller* (DSMC). The controller off-loads all DSM related processing from the host by servicing all DSM related requests, thereby freeing the host to perform other useful work.

The DSMC is a coprocessor board that sits on the system backplane, communicating with the host via the system bus. The board is self-contained, i.e., it has its own processor, private memory, and control logic needed to service any

DSM-related requests. It interacts with the host through a well-defined interface. In the following sections, we discuss the various aspects of the design. We conclude the discussion with expected system performance improvement due to the controller. In the present design, the controller does not have direct access to the network. We assume that the controller can access the network adapter via the host, and is able to receive and send messages to the network. Such a design may cause the communication subsystem to become a bottleneck if large amounts of data traffic is generated by the controller. One solution to this problem would be to allow the controller to directly access the network.

## C.1 Hardware Design of the Controller Board

Figure 32 shows the layout of the DSM coprocessor board. The board consists of a 32-bit microprocessor, memory modules, system-bus interface logic, and specialized chip set to perform compression/decompression of data. The microprocessor communicates with its local memory via a local bus, while it communicates with the host via its VME bus interface. The VME bus interface consists of the VIC 068 chip [10]. The VIC 068 chip allows easy access to the board from the host. It also offers a facility for inter-process communication via a dedicated set of inter-process communication registers and mailboxes. Figure 33 shows the functional description of the VIC 068 chip. The resident memory on the board is used for storing the control software, and private data. A portion of the resident memory is also mapped onto host's address space for access via the VME bus. Thus, if the host writes to the overlapped portions of memory, the data can be read directly

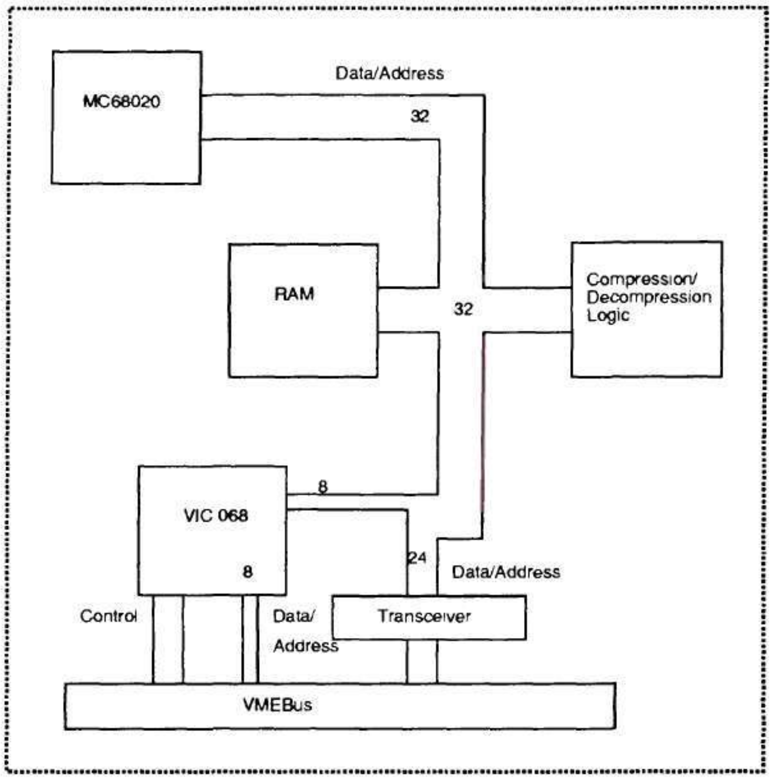

Figure 32: Layout of the DSMC coprocessor board

by the DSM coprocessor. Similarly, data written to this portion of memory by the DSMC can be accessed by the host. Synchronization between the coprocessor and the host is achieved using the inter-processor communication module switches provided by the VIC 068 chip. The interaction between the host processor and the controller is non-blocking. This means that while a data request is being serviced by the controller, the host processor is free to perform other activities. Only the thread/process that caused the page-fault gets blocked. Similarly, while a reply is pending from a remote controller, the local controller is free to service other requests from the processor.

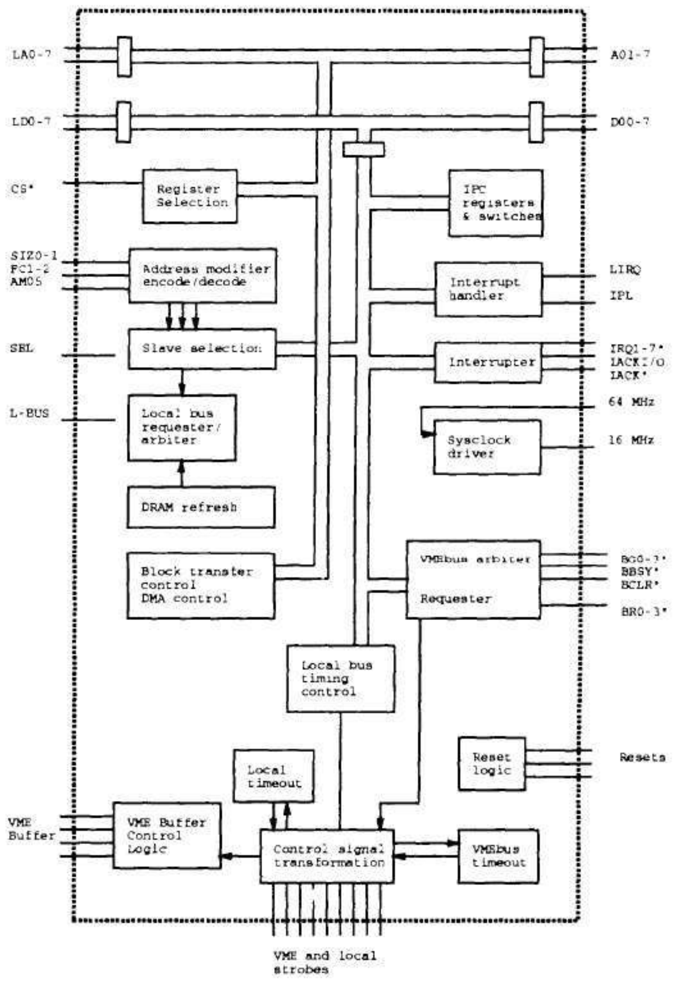

Figure 33: Functional description of the VIC 068 VME bus interface chip

## C.2 Software Design for the Controller Board

The software organization of the controller consists of three modules: control software for the controller, interface between the controller and the host, and interface between two controllers.

## Control Software for the Controller

The control software implements the protocol for managing coherence of shared data. It also maintains data structures for storing state information about resident data pages. As mentioned in section C.1, some portion of the controller memory is shared with the host's address space. At the time of system initialization, the host sets up two circular buffers in this portion of the shared memory. These two buffers, the **`request-buf`** and the **`reply-buf`**, are used for communication between the controller and the host. All data requests made by the host are put in the **`request-buf`** while all replies sent by the controller are put in the **`reply-buf`**. Both, the host and the controller, maintain a pointer to the last processed entry in the buffer. Figure 34 shows the structure of a buffer entry. Each entry consists of the request type, and information about the data-page being operated upon. The address field in the entry points to location in host's memory where data might be located.

```
enum MsgType = {Get, Discard, Flush, Install, Invalidate};
enum Mode = {Read, Write, WeakRead, None};
struct message {
 enum  MsgType type;       /* Type of message; request, reply */
 long          segment-id; /* segment id */
 long          page-num;   /* page number */
 enum  Mode    mode;       /* mode, if applicable */
 long          address;    /* Address where data should be put */
};
```

Figure 34: Structure of a buffer entry

```
struct dsm_entry {
 long      segment-id; /* segment id */
 long      page-num;   /* page number */
 enum Mode mode;       /* Mode is which page is installed */
 long      orig-page;  /* pointer to original copy */
};
```

Figure 35: Structure of a state table entry

State information about all shared data pages resident at a node is maintained in separate state tables. These tables contain information about the owner of a page, mode in which the page is present at the node, pointer to the original copy of the data page in controller's memory, and any coherence protocol specific state information. The structure of an entry of the state table is shown in Figure 35.

## Controller - Host interface

The following set of primitives define the interface between the controller and the host.

• `get(segment, page, mode, physical address)`: The `get` primitive, issued by the host on a page-fault on shared data, indicates to the controller that the host is interested in acquiring the data-page for the segment in the specified mode.

• `discard(segment, page, physical address)`: The `discard` primitive allows a host to throw out a shared data-page from its memory. The controller is responsible for taking appropriate coherence actions on the discarded page. These actions could include throwing out the data-page, sending it back to the owner, or taking no action at all. The actions taken by the controller are based on the coherence protocol being used by the controller.

• `flush(segment, page, physical address)`: The `flush` primitive is functionally similar to the `discard` primitive. The only difference is that on a `flush`, only the modifications to a page (using the *diff* page) are sent to the owner. The *diff* page is constructed by *XORing* the contents of the modified page and the original data-page. This primitive is used in the implementation of the write-update protocol.

• `install(segment, page)`: The `install` primitive is issued by the controller to the host, indicating that a data-page requested by the host is now available for installation. The controller issues `install` after it has serviced a `get` request for the data-page.

• `invalidate(segment, page)`: Certain coherence schemes may require that a data-page in host memory be invalidated, i.e., memory mappings associated with a data-page be invalidated. The `invalidate` message is sent by the controller to the host indicating that a particular data-page be invalidated. One example where this primitive can be used is in the implementation of the write-invalidate scheme, where read-only copies of the data-pages need to be invalidated before a write to the data-page can occur. This primitive can also be used to force the host to request a fresh copy of a data-page.

• `receive_updates(segment, page, length, flag)`: This primitive is specific to the write-update coherence protocol based on the buffered-consistency memory model (see Appendix B). Under user-direction, the primitive allows a program to request updates for a specific data-page. The request for updates is sent to the owner. The owner then continues to send fresh updates for the data-page until the request is explicitly turned off by the user. For page-oriented coherence protocols, the host specifies the starting page of the segment that the user is interested in receiving updates for.

## Controller - Controller interface

The following set of primitives are defined for co-ordinating activities among controllers. These primitives provide the basic functionality to implement different types of coherence protocols.

- `c_get(segment-id, page, mode)`: If a controller is not able to service a request for a data-page, it sends a `c_get` message to the owner, requesting the data-page on behalf of the host. The owner takes appropriate actions to service the data request.

- `c_data(segment-id, page)`: In response to the `c_get` request, the owner sends the data to the requesting controller using the `c_data` primitive.

- `c_discard(segment-id, page)`: Information that a data-page has been discarded by a node is sent to the owner using the `c_discard` primitive. If needed, the data-page is also sent to the owner.

- `c_flush(segment-id, page)`: The `c_flush` primitive is similar to `c_discard`, except that only modifications to the data are sent to the owner.

- `c_forward(segment-id, page)`: Sometimes an owner of a page may need to request another node to forward the data to the requesting node. The owner can do so using the `c_forward` primitive.

• `c_update(segment-id, page)`: Some weaker coherence protocols require updates to a data-page be propagated to other nodes. A controller can send updates to other nodes using the `c_update` primitive.

• `c_invalidate(segment-id, page)`: A controller can instruct other controllers to invalidate their copy of a data-page by sending the `c_invalidate` message to them.

• `c_receive_updates(segment-id, page, size, flag)`: A controller sends a request to the owner if the host is interested in receiving/reseting updates for a data-page. If updates are requested, any future updates to the data-page will be sent by the owner using the `c_update` primitive.

## C.3 Functional Description of the Controller

In this section, we describe the events that occur during processing of different requests by the controller.

### Handling page-faults on shared data

When a page-fault occurs on data belonging to shared memory, the kernel determines the segment-id, the page and the mode in which the data needs to be fetched. A `get` message is constructed using this information. A physical page is allocated in memory and its physical address added to the contents of the message. The physical address is used by the controller for storing the data. The get message is then added to `request-buf`. The host then indicates to the controller that a request is pending in the request queue. This is be done by writing to the VME bus interface chip, which then generates an interrupt to the controller.

The controller, upon receiving an interrupt, sets out to service the request. It removes the message from the `request-buf` and adds it to its private `request-queue`. Using the <segment-id, page-num> as a key, the controller looks through its local tables to determine if it has a valid copy of the data-page in its private memory. If a valid copy is found, the controller initiates a DMA transfer of the data from its private memory to the host's memory using the physical address supplied by the host as the target address. Upon successful completion of the DMA transfer, the controller sends the `install` message to the host. This is done by writing the message to the `reply-buf` and interrupting the host. However, if the controller is unable to locate the data-page corresponding to the segment-id and page-num, it sends a `c_get` message to the owner, requesting the data page.

When an owner receives a `c_get` message, it locates the data-page and sends the data back to the requesting controller using the `c_data` primitive. If the data page is not found then it can request (via `c_forward`) another node to forward the data to the requester. The owner also updates its local state tables to reflect any coherence specific information. Upon receipt of the data page, the controller copies the page into its local memory. The controller identifies the request corresponding to the page. Using DMA, the data is transferred to host's memory. After successful completion of DMA, the controller writes the `c_install` in the `reply-buf` and interrupts the host. Upon receiving an interrupt from the controller, the host removes the reply from the `reply-buf` and services the page-fault.

### Discarding a page

When a shared data page is no longer needed by a host, it discards the data. This is done by sending a `discard` message to the controller. The controller copies the data page into its private memory, and takes appropriate action depending on the coherence protocol being used. If the coherence protocol requires that the data be returned to the owner, the controller sends the data to the owner using the `c_discard` message. After all the processing is done, the host is informed that the page has been discarded.

### Flushing a page

When the host wishes to flush out all modification made to a data page, it does so by issuing a `flush` message to the controller. The controller copies the data page into its private memory and constructs the difference page for the data. The difference page represents the modifications made to the data page and is constructed by XORing the contents of the modified page and the original page. The difference page is then sent to the owner.

### Invalidating a page

The coherence protocol may require that all copies of the data-page be invalidated before any more activity on the page can occur. This can be achieved by sending a `c_invalidate` message to all nodes that have a copy of the page. The controllers at these nodes instruct their hosts to invalidate the copies of the data-page by sending an `invalidate` message. Upon receiving an `invalidate` message from the controller, the host invalidates the corresponding data-page mappings in its MMU. Subsequent access to the page will cause a page-fault. This page-fault will be handled as explained earlier.

### Installing updates for a page

If a controller wishes to update all copies of a data-page resident at other nodes, it sends the data-page to all controllers via the `c_update` message. The controller installs the data copy in its private memory and sends an `invalidate` message to the host. Subsequent access to the page would cause a page-fault, resulting in the new copy of the page being installed in host's memory.

## C.4 Expected Performance of the Controller

In this section, we examine the performance implications of a DSM controller on the host system performance. The examination is done in context of the write-update protocol based on the buffered-consistency memory model. More details about this protocol can be found in Appendix B. In our analysis, we assume the controller uses the Motorola MC68020RC16 microprocessor with 60ns cycle time. A typical 32-bit read or write on MC68020 takes about 3 cycles. We also assume that a 32-bit DMA transfer between the controller board and host memory takes about 6 cycles (worst case).

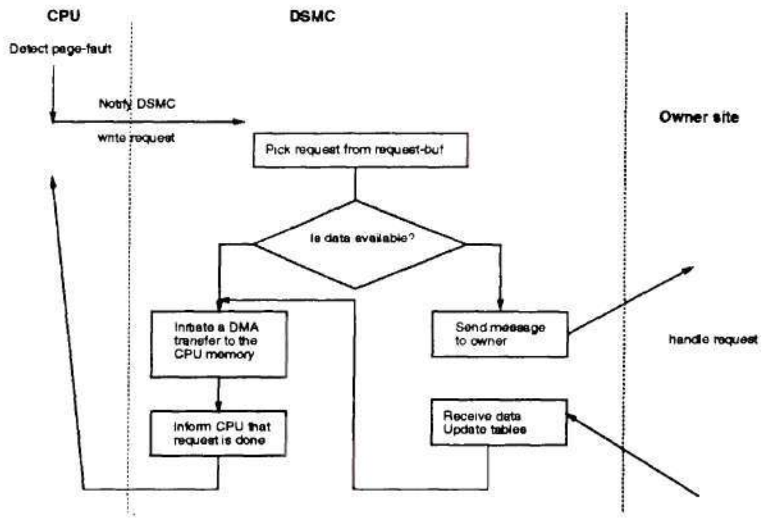
CPU
DSMC
Detect page-fault
Notify DSMC
write request
Owner site
Pick request from request-buf
Is data available?
Initiate a DMA transfer to the CPU memory
Send message to owner
handle request
Inform CPU that request is done
Receive data Update tables

Figure 36: Page-fault servicing by the controller

## Page-fault servicing

Figure 36 shows the sequence of events that occur upon a page-fault on shared data. The breakdown for expected service time of 1.739 ms is shown in Table 30. Data that needs to be requested from another controller will incur additional cost of constructing the message to be sent, receiving the data over the network, and updating the local data structures. This overhead would be approximately 15 ms per data page over an Ethernet-like communication network.

## Servicing a cp_synch() call

Figure 37 shows the sequence of events that occur when a `cp_synch()` call is issued. The `cp_synch()` call is issued by the host to enforce global writes to shared data before the program completes the synchronization call. This is an artifact of the coherence protocol implemented by the controller. The expected

| Event | Operations | Number of cycles | Time (in ns) |
|---|---|---|---|
| -Host: writes request in `request-buf` | 4-word write | 12 | 720 |
| -Host: notify DSMC | 1-word write | 3 | 180 |
| -DSMC: read request from `request-buf` | 4-word read | 12 | 720 |
| -DSMC: locate data in the state tables | | | 1000000 |
| -DSMC: initiate DMA transfer to host memory (8192 bytes) | 2048-word transfer | 12288 | 737280 |
| -DSMC: write reply in `reply-buf` | 4-word write | 12 | 720 |
| -DSMC: signal the host | 1-word write | 3 | 180 |
| TOTAL TIME | | | 1739080 |

Table 30: DSMC: Times for page-fault servicing for resident data

time for individual events is summarized in Table 31. Performing a `cp_synch` operation costs approximately 18.37 ms. The bulk of the time is spent in sending the data to the owner. Computing the difference page takes about 21 cycles per 32-bit word (2.5 ms/8192 bytes). This entails reading two words, performing an exclusive-or operation and writing the result back. Additional cycles are needed for executing the loop 2048 times. Compressing a data page costs an additional 602 $\mu$s.[1] The advantage of using data compression is that one can save on data transmission costs. For example, assuming on an average 50% reduction in size of a data page, transmitting an 8192-byte data page over Ethernet at 10Mbps would save approximately 3.25 ms per page.

[1]Currently available data compression hardware can operate at speeds from 13.6 to 20MB/sec [6].

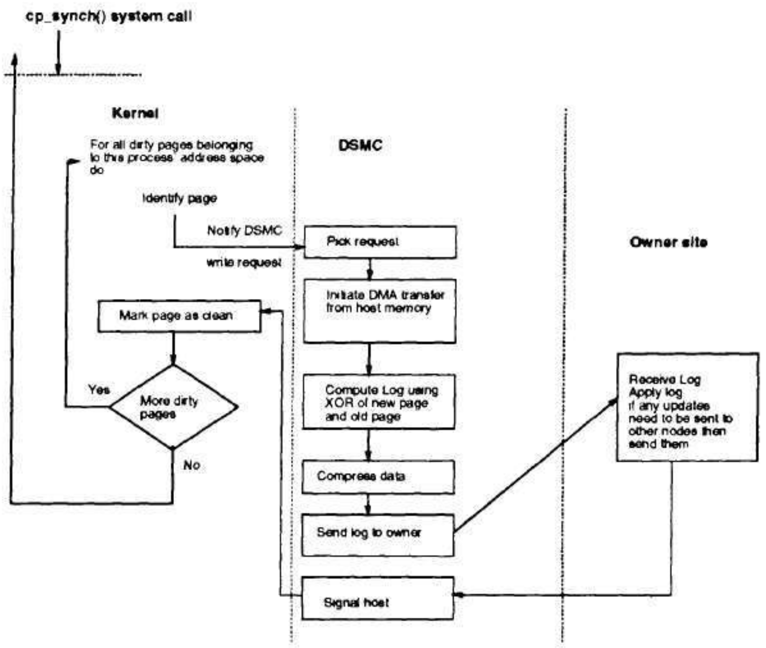
cp_synch() system call
Kernel
For all dirty pages belonging to this process' address space do
Identify page
Notify DSMC
write request
Mark page as clean
More dirty pages
Yes
No
DSMC
Pick request
Initiate DMA transfer from host memory
Compute Log using XOR of new page and old page
Compress data
Send log to owner
Signal host
Owner site
Receive Log
Apply log
if any updates need to be sent to other nodes then send them

Figure 37: Schematic of DSMC actions subsequent to a `cp_synch()` system call

## Servicing an update request

Figure 38 shows the sequence of events that occur when an update message is received by the controller. The expected times for individual events are summarized in Table 32. On an average, approximately 4 ms is spent processing an update message, bulk of which is spent applying the modifications to the page.

## Advantages of data compression

Data transmission is the major costs incurred in distributed shared memory systems. Several coherence protocols try to reduce data communication costs by reducing the cost of maintaining coherence. Another way to reduce data communication cost is the use of data compression techniques. Current state of the

| Event | Operations | Number of cycles | Time (in ns) |
|---|---|---|---|
| -Host: write request in `request-buf` | 4-word write | 12 | 720 |
| -Host: notify DSMC | 1-word write | 3 | 180 |
| -DSMC: read request from `request-buf` | 4-word read | 12 | 720 |
| -DSMC: initiate DMA transfer for data page to be flushed from host memory (8192 bytes) | 2048-word transfer | 12288 | 737280 |
| -DSMC: compute the difference page | 2048-word XOR | 43008 | 2580480 |
| -DSMC: compress the difference page | 8192-bytes @ 13.6 MB/sec | 8192 | 602352 |
| -DSMC: sending the difference page using `msg_flush` and waiting for an acknowledgement | | | 13450000[a] |
| -DSMC: write reply in `reply-buf` | 4-word write | 12 | 720 |
| -DSMC: signal the host | 1-word write | 3 | 180 |
| TOTAL TIME | | | 18372632 |

[a]Taken from software measurements

Table 31: DSMC: Service time for a `cp_synch` call

art hardware technology has made it possible to use on-the-fly data compression to substantially reduce the data transmission costs. We use data compression for our DSMC implementation because the difference pages generated using XOR in the buffered consistency coherence protocol are most likely to contain a high percentage of strings of 0s, thereby yielding good compression ratios. Thus, such a protocol will benefit with the use of data compression techniques especially for slower communication mediums such as Ethernet. For example, in an Ethernet-like media with a communication speed of 10 Mbps, a compression ratio of more

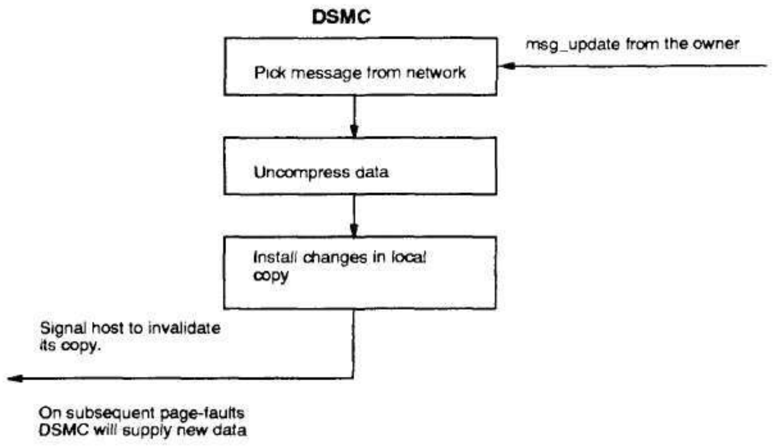
DSMC
msg_update from the owner
Pick message from network
Uncompress data
Install changes in local copy
Signal host to invalidate its copy.
On subsequent page-faults DSMC will supply new data

Figure 38: Schematic of DSMC actions subsequent to the receipt of a `msg_update` message

than 0.82 would yield good system performance.

## Effects of a DSM Controller

In the implementation of distributed shared memory on CLOUDS, a processor performs all distributed shared memory related activity along with any computation that needs to be performed at the node. As mentioned earlier, at least 20% of the page-fault servicing a time is related to DSM related activity. Such overhead is acceptable if only one thread or process is executing on a processor. In real life, however, this is not the case. A processor may be simultaneously executing several processes (using time-sharing), and the 20% DSM related overhead per page fault can degrade the overall throughput for the node. Similarly, an increase in DSM activity in the system may require a processor to handle a large number of control requests (such as invalidation, forward, and updates) from other processors for maintaining coherence of shared data. Such overheads can adversely affect the

| Event | Operations | Number of cycles | Time (in ns) |
|---|---|---|---|
| -DSMC: receive `msg_update`, including DMA transfer from host memory (8192 bytes) | | | 1000000[a] |
| -DSMC: uncompress data | 8192-bytes @ 13.6 MB/sec | 8192 | 602352 |
| -DSMC: apply the difference page | 2048-word XOR | 43008 | 2580480 |
| -DSMC: write invalidation request in `reply-buf` | 4-word write | 12 | 720 |
| -DSMC: signal the host | 1-word write | 3 | 180 |
| TOTAL TIME | | | 4183732 |

[a]Approximately, includes 737280ns for DMA.

Table 32: DSMC: Service time for a `msg_update` message

application performance by reducing the number of cycles that are available for doing the computation.

To study the performance gains of having a separate controller to handle distributed shared memory related requests, we extended the simulation studies of chapter 6. Figure 39 compares the results for two such system configurations; one with a DSM controller and one without a DSM controller. As shown in Figure 39, the system with a DSM controller shows much better performance than than one without. This experiment shows the performance with respect to a transaction workload model, executing on 16-node system. The improvement is more significant for small data transfer sizes than for large data sizes. The reason is that for small data transfer sizes, more number of DSM requests are generated in the system, thereby requiring more DSM related work to be performed at each node.

Under such conditions, existence of a DSM controller improves the system performance. Benefits accrued due to the presence of the DSM controller become more significant as the number of processors used for solving a problem is increased. For example, for a data transfer size of 2 Kbytes, 13% performance improvement is observed due to the presence of the controller in a 4-node configuration, 25% in a 8-node configuration, and 47% in a 16-node configuration. This behavior is consistent with the observation that an increase in the number of nodes is accompanied by an increase in the amount of DSM related activity in the system.

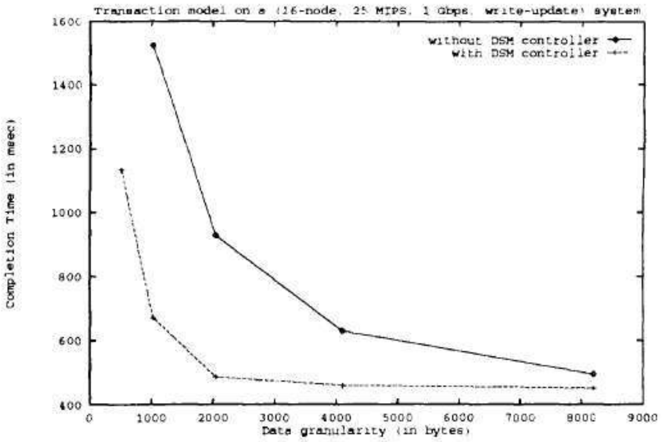

Figure 39: Effect of using a DSM controller on the performance

In this chapter, we have presented a detailed design and performance analysis of a distributed shared memory controller. The controller has been designed to improve a node's throughput by off-loading all DSM related processing from the host processor.

# Appendix D

# Number of Messages for the Coherence Protocols

Let

| | |
|---|---|
| $\mathcal{S}$ | Number of synchronization phases |
| M | Amount of memory operated by a processor during a computation |
| $w$ | Probability that an access is a write operation |
| $h$ | Hit ratio |
| $c$ | Probability that an access read/write will cause coherence messages to be sent to other nodes |
| $r$ | Number of nodes involved in receiving coherence messages |
| $\mathcal{G}$ | Unit of data transfer |
| $\mathcal{P}$ | Number of messages needed to bring in $\mathcal{M}$ bytes of memory. $\mathcal{P} = \frac{\mathcal{M}}{\mathcal{G}}$ |

Each of the following activity is counted as one message:

- Requesting a lock from the lock-server
- Granting a lock by the lock-server

- Releasing a lock by a client
- Requesting data from a data-server
- Sending data to the client by the data-server
- Sending an invalidation-message to a node
- Receiving an acknowledgement for an invalidation-message from a node
- Sending an update-message to a node
- Receiving an acknowledgement for an update-message from a node
- Forwarding of a data request by the data-server to another node (implementing mode `none` semantics in the lock-based protocol).

The total number of messages generated by a node during the execution of a computation for the write-update protocol is shown in 33; for the write-invalidate protocol is shown in Table 34; and for the lock-based protocol is shown in Table 35;

| Activity | Number of messages |
|---|---|
| **Synchronization phase** | |
| Write-access | $\mathcal{S}w(5 + 2r)$ |
| Read-access | $5\mathcal{S}(1\text{-}w)$ |
| **Computation phase** | |
| Write-miss | $2\mathcal{P}w(1\text{-}h)$ |
| Read-miss | $2\mathcal{P}(1\text{-}w)(1\text{-}h)$ |
| Total | $\mathcal{S}(5 + 2rw) + 2\mathcal{P}(1\text{-}h)$ |

Table 33: Number of messages generated in the write-update protocol

| Activity | Number of messages |
|---|---|
| **Synchronization phase** | |
| Write-access, no coherence activity | $5\mathcal{S}w(1\text{-}c)$ |
| Write-access, coherence activity | $\mathcal{S}(5 + 2r)wc$ |
| Read-access, no coherence activity | $5\mathcal{S}(1\text{-}w)(1\text{-}c)$ |
| Read-access, coherence activity | $6\mathcal{S}(1\text{-}w)c$ |
| **Computation phase** | |
| Write-miss, no coherence activity | $2\mathcal{P}w(1\text{-}h)(1\text{-}c)$ |
| Write-miss, coherence activity | $\mathcal{P}(2 + 2r)w(1\text{-}h)c$ |
| Read-miss, no coherence activity | $2\mathcal{P}(1\text{-}w)(1\text{-}h)(1\text{-}c)$ |
| Read-miss, coherence activity | $3\mathcal{P}(1\text{-}w)(1\text{-}h)c$ |
| Total | $\mathcal{S}(5 + 2rwc + c(1\text{-}w)) +$ $\mathcal{P}(1\text{-}h)(2+ c(w(5 + 2r) + 1))$ |

Table 34: Number of messages generated in the write-invalidate protocol

| Activity | Number of messages |
|---|---|
| **Synchronization phase** | |
| Write-access | $3\mathcal{S}w$ |
| Read-access | $3\mathcal{S}(1\text{-}w)$ |
| **Computation phase** | |
| Write-miss, data with the server | $2\mathcal{P}w(1\text{-}h)(1\text{-}c)$ |
| Write-miss, data with another node | $3\mathcal{P}w(1\text{-}h)c$ |
| Read-miss, data with the server | $2\mathcal{P}(1\text{-}w)(1\text{-}h)(1\text{-}c)$ |
| Read-miss, data with another node | $3\mathcal{P}(1\text{-}w)(1\text{-}h)c$ |
| Total | $3\mathcal{S} + \mathcal{P}(1\text{-}h)(2 + c)$ |

Table 35: Number of messages generated in the lock-based protocol

# Bibliography

[1] M. Ahamad, P. W. Hutto, and R. John. Implementing and programming causal distributed memory. In *11th International Conference on Distributed Computing Systems*, pages 274–281, 1991.

[2] R. Ananthanarayanan, R. John, A. Mohindra, M. Ahamad, and U. Ramachandran. An evaluation of state sharing techniques in distributed operating systems. Unpublished, April 1993.

[3] J. Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273 – 298, Nov 1986.

[4] G. M. Baudet. Asynchronous iterative methods for multiprocessors. *Journal of the ACM*, 25(2):226–244, April 1978.

[5] Roberto Bisiani and Alessandro Forin. Multilingual parallel programming of heterogeneous machines. *IEEE Transactions on Computers*, 37(8):930–945, August 1988.

[6] Suzanne Bunton and Gaetano Borriello. Practical dictionary management for hardware data compression. *Communications of the ACM*, 35(1):95–104, January 1992.

[7] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *The 13th ACM Symposium on Operating Systems Principles*, Oct 1991.

[8] Peter B. Danzig and Stephen Melvin. High resolution timing with low resultion clocks and A microsecond resolution timer for Sun workstations. *Operating Systems Review*, 24(1):23–26, 1990.

[9] Partha Dasgupta, Richard LeBlanc, Mustaque Ahamad, and Umakishore Ramachandran. The CLOUDS distributed operating system. *IEEE Computer*, April 1991.

[10] Colin Davis. Features of a VMEbus interface chip design. *Electronic Engineering*, pages 83–89, March 1989.

[11] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor simulation and tracing using TANGO. In *International Conference on Parallel Processing*, pages II–99–107, 1991.

[12] Gary S. Delp, Adarshpal S. Sethi, and David J. Farber. An analysis of Memnet: An experiment in high-speed shared-memory local networking. In *Computer Communication Review*, volume 18, pages 165–174, Stanford, California, August 1988. ACM SIGCOMM.

[13] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In *The 13th International Symposium on Computer Architecture*, pages 434–442, June 1986.

[14] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proc. 14th Int'l. Symp. on Computer Architecture*, pages 373–382, June 1988.

[15] Anon *et al.* A measure of transaction processing power. *Datamation*, pages 112–118, April 1985.

[16] Brett D Fleisch. Reliable distributed shared memory. In *Second IEEE Workshop on Experimental Distributed Systems*, pages 102–105, Huntsville, Alabama, October 1990. Extended Abstract.

[17] Brett D. Fleisch and Gerald J. Popek. Mirage: A coherent distributed shared memory design. *Operating Systems Review*, 23(5):211–223, Dec 1989.

[18] J. R. Goodman and P. J. Woest. The Wisconsin Multicube: a new large-scale cache-coherent multiprocessor. In *Proc. 15th Int'l. Symp. on Computer Architecture*, pages 422–431, June 1988.

[19] R. E. Kessler and Miron Livny. An analysis of distributed shared memory algorithms. In *9th Intl Conference on DIstributed Computing Systems*, pages 498–505, 1989.

[20] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transaction on Computers*, C-28(9):690–691, Sep 1979.

[21] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *Operating Systems Review*, 13(2):3–19, April 1979.

[22] Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A Hamilton, David L Nelson, and Bernard L. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, 1(5):842–857, November 1983.

[23] Joonwon Lee and Umakishore Ramachandran. Synchronization with multiprocessor caches. In *Proc. 17th Int'l. Symp. on Computer Architecture*, pages 27–37, May 1990.

[24] Joonwon Lee and Umakishore Ramachandran. Architectural primitives for a scalable shared memory multiprocessor. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 103–114, july 1991.

[25] Joonwon Lee and Umakishore Ramachandran. Locks, directories, and weak coherence - a recipe for scalable shared memory multiprocessors. In *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991.

[26] D Lenoski, J. Laudon, K Gharachorloo, A Gupta, and J Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. *Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 148–159, 1990.

[27] W. G. Levelt, M. F. Kaashoek, H. E. Bal, and A. S. Tanenbaum. A comparison of two paradigms for distributed shared memory. *Software - Practice and Experience*, 22(11):985–1010, Nov 1992.

[28] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *tocs*, 7(4):321–359, November 1989.

[29] Ronald G. Minnich and David J. Farber. Reducing host load, network load, and latency in a distributed shared memory. In *10th International Conference on Distributed Computing Systems*, pages 468–475, May 1990.

[30] 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*, pages 160–169, Aug 1989.

[31] Umakishore Ramachandran, Gautam Shah, S. Ravikumar, and Jeyakumar Muthukumarasamy. Scalability study of the KSR-1. Technical Report GIT-CC 93/03, Georgia Institute of Technology, Atlanta, GA 30332-0280, 1993.

[32] Richard Rashid, Avadis Tevanian, Micheal Young, David Golub, Robert Baron, David Black, Willian Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, 1987.

[33] Kendall Square Research. *KSR1 Principles of Operations*, 1991.

[34] A. Sane, K. MacGregor, and Roy H. Campbell. Distributed virtual memory consistency protocols: Design and performance. In *Second IEEE Workshop on Experimental Distributed System s*, pages 91–96, Oct 1990.

[35] H. D. Schwetman. CSIM: A C-based, process-oriented simulation language. In *Proceedings of the 1986 Winter Simulation Conference*, pages 387–396, December 1986.

[36] John F. Shoch, Yogen K. Dalal, David D. Redell, and Ronald C. Crane. Evolution of the ETHERNET local computer network. *IEEE Computer*, pages 1–27, Aug 1982.

[37] Michael Stumm and Songnian Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, pages 54–64, May 1990.

[38] Christopher J. Wilkenloh. RaTP: A transaction support protocol for *ra*. Master's thesis, School of Information and Computer Science, Georgia Institute of Technology, 1989.

[39] Kung-Lung Wu and W. Kent Fuchs. Recoverable distributed shared virtual memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.

# Vita

Ajay Mohindra was born in Kota, India on October 17th, 1964. In 1983, he completed his secondary school education from Delhi Public School, New Delhi. Next, he attended the Birla Institute of Technology and Science (BITS) at Pilani, where he earned his M.Sc (Tech) in Computer Science. At BITS, he gained his first exposure to computers on an IBM 1130 – a 16 bit mainframe with punch cards. Not satisfied with his knowledge about computers, he decided to pursue higher studies in the U.S.A. He joined the Georgia Institute of Technology in Atlanta, where he earned M.S. in Information and Computer Science in 1989. Still not sure if he had learned enough about computing, Mr. Mohindra decided to join the Ph.D. program in Computer Science at Georgia Tech. Now, that he has earned his Ph.D., he realizes that it is not possible to learn it all, but is ready to move on to bigger and better things in life.