Implementing and Programming Weakly Consistent Memories

GIT-CC-95-12

A Thesis
Presented to
The Academic Faculty

by

Ranjit John

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

Georgia Institute of Technology
March 1995

Implementing and Programming Weakly Consistent Memories

GIT-CC-95-12

Approved:

Mustaque Ahamad, Chairman

Gil Neiger

Umakishore Ramachandran

Karsten Schwan

Divyakant Agrawal

Date Approved by Chairman _____

To My Parents

Acknowledgments

I would like to begin by expressing my gratitude to my advisor, Mustaque Ahamad. It was his guidance and encouragement which has made this thesis possible. I would also like to thank the members of my dissertation committee, Gil Neiger, Umakishore Ramachandran, Karsten Schwan and Divyakant Agrawal, for their time and helpful suggestions which have vastly improved this thesis.

Several people associated with the Clouds project have contributed with their assistance, ideas and coding: In particular, Sathis Menon who helped me find my bearings in the Clouds project and was always around when I needed help; R. Ananthanarayanan and Ajay Mohindra for the discussions on DSM related issues which to a great extent simplified my implementations.

I would like to thank my cubicle-mates Vibby Gottemukkala, M. Chelliah and Srinivas Doddapaneni for their companionship during the last five years. Also, Rida Bazzi for the late night tea sessions, while we were both writing our theses, where we argued about everything from the format of the thesis to life after school.

The work leading to this thesis was done in part with funding from the National Science Foundation and financial support from the College of Computing. Their assistance is gratefully acknowledged.

Finally, I would like to thank my parents who deserve the credit for encouraging me to continue studying through all these years. This thesis is dedicated to them.

Contents

List of Figures

Summary

A distributed operating system should provide abstractions that make it easy to program applications, provide good performance and allow applications to scale. Operating systems structured around message passing kernels typically ensure good performance and are scalable. On the other hand, Distributed Shared Memory (DSM) systems are much easier to program since the programmer need not be directly concerned with the data placement and data movement between processors.

The ease of programming applications using DSM comes at a cost. DSM systems usually replicate data for efficient implementations. Replication allows shared data to be accessed faster but introduces the problem of maintaining consistency among these copies. Early DSM implementations used variants of multiprocessor cache consistency algorithms that provided sequential consistency. These, however, do not perform very well in distributed systems where the message latencies are much higher.

Maintaining sequential consistency, which requires that all replicas are kept consistent, has been shown to limit performance. Weakly consistent memory models do not require that caches be kept consistent but control the inconsistencies so that most applications still run correctly.

This thesis explores a memory consistency model called causal consistency which provides weaker consistency guarantees than sequential consistency. Causal memory is a weakly consistent memory which requires that a read of a location return a value which is consistent with all causally preceding reads

and writes to that location. Many applications which execute correctly on a sequentially consistent DSM can run correctly without any change in code on a causal DSM.

By programming applications that have a variety of data sharing patterns, it is shown that performance comparable to the message passing implementations of the applications can be achieved on the causal DSM system. The improved performance is due to a significant reduction (70 – 90%) in communication costs compared to the implementation of a sequentially consistent DSM system. These results show that causal memory can meet the consistency and performance requirements of many distributed and parallel applications.

Chapter 1

Introduction

Computers are becoming faster, cheaper and commonplace. It is not unusual today to find workstations on every desk in an organization. When networked together, their cumulative computing power can provide performance at a cost much less than any equivalent centralized system. It is not surprising that much research is being targeted towards utilizing the computing power on the network to solve problems which were previously the exclusive domain of main-frames and super-computers.

A distributed operating system attempts to facilitate the sharing of computing and storage resources provided by these multiple autonomous computing elements. Such systems not only allow *resource sharing* but can also be used to provide *fault tolerance* and *parallelism*. To take advantage of these added benefits, distributed systems should allow programmers to write distributed applications without increasing the complexity of programming.

1.1 Programming Distributed Systems

A distributed system consists of a set of processors connected by a communication network. A process is the computational entity at a processor. There could be multiple processes at a processor. A distributed application, in general, consists of a number of co-operating processes which execute on different

processors. Processors in a distributed system do not share memory and the interactions between them is through messages.

Programming a distributed application requires the specification of the computation at different processors and the interactions between them. Distributed operating systems provide the necessary abstractions to specify the computation and the interactions. Message passing, Remote Procedure Call (RPC) and Distributed Shared Memory (DSM) are a few of the programming abstractions commonly supported by distributed operating systems.

The early distributed systems were built with message passing as the underlying model for distributed computation. Amoeba [47] and the V system [18] are examples of message-based distributed systems. Message-based operating systems support processes that communicate via explicit message send and receive calls. Message-based systems require the programmer to pack data into a buffer which is given to the send call. The receive call receives this buffer and it is the programmer's responsibility to unpack and interpret the data. Such systems typically give good performance since the programmer controls the data placement and data movement between processors. However, programming is more complex since the programmer has to do the packing and unpacking of data and control explicitly the communication between the processors.

Remote Procedure Calls were proposed for remote communication, as it was felt that message passing as an abstraction was too primitive to provide to programmers of distributed applications. Cedar [48], Sun RPC [1] and Argus [42] are among several notable RPC-based systems which have been built. RPC-based systems support a procedural interface for executing remote operations where the actual packing of parameters and communication is hidden

away in library procedures. Similar to RPC's, object oriented systems use Remote Object Invocations (ROI) where communication between processors takes place by invoking a method on a remote object. In addition, ROI's allow the benefits of object orientation, namely encapsulation and inheritance to be exploited by application programmers. Spring [28] and Chorus [51] are examples of systems which provide ROI.

The procedural paradigm supported by RPC-based systems, however, is inadequate for programs which pass complex data structures, use global variables or have dynamic communication patterns. Another way of supporting distributed applications is based on providing a "virtual" shared memory across processors. Systems which support Distributed Shared Memory (DSM) provide a programmer with the illusion of a global shared memory across multiple processors. Kai Li's Ivy [39] system was the first instance of an implementation of DSM. Such systems simplify the programming of distributed applications since the programmer can make use of a uniform mechanism to access both local and remote data. They also allow programs written for shared memory multiprocessors to be ported fairly easily onto a network of workstations.

1.2 Problem Statement

The abstractions provided by a distributed operating system should be efficient to implement, allow the system to scale and make it easy to program applications. Distributed operating systems structured around message passing kernels typically ensure good performance and are scalable since programmers can control the data placement and data movement. On the other hand, DSM systems are much easier to program since the programmer need not be directly concerned with data placement and data movement between processors.

The ease of programming applications using DSM comes at a cost. DSM systems usually cache or replicate data for efficient access. Replication allows shared data to be accessed faster but introduces the problem of maintaining consistency among these copies. Memory has to be kept consistent across processors since multiple processors could simultaneously read or write the same data item. Early DSM implementations used variants of multiprocessor cache consistency algorithms which provided *sequential consistency* [38].

Maintaining sequential consistency has been shown to limit performance. Several researchers have proposed memory consistency conditions which offer weaker guarantees than sequential consistency. Weakening the consistency guarantees provided by a memory system allows simpler and more efficient implementations of the memory system but on the other hand may also increase the complexity of programming applications. This thesis explores one such weakly consistent memory model called causal memory [5] and shows that it can be efficiently implemented and easily programmed. The goal of this thesis is to demonstrate that:

- Causal memory can be efficiently implemented and is a viable architecture for programming distributed applications.
- The execution of the applications on causal memory provides performance comparable to message passing systems for most applications.
- Scalable shared memory systems can be built using the causal memory model, since no form of global synchronization (e.g., messages which need to be sent atomically to all processors in a system) is required.

1.3 Organization of the Thesis

The rest of the thesis is divided into eight chapters. Chapter 2 discusses the related work. It starts by describing the Ivy system and then goes on to describe the newer approaches which have been proposed for building a high performance DSM system.

Chapter 3 motivates the need for weakening the memory consistency requirement and introduces causal memory. Causal memory is formally defined using a history based method by putting constraints on the allowable memory executions. Several examples are shown to illustrate how causal memory differs from other related memory models. Chapter 4 shows how to write programs for causal memory. It introduces a class of programs which would run on causal memory without any change in code. To illustrate, a linear equation solver and a distributed calendar service are programmed on causal memory. Also, examples are shown of programs which fail to execute correctly on causal memory.

Chapter 5 describes a simple protocol for implementing causal memory, which defines the consistency actions that need to be executed on access to each memory location, and argues its correctness. This protocol is used to develop a page based DSM system. Several optimizations are possible when programs are free of data races. For such programs, the consistency operations can be limited to certain synchronization points in a program. We show how the general implementation can be optimized if programs are known to be data-race-free in Chapter 6. We then give an alternative implementation which has the same structure as the previous protocols but optimizes on the earlier implementations.

Chapter 7 is a detailed experimental evaluation of the causal DSM. The

chapter describes the benchmark applications that were used and the performance metrics that were collected. The performance of the applications on three systems is reported; the first supports message passing, the second is a DSM protocol which is sequentially consistent and the third is the causal DSM protocol. Further, a performance comparison with other related work is reported. Chapter 8 concludes the thesis and also suggests possible directions for future work.

Chapter 2

Related Work

Over the last decade, several researchers have described implementations of Distributed Shared Memories across a network of workstations. This chapter surveys the work which has been done in the area of DSM. The first instance of such an implementation was Kai Li's Ivy system [39]. Since then there have been several other implementations which extend the work done by Kai Li. The Ivy implementation is discussed first and then more recent work, which address some of the problems the early systems had, is described.

2.1 Traditional Implementations

The Ivy system supported DSM across a local area network of Apollo workstations. DSM was supported by integrating the virtual memory mechanisms (page-faults and access rights to pages) with the consistency maintenance algorithms. The consistency maintenance algorithms manifested as page-fault handlers and server processes which handled the requests for a page.

Ivy supported sequential consistency, where the value returned by a read operation was the value written by the most recent write operation to the same address. Sequential consistency was maintained by restricting a page to a single writer or multiple readers at any time. This was done using a writer-invalidate-readers protocol where all the copies of a page are invalidated before

letting a processor write to a page.

To implement the protocol, the Ivy system introduced the notion of an *owner* and a *manager*. The owner of a page was the most recent processor which had write access to the page. The owner also maintained information about the processors that had a copy of the page. A manager processor knew the identity of the owner. The manager could be centralized or distributed. A centralized manager maintained a table which tracked the owners of all the shared pages. In the fixed distributed manager scheme, every processor is given a pre-determined subset of shared pages to manage. In the dynamic distributed manager scheme, each processor tries to keep track of the ownership of the page. This is achieved by using a field *prob_owner* with every page. If a processor faults on a page, a request is sent to the *prob_owner*. If that processor is the true owner, it sends the page, otherwise it forwards the request to the processor indicated by its *prob_owner* field. By following the forwarding chain of pointers, the message would eventually get to the current owner.

On a read-fault, the fault handler sends a message to the owner of the page. The owner would respond by sending a copy of the page. If the owner had write access to the page, it would downgrade its access to the page to a read. On a write-fault, a similar message was sent to the owner which would then supply the page and the list of processors with read access to that page. The faulting processor would now invalidate all other copies of the page. This ensured that while a processor was writing to a page, no other processor could read from the page simultaneously.

The Mirage system [23] is an extension of the Ivy system where a processor could retain access to a page for a certain amount of time irrespective of pending requests for the page. This controlled page thrashing, where a page is moved between processors continuously preventing the processors from doing any

useful work.

The approach these early systems took was to adapt a cache consistency protocol developed for shared memory multiprocessors and implement it in software on a network of workstations. However distributed systems differ from multiprocessor systems in several respects and using a multiprocessor cache consistency protocol is inappropriate because of the following differences:

- **Message Latency** — Accessing a page of memory which is not locally cached at a processor requires communication with some other processor. Message latencies on a multiprocessor are of the order of microseconds. In contrast, the message latencies in a distributed system are typically in the range of milliseconds. This leads to much lower processor utilization, since the process has to block while its page request is being serviced.

- **Page as the unit of sharing** — The virtual memory page does not correspond to any logical entity in a program. This lead to the problem of *false sharing* where unrelated data may reside on the same page. Processors accessing data residing on different parts of a page would still cause contention for the page. Also, sending data in units of pages sometimes results in excessive communication when the size of the actual shared data is small compared to the page size.

- **Use of multicasts/broadcasts** — These systems rely on multicasts or broadcasts to invalidate copies of pages. In multiprocessors, broadcasts are done in hardware. In distributed systems, multicasts and broadcasts are costly operations and do not scale well when there are a large number of processors.

- **Network Processor** — Multiprocessors have a separate network processor which handles requests from other processors without interrupting the CPU. In a workstation environment, the CPU has to be interrupted to service the network messages. In such an environment, messages do not just have the latency cost but in addition have the cost of traversing several software layers and context switching overheads.

The key to building a high performance DSM is to reduce the number of messages which need to be sent to maintain consistency of data. The next section discusses more recent work which addresses this problem.

2.2 Newer Approaches to building DSM

More recently, many researchers have looked at the problem of providing a high performance and scalable DSM. Their work can be categorized into the following approaches which are discussed in detail:

- **Data-annotation-based systems** — This approach is based on the fact that data movement protocols can be optimized if the data access patterns of the programs are known. These systems require the application programmer to annotate the data in the program based on their data sharing patterns.

- **Weakly consistent systems** — This approach is motivated by the reasoning that sequential consistency is the bottleneck and weakening it would allow improved performance.

- **Weakly ordered systems** — This approach unifies data transfer with synchronization. These systems require that the synchronization operations are made explicit to the memory system and consistency is only guaranteed at synchronization points.

- **Non-Page-Based systems** — These systems provide consistency at the level of language level objects and require applications to be structured into objects or shared data structures which are made explicit to the underlying system.

2.2.1 Data-Annotation-Based Systems

Accesses to shared data in parallel programs follow some common patterns. Bennett et. al. [13] analyzed the data sharing behavior of common parallel programs and found that the data accesses can be divided into the following categories: *write once*, *private*, *write many*, *result*, *migratory*, *producer/consumer*, *read mostly* and *general read write*.

The cost of maintaining consistency can be reduced if the knowledge of the data sharing behavior in an application is made available to the system. Instead of supporting one protocol, the data-annotation-based systems support multiple protocols each optimized for a particular kind of data sharing behavior. The Munin system [14] is a data-annotation-based system where the application programmer annotates the data in the program based on its access pattern. Accessing a particular data item causes the appropriate protocol for that data to be employed. The Munin system also handled the false sharing problem by allowing multiple writers to access a page and merging the changes at a later time. Allowing multiple writers on a page prevented contention for the page but required some mechanism for merging the changes

made by the multiple writers. Systems that require annotations provide good performance, but the application writer has the onus of providing the protocol specification for the data in the program.

2.2.2 Weakly Consistent Systems

Several researchers [9, 41] have shown that a sequentially consistent memory cannot scale. This is because any implementation of sequentially consistent memory cannot avoid either a read or write operation from incurring the network latency cost of communicating with some other processor. Weakly consistent systems improve the performance of shared memory systems by weakening the consistency guarantees provided by the system. The weaker consistency does not guarantee that the execution of memory operations of all processes is equivalent to some sequential execution of these operations as in a sequentially consistent system. Examples of weakly consistent memory systems include *Pipelined RAM* (PRAM) [41], *processor consistency* [4, 24, 26], and *causal* memory [5].

PRAM - Lipton and Sandberg proposed a memory system, Pipelined RAM [41], which offered weaker consistency guarantees than sequential consistency. In a PRAM memory system, a read operation on a processor would just return the value in the local memory of the processor. A write operation on a processor would write the value to its local memory and then send a message about the write to all other processors. When another processor receives the write message, the write is applied to its local copy of memory. A PRAM memory guarantees that write operations by a processor are seen in the order they were executed by all processors. Writes from different processors can be seen in a different order by different processors.

Processor Consistency - Processor Consistency was proposed by Goodman [26] and a slightly different version was implemented by the DASH multiprocessor [4, 24]. Processor Consistency as defined by Goodman requires that the memory be PRAM consistent and coherent. Memory coherence is the property that all read and write operations to a single location in memory are observed in the same order by all processors.

Causal Memory - Causal memory was proposed by Hutto and Ahamad [30] and requires that all read operations return values which are consistent with all causally related reads and writes of that same location. The causal relation is similar to Lamport's *happens-before* relation [37] used to order events in a distributed system but applied to a shared memory environment. Two operations are causally related if they occur in program order or if one of the operations is a read and the other is a write and the read operation returns the value written by the write operation. The causal relation applies transitively. Write operations which are causally related are seen in the same order by all processors. Concurrent write operations can be seen in different orders by different processors.

To execute applications in such weakly consistent memory systems, either the applications must have data sharing patterns that are not effected by the weaker consistency or the program must explicitly deal with the lack of strong consistency. Programming on these memory systems is different from programming on a system supporting sequential consistency. All represent weakenings where it is no longer required that all processors agree on a global view in which all the memory operations occur in some serial order. Many programs written assuming sequential consistency, however, do run correctly on the weaker memory systems.

Several systems have been proposed which support multiple consistency

models. This is useful where applications or the data within the applications demand different levels of consistency. The Maya [3] and GARF [27] systems provide a DSM abstraction but do not enforce any particular consistency criterion. The Maya system offers a programming model called mixed consistency which supports both causal memory and pipelined RAM and provides explicit synchronization operations. The memory and synchronization operations can be tailored for efficient implementations of applications. The GARF system provides a library of consistency criteria and applications can bind to the appropriate library.

2.2.3 Weakly Ordered Systems

One way to reduce communication between processors in shared memory system is to guarantee consistency only at certain points in the execution of a process. This approach was first outlined by Dubois et al. [21]. They observed that parallel programs define their own consistency requirements through the means of synchronization operations. They define a *weakly ordered* system, where synchronization operations are made explicit to the memory system, and consistency maintenance is done only at synchronization points. One of the earliest implementations where the data transfer is linked with the synchronization operations is reported in [50]. The DASH multiprocessor [24] is a weakly ordered system which implements a memory model called *release consistency*. Release consistency refined the weakly ordered approach by dividing synchronization operations into two types: acquire and release. Release consistency allows remote memory accesses to be propagated asynchronously, as long as they complete before a release operation (e.g., an *unlock* operation on a synchronization variable).

Weakly ordered systems require a certain behavior from application programs to ensure their correct execution. Such systems guarantee correct execution only for programs which are data-race-free [2] or properly labeled [24]. In other words, when these programs are executed on a sequentially consistent memory system, conflicting accesses to the same shared location (two writes or a read and write to the same location conflict) will always be separated by accesses to synchronization variables in the equivalent serial order. Since consistency is guaranteed at synchronization points, data-race-free programs can be written on weakly ordered memory systems assuming a sequentially consistent system.

Hardware implementations of weakly ordered systems use optimizations such as instruction reordering, pipelining and write buffering but still maintain a coherent cache. In contrast, software implementations sacrifice coherence by delaying consistency related operations to certain specific points in the program execution. The Munin system [14] implements release consistency in software, by delaying the propagation of the changes made inside a critical section till the lock is released. More recently *lazy release consistency* [33] and *entry consistency* [15] memory models have been proposed which use synchronization information to further reduce communication.

Lazy Release Consistency (LRC) - Release consistency requires that all processors are made consistent at the release of a synchronization variable. This is done in Munin by propagating all modified data to all the processors with a cached copy of the data. LRC is an optimization where the propagation of modifications is postponed until the time of the acquire. At this time, the acquiring processor determines which modifications it needs to see according to the definition of release consistency. LRC is implemented in the TreadMarks [34] system at Rice University. No consistency actions are done at a

release. At an acquire, the processor doing the acquire has to make sure that its cache is updated correctly. LRC based systems send far fewer messages as compared to release consistent systems since modified data is propagated only to the next processor that acquires the synchronization variable.

Entry Consistency (EC) - Entry consistent systems require the explicit association of data with synchronization variables. This allows further optimizations since only the data associated with a synchronization variable needs to be made consistent. The Midway system [15] supports entry consistency. A special compiler keeps track of the data which has been modified inside of a critical section. The time of modification is kept track by attaching a Lamport's clock [37] with each shared data item. On an acquire, the acquiring processor sends the Lamport time of the data associated with the synchronization variable to the last releaser. The releaser sends all the modifications to data associated with the variable that were made since the acquirer's Lamport time. Entry consistent systems typically give performance very close to message passing implementations. However, programming applications is more involved as programmers have to explicitly associate data with synchronization variables and in some cases may have to include more synchronization operations into their programs.

2.2.4 Non-Page-Based Systems

The Midway implementation is a non-page-based approach where the programmer is responsible for providing the information about the units of sharing. Applications on Midway are C or ML programs, extended with the data associations. Many programming languages allow related data to be encapsulated into objects, modules or other similar structures. In non-page-based systems

data is communicated in units of language objects instead of pages. If a processor accesses some part of an object, the whole object is moved to that processor as it is likely that the rest of the object would be accessed too. Orca [11] is another example of a shared memory system that maintains consistency at the level of language level objects.

2.3 Why Causal Memory ?

A large number of parallel applications exist which have been developed for shared-memory multiprocessors. Shared-memory multiprocessor vendors have typically relied on proprietary hardware which has not kept pace with the commodity processor technologies. The recent demise of multiprocessor manufacturers like Thinking Machines and Kendall Square Research further endorse this argument. With current processor speeds and increasing network bandwidths, networks of workstations provide a competitive price/performance alternative to shared-memory multiprocessors. DSM systems allow the parallel programs developed for shared-memory multiprocessors to be ported to a network of workstations without too much effort.

Using data-annotation-based systems, however, requires determining the data sharing behavior of these applications which would require a detailed understanding of the programs. Weakly ordered systems require a certain contract by the applications. Running applications which do not confirm to the contract can result in unexplained results. Non-page-based systems would require that the applications be rewritten as language level objects which would be a non-trivial task. Causal memory is interesting since, as we show later, it allows most applications to port without change in the code. This is not true for consistency conditions which are weaker than causal consistency.

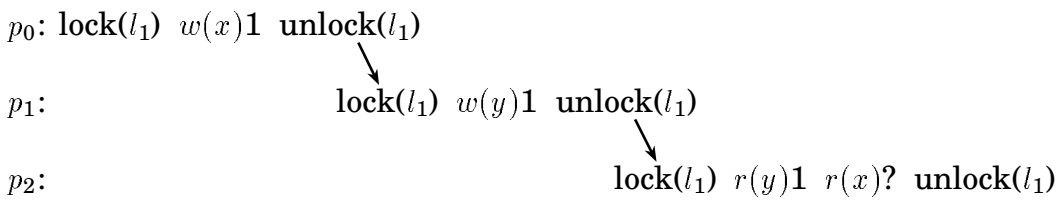


Figure 1: A PRAM execution

For example, consider a PRAM execution shown in Figure 1. When processor p_2 reads the location y, it would read the value written by p_1 as the write happened before p_1 did the unlock. However, when p_2 reads the location x, it is not guaranteed that it would see the write by p_0, leading to possible incorrect executions. This implies that either programs have to be rewritten to avoid such anomalies, or the synchronization operations have to be extended to enforce stronger semantics.

Another area where the causal semantics seem appropriate are cooperative distributed applications where users asynchronously interact to share data. Such applications include conferencing systems, distributed calendars etc. The sharing requirements of these applications usually do not require that modifications to shared data are seen at once or in the same order by all users. Causal consistency also seems appropriate for applications like mail agents and news readers. Consider the scenario where a user A sends a mail message to users B and C. B now responds to this message to A with a copy to C. We would ideally like C to receive A's mail before she gets the response from B. The response by B would not make any sense to C unless she has received the original mail. Current systems rarely support this level of consistency because of the overhead involved in guaranteeing this message delivery order for large systems. The ISIS [16] system provides a group communication

primitive called CBCAST which guarantees such causal delivery of messages within the group. The GARF system uses the CBCAST protocol to ensure causal consistency for shared objects. However, for page-based DSM systems, update-based protocols which rely on multicasts or broadcasts, do not scale well.

The next chapter defines causal memory more precisely. Later chapters show that programming is not complicated by the weaker consistency of causal memory and that it can be efficiently implemented.

Chapter 3

Causal Memory

There is at present no consensus on the best way of defining formally memory consistency models. Different researchers have used different approaches and notations which makes it difficult to compare the various memory models which have been defined in literature. Goodman [26] and Gharachorloo et. al. [24] used an operational characterization which lead to many misinterpretations [4]. Gibbons et. al. [25] used an automata based characterization for describing release consistency. Cekelov et. al. [55] used an axiomatic approach to formally describe Total Store Order (TSO) and the Partial Store Order (PSO). Both the automata based and axiomatic approaches have the formal rigidity but fail to provide an easy way of comparing different memory models. Initially, causal memory was defined [6] by characterizing the possible values that could be returned when a read operation is executed by a processor. A more general framework described by Ahamad et. al. [5] is used here, as it allows us to define and easily relate causal memory to a range of memory models that have been proposed.

3.1 The Model

The model is motivated by the ones used by Misra [45] and by Herlihy and Wing [29]. A system is defined to be a finite set of *processors* that interact via a

shared memory consisting of a finite set of *locations*. Processors execute *read* and *write* operations. Each such operation acts on a named location and has an associated value. For example, a write operation executed by processor p, denoted by $w_p(x)v$, stores the value v in location x; a similarly denoted read operation, $r_p(x)v$, reports that v is stored in location x. The execution of a processor is defined by a *processor execution history*, which is a sequence of read and write operations. The execution history of processor p, denoted by H_p, is the sequence $o_{p,1}$, $o_{p,2}$, ..., $o_{p,i}$, ... where $o_{p,i}$ is the i^{th} operation issued by processor p. A *system execution history* is a collection of processor execution histories, one for each processor in the system. Thus, a system execution history $H = \{H_p \mid p \in \mathcal{P}\}$ where $\mathcal{P}$ is the set of processors in the system.

It is possible to establish orderings on the operations that appear in a system execution history H. The following orders are used in defining causal memory.

- **Program order:** For operations $o_{p,i}$ and $o_{p,j}$, we say $o_{p,i} \stackrel{po}{\rightarrow} o_{p,j}$ when $o_{p,i}$ precedes $o_{p,j}$ in the program, i.e., $i < j$. In this case, we say $o_{p,i}$ is ordered before $o_{p,j}$ by the program order. This defines program order to be total; it orders all operations of a given processor[1].

- **Writes-before order:** If o_i (we drop the first subscript where it is unimportant) is a write to some location, and o_j is a read by a processor (which may be different from the writer), and o_j returns the value written by o_i, then $o_i \stackrel{wb}{\rightarrow} o_j$. We call this the writes-before order and it captures the natural requirement that if a read operation returns the value written by a certain write operation, then the write operation must be ordered

[1]As defined here, program order totally orders the operations of each processor. In other memory models, the ordering between the local operations of a processor could be partial [24].

before the read.

- **Causal order:** The *happens-before* relation defined by Lamport [37] can be adapted to a shared memory system; this captures the causal relationship between the read and write operations. For any two operations o_1 and o_2 in H, $o_1 \stackrel{co}{\rightarrow} o_2$ if
  - $o_1 \stackrel{po}{\rightarrow} o_2$ or
  - $o_1 \stackrel{wb}{\rightarrow} o_2$ or
  - for some operation o', $o_1 \stackrel{co}{\rightarrow} o'$ and $o' \stackrel{co}{\rightarrow} o_2$.

Ideally, a processor should be able to assume that a shared memory system executes a set of read and write operations one at a time, in a sequential order, and that the value returned by a read of location x was stored by the most recent write to x preceding the read in the sequential order. We call such an ordered sequence of memory operations the *view* of the processor. A memory model can be characterized by the types of views that result when processors execute with that type of memory system. For example, if the memory is sequentially consistent, all processors have a single view. Furthermore, the order in which the operations appear in the view is consistent with program order. Weakly consistent memories can be defined by allowing each processor to develop a different view. Views can be different because they may differ in the set of operations included in them, or in the order in which common operations appear in the views. By choosing the set of operations to be included in a processor view and the orders that must be maintained between them, implementation independent definitions of several memory systems can be developed [4, 36].

3.2 Synchronization Operations

In parallel applications, communication between processors is not restricted to the messages generated by sharing of data. Processors also communicate to achieve synchronization and also when one process forks another process on some other processor. Synchronization is used to ensure that a sequence of memory operations (e.g., a critical section) are executed atomically. When a processor p acquires a lock released by another processor q, memory operations of q that precede the unlock operation on the lock, are ordered before memory operations of p that follow the operation in which the lock is acquired. In addition, parallel and distributed programs achieve parallelism by *forking* computation onto different processors. Domain decomposition is a commonly used method for developing parallel programs, where a "parent" process initializes the domains and then creates "child" processes on different processors, each working on a different partition. The semantics of fork assumes that the initializations done by the parent will be visible to the children. Similarly, at *fork-joins*, the programmer assumes that the changes made by the children will be visible to the parent. The synchronization and fork operations are typically implemented outside the memory system on distributed systems. However, these operations would create additional orderings between the memory operations.

The model can be extended to include the orderings induced by the synchronization and forking operations. Causal orderings would now arise between memory operations due to synchronization acquires and releases and also between the forking parent process and the forked child processes. We can now define the following orders induced between memory operations o_i and o_j by lock, barrier and fork-join operations (similar orders can be defined for

other synchronization constructs which are implemented outside the memory system).

- **Lock order:** We say $o_i \stackrel{lo}{\rightarrow} o_j$, when o_i and o_j are two memory operations such that o_i immediately precedes an unlock operation and o_j immediately follows the corresponding lock acquire. This order captures the orderings induced by read-write locks and semaphores.

- **Barrier order:** We say $o_i \stackrel{bo}{\rightarrow} o_{k,j}, k = 1..n$, when o_i immediately precedes a n-process barrier operation and $o_{k,j}$ immediately follows the matching barrier operation in process p_k.

- **Fork order:** We say $o_i \stackrel{fo}{\rightarrow} o_j$, when o_i immediately precedes a fork (or join) operation and o_j is the first memory operation executed by the forked child process (or after the join).

Let $o_1 \stackrel{so}{\rightarrow} o_2$ be the order induced by the synchronization operations. Two memory operations o_1 and o_2 are related by $\stackrel{so}{\rightarrow}$ if

  - $o_1 \stackrel{po}{\rightarrow} o_2$ or
  - $o_1 \stackrel{lo}{\rightarrow} o_2$ or
  - $o_1 \stackrel{bo}{\rightarrow} o_2$ or
  - $o_1 \stackrel{fo}{\rightarrow} o_2$ or
  - there exists an o' such that $o_1 \stackrel{so}{\rightarrow} o'$ and $o' \stackrel{so}{\rightarrow} o_2$.

The causal order definition can now be extended to include these orderings. Two operations o_1 and o_2 in H are ordered by the *extended causal order*, $o_1 \stackrel{eco}{\rightarrow} o_2$,

if $o_1 \stackrel{co}{\rightarrow} o_2$ or $o_1 \stackrel{so}{\rightarrow} o_2$ or for some operation o', $o_1 \stackrel{eco}{\rightarrow} o'$ and $o' \stackrel{eco}{\rightarrow} o_2$. Causal memory can now be defined using this extended causal ordering [2].

3.3 Defining Causal Memory

Causal memory requires that values returned by read operations respect the extended causal order between memory operations. Since the effects of concurrent operations (operations not related by the causal order) can be observed in different order at different processors, causal memory allows each processor to develop a different view of shared memory.

Causal orderings between the operations of processor p and the operations of other processors are established when p reads values written by other processors. Since write operations update the state of shared memory and p's reads can return values written by other processors, a processor's view in causal memory needs to include all write operations. The causal ordering established by a read operation of processor p can propagate to another processor q but that happens only as a result of p writing a value that q reads (or through the synchronization and fork operations). Thus, the values that can be read by q are affected by only the write operations of other processors. As a result, p does not become aware of the read operations of other processors directly. This observation, coupled with the fact that orderings of operations in views must respect causality, leads to the following definition of causal memory. For system execution history H, H_{p+w} refers to the history resulting after read operations of all processors other than p are deleted from their processor execution histories.

[2]Recently, the Maya system has also proposed a causal memory system that includes orderings induced by synchronization operations [3].

Causal Memory A history H is causal if, for each processor p, there is a *view* S_p, such that, for all operations o_1 and o_2 in H_{p+w}, if $o_1 \stackrel{ecq}{\rightarrow} o_2$ then o_1 precedes o_2 in S_p.

A memory system implements causal memory if all histories allowed by it are causal. The example execution in Figure 2(a) is possible with causal memory because the corresponding views (shown in Figure 2(b)) exist for each processor, as required by the definition of causal memory. We assume that all variables have an initial value of zero. This history is not sequentially consistent since both processors would not "agree" on a common sequence of operations (there is no single view that includes all operations and respects the program order established by each processor).

$p_0:$	$w(x)1\ w(y)2\ r(z)0$	$S_{p_0}:$	$w_0(x)1\ w_0(y)2\ r_0(z)0\ w_1(z)1$
$p_1:$	$w(z)1\ r(x)0\ r(y)2\ r(x)1$	$S_{p_1}:$	$w_1(z)1\ r_1(x)0\ w_0(x)1\ w_0(y)2\ r_1(y)2\ r_1(x)1$
(a) Two Processor Execution		(b) Causal Views	

Figure 2: An execution which is causal but not sequentially consistent

Causal memory differs from other weakly consistent memories. Figure 3 shows an execution that is permitted by causal memory which is not allowed by processor consistent memory as implemented by the DASH multiprocessor [24]. Causal memory allows concurrent writes to a memory location to be read in any order by different processors. The DASH implementation of processor consistency requires memory to be coherent, that is, writes to a single memory location are serialized and observed in the same order by all processors. For this reason, the execution would not be permitted by processor consistent memory. Pipelined RAM [41] is strictly weaker than causal memory

because it only requires that processors order two write operations in the same order in their views only if the writes are executed by the same processor.

$$
\begin{array}{ll}
p_0: & w(x)1 \\
p_1: & w(x)2 \\
p_2: & r(x)1\ r(x)2 \\
p_3: & r(x)2\ r(x)1
\end{array}
$$

Figure 3: Causal but not processor consistent execution

3.4 Concluding Remarks

This chapter developed a history based approach for formally defining causal memory. Causal ordering was defined as the transitive closure of the program order and the writes-before order. Synchronization and fork operations, which are normally implemented outside the memory system, cause additional orderings on the memory operations. The causal order was extended to include the effects of these non-memory operations. A memory was defined to be causally consistent if the values returned by read operations were consistent with the extended causal order. Causal memory is weaker than sequential consistency as it allows multiple processors to have different views of the memory operations. The following chapters show that causal memory can be easily programmed and that efficient implementations of causal memory are possible.

Chapter 4

Programming on Causal Memory

The weaker consistency of causal memory can provide more efficient implementations but it should not have an adverse impact on the ease of programming with DSM's. This chapter shows that programming with causal memory is not more complex than programming with sequentially consistent memory for a wide range of applications. These include data-race-free programs that employ synchronization operations to ensure atomicity of critical section code as well as applications which have data races but where causality introduces enough orderings between memory operations such that there are no concurrent writes. Towards the end, programs with data access patterns which do not execute correctly on causal memory are discussed.

4.1 Data-Race-Free Programs

In parallel and distributed programs where processes share data, access to data is controlled by synchronization operations. When programs use "sufficient" synchronization to control access to shared data, consistency maintenance can be limited to the synchronization points [52]. Adve and Hill [2] formalized this and introduced the notion of data-race-free programs. In these programs, conflicting accesses to a shared location by different processors (two accesses to a memory location conflict when they are not both reads) are always separated

by one or more synchronization operations. More precisely, all conflicting memory operations must be ordered by a *happens-before* relation, $\stackrel{hb}{\rightarrow}$, that is derived from program order and the order in which synchronization operations are executed. Data-race-free programs can be developed on causal memory the same way as on sequentially consistent memory. Thus, programming of this class of applications is not made more complex when we use causal instead of sequentially consistent memory. This is formally proved in [7]. Intuitively, it follows from the fact that writes to a location must all be ordered by the happens before relation, $\stackrel{hb}{\rightarrow}$, as there can be no conflicting writes. The extended causal order, $\stackrel{eco}{\rightarrow}$, includes all orderings induced by $\stackrel{hb}{\rightarrow}$, and hence it follows that writes to a single location must appear in the same order in all processor views. Since this holds for each location and causality is respected across memory operations, it can be shown that an execution of a program on causal memory is also possible on sequentially consistent memory. Thus, if a data-race-free program executes correctly on sequentially consistent memory, its execution on causal memory is also correct. We illustrate this next by showing that a linear equation solver, programmed assuming sequentially consistent memory, executes correctly even on causally consistent memory.

4.1.1 Linear Equation Solver

Very large systems of linear equations often arise in many scientific and engineering applications. Iterative methods such as the Gauss-Seidel algorithm are particularly amenable to parallelization. Iterative methods work in a sequence of phases. Each phase uses the results of the previous phase to generate a new, more accurate solution. Computation terminates when the result of the last phase passes a convergence test or has been computed to sufficient

accuracy.

Consider a parallel iterative algorithm that solves $Ax = b$, where A is a $n \times n$ matrix and x and b are vectors of size n. $a_{i,j}$ refers to the element of A in row i and column j and b_i (similarly x_i) refers to the ith element of vector b (x). x_i^{k+1} represents the value of the ith component of x in phase $k+1$. x_i^{k+1} is computed using the following equation: $x_i^{k+1} = \left(b_i - \sum_{j=1}^{i-1} a_{i,j}\, x_j^k - \sum_{j=i+1}^{n} a_{i,j}\, x_j^k\right) \Big/ a_{i,i}$. A and b are inputs and remain constant but computing x_i^{k+1} requires access to all x_j^k $(i \neq j)$ from the previous iteration.

A parallel implementation of the iterative method partitions the tasks of computing the new x_i's among available processes. At the beginning of each iteration, a process reads the results of the previous iteration from the shared global vector x and computes and stores the new x_i in a private local variable t_i. Since processes may proceed at different rates, a *synchronous* implementation requires processes to synchronize twice per iteration[1]. Before beginning phase $k+1$, each process waits until all results from the previous phase have been copied to the global vector x. Then, before copying the newly computed value t_i to global x_i, each process waits until all other processes have completed computation of their new t_i (allowing the old x_i to be overwritten). Barriers are used to achieve this synchronization. Figure 4 shows the code for the linear equation solver.

There are *N* worker processes. The vector *x* is partitioned among the processes, such that, each process computes approximately an equal number of elements. As described previously, each worker process begins by assuming access to the previous (or initial) values in the global vector *x* and proceeds to compute and store the new value in the local variable t_i. The processes

[1]It is possible to synchronize just once per iteration by having the processes read alternately from x_i and t_i, but we do not consider that here.

```
object linear_solver {
  float A[n][n];
  float b[n];
  float x[n];

  main()
  {
  initialize A & b;
  create N worker processes;
  }

  worker(int from, int to, int n)
  {
  int i;
  while (¬ converged())
    for (i = from; i < to; i++)
      t_i := ( b_i − Σ_{j=1}^{i−1} a_{i,j} x_j − Σ_{j=i+1}^{n} a_{i,j} x_j ) / a_{i,i};
    barrier();
    for (i = from; i < to; i++)
      x_i := t_i;
    barrier();
  }
}
```

Figure 4: Synchronous iterative linear equation solver

then synchronize at a barrier to ensure that all the processes would have computed the t_i's. After synchronizing, the processes copy the new value t_i to the global x_i and goes through a similar barrier before resuming the next phase of computation or terminating.

The code in Figure 4 correctly solves the system $Ax = b$ on both sequentially consistent memory and causal memory. Consider a read of some x_j by p_i in phase k, where x_j is in the partition of p_j. We need to show that the only value that may be correctly returned is the write to x_j by p_j in phase $k-1$. In other words, values from all earlier iterations are overwritten and causal memory behaves like sequentially consistent memory in this instance. Following our notation for x_i^k, we use superscripts on read and write operations to denote the phase in which the operation was performed. For example, $w_i^k(x_i)v$ denotes a write of x_i by p_i in phase k. Consider the causal relations established between p_j's write of x_j in phase $k-2$ and p_i's read of x_j in phase k by the interactions between the two processes (i and j).

$$
\begin{array}{ll}
(1) & w_j^{k-2}(x_j)v \overset{po^*}{\rightarrow} w_j^{k-1}(x_j)v' \\
(2) & w_j^{k-1}(x_j)v \overset{bo}{\rightarrow} o_i \\
(3) & o_i \overset{po^*}{\rightarrow} r_i^k(x_j)v''
\end{array}
$$

Above, (1) holds because the two writes happen in consecutive iterations on p_j. (2) relates the write to x_j in the $(k-1)$'th iteration to the first memory operation after the barrier on p_i. (3) holds because after p_i has checked for convergence, it reads the value of x_j in the next iteration (assuming that the convergence test fails). Taken together (1)–(3) imply that $w_j^{k-2}(x_j)v \overset{co}{\rightarrow} w_j^{k-1}(x_j)v' \overset{co}{\rightarrow} r_i^k(x_j)v''$. This ordering must be respected by the individual views of the processors and therefore the read operation on x_j in the k'th iteration must return the value written in the $(k-1)$'th iteration (i.e, $v' = v''$). Since we assumed an arbitrary

i and j, this argument shows that all reads of x in the computation return the value computed in the previous iteration, the same value returned when the computation is executed on sequentially consistent memory.

4.2 Non Data-Race-Free Programs

Several programs execute correctly on causal memory even when they have data races. These programs include applications where causality introduces enough orderings between memory operations such that there are no concurrent writes to a single location. The effect of such a data race is that a process could continue reading an old value of a data item residing in its cache. Some applications converge to a solution even with older values and some interactive applications can tolerate old values.

4.2.1 Asynchronous Linear Equation Solver

It is possible to eliminate the synchronization entirely by using an *asynchronous* algorithm. The asynchronous algorithm is based on the observation that the iterative procedure may be used to *improve* close but inexact results. If we eliminate synchronization, processes may read, in a single phase, x_i's computed in several previous phases. Fortunately, due to the error-correcting properties of the iterative method, the algorithm will still converge on the correct solution even when the x_i used to compute t_i^k are not all from the same iteration. All that is necessary is that processes never observe an x_i older than the values already read. This is precisely the property preserved by causal memory. Reads respect the order of causally related writes. An asynchronous iterative algorithm given in [12] is shown in Figure 5. The code works cor-

while ($\neg$*converged*())
$x_i := \left(b_i - \sum_{j=1}^{i-1} a_{i,j}\, x_j - \sum_{j=i+1}^{n} a_{i,j}\, x_j\right) \Big/ a_{i,i}$

Figure 5: Asynchronous iterative linear solver on causal memory

rectly on both sequentially consistent and causal memory. However, unlike in sequentially consistent memory, each worker may execute different number of iterations as the new values propagate at different times and as a result the convergence test would not return true at the same time for all the processes.

4.2.2 Distributed Calendar

Consider the problem of implementing a distributed calendar service. Every user in the system maintains an appointment calendar. The calendar is used to record appointments and daily schedules. In addition, we allow a user to browse through other people's calendars. We will not consider privacy of information issues; if it is a problem then data could be marked as private or public. Looking into someone else's calendar would only show the public information. Occasionally there could be a need to schedule a common meeting time for some group of people. We would also like a feature where a user is alerted if his/her calendar has been updated when someone schedules a meeting. The code for such a service is shown in Figure 6.

Access to a user's calendar is protected by a read-write lock. This is required since we allow users to update and browse any other person's calendar. *browse_calendar* allows a user to look at anyone's appointment calendar by specifying the appropriate user-id. To bring up one's own calendar the user has to specify his/her user-id. The function *update_calendar* permits one to add

```
object calendar {
   calendar_type calendar[MAX_USERS];
   lock mutex[MAX_USERS];
   boolean changed[MAX_USERS];

browse_calendar(user_id i)
{
   acquire_read_lock(mutex[i]);
   display calendar;
   release_lock(mutex[i]);
}

update_calendar()
{
   acquire_write_lock(mutex[my_id]);
   update entry in calendar;
   release_lock(mutex[my_id]);
}

set_up_meeting(group_id group)
{
   ∀ members i in group
      acquire_write_lock(mutex[i]);
      insert entry into calendar of each member of group;
      ∀ i changed[i] = TRUE;
      ∀ i release_lock(mutex[i]);
}

listener_daemon()
{
   while(1) {
      if (changed[my_id] == TRUE) {
         beep();
         changed[my_id] = FALSE;
      }
      sleep(NUM_SECS);
   }
}

} // end of calendar object.
```

Figure 6: Shared memory distributed calendar

and delete appointments from one's own calendar. The function *set_up_meeting* is invoked when a user needs to schedule a group meeting. The locks are acquired in some pre-defined order to prevent deadlocks. The *listener_daemon* is responsible for notifying a user that his calendar has been modified. It periodically reads a flag to determine if any change has been done to the calendar.

The data race happens since we allow the flag *changed* to be read without locking it first. While a process is reading the flag, another process could be concurrently updating the value of the flag, leading to the data race.

A user will see his/her updates to the calendar immediately. If someone else schedules a meeting, we need to show that the daemon would alert us of a change and if we bring up our calendar, it would have a record of the scheduled meeting. Since the daemon, keeps reading the flag *changed*, it would eventually see the value *true* and alert the user. The user would then use *browse_calendar* to read the calendar. Before displaying the calendar, a read lock has to be obtained. Since all accesses to the calendar are guarded by a read-write lock, there can be no concurrent updates to the calendar and hence all processes must see the order of updates in the same order. The lock order would guarantee that the user sees the last update of his/her calendar.

Programming on causal memory is remarkably similar to programming on sequentially consistent memory, identical in fact, for the synchronous and asynchronous linear equation solver and the distributed calendar service.

```
var flag: array [0..1] of boolean;
    turn: 0..1;

repeat
    flag[i] = true;
    turn = j;
    while (flag[j] and turn == j);

        critical section

    flag[i] = false;

        remainder section

until false;
```

Figure 7: Peterson's 2-process mutual exclusion

4.3 Programs with Incorrect Executions

There exist several programs which rely on a memory system being sequentially consistent for correct execution. Such programs include software solutions to the mutual exclusion problem. For example, the Peterson's two-process mutual exclusion algorithm [54] and Lamport's bakery algorithm [54] will not execute correctly on causal memory. We show here how Peterson's algorithm fails to provide mutual exclusion on causal memory.

Figure 7 shows the algorithm for achieving two-process mutual exclusion. To show that the above solution is incorrect on causal memory, consider the following execution history shown in Figure 8(a). This history is possible on a causally consistent memory as the two processes could have observed views as shown in the Figure 8(b). Since each process could see the value

$$p_0: \quad w(flag[0])true\ w(turn)1\ r(flag[1])false$$
$$p_1: \quad w(flag[1])true\ w(turn)0\ r(flag[0])false$$

(a) Execution History

$$S_{p_0}: \quad w_0(flag[0])true\ w_0(turn)1\ r_0(flag[1])false\ w_1(flag[1])true\ w_1(turn)0$$
$$S_{p_1}: \quad w_1(flag[1])true\ w_1(turn)0\ r_1(flag[0])false\ w_0(flag[0])true\ w_0(turn)1$$

(b) Processor Views

Figure 8: An execution history and processor views for Peterson's algorithm

of the *flag* variable as *false*, both would enter the critical section and mutual exclusion would not be preserved. A similar argument can be made for the Bakery algorithm that implements a solution to the n-process mutual exclusion problem.

4.4 Concluding Remarks

In this chapter, we investigated how to program applications on a memory system supporting causal consistency. Programming on a causally consistent memory system seems remarkably similar to programming assuming a sequentially consistent memory system. In fact, if a program is free of data races, it would run without any change in code on causal memory.

The synchronous linear equation solver is a data-race-free program. The asynchronous linear equation and the distributed calendar are both examples of programs with data races. They execute correctly as the former converges to a solution even with older values and the latter due to the interactive nature of the application can tolerate old values of the boolean variable *changed*.

There are, however, programs which do not execute correctly. The Peterson's algorithm and the Bakery algorithm require sequentially consistent memory for correct execution.

The next chapter discusses how to actually build a system supporting causal consistency. We will also analyze and compare the performance of the applications discussed here, in terms of the number of messages required for maintaining consistency of data, on a system supporting sequentially consistent memory and a causally consistent memory system.

Chapter 5

Implementing Causal Memory

This chapter investigates the problem of implementing a causal DSM. We first present a simple protocol to support a causally consistent shared memory. We initially restrict the implementation to programs in which processes only communicate via read and write operations to shared locations in memory. This allows us to study the issues involved in building a weakly consistent DSM. We extend this implementation to a page based DSM and introduce the actions which need to be performed at non-memory operations, namely the synchronization and forking operations. We then analyze the performance advantage, in terms of number of messages, of causal memory over sequentially consistent memory for some of the applications described in the last chapter.

5.1 A Simple Owner Based Protocol

We assume a system where the shared causal memory is partitioned among the processors. The locations assigned to a processor are *owned* by that processor. Each processor p_i has a local memory M_i indexed by location names (addresses). The locations owned by a processor are always stored in the local memory of that processor so that, if $i = \mathit{owner}(x)$ then $M_i[x]$ always contains a value of x. The remaining locations in a processor's local memory are used to *cache* copies of locations owned by other processors. The distinguished value $\perp$ is

p_0: w(x)1 w(y)1
p_1: r(y)1 w(x)2 w(y)2
p_2: r(x)1 r(y)2

Figure 9: Example to show causal over-writing of data

used to indicate that a processor does *not* possess a cached copy of a location. If $M_i[x] = \perp$ then x is *invalid* (not cached) at p_i. Also, assigning $\perp$ to a location ($M_i[x] := \perp$) *invalidates* that location at p_i. The locations owned by a processor can never be invalidated by that processor. C_i is the set of locations currently cached by processor p_i, that is, locations that are not owned by p_i and that are not invalid in M_i.

Causal orderings between memory operations at different processors are established when a processor reads a value written by another. Thus, each time a processor caches a data value written at another processor, it must ensure that the newly cached value is causally consistent with the data values already existing in its memory. In particular, any new causal orderings that get established by the reading of the newly cached data value must not cause the existing data to be over-written in the causal sense. In the example shown in Figure 9, when p_2 caches the value of y written by p_1 and reads it, the cached value of x at p_2 has become over-written (the write to x by p_0 causally precedes the write to x by p_1), and the value 1 must not be returned by a future read to x by p_2. The implementation maintains correctness by invalidating cached copies that might violate causal memory correctness if read. This is done by detecting cached data which could be causally over-written each time a new value is introduced into the cache. Causally over-written data is detected using a mechanism called vector timestamps which is explained below.

5.1.1 Vector Timestamps

Causal memory correctness is intimately related to causality. A simple *vector clock protocol* [44] may be used to capture precisely the evolving partial ordering of events in a distributed system, and thereby, causality. A vector clock represents logical time as a integer vector of size *n*, where *n* is the number of processors. Each processor in the system maintains a separate vector clock VT_i that may be *incremented*, *updated*, and *compared*. The three operations are described below.

- *increment*: *inc(VT)*, when executed by processor p_i, adds one to the i th component of *VT* and returns the incremented vector time.
- *maximum*: *max(VT′, VT″)* returns the component-wise maximum of the vector timestamps VT' and VT''. We will also refer to this operation as the clock update operation.
- *comparison*: $VT' < VT''$ returns true if, for all i, $VT'[i] \leq VT''[i]$, and there is at least one component of VT' that is less than the corresponding component of VT''.

Vector clocks have been used extensively in distributed system to do distributed debugging [22], achieving causal ordering of messages [53], building highly available distributed services [43], and checkpointing for optimistic recovery [32] among other applications.

5.1.2 The Basic Algorithm

Attempts to read a location not locally owned nor cached (a *read miss*) generate a message to the owner requesting a current copy. The requesting processor

blocks until a reply is received, caches the copy received, and then completes the read. Similarly, writes to a location not owned by the writer require a *write-through* to the owner. The writing processor blocks until the owner's value is updated and a reply is received from the owner. Essentially, all such writes are completed co-operatively between the writing processor and the owner. The algorithm is shown in Figure 10.

On every write attempt, the writing processor increments its vector clock and associates the resulting vector time, called a *writestamp*, with the value written. Thus each location x in a processor's local memory M_i contains a *value-writestamp* pair $M_i[x] = (v, VT)$. All writes by a processor are totally ordered by these writestamps and all writestamps ever generated in the system are unique. Two writes not ordered by their associated timestamps are *concurrent*. When a processor introduces a value into its cache, it updates its vector clock with the writestamp associated with the value being introduced.

Identifying precisely the values that may violate correctness when a new value is brought into the cache requires more overhead than we are willing to pay in our simple owner protocol. Instead, each time a "new" value is introduced into local memory by a read or write, we invalidate all cached values that could potentially lead to a violation of causality; that is, all cached values that are "older" (via the causality relation) than the value being introduced. This approach may invalidate more cached values than strictly necessary but it requires little book-keeping overhead and ensures correctness.

Five procedures are shown in the Figure. The first two are executed whenever p_i performs a read or write. The next two are executed by p_i on receipt of READ and WRITE requests for locations owned by p_i. The final operation, $discard$, may be performed by p_i under a variety of circumstances described later. Notice the handling of writestamps of locations not locally

$r_i(x)v$ ::
  if $M_i[x] = \perp$
    send $[\mathit{READ}, x]$ **to** $\mathit{owner}(x)$
    receive $[\mathit{R\_REPLY}, x, v', \mathit{VT}']$ **from** $\mathit{owner}(x)$
    $\mathit{VT}_i := \mathit{update}(\mathit{VT}_i, \mathit{VT}')$
    $M_i[x] := (v', \mathit{VT}')$
    $\forall y \in C_i : M_i[y].\mathit{VT} < \mathit{VT}'$
      $M_i[y] := \perp$
  $v := M_i[x].\mathit{value}$

$w_i(x)v$ ::
  $VT_i := \mathit{increment}(\mathit{VT}_i)$
  if $\mathit{owner}(x) \neq i$
    send $[\mathit{WRITE}, x, v, \mathit{VT}_i]$ **to** $\mathit{owner}(x)$
    receive $[\mathit{W\_REPLY}, x, v, \mathit{VT}']$ **from** $\mathit{owner}(x)$
    $\mathit{VT}_i := \mathit{update}(\mathit{VT}_i, \mathit{VT}')$
  $M_i[x] := (v, \mathit{VT}_i)$

$[\mathit{READ}, x]$ ::
  receive $[\mathit{READ}, x]$ **from** j
  send $[\mathit{R\_REPLY}, x, M_i[x].\mathit{value}, M_i[x].\mathit{VT}]$ **to** j

$[\mathit{WRITE}, x, v, \mathit{VT}]$ ::
  receive $[\mathit{WRITE}, x, v, \mathit{VT}]$ **from** j
  $\mathit{VT}_i := \mathit{update}(\mathit{VT}_i, \mathit{VT})$
  $M_i[x] := (v, \mathit{VT}_i)$
  $\forall y \in C_i : M_i[y].\mathit{VT} < \mathit{VT}_i$
    $M_i[y] := \perp$
  send $[\mathit{W\_REPLY}, x, v, \mathit{VT}_i]$ **to** j

discard ::
  $M_i[y] := \perp : \exists y \in C_i$

Figure 10: A simple owner protocol

p_0: r(y)0 w(x)1 r(y)0
p_1: r(x)0 w(y)1 r(x)0

Figure 11: A weakly consistent execution

owned. The writer increments the local timestamp and sends this to the owner, along with the value being written. The owner's local vector timestamp is then updated based on the incoming vector timestamp. The owner's updated timestamp is finally sent back to the initiating writer who performs a second update operation. Thus each write to a non-local memory location involves an increment and two updates of the associated writestamp.

Although our implementation may generate more invalidations than necessary, it still admits weakly consistent executions, not allowed by strongly consistent memories. The weakly consistent execution in Figure 11 is allowed both by causal memory correctness and by our implementation if p_0 is the owner of x and p_1 is the owner of y.

Finally, notice that our implementation includes a simple *discard* action. *discard* may be used as part of a *replacement* policy to make room for new values to be cached. Occasional execution of *discard* can also be used to ensure eventual communication and to provide *liveness*. Without *discard* two processors that initially cache all locations and only write locations owned by them need never communicate.

5.1.3 Correctness

Informally, our implementation maintains correctness by invalidating any locally cached values that could cause a violation of correctness if read. Violations are possible anytime a new value is introduced into the cache. Thus, our algorithm performs invalidations whenever a value is received as the result of a read or write request. Since a cached value can be read anytime, our algorithm invalidates values that, if read, can potentially violate correctness. Owners also perform invalidations when servicing write requests since this could lead to a potential causal interaction between the writing process and the owner if the owner reads that location in the future.

Two observations lead to the correctness of our implementation. First, violations of causal memory correctness are always related to violations of causality. A read of x by p_i returning v can only violate correctness if p_i "knows" of some other value v' whose read or write causally follows the write of v. Second, a read of x by p_i resulting in a request to the owner (a *remote* read) can never violate correctness since the owner is guaranteed to return a value that causally follows any value of x that p_i could previously have seen. Thus, a simple strategy to maintain correctness is to force a request to the owner on every read. This strategy results in a memory that satisfies sequential consistency, not just causal correctness, but we lose all the benefits of caching. A better strategy, the one used by our implementation, is to allow a process p_i to read cached values that are concurrent with or that causally follow the value most recently introduced into the cache. Reads of any other value (not locally owned nor cached) must generate a read request to the owner. Note that subsequent remote reads might introduce values that causally precede all other cached values so this strategy allows the cache to contain values with a

p_0: w(x)0 w(y)1
p_1: r(x)0 r(y)1

Figure 12: An example to illustrate unnecessary invalidation

wide range of writestamps.

5.1.4 Limitations of the Owner Protocol

The protocol embodies a *sufficient* but not *necessary* condition for correctness. This condition may be overly pessimistic in some cases but it provides a simple, easily implemented rule which guarantees that no read may return an over-written (causally prior) value. As an example of such a case, consider the process histories shown in Figure 12. Assume that p_0 owns both x and y. If neither location is initially cached at p_1 then our algorithm will invalidate x in p_1's cache when y is introduced even though x's value has not been over-written.

Also, the protocol is structured in terms of accesses to individual locations and as such is not very practical in distributed systems where it is costly to transmit small amounts of data over a network. The protocol also requires that a write operation to a location block the processor until the owner of the location has been informed of the write. Implementing write-through for memory locations is expensive in distributed systems where the message latencies are high. The static owner protocol is impractical in a distributed setting.

Another protocol for implementing causal memory was described by Ahamad et. al. [5]. Their protocol is update-based and requires each write operation to send a message to all processors informing them of the write. This generates a

large number of messages but does not suffer from the unnecessary invalidations. However, update-based protocols are not a feasible choice for page-based DSM's, since trapping each write operation is expensive.

The basic protocol described here can be improved in several ways. These include scaling the unit of sharing to a page, eliminating the unnecessary invalidations, and reducing the blocking of processors. We discuss this in the next section.

5.2 A Page Based Protocol

In this section, we develop a protocol where the unit of sharing is a page. This allows us to integrate consistency maintenance operations with virtual memory operations such as page faults. First, we discuss the issues that arise when we scale the unit of sharing from a single location to a page. We also consider the effects of synchronization and forking operations later in the section.

5.2.1 Issues in Scaling the Unit of Sharing

We can associate timestamps with pages instead of individual memory locations. The page now becomes the unit of consistency maintenance. Although causal memory permits multiple writers and readers to access a page concurrently, concurrent writers introduce the problem of merging the modifications to a page done at multiple processors. There could be concurrent writes to the page because of false sharing even when synchronization controls access to data stored in the page. The *diff* mechanism employed in [14, 33] can be used to handle the merging problem but it does have copying and processing

overheads, which sometimes can be quite high [31]. We deal with false sharing by making the restriction that a page can only be accessed by a single writer and multiple readers at a given time. Thus, a page cannot be cached at multiple processors with write access at any time. This does, however, serialize concurrent writes to a page. We pin a page to a processor [23] for a certain amount of time to control the situation where concurrent writers move a page between processors continuously (page thrashing). It must be noted that false sharing, where a single writer is concurrent with multiple readers, does not result in any communication in our implementation of causal memory.

The static owner scheme used in Section 5.1 is not appropriate in a distributed system since any write to a page would result in communication with the owner of the page. Instead we will use a dynamic-owner-fixed-manager protocol as described by Li and Hudak [39]. A *manager* is a processor that either caches a data item x assigned to it or knows the identity of the processor that caches a current copy of x. One of the processors that caches a current copy of x is also called its *owner*. The identity of the owner processor for a data item is known to the manager of the data item.

We develop an implementation where access to causally consistent shared data is provided by caching data pages in the memories of the processors. Processes at a processor can freely read data that is cached locally. Communication with other processors may be required when the data to be accessed is not locally cached or when it is written.

5.2.2 Implementation Approach

The key features of our implementation of causal memory include (1) use of page fault mechanisms for providing access to shared data to processes on

different processors, (2) use of vector timestamps for maintenance of causality information, (3) allowing a single writer and multiple readers to access a page concurrently.

Our implementation maintains consistency of shared data at the level of a page. Thus, each processor's memory caches a subset of the shared pages. When a page not present in the local memory is accessed, it generates an *access* fault. A write on a page that is cached with read access causes a *protection* fault.

Each processor maintains a vector clock and timestamps derived from this clock are stored with each copy of a page. More specifically, the timestamp on a page copy reflects the vector time at the processor that last wrote this copy of the page.

5.2.2.1 Data Structures

Each processor maintains several data structures to provide access to shared pages. In particular, processor p_i maintains a vector clock VT_i, which is used to timestamp pages written at p_i. A table that has an entry for each shared page is also kept at each processor. An entry for page x in this table indicates if the page is currently cached, the page's vector timestamp, the manager and owner processors of the page, and its access information. Since we allow only a single writer to a page, the owner field in the entry stores the identifier of the processor that has the most recent copy of the page. The access field specifies the access privileges to the page at the local processor. It can be *null*, *readonly* or *readwrite*.

Initially, each component of the vector clock is set to 0 at all processors. Only the manager processor of a page has the owner and manager fields set to

itself for that page. Processors other than the manager have the access field set to *null* for the page and the manager field appropriately initialized (other fields for a page do not need to be defined at these processors). We now discuss the handling of access and protection violations and the manipulation of the data structures stored at each processor.

5.2.2.2 Handling Page Faults

The actions executed when page faults occur at processor p_i are shown in Figure 13. We also show the actions executed by the owner and manager processors for servicing page faults.

On a read access fault, a processor sends a READ message to the manager of the page. If the fault is due to a write access, a WRITE message is sent. On a protection fault, the faulting processor checks first whether it is the owner of the page. If it is, it upgrades the access to the page to *readwrite*. Otherwise, a WRITE message is sent to the manager. The manager on getting a request, supplies the page if it is the owner or else forwards the request to the current owner. If a WRITE message was sent, the current owner downgrades its access to the page to *readonly* (we do not need to make its access *null* since we allow readers to concurrently exist with a writer), and sends a copy of the page to the faulting processor, which assumes ownership. If the fault was due to a read, the owner supplies a copy of the page and downgrades its access to *readonly* but retains ownership of the page. This is done so that the processor's vector time can be incremented the next time it writes to that page. The faulting processor, on receiving the page in a DATA message, installs the page with the appropriate access rights.

```
Read Access Fault::
  send [READ, x] to x.manager
  recv [DATA, x, VT′]
  x.access := readonly
  x.VT := VT′
  VT_i := max(VT_i, VT′)
  invalidate(VT′)

Write Access Fault::
  send [WRITE, x] to x.manager
  recv [DATA, x, VT′]
  x.access := readwrite
  x.owner := self
  VT_i := max(inc(VT_i), VT′)
  x.VT := VT_i
  invalidate(VT′)

Protection Fault::
  if (x.owner != self)
      Handle similar to
      Write Access Fault
  else
      x.access := readwrite
      VT_i := inc(VT_i)
      x.VT := VT_i
```

(a) Actions Executed at p_i

```
[READ, x]::
  recv [READ, x] from p_i
  if (x.owner = self)
      x.access := readonly
      send [DATA, x, x.VT] to p_i
  else
      send [FWD, x, i, read] to x.owner

[WRITE, x]::
  recv [WRITE, x] from p_i
  x.owner := i
  if (x.owner = self)
      x.access := readonly
      send [DATA, x, x.VT] to p_i
  else
      send [FWD, x, i, write] to x.owner

[FWD, x, i, mode]::
  recv [FWD, x, i, mode] from x.manager
  if (mode = write) x.owner := i
  x.access := readonly
  send [DATA, x, x.VT] to p_i
```

(b) Manager & Owner Actions

Figure 13: Causal memory implementation using vector timestamps

5.2.2.3 Vector Clocks and Timestamps

In general, vector timestamps are incremented between local events and are also included in all messages. A processor's clock is also updated when a message is received by performing a *max* operation using the current value of the clock and the timestamp received with the message. The result is assigned to the vector clock of the receiving processor.

In our implementation, the value of clock, VT_i, is not sent when p_i sends request messages to other processors. This is because causal dependencies between processors in a shared memory system are only created when data written by one processor is read by another. Thus, a request message does not create a causal dependency. The causal order created by a read orders the associated write operation before the read. Once a page is mapped with *read-write* access, there is no mechanism to track when the last write was made to the page. The page could potentially have been written as late as the current vector time at the processor. The timestamp sent in a DATA message is, therefore, the current value of the sender's clock. Thus, in Figure 13, only DATA messages carry the timestamp associated with the page being sent. When a page is received in a DATA message due to a write fault, the clock is incremented and updated because a timestamp read from the clock is assigned to the new version of the page data that will be created by the processor.

The actions that handle the various types of faults in Figure 13 show how page timestamps are determined. When p_i requests and caches a page in *readonly* mode, the timestamp associated with it is received in a DATA message with the copy of the page. Page timestamps are used to decide when the page copy may be overwritten according to causality. If a processor is caching a page in *readwrite* mode, its timestamp stores the time at which the

processor last write faulted on the page. Multiple writes at a processor that fall within a single page will result in a single increment operation to the writer's vector clock because only the first write generates a fault. However, there are situations when the clock needs to be incremented several times. This happens when other processors get copies of the page while a processor is writing it (we allow concurrent readers with a writer). The clock is incremented to ensure that different versions of a page data have different timestamps associated with them. We achieve this by downgrading the owner processor's access to a page to *readonly* when it sends a copy of the page in response to a READ message (or FWD when the owner is different from the manager). By making the page *readonly*, we ensure that a future write by the owner would generate a protection fault which will result in incrementing the clock and a new timestamp being assigned to the page. Thus, the new version of the page data will have a higher timestamp than the preceding version that has been supplied to another processor.

5.2.2.4 Maintaining Data Consistency

Our implementation must ensure that when a processor reads data from a page, the locally cached copy of a page has not been overwritten. In other words, if the page contains a value for location x which was written by operation o_1, then it is not the case that there is another write operation o_2, such that $o_1 \overset{ecg}{\rightarrow} o_2$ and o_2 causally precedes the read that returns the value written by o_1. We ensure that only causally consistent data is read by locally invalidating cached pages when it is suspected that they contain causally overwritten data. The vector timestamps maintained for cached pages are used to determine when they may contain potentially overwritten data.

Causal orderings between operations are established when a processor reads data written by another processor. In our system, this would happen because a processor faults on a page and receives it from another processor. Since reading the data in a newly cached page could result in new causal orderings, consistency operations need to be executed when a page is added to the cache of a processor.

The basic consistency maintenance operation, *invalidate(timestamp TS)*, is shown in Figure 14. It is performed for a set of pages with respect to the timestamp *TS*. When it is executed at processor p_i, pages in C_i, which is the set of shared pages cached at p_i, are checked. If a page in C_i is cached with *readonly* access, p_i is not the owner of the page, and the page timestamp is less than *TS*, the page is locally invalidated by setting its access to *null*. Since the invalidation is local, no messages are sent to other processors that cache the page.

```
invalidate(timestamp TS)
{
    ∀ y ∈ C_i
        if ( (y.access = readonly) ∧ (y.owner ≠ self) ∧ (y.VT < TS) )
            y.access := null
}
```

Figure 14: The invalidate operator

5.2.2.5 Synchronization Operations

In parallel programs, communication between processors also takes place through synchronization operations and forking operations. Synchronization operations are used for access control (e.g., semaphores and locks ensure that a

sequence of memory operations are executed atomically) or to provide sequence control (e.g., barriers). Forking operations are the mechanism by which computation is assigned to a particular processor.

Thus, we need to consider the effects of synchronization operations and forking operations on the implementation of causal memory. In distributed systems, synchronization operations are implemented by a server (many distributed synchronization algorithms exist but they have high message costs or latency). Since synchronization operations order memory operations, their implementation must be modified to carry the ordering information. We do this by associating a vector timestamp with each synchronization variable.

The acquire and release actions on a lock variable are shown in Figure 15(a). Each lock l has an associated vector timestamp $l.ts$. On a release operation by processor p_i on lock l, p_i assigns the current value of VT_i to $l.ts$. When l is acquired by another process p_j, VT_j is updated by assigning to it the component-wise maximum of the current value of VT_j and $l.ts$. By updating its clock, processor p_j ensures that its clock orders all memory operations at p_i that were executed before the corresponding release operation on lock l. The *invalidate* operation shown in Figure 15 is the same as described before.

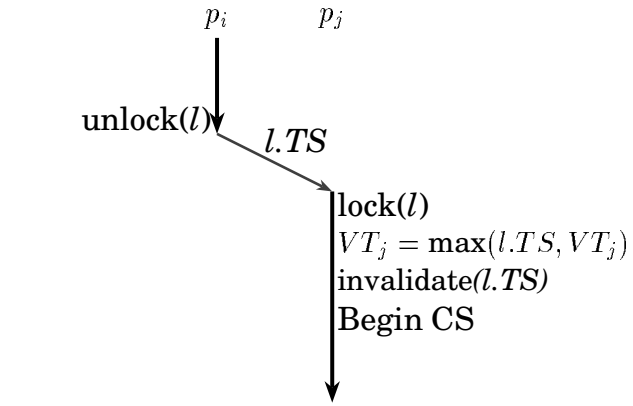
Barriers are implemented by having a barrier server do an update of its clock based on the timestamps received from every processor in the request messages that are sent when a process reaches a barrier. A timestamp read from the updated clock is transmitted with the barrier release and every processor updates its vector clock to reflect the timestamp received. Thus, each processor participating in the barrier call orders memory operations at all processors that are executed before the barrier call.

Object invocations (or forks) also induce orderings between memory operations. Their handling is shown in Figure 15(b). When processor p_i invokes an

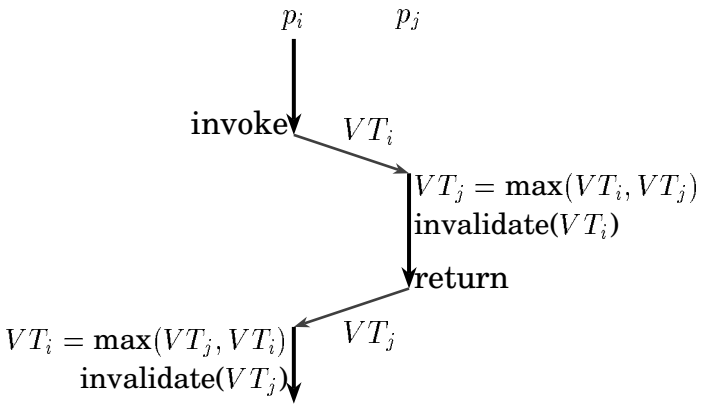
object on p_j, the value of the vector clock at p_i, VT_i, is sent with the invocation request. Before executing the invocation, processor p_j updates its clock to reflect the timestamp received with the invocation request. On return from the invocation, VT_i is updated to reflect the timestamp that comes from p_j in the return message.

5.2.2.6 Reducing the Unnecessary Invalidations

The implementation that we sketched handles two of the problems mentioned earlier, namely scaling the unit of sharing and the blocking of processors. But we still have the problem where we could be invalidating more pages than are absolutely necessary. This occurs for two reasons. Pages which are not written frequently enough would not have their timestamps increasing and thus would end up getting invalidated. Also, the discard operation could invalidate pages which still have current values. To reduce the occurrence of unnecessary invalidations, we provide language support to annotate data. Data which is read-only or write-once are tagged *Static*. These pages are not considered for invalidation when the local invalidations are done, since their values never change during the course of the execution. Data which needs to be discarded periodically to preserve liveness is tagged as *Transient*. Only transient data is periodically discarded. Figure 16 shows the data structures and their annotations for the linear equation solver and the distributed calendar applications that were discussed in Chapter 4. In the linear equation solver, the array A and the vector b are inputs and remain constant during the execution of the application and thus are tagged as *Static*. In the distributed calendar, the boolean *changed* has to be discarded periodically to ensure that new values would be seen. So it is tagged as *Transient*.



(a) Handling Synchronization



(b) Handling Invocations

Figure 15: Synchronization and Invocations

```
float   [Static]   A[n][n];
float   [Static]   b[n];
float              x[n];
```

(a) Linear equation solver data with annotations

```
calendar_type                calendar[MAX_USERS];
lock                         mutex[MAX_USERS];
boolean         [Transient]  changed[MAX_USERS];
```

(b) Distributed calendar data with annotations

Figure 16: Applications with annotations

5.2.3 Performance Analysis

The implementation of causal memory should provide improved performance for applications compared to a sequentially consistent memory. We demonstrate improved performance by showing that causal memory requires much fewer number of messages to maintain consistency of data compared to a sequentially consistent memory.

We compare a causal and a sequentially consistent DSM, both running the linear equation solver and the distributed calendar. We assume a comparable static-manager-dynamic-owner protocol [39] for sequentially consistent memory. A write requires that all cached copies in the system be invalidated. (In Ivy [39], a representative sequentially consistent DSM, a read set is maintained by the owner and invalidation messages are sent to all processors in the read set.) In comparison, a causal write requires at most three messages.

A simple message counting argument shows the advantage of causal memory when running the synchronous linear solver. For simplicity, assume that each processor in the system runs a single worker process and that the data

is partitioned such that it fits exactly into one page. Assume also that each processor also manages the data assigned to it. We also do not consider the number of synchronization messages, as they would be the same in either implementation.

First consider the causal memory implementation. In every iteration, p_i must issue read requests to all other processors when reading values not in its data partition. This results in $2(n-1)$ messages ($n-1$ requests to the manager, and $n-1$ replies). When processor p_i writes to x_i, it does not generate any messages. Now consider the same execution on sequentially consistent memory. The same number of messages are generated for reading the pages not in one's partition. However, when processor p_i writes x_i at the end of a phase, sequentially consistent memory requires that all cached copies be invalidated. Since every process other than p_i has a copy of x_i, this results in $2(n-1)$ messages ($n-1$ invalidation messages and corresponding acknowledgements) per processor, a cost not incurred by causal memory. Thus each phase of the synchronous linear solver requires $4n-4$ messages per processor when executed on sequentially consistent memory compared to $2n-2$ when executed on causal memory. This represents a substantial savings of $2n(n-1)$ messages per iteration and causal memory will lead to performance gains because its consistency actions are local and do not require any communication.

In Figure 17 we compare the number of messages when the calendar application is executed on causal memory and sequentially consistent DSM. We have assumed that each user's calendar data would not require storage more than a page. This assumption just simplifies the message count analysis.

We consider the maximum number of messages which could be exchanged. This would happen when the manager processor is not the current owner of a requested page. Causal memory requires at a maximum three messages

Operations	browse	update	set_up_meeting	daemon
Max Messages(SC)	3	2r+3	k(2r+3)	(3,6+2r)
Max Messages(Causal)	3	3	3k	(3,6)

r: number of readers
k: number of members in meeting group

Figure 17: Message counts for different memory models

to service a page fault. On the other hand, the number of messages on sequentially consistent memory depends on the number of concurrent readers. The two values for the *listener_daemon* case correspond to the cases where the *if* statement evaluates to true or false. Since the number of messages for each operation is bounded, scalable implementations are possible for causal memory.

5.3 Concluding Remarks

This chapter showed how to implement a causally consistent DSM. We started with a simple protocol and extended that to a practical page-based implementation.

One of the problems that the implementation had was that of unnecessary invalidations. We provided a mechanism whereby the programmer could annotate data in the program. This reduces the number of unnecessary invalidations but does not eliminate it. In the next chapter, we propose an alternative implementation which eliminates the problem of unnecessary invalidations.

We also presented message count arguments to show that causal memory

has much lower communication overhead compared to a sequentially consistent memory implementation. In a later chapter, we provide a detailed experimental evaluation to quantify exactly the performance improvements made possible by the weaker consistency of causal memory.

Chapter 6

Optimizing for Data-Race-Free Programs

A large number of shared memory parallel programs tend to be data-race-free (DRF) programs. Several optimizations are possible in the implementation of causal memory if we restrict ourselves to this class of programs. These programs are interesting since they execute correctly on causal memory without any change in the code. In the protocol presented in the previous chapter, we invalidated cached data items that could be potentially overwritten when a new data item was added to the cache. In the implementations developed in this paper, we assume that the data is shared by DRF programs, and this allows us to develop more efficient schemes for maintaining causal consistency. In this chapter, we first modify the vector timestamp protocol to reduce the consistency maintenance overhead for such programs. We then introduce a new implementation of causal memory based on versioned pages which eliminates the problem of unnecessary invalidations.

6.1 Causal Memory and DRF Programs

If programs are known to be free of data races, we can get more efficient implementations of causal memory. Causal orderings between memory operations at different processors are established when a processor reads a value written by another. Thus, each time a processor caches a data value written at another

processor, it must ensure that the newly cached value is causally consistent with the data values already existing in its memory. In particular, any new causal orderings that get established by the reading of the newly cached data value must not cause the existing data to be overwritten in the causal sense.

When programs are known to be DRF, the check to determine that existing data remains causally consistent with information received from another processor only needs to be performed when *acquire* synchronization operations complete. This optimization is possible because the order induced between memory operations by synchronization operations, $\stackrel{so}{\rightarrow}$ (see Section 3.2), is identical to the extended causal order $\stackrel{eco}{\rightarrow}$. This follows from the fact that conflicting operations, for example two operations $o_1 = w(x)v$ and $o_2 = r(x)v$, must be ordered by the order defined by synchronization operations when o_1 and o_2 are executed by different processors. Thus, the synchronization operations will order o_1 and o_2, and the writes-before order, $\stackrel{wb}{\rightarrow}$, between o_1 and o_2 cannot create any new orderings between memory operations. As a result, we do not need to check for causal consistency of data at a processor when a new data value is added to its memory. The protocol that we develop in the next section makes use of this fact to avoid extra processing overhead for programs known to be free of data races.

6.2 Causal Memory Implementation Based on Vector Timestamps

When programs are free of data races, the local invalidations need not be performed when pages are brought into the cache. Also, the updating of clocks when acquire operations complete make it unnecessary to advance clocks when

DATA messages containing a page are received due to read faults. In a shared memory environment, causal relationships arise between processors either when one processor reads what is written by another processor, or due to synchronization operations. Since timestamps are transferred with synchronization variables, and the vector clock of the processor acquiring a synchronization variable is updated, we consider the case when a processor reads a value written by another processor. In particular, when p_i reads a value of location x which is written by p_j, VT_i should be updated to include the value of VT_j at the time p_j wrote x. In our implementation, for p_i to read the value of x, it must have generated a page fault to fetch the page that contains x after the page was written by p_j. Although VT_i is not updated when the DATA message is received at p_i, VT_i still orders all operations of p_j including the write that produced the value being read. This is because we assume DRF programs and hence p_j must have done a release on a synchronization variable after its write to x. Furthermore, p_i must have acquired the synchronization variable which would advance VT_i to include all operations up to the release by p_j. Thus, the value of VT_i will be greater than the time at which x was written and it is not necessary to update the clock when DATA messages are received for servicing read faults.

However, the write fault action does need to update the vector clock VT_i before generating a timestamp for the page being written. This is necessary, even when programs are DRF and a lock is acquired before the write is done, due to false sharing problems. Consider the execution shown in Figure 18. Assume that both data items, x and y, are stored in a single page. p_0 first acquires a lock, l_1, that controls access to x, writes x and then releases l_1. It then acquires l_2 that controls access to y and writes it. Assume that p_1 now acquires l_1 and reads x after p_0 has written y. Clearly, the timestamp received

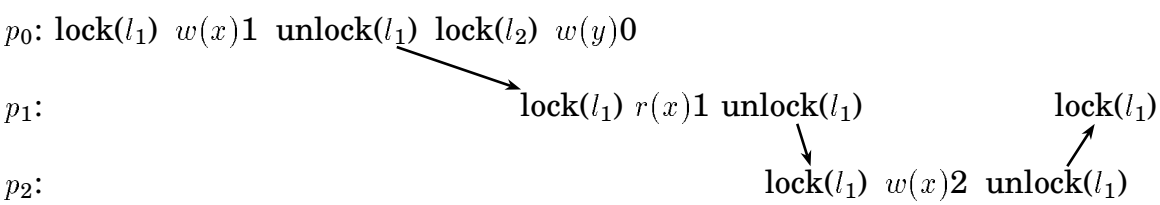


Figure 18: Example to show why clock update is necessary on write faults

with l_1 will be less than the timestamp on the page when p_1 read faults and receives it after acquiring l_1. If p_2 now acquires the lock and writes to location x without having first updated its clock, the page cached at p_1 will not have a timestamp that is smaller than the timestamp assigned to the new version of the page at p_2. Our consistency maintenance operations require that writes to a page be totally ordered, and this be reflected in the timestamps associated with the copies of the page. An increment followed by a clock update in the write fault action in Figure 19 guarantees that this property holds.

False sharing could lead to a similar situation when p_2 only reads the page. In this case, p_2's clock need not be advanced because only the data written by p_0 before it released l_1 is read. In our example, p_2 advances the clock because it writes the page. Thus, when p_1 acquires l_1 again, its copy of the page will have a lower timestamp than the timestamp received with the lock. The modified protocol is shown in Figure 19.

The consistency action, now executed only at synchronization and fork points, remain the same as in the previous chapter, and is shown again in Figure 20.

The local invalidations guarantee that if a page is locally cached, the data stored in the page is not causally overwritten according to the view of the

Read Access Fault::
send [READ, x] **to** x.manager
recv [DATA, x, VT′]
x.access := readonly
x.VT := VT′

Write Access Fault::
send [WRITE, x] **to** x.manager
recv [DATA, x, VT′]
x.access := readwrite
x.owner := self
VT_i := max(inc(VT_i), VT′)
x.VT := VT_i

Protection Fault::
if (x.owner != self)
Handle similar to Write Access Fault
else
x.access := readwrite
VT_i := inc(VT_i)
x.VT := VT_i

(a) Actions Executed at p_i

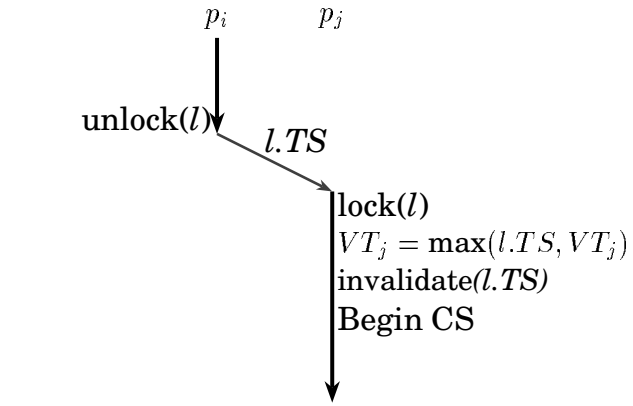
[READ, x]::
recv [READ, x] **from** p_i
if (x.owner = self)
x.access := readonly
send [DATA, x, x.VT] **to** p_i
else
send [FWD, x, i, read] **to** x.owner

[WRITE, x]::
recv [WRITE, x] **from** p_i
x.owner := i
if (x.owner = self)
x.access := readonly
send [DATA, x, x.VT] **to** p_i
else
send [FWD, x, i, write] **to** x.owner

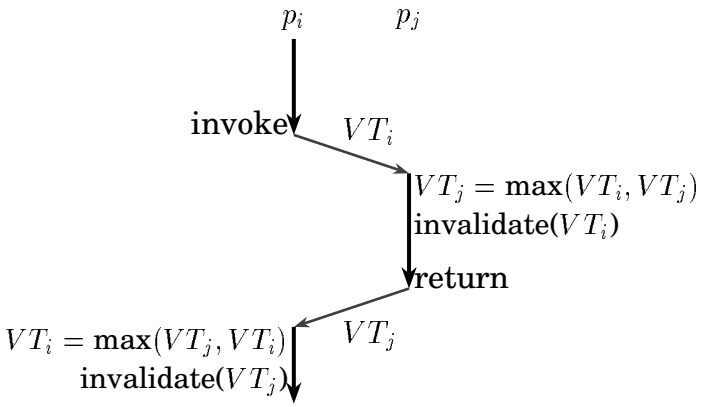
[FWD, x, i, mode]::
recv [FWD, x, i, mode] **from** x.manager
if (mode = write) x.owner := i
x.access := readonly
send [DATA, x, x.VT] **to** p_i

(b) Manager & Owner Actions

Figure 19: Causal memory implementation for data race free programs



(a) Handling Synchronization



(b) Handling Invocations

Figure 20: Synchronization and Invocations

processor. Consider data item x and the time at which the page containing x was brought to the memory of processor p_i for reading. For simplicity, assume that the page only contains the data item x. The vector timestamp stored at p_i for this page is assigned from the clock of the processor p_j that last wrote x. In particular, the timestamp was the value of VT_j when p_j wrote x. Since p_j was the owner of the page at the time the request due to p_i's read fault was serviced, the value of *x* received by p_i in the page was up-to-date. Assume that x is now written again by another processor p_k. This processor must have acquired a synchronization variable such as a lock before it writes x. Furthermore, the lock must have been released by p_i which had acquired it to read x. A lock carries a timestamp that is the value of the vector clock at the processor that last executed the release operation. Also, a processor updates its clock when it acquires a lock. Therefore, before p_k's write, the value of VT_k will be greater than or equal to the timestamp that was assigned to the page by p_j (this timestamp was assigned before p_j released the lock to p_i and is also stored with the page at p_i). Furthermore, VT_k is incremented when p_k performs its write to x. To read a page that has been written since the time the page was cached, p_i must acquire the lock again. It is easy to see that the timestamp received with the lock will be greater than the timestamp of the cached page. As a result, the consistency actions executed at the time the acquire operation completes will invalidate the old copy of the page at p_i. Thus, when the page is read again, it will be requested from the current owner and the causally overwritten data will not be read.

A page can store multiple data items and hence several locks may be used to control access to the data stored in it. Causal consistency is maintained for shared data even when we have such false sharing. For example, the page containing data item x in the discussion in the previous paragraph, could have

been written by another processor after p_j wrote x (such a processor could have written data item y which is also stored in the same page). In this case, the timestamp p_i receives from p_j with the lock that controls access to x will be lower than the timestamp p_i receives with the page when it reads x. However, because a processor updates its clock when it receives a page for writing, p_k's clock will be advanced beyond the timestamp that p_i stores for the page that contains x. Thus, when p_i acquires the lock again to read x the second time, it will receive a timestamp that is higher than the timestamp it stores for x's page. As a result, the page will be locally invalidated and hence the overwritten value of x cannot be read.

Since we allow a single writer for a page, reading a page that is cached with *readwrite* access rights can never result in overwritten data being read. Thus, a cached page at an owner processor always contains data that is causally consistent.

We can consider more complex situations but because of the the fact that programs are DRF, a processor will first acquire a synchronization variable before it accesses the data in a page. Since timestamps are transmitted with synchronization variables, a page that has been written by a causally later write and the release operation that follows it, will ensure that the synchronization variable carries a timestamp higher than the timestamp associated with the page copy that stores the value of the old write. This will always guarantee that causally overwritten data is invalidated before a processor can access it.

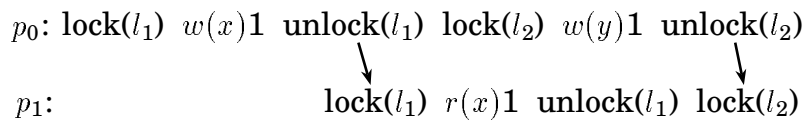


Figure 21: Unnecessary invalidation of page containing data item x

6.2.1 Unnecessary Invalidations

As in the implementations in the last chapter, the local invalidations are sufficient to ensure correctness but they are not always necessary. This is because not all pages that are older with respect to a timestamp are causally overwritten. For instance, consider the execution shown in Figure 21. Assume that x and y are on different pages and that the read by p_1 of x returns the value written by p_0. When p_1 completes lock(l_2), the page containing x would get invalidated even though it still holds the current value of x. This is because the clock at p_0, VT_0, gets incremented when y is written so the timestamp received with l_2 is greater than the timestamp associated with page x. The problem is that vector timestamps, with a component for each process, do not accurately track the exact set of pages that have been modified. Because a write by p_0 to either x or y would have resulted in the same timestamp being received at p_1 when lock(l_2) completes, p_1 must assume that x could be overwritten and invalidate the page.

We modify the protocol given here in the next section to ensure that only those pages that have been causally overwritten are invalidated when an *invalidate* operation is executed. This requires that we expand the vector timestamp to include a component for each shared data page.

6.3 Causal Memory Implementation Based on Versioned Pages

In our implementation of causal memory, we allow only a single writer to access a page concurrently with multiple readers. As a consequence, a sequence of writes to a page can be tracked with a version number. In the second implementation of causal memory shown in Figure 22, which eliminates the unnecessary invalidations, version numbers associated with pages are used to determine when the data in a page are causally overwritten[1]. Each processor maintains a version number (instead of a vector timestamp) with each of its cached pages. Also, the vector clock at processor p_i is replaced by a *version vector*, VV_i, which stores the latest version number known to p_i of each of the shared pages. $VV_i[x.num]$ is the latest version of page x (the field $x.num$ stores the index of page x) as known to p_i. The basic idea is to transfer the value of VV_i with synchronization variables and to use these versions in consistency maintenance operations. In particular, when an acquire operation completes on a synchronization variable, a cached page x is locally invalidated if the version number stored with it is less than the version number received for x with the synchronization variable. We now explain the operation of the second implementation and discuss how it differs from the first one.

6.3.1 Page Fault Handling and Version Management

On a page fault, the same actions are executed as before. A read fault results in a request message for the page which is sent to the manager which

[1]Version numbers were used in the Locus file system for detecting concurrent updates to a single file [49]. Our version vectors are used to ensure causal consistency for a set of data pages.

```
Read Access Fault::
  send [READ, x] to x.manager
  recv [DATA, x, version′]
  x.access := readonly
  x.version := version′

Write Access Fault::
  send [WRITE, x] to x.manager
  recv [DATA, x, version′]
  x.access := readwrite
  x.owner := self
  x.version := version′ + 1
  VA_i[x.num] := x.version

Protection Fault::
  if (x.owner != self)
      Handle similar to
      Write Access Fault
  else
      x.access := readwrite
      x.version := x.version + 1
      VA_i[x.num] := x.version
```

(a) Actions Executed at p_i

```
[READ, x]::
  recv [READ, x] from p_i
  if (x.owner = self)
    x.access = readonly
    send [DATA, x, x.version] to p_i
  else
    send [FWD, x, i, read] to x.owner

[WRITE, x]::
  recv [WRITE, x] from p_i
  x.owner := i
  if (x.owner = self)
    x.access = readonly
    send [DATA, x, x.version] to p_i
  else
    send [FWD, x, i, write] to x.owner

[FWD, x, i, mode]::
  recv [FWD, x, i, mode] from x.manager
  if (mode = write) x.owner := i
  x.access := readonly
  send [DATA, x, x.version] to p_i
```

(b) Manager & Owner Actions

Figure 22: Causal memory implementation using versioned pages

either sends the page or forwards the request to its current owner. The DATA message that contains the requested page also includes the version number of the page. On a write fault, since a new version of the page data will be created by the processor, the received version number of the page is incremented and the corresponding entry in VV_i is updated. If a processor owns a page, a protection fault on that page does not require any communication but the version number and the version vector are similarly updated. In the case where the processor is not the owner of a page, a protection fault is handled similar to a write access fault.

To ensure that different page versions have different numbers, an owner downgrades its access to a page to *readonly* when it sends a copy of the page to another processor even when the request for the page was due to a read fault. This guarantees that the version number is incremented if the owner writes the page again. This is a local operation and does not require any messages. The same page, cached at different processors, may have different version numbers. The current *owner* of the page has the latest copy of the page and will always have the highest version number.

Although DATA messages contain only the version number of the page being transferred, synchronization operations need to transfer the version vector information between processors. For example, when a lock is released by processor p_i, the lock is assigned a timestamp which is the current value of VV_i. When processor p_j next acquires this lock, the lock's timestamp is used to update VV_j. Similar to Figure 15 for vector clocks, a maximum operation is performed on each component of VV_j and the timestamp received with the lock, and the result is assigned to VV_j. Other synchronization constructs such as barriers similarly update the version vectors at the participating processors.

```
invalidate(version_vector VV)
{
    ∀ y ∈ C_i
        if (y.version < VV[y.num])
            y.access = null
}
```

Figure 23: The invalidate operator for versioned pages

6.3.2 Maintaining Data Consistency

Causal consistency of shared data is maintained by locally invalidating pages at synchronization acquires and at fork-join points as shown in Figure 20. The *invalidate* operation is shown in Figure 23. As before, C_i is the set of all pages in the cache of processor p_i. The timestamp received with the synchronization variable is passed as a parameter to the *invalidate* operation. It locally invalidates a page if the version number associated with the page is smaller than its version in the timestamp (y.version is the version number of page y and y.num is the page number).

We do not need to check if a processor owns a particular page or if it caches it only in *readonly* mode before invalidating the page. This was done in the previous implementation because a vector clock could have been advanced due to writes to other pages. Thus, a timestamp received with a synchronization variable could have been greater than the page's timestamp at the owner even when it had the most recent copy of the page. Since the owner of the page has the highest version number for the page, a page at the owner will never be invalidated.

6.4 Comparing the Two Implementations

The two implementations differ in the state maintained and exchanged by processors. In the first implementation, vector clocks are maintained which have a component for each processor in the system. The second implementation based on version numbers, has a component for each shared data page of the application. The storage and communication costs of the implementations depend on the sizes of these data structures. It may appear that a version vector may have a much larger number of components compared to a vector clock (when the number of processors is smaller than the number of shared pages). This could result in higher storage and communication overheads for the second implementation. This is not the case for many of the applications that we studied. One reason is the relatively large page size (typical page sizes for most workstations are 4K and 8K bytes). Also, only a single version number is sent in DATA messages in the second implementation whereas these messages include vector timestamps in the first implementation. Thus, only the messages that transfer synchronization variables incur the overhead of transmitting the version vector. Version numbers also reduce the processing overhead in consistency maintenance operations because version numbers instead of vector timestamps are compared.

A version based implementation could have excessive storage and communication overhead when the shared data space is very large. The annotations discussed in the previous section can also be used to reduce the size of version vectors. Pages that are *Static* are not considered for invalidation and do not need to have version numbers associated with them.

6.5 Additional Optimizations

The implementations of causal memory also allow several other optimizations. They are described next.

6.5.1 Avoiding Page Copying

Our implementation allows a writer to be concurrent with readers. While a page is being transferred by the owner of a page in response to a READ request, it is quite possible that the owner processor continues to write into that page. This is because a page has to be broken down into several messages, each the size of a protocol data unit, and the application could run while the protocol code is blocked waiting for an acknowledgement for a previous message. In general, the protocol code copies the data to be transmitted in a separate buffer to avoid the problem of the data being modified while it is sent. In our system, such copying is not necessary because programs are assumed to be DRF.

Consider the case in which an owner processor p_0 receives a request for a page due to a read fault at processor p_1. At p_0, the execution of the communication protocol code can be interleaved with the execution of the application code. Since we downgrade the access to the page at p_1 on processing a READ or FWD message to *readonly* before sending the page, the application code at p_0 would raise a protection fault on a write, which would be handled locally as p_0 is still the owner. Also, because programs are free of data races, the data in the page being written by p_0 will not be read by p_1, the processor to which the page is being sent. In fact, the concurrent access to the page must be due to false sharing and p_0 writes to parts of the page for which p_0 holds an exclusive lock. Thus writes by p_0 while the page is being transmitted to p_1 will not change any data that is accessed by p_1 and hence the page need not be copied by the

communication protocol. If p_1 acquires a lock in the future for data in the page which got modified while the page was being transferred previously, its copy of the page would be invalidated because p_0 generated a newer version since the last acquire operation of p_1. Thus, consistency is guaranteed even when copying is not done. Recently techniques have been proposed that can be used to reduce copying overhead in message passing systems [20, 40]. However, they require additional flexibility from the underlying operating system.

6.5.2 Avoiding Page Transfers on Double Faults

A *double fault* occurs when a page that is not cached locally is first read and then written [35]. This would cause a page to be transferred once due to the read and again due to the write. The second page transfer will occur because the processor is not the owner of the page when the protection fault is handled. In cases where the page has not been modified since it was fetched as a result of the read fault, transferring the page the second time is wasteful since the faulting processor already has the current version of the page. Although not shown in Figures 19 and 22, a processor includes the version number (or the vector timestamp) of the locally cached page in the request message sent to the manager on a protection fault. The current owner compares this version number (timestamp) with its version (timestamp) for the page. If they are the same, the owner does not transmit the page in the DATA message.

6.6 Concluding Remarks

In this chapter, we presented two protocols for guaranteeing causal consistency for DRF programs. The absence of conflicting operations in a DRF program

allows us to optimize on the general protocol described in the last chapter by restricting the consistency maintenance operations to certain synchronization points.

The vector timestamp protocol has the problem of unnecessary invalidations, where data not causally overwritten could still get invalidated. The version vector protocol eliminates the unnecessary invalidations but typically at the cost of an increase in the size of the vector which needs to be transmitted at synchronization points. The data annotation can be used to reduce the number of unnecessary invalidations in the vector timestamp protocol and reduce the size of the version vector in the second protocol.

Both protocols allow a single writer to coexist with multiple readers. DRF programs allow the protocols to avoid the overhead of copying a page before transmitting the page, resulting in the possibility of the page being modified while it is concurrently being transmitted. This leads to the situation where processors would not just have different versions of a page in their caches but also inconsistent versions. These inconsistencies get resolved at synchronization acquire points when the local invalidations are performed.

Chapter 7

Performance Evaluation of Causal Memory

This chapter presents a detailed experimental evaluation of the implementations of causal DSM described previously. Causal DSM is compared with an implementation of a sequentially consistent DSM and a message passing implementation. We start by describing the two other systems that were implemented for comparison. To compare the performance of these systems we used a set of applications commonly used to benchmark DSM system performance. These applications and their implementations are described next. We then characterize the performance metrics and analyze in detail the performance with respect to these metrics.

7.1 System Descriptions

To compare causal memory with a strongly consistent memory, we implemented a DSM system that provides sequential consistency[1]. In addition, our system provides support for message passing on the same platform. The sequentially consistent DSM protocol implements a fixed manager writer-invalidate-readers protocol similar to the one described by Li and Hudak [39].

[1] In Section 7.6, we address how the performance of causal memory compares with other memory systems such as release consistency, entry consistency and others.

The protocol was optimized in several ways. For example, we use pinning to control thrashing and also use a technique similar to the one described by Kessler and Livny [35] to avoid re-sending a page due to a double page fault. All of the DSM protocols were implemented in the Clouds operating system [19] using low level communication mechanisms. Thus, consistency related activities, which are performed on page or protection faults (and also on certain synchronization events), are implemented in the kernel. The synchronization constructs used by the memory system are implemented by central servers. For a given synchronization variable, a single server maintains its state and the queue of processes blocked on it.

For the message passing system, we provide two system calls, *msgsend* and *msgreceive* to the application. A *msgsend* call results in copying of the data being sent to a buffer in kernel space. At the receiving processor, the received data buffer is enqueued until a *msgreceive* is executed by the process. The *msgreceive* call is synchronous – it blocks the process until the data is received. Since message passing only sends the data that needs to be shared and only to those processes that will use it, it provides a lower bound for execution time for most applications and also allows us to quantify the extra overhead in providing a shared memory abstraction.

7.2 Applications

A number of applications were implemented to evaluate causal memory. The applications include Embarrassingly Parallel (EP), Integer Sort (IS), and Conjugate Gradient Method (CGM) from the NASA Ames NAS kernels [10], and traveling salesperson (TSP), matrix multiplication (MM), and successive over-relaxation (SOR). These applications have been used in the study of several

distributed shared memory systems. We chose them to ensure that we evaluate causal memory for a variety of data access patterns, synchronization patterns, communication patterns, computation granularity (which is the amount of work done between synchronization points), and data granularity (which is the amount of data manipulated between synchronization points). The last two together define the task granularity of a parallel application. If the application is implemented using the message passing style, then the data access pattern becomes unimportant (except for any cache effects) since all data accesses are to private memory. Further, the synchronization pattern is usually merged with the communication pattern in such an implementation. On the other hand, if a shared memory style programming is used, the communication pattern is not explicit and gets merged with the data access pattern.

- Embarrassingly Parallel (EP) kernel evaluates integrals by means of pseudo-random trials and is used in many Monte Carlo simulations. As the name suggests, the kernel requires very little synchronization and communication among the parallel threads executing on different processors. Each thread computes an equally large number of floating point random numbers and performs certain floating point operations on them. The only communication that happens is toward the very end when all the processes participate in a reduction operation to generate a global sum. The kernel also has a very high task granularity.

- Integer Sort (IS) kernel uses bucket sort to rank a large set of integers. The input data is partitioned among participating processors. A reasonable parallel kernel for this problem would replicate the buckets at each processor, with each processor sorting the partition assigned to it using the local buckets (phase I). These buckets are then merged at a single

processor, which then generates the ranks for the keys in the input data (phase II). The algorithm uses barrier synchronization between phases to synchronize the processors. We chose an input size of 4M integers. There is very little communication (non-local data access) in phase I, while phase II involves considerable amount of data communication for merging the replicated buckets.

- Conjugate Gradient Method (CGM) kernel computes the smallest eigen value of a sparse symmetric positive definite matrix. There are alternating phases of parallel and sequential parts in each iteration of this kernel. The computation intensive part of this kernel is the multiplication of this sparse matrix by a vector. The sparse matrix is represented using a row-start, column index format to reduce the amount of data transfer during the vector-matrix multiplication. Each processor is pre-assigned a set of rows of the sparse matrix on which to work. Thus, each processor computes the elements of the result vector assigned to it with very little communication or synchronization with the other processors. The parallel part is followed by a sequential part that uses the result vector in a dot product operation. While there is considerable task granularity during the parallel part, the data movement for the serial part increases with the number of processors used in the algorithm. The matrix size was 14000×14000.

- Matrix multiplication (MM) multiplies two square matrices. The job is partitioned such that each processor computes a set of contiguous rows of the output matrix. The task granularity is large and there could be some amount of false sharing but, since the writes to shared data at a processor display a high spatial locality, it does not interfere with activities at other

processors. The matrices were of size 256×256.

- SOR is an iterative method for solving discretized Laplace equations on a grid. The program is based on the parallel red/black SOR algorithm as described by Chase et. al. [17]. The grid is partitioned among the processors and all the communication occurs between neighboring processors. Only the boundary elements of the grid need to be communicated between iterations. We ran the program for a 512×512 size grid.

- TSP is unique because of the high degree of dynamic behavior of data sharing exhibited by it. The implementation is similar to the one reported by Bal et al. [11] and uses a branch-and-bound method. A set of partial tours are generated and processors evaluate these partial tours in parallel. They all share a work queue that stores the partial tours. The value of the best tour that has been found so far is also shared. If the value of a certain tour being explored exceeds the current best value, the tour is abandoned and the process starts on another pending tour. The application completes when all tours have been explored. To prevent excessive synchronization, processes read the best-tour value without locking the variable, leading to a program that is not data-race-free. The application was run for a 13 city tour.

The six applications have differing types of data sharing characteristics. For instance, EP is characterized by no or very little sharing; MM and IS both have large shared state but spatially dispersed accesses. CGM is an iterative program that exhibits producer-consumer type sharing and also write-write false sharing. SOR is also an iterative algorithm but a processor shares data only with its neighbors. It also exhibits write-read false sharing. Finally, we chose TSP because it has data dependent sharing patterns.

All of these applications, except TSP, are data-race-free. Thus, the same code for these applications was used for both causal and the sequentially consistent memory systems. The same is also true for TSP. This is because the only data race is for reading the best-tour value, and the program still executes correctly if a process reads older values of the best tour variable. Some simple annotations were done to the program data for causal memory. For example, *readonly* data was tagged to reduce the size of the version array in the second implementation of causal memory.

The programming of the message passing implementations of the applications was significantly different. If the process that needed the new data values is known, new values were sent to such a process directly. For example, data produced by the processes in the parallel phase in SOR is sent to the process that executes the sequential part. When the process that needs a new data value is not known because the data item is shared between several processes, it was maintained by a server process. A new value of such an item was sent to the server. Other processes get the new value by communicating with the server.

7.3 Performance Metrics

To quantify the performance of the applications on the memory systems and with message passing, we use several measures. *Completion time* is the total execution time of an application in a given system. Completion times are measured when the application is the only computation in the system and there is no extraneous communication on the network. To gain a better understanding, we also measured the following four component times that define completion time. These represent the costs of the corresponding activities of

the application and are accumulated over the execution of the application.

- **Computation time:** The time spent by the processor actually executing application code. Thus, during this time the processor is not blocked waiting for synchronization, communication or coherence activities to complete. For an application that does not have time or data dependent behavior, this component must be the same on all the systems.

- **Synchronization time:** The time the processor spends blocked on a synchronization call. This time will be zero in the message passing system since processes do not execute any synchronization operations.

- **Communication time:** For the memory systems, this is the time spent in handling page faults and installing pages. The clock is started at a page fault trap and read just before returning from the fault handler. For the message passing system, this is the time spent in the *msgsend* and *msgreceive* system calls.

- **Network Handling time:** This time is spent in responding to network messages (invalidation and forward requests for the memory systems). A processor may get a message while the user process is blocked on a synchronization call or if it has requested a page. In these cases, the time spent in handling the message is not included, since it is accounted for in the other costs. For the message passing case, this is the time spent in handling a message that arrives before the process does a corresponding receive.

Apart from completion time and its four component times described above, we also recorded page fault counts (for the memory systems), the number of messages exchanged, and the size of data communicated in these messages.

A general overview of the results and then a detailed analysis for three of the applications is presented next.

7.4 Results

The applications were run on SUN 3/60's connected over a 10Mbits/s ethernet[2]. The page size, which is the unit of sharing in the memory systems, was 8K bytes. All applications were run using 1 to 8 processors. The same program was run without any synchronization calls to get the time for the single processor case. For causal memory, the results are for the implementation based on versioned pages. Thus, the timestamp had a component for each shared page. The completion times for the two implementations of causal memory were not significantly different and is discussed later.

Figure 24 shows the completion times of the six applications with causal and sequentially consistent memories and also with message passing. The different systems do not have a significant impact on completion times if one or more of the following attributes hold for an application:

1. There is very little sharing or most sharing is by concurrent readers and hence the execution of the application does not result in much communication between processors.

2. The computation granularity is sufficiently large. In this case, the computation time between communication points dominates the time spent in communication.

3. The writes to shared data at different processors are to disjoint parts

[2]The impact of faster processors and networks is discussed in Section 7.5.4

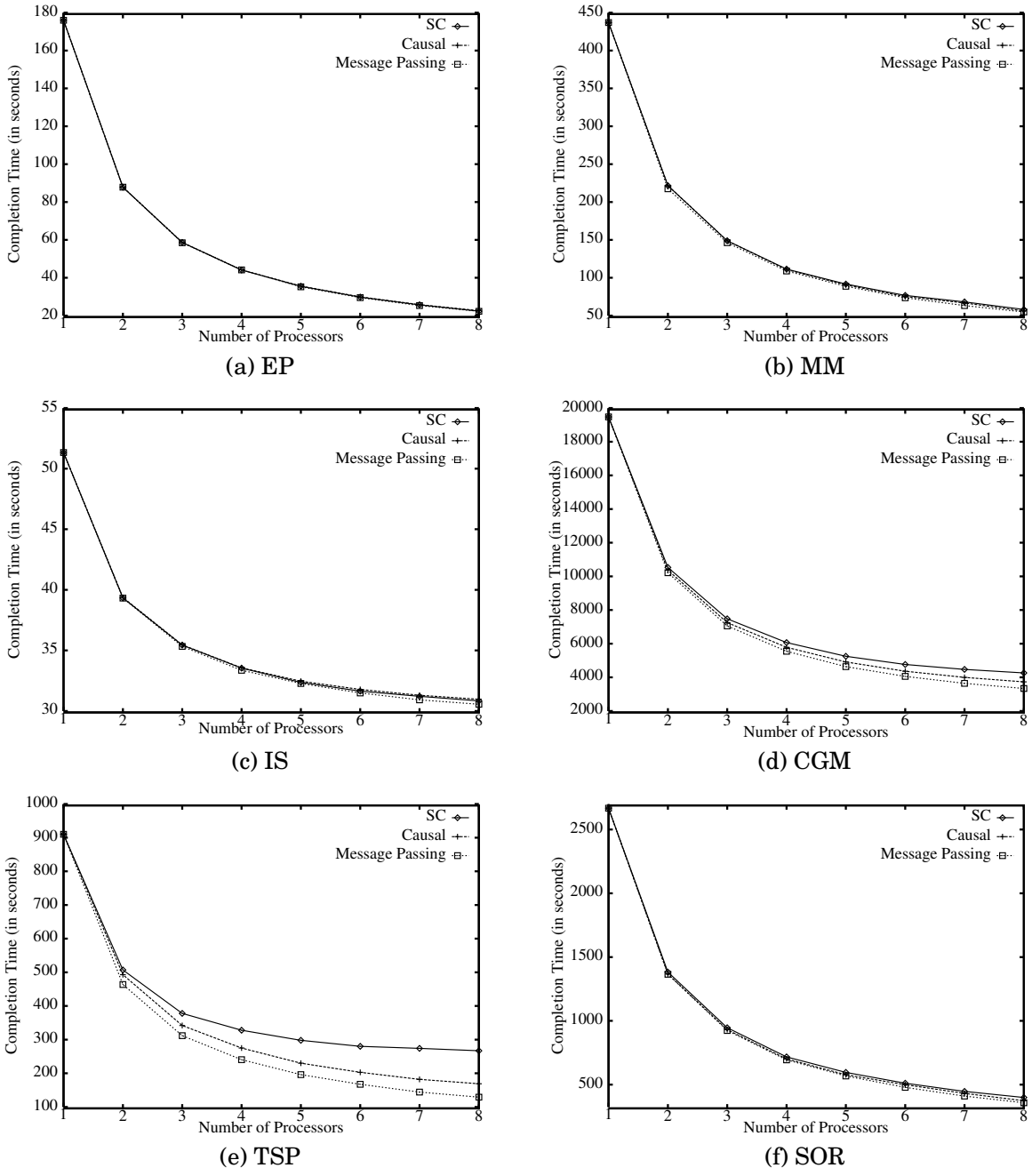


Figure 24: Execution times

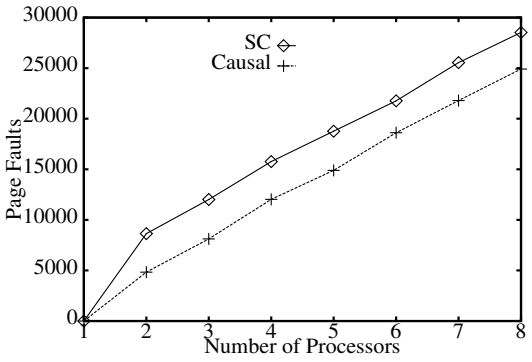
of the data and thus do not interfere with the data accesses at other processors.

EP and IS exhibit the first two attributes while MM exhibits all three. All three applications, EP, IS and MM, give almost identical speedups for the three systems. Both EP and MM show good speedups. IS, because of its large serial fraction, shows poor speedups but its execution time with the memory systems is within 1-2 % of the message passing system. In all the applications, the performance with causal memory is between sequentially consistent memory and message passing. Since the differences between the three systems are appreciable only for CGM, TSP and SOR, these three applications are discussed in more detail in the following subsections.

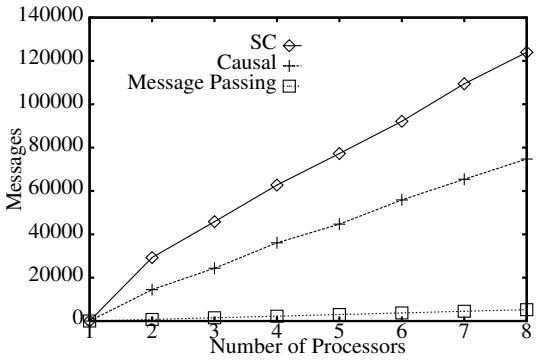
7.4.1 CGM

Data Sharing Characteristics: CGM uses two temporary arrays of floats that are actively shared. The first array exhibits a producer-consumer sharing, where processor p_0 (processors are numbered from p_0 to p_7) writes all elements in the array in the sequential phase, which is followed by a parallel phase in which all processors read the array. The second array is write-shared; all processors write to different parts of the array in the parallel phase and only processor p_0 reads it in the serial phase. The program was run with matrices of size 14000 $\times$ 14000. The temporary arrays are of size 14000.

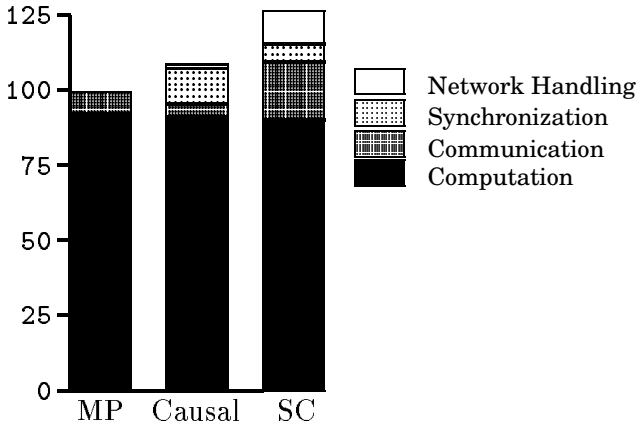
1. **Completion time:** Figure 24(d) compares the performance of CGM on the three systems. While this application on sequentially consistent memory has around 28% higher completion time compared to message passing when the application is executed on 8 processors, the completion



(a) Page Faults



(b) Messages



(c) Execution Profile

Figure 25: CGM analysis

time on causal memory is within 12% of message passing. Thus, causal memory does improve the completion time of CGM over sequentially consistent memory. The execution profile shown in Figure 25(c) breaks down the extra overhead for the memory systems, explaining this difference in completion time. For the 8 processor case, causal memory reduces communication time by 78% compared to sequentially consistent memory. However, with causal memory, the application spends more than twice the time in synchronization calls. This is to be expected since causal memory does all the consistency related actions at synchronization time. The communication time is reduced because causal memory uses local invalidations whereas invalidation messages are sent on writes in the sequentially consistent memory system. This also explains the fact that causal memory has lower network-handling time.

2. **Page Faults:** Figure 25(a) compares the number of page faults on the two memory systems. For 8 processors, the page fault count is around 15% more on the sequentially consistent memory system. In this system, when processors read the array having the producer-consumer data sharing pattern, access to the page at the producer (p_0) is downgraded to *readonly*. In the next iteration the producer has to fault again before it can write the page. In contrast, with causal memory, a producer can write the page without communicating with other processors, as a writer can co-exist with readers. This leads to fewer page faults in the causal memory system. Note that, in the causal memory implementation, a producer processor does downgrade access to *readonly* when another processor gets a copy of the page for reading. However, such protection faults are handled locally and do not result in messages to other processors.

3. **Messages:** Figure 25(b) shows the number of messages sent when the CGM application is executed on the three systems. The message passing system provides a lower bound on the number of messages that need to be sent. The causal memory system sends 41% fewer messages compared to the sequentially consistent system when the application is executed on 8 processors. The message passing system sends significantly fewer messages because the shared array of size 14000 floats (56000 bytes) can be sent as a single message[3] whereas a separate message is sent for each page in the memory systems. As a result, these systems send 7 messages to transfer the array. Although the message counts are significantly different, the amount of data transferred, which is 13.1 Mbytes and 12.9 Mbytes, for sequentially consistent and causal memory systems, is close. Furthermore, in the message passing system, 10.5 Mbytes are sent, which is only about 20% less than causal memory. Both memory systems send more data because they transmit in units of 8 Kbytes, while only the actual data is sent in the message passing system.

7.4.2 TSP

Data Sharing Characteristics: In TSP, two data structures are shared between processors. The first is a global shared queue. The actual queue is shared in *readonly* mode and only a next-job pointer, which points to the job in the queue that has to be searched next, is written to when processors choose jobs to work on. The other shared data item is a best-tour variable which is both read and written. The best-tour value is typically cached at all processors, which read it to compare it with the current tour value and is updated by a

[3]The underlying protocol could fragment this message but we are counting only the number of times the protocol is invoked to send a message.

processor only if it has found a better tour.

The behavior of TSP on the three systems is very different. With sequentially consistent memory, the best-tour value gets propagated immediately to all processors whenever it gets updated, since any cached value would be invalidated before the write. Causal memory allows out-of-date values of best-tour because a writer can co-exist with readers. A processor gets a new value of best-tour only when the page containing its old value gets locally invalidated as a result of a synchronization operation. This is done when a processor chooses the next job to be searched (since the queue is shared, a lock has to be acquired before the processor chooses the next job). The message passing implementation of TSP has a server that maintains the work queue and the best-tour value. Whenever a processor needs new work or if it has found a better tour, it communicates with the central server. As the best-tour value does not get propagated to all processors immediately in the message passing version of TSP, it may search more nodes in the search tree than sequentially consistent memory. This extra computation overhead could become significant if we allow processors to continue with very old values of best-tour[4]. This was observed in the experiments that we ran. The number of nodes searched in the three cases are shown in Figure 26. The number of nodes searched by the message passing code is 7% higher than sequentially consistent memory. The various component times for TSP below are discussed below.

1. **Completion time:** Figure 24(e) shows the performance of TSP with the three systems. Its execution on the sequentially consistent system, with 8 processors, takes 102% more time than the message passing system.

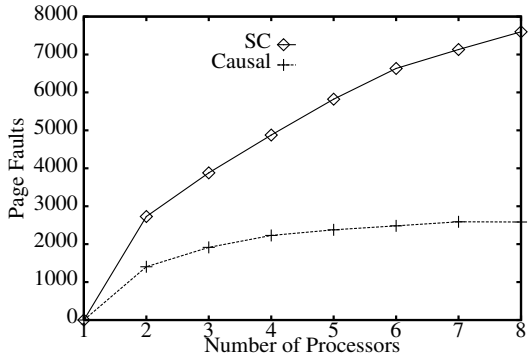
[4]We can make the sender transmit new best-tour values as soon as they arrive but this does not solve the problem. The other processors must receive these values and due to the asynchronous nature of when these values arrive, one cannot code points in the program where a *msgreceive* should be executed.

Protocol	SC DSM	Causal DSM	Message Passing
Nodes Searched	2,226,682	2,383,553	2,386,386

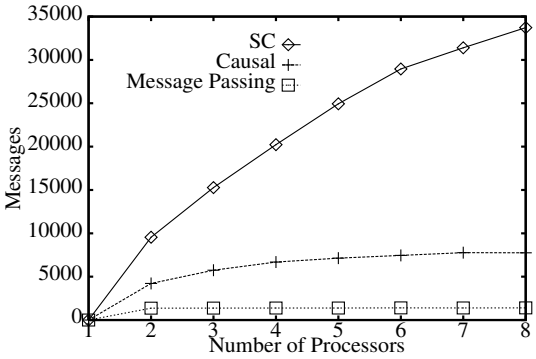
Figure 26: Nodes visited in TSP

In contrast, its completion time with causal memory is within 28% of the message passing time. Although more nodes are searched when TSP is executed on causal memory (see computation time in Figure 15(c)), it has significantly lower communication and synchronization times than the sequentially consistent system. Causal memory has lower synchronization costs, since it reduces the contention for the synchronization variables. The contention is reduced, since the lower communication cost for maintaining causal consisteny actually reduces the length of the critical section (servicing a page fault which happens inside a critical section is cheaper for causal memory. The sequentially consistent memory has to send invalidation messages to all the processors, since all processors would have a copy of the data). Message passing provides better completion time as it does not have the synchronization overhead. Also, the memory systems suffer because of the mismatch between the amount of shared data and the page size, which is the unit of coherence.

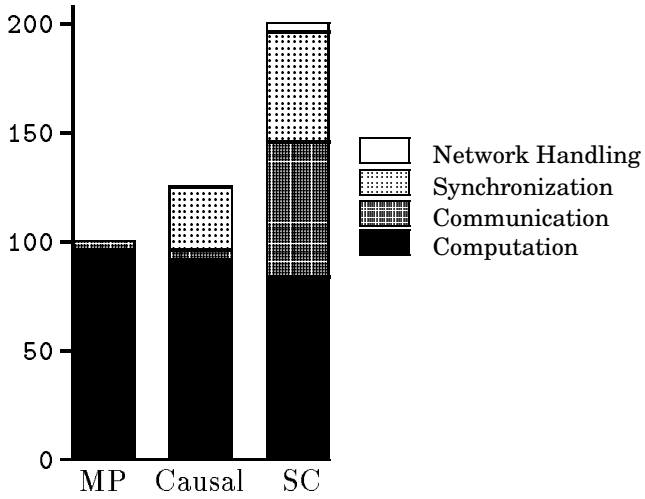
2. **Page faults:** The causal memory system has significantly fewer page faults compared to the sequentially consistent memory system. For example, for 8 processors, causal memory has almost two thirds less number of page faults. This is easily explained. In sequentially consistent memory, whenever a new best-tour is found, its update results in invalidation messages being sent to all processors since this variable is used and



(a) Page Faults



(b) Messages



(c) Execution Profile

Figure 27: TSP analysis

cached at all processors. A subsequent access to best-tour will generate a page fault at all processors. With causal memory, other processors can continue to read old values and the new value is requested by them only when an acquire operation on a lock variable is executed to find the next job.

3. **Messages:** Figure 27(b) compares the number of messages sent in the three systems when the TSP application is executed. The sequentially consistent system requires significantly more messages because invalidation messages are sent whenever the next-job and the best-tour variables are updated. For 8 processors, the number of messages sent is 33723, 7913, and 1403 for sequentially consistent, causal and message passing system. Another interesting observation is that the number of messages exchanged does not increase after 3 processors for causal memory and after 2 processors for message passing because the same number of jobs are searched by the processors. In contrast, the message count increases almost linearly in the sequentially consistent system. This, again, is due to the invalidation messages that are sent to all processors. Causal memory never requires more than 3 messages to complete a memory request, which explains why the number of messages does not increase as more processors execute the application.

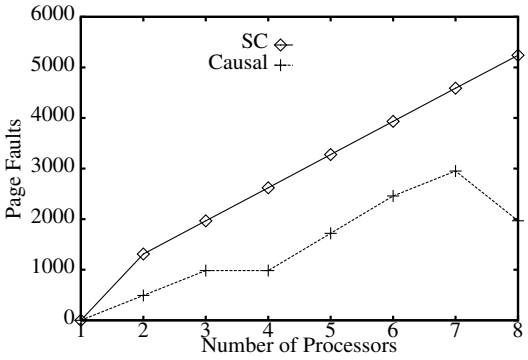
   As mentioned earlier, in both memory systems, the unit of coherence is a page and hence 8K bytes are transferred even when two integers (current-job and best-tour) are shared. This explains why the message passing system sends around 18 Kbytes of data, whereas the sequentially consistent and causal memory systems send 48.8 Mbytes and 20.2 Mbytes of

data respectively. As seen in Figure 27, avoiding unnecessary data transfer does result in better completion time for message passing because it reduces the communication time significantly.

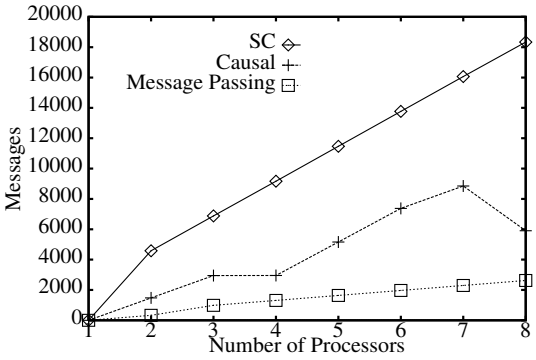
7.4.3 SOR

Data Sharing Characteristics: The data sharing pattern in SOR is quite different from the applications discussed so far. The computation consists of a sequence of iterations. Each iteration consists of two phases, an odd phase and an even phase, which are separated by barriers. Each processor computes grid elements that are assigned to it (the grid is partitioned horizontally and each processor is assigned equal number of elements). The computation of a grid element requires the reading of its four neighbor elements. Thus, there is a producer-consumer data sharing pattern because one processor reads the values of grid elements written by another processor (this is true only for boundary elements). Also, only the processor that is assigned a partition writes to the elements in its partition; neighbors only read the elements in this partition. The two phases in an iteration help avoid the synchronization that will be necessary before reading the elements of neighbor processors.

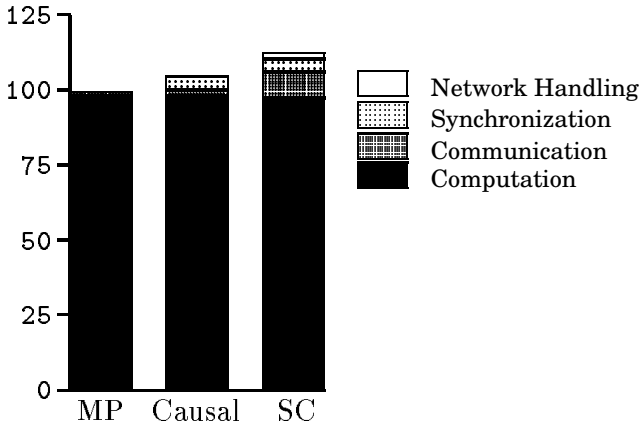
1. **Completion time:** Figure 24(f) shows the completion times for SOR on the three systems for a 512×512 grid. The completion times are not significantly different on the three systems. For example, with 8 processors, the completion time with sequentially consistent memory is within 10% of the message passing time. The completion time for causal memory is almost the same as message passing (2% difference for 8 processors). As can be seen from Figure 28(c), the computation time dominates the completion time and, since it is the same in all three systems, the completion



(a) Page Faults



(b) Messages



(c) Execution Profile

Figure 28: SOR analysis

times are not significantly different. The sequentially consistent memory system does have higher communication time because it generates additional page faults. The small difference between message passing and causal memory is due to the synchronization overhead on causal memory.

2. **Page faults:** Figure 28(a) shows the number of page faults on the two memory systems. SOR on causal memory takes 62% fewer page faults compared to the sequentially consistent memory when the application is run on 8 processors. This is for two reasons. First, because of the producer-consumer nature of data sharing, a processor will always be able to write pages in its partition without requiring communication with other processors in the causal memory system. Although page access will be downgraded to *readonly* when the neighbor processor gets a copy of the page for reading, this will result only in a protection fault which is handled locally. In contrast, in the sequentially consistent memory system, such a fault requires communication with the neighbor processor whose copy of the page has to invalidated.

   The second reason why the sequentially consistent memory system has higher communication time is because it transmits more pages due to faults. In fact, it generates four faults in each iteration (twice during each phase) to get the boundary elements whereas, with causal memory, only three faults are generated. Figure 29 explains this difference. In the first phase of the iteration, because of the order in which processor p_i computes the grid elements assigned to it, p_i first read faults and receives page x_{i-1} from its left neighbor, processor p_{i-1}. At the very end of the phase, p_i read faults again and receives page x_{i+1} from its right

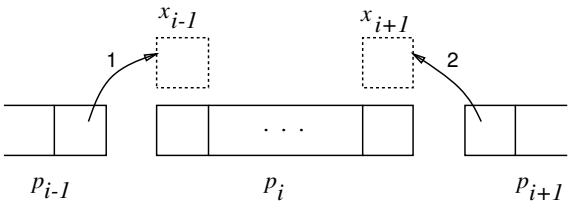


Figure 29: Data sharing for SOR

neighbor, processor p_{i+1}. By this time, processor p_{i+1} has already finished computing the new values of the elements on page x_{i+1} that will be read in the next phase by p_i. Notice that the writing by p_{i+1} and reading by p_i do not result in data races because different elements on the page are accessed by the two processors. In causal memory, when the processor arrives at a barrier after these data accesses in the first phase, the version of x_{i+1} that p_i caches has the same version number as p_{i+1} and is thus not invalidated. Page x_{i-1}, however, does get invalidated as it should, since it was received from processor p_{i-1} before p_{i-1} wrote it. In the next phase, p_i read faults on page x_{i-1} but reads the cached copy of x_{i+1}. Thus, only three faults are generated in the two phases. At the barrier after the second phase completes, both pages will be invalidated because they have been written again by the neighbor processors is the second phase. In the sequentially consistent system, the computation of the second phase will generate two faults. This is because, x_{i+1} will be written by p_{i+1} as soon as the second phase starts, which will result in an invalidation at p_i. This invalidation is a result of false sharing because, in the second phase, p_i does not read the values written by p_{i+1} in this phase. Thus, when p_i reads x_{i+1} towards the end of the second phase, an extra page

fault will be generated and hence a total of four faults per iteration are experienced. Causal memory has one less fault per iteration because it allows a writer to co-exist with readers.

3. **Messages:** Figure 28(b) compares the number of messages sent by the three systems while executing SOR. Causal memory sends 68% fewer messages than sequentially consistent memory because of fewer page faults and the fact that invalidations are local. The amounts of data transferred in the messages in sequentially consistent, causal and message passing systems are 20.6 Mbytes, 15.4 Mbytes and 5.1 Mbytes respectively. The memory systems send more data due to the mismatch between the data granularity (512 elements of floats — 2048 bytes) and the large page size (8192 bytes).

7.5 Discussion

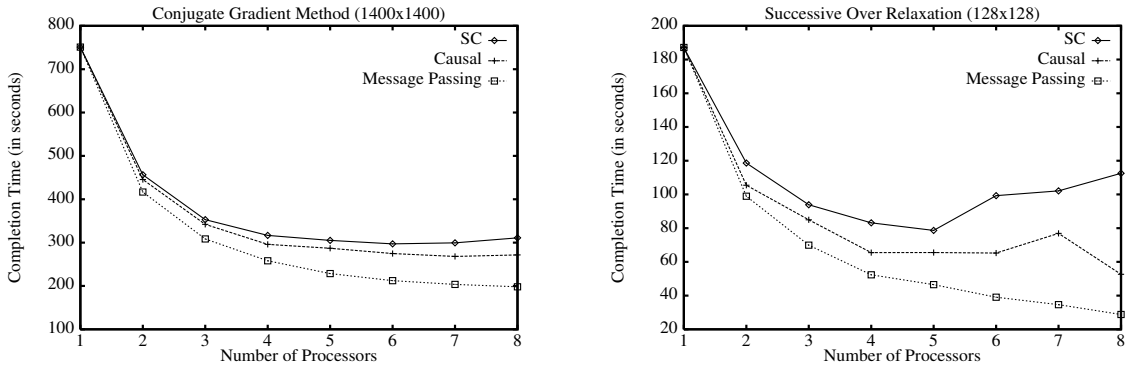
In the results, it can be seen that causal memory performs better than sequentially consistent memory system. This is due to two primary reasons: it tolerates false sharing between readers and a writer, and it sends fewer messages. We discuss both of these issues and also comment on performance of causal memory when vector timestamps instead of version numbers are used in the implementation. Causal memory is compared with other memory systems in Section 7.6.

7.5.1 False Sharing

To study the effects of false sharing, we ran CGM and SOR with several problem sizes. While CGM illustrates the effects of write-write false sharing on the

performance of the memory systems, SOR shows how the systems handle read-write false sharing. Figure 30(a) shows the completion times for CGM when the problem size is 1400 (the result discussed earlier were for size 14000). This size was chosen so that the array which is write-shared would fit in one page. Consequently, when the processors concurrently write to different parts of the array, the same page would experience write-write false sharing. Both memory systems suffer and have much higher completion times than message passing because the writes to the single page get serialized. Message passing allows these writes to be done in parallel which leads to much better performance. For example, with 8 processors, the percentage difference between causal memory and message passing is 37%. This difference was only 12% when the problem size was 14000 because writes to different pages could be done in parallel.

SOR was run with a grid size 128×128, which results in only 8 pages of shared data (the earlier results were for a 512×512 grid). For the 8 processor case, each processor would compute and write elements on exactly one page. Thus, there is no write-write false sharing. However, there is read-write false sharing because some of the elements read by a processor are written by its neighbor. As shown in Figure 30(b), causal memory has much better completion time than sequentially consistent memory because it does not require any communication between processors when there is read-write false sharing. For the 8 processor case, causal memory has 53% lower completion time. The completion time with causal memory does not decrease monotonically as the number of processors is increased. This is because for some number of processors, the data partitioning leads to load imbalances (due to the fact that a processor's partition could lie across different number of pages) and write-write false sharing.



(a) CGM (b) SOR

Figure 30: Effect of false sharing

7.5.2 Causal Memory Implementations

All the applications were run with both the implementations of causal memory that have been described in Chapter 6. The implementation based on vector timestamps that have a component for each processor does suffer from the problem of unnecessary invalidations. As a result, its performance is not as good as the implementation based on page versions (the results discussed so far are for this implementation of causal memory). Since the state sharing method has minimal impact on the performance of EP, IS and MM, we only show the completion times for CGM (size 1400x1400), TSP (13 cities) and SOR (size 512x512) for the two implementations of causal memory in Figure 31.

As seen in the table, the applications take between 1% to about 12 % more time to execute with the vector timestamp based implementation compared to the version based implementation. For these application sizes, the size of the shared data space which was actively shared for CGM, TSP and SOR was 2, 1 and 128 pages respectively. These small sizes favor the version based

	Vector Timestamp			Version Pages		
	# of Processors			# of Processors		
	2	4	8	2	4	8
CGM	455.96	316.56	291.08	445.46	295.85	271.64
TSP	507.79	292.02	188.84	493.13	275.64	169.29
SOR	1370.51	706.45	378.03	1366.88	701.60	373.77

Figure 31: Comparing the two implementations

implementation. The vector timestamp based implementation also suffers from unnecessary invalidations in TSP and SOR.

7.5.3 Scalability

The size of the experimental test-bed limited us to only 8 processors. We can extrapolate the behavior of causal memory for a larger size system. We believe that causal memory provides a more scalable implementation of DSM because any memory access can be completed by exchanging at most three messages. Thus, a constant number of messages are exchanged even when a page cached by many processors is written. We see this behavior of causal memory in Figures 25, 27, and 28 where we show the number of messages for CGM, TSP and SOR. In a sequentially consistent memory system (and also in a release consistency system – see Section 7.6), the number of messages required to complete a memory operation can increase with system size because all processors may have a page copy that has to be invalidated.

Although the communication required for completing a memory access does not increase with system size in causal memory, the size of the vector timestamps or version vectors could limit the scalability of the implementations. Since version vectors only need to be sent with synchronization variables, we

believe that this is not a problem for data-race-free programs. Furthermore, there exist techniques that can be used to limit the information carried in timestamps [57] by maintaining at each processor how far the clocks at other processors have advanced.

The performance benefits due to better scalability of causal memory or message passing are not easily seen for the applications and their sizes that we used in the study. This is because in all of them, the completion time curves become flat because of reduced computation granularity as the number of processors are increased. Thus, a large number of processors will only be useful when the application has very large computation granularity.

7.5.4 Impact of Faster Processors and Network

The experimental test-bed consisted of Sun 3/60 machines connected by a 10Mbits/s ethernet. An obvious question is if the differences in the three systems would persist with faster processors or when the processors are connected by a high speed network such as an ATM. It can be easily argued that the results will still be valid with increased CPU speed because that will reduce computation time which will shift all the completion time curves down. Furthermore, causal memory will experience further improvements because synchronization time will also be reduced. This is because on certain synchronization operations, causal memory incurs considerable processing overheads. A faster network will make the difference among the different systems less significant. This is because the memory systems that send large messages (pages) will benefit more from the increased network speed [56]. The memory systems will also become more competitive with message passing in architectures that have smaller page sizes or support multiple page sizes.

7.6 Comparison with Related Work

In this section, we compare our implementation with other related work that have been proposed for implementing DSM systems. Our focus is on quantifying the performance gains made possible by the different schemes. The first implementation is the sequentially consistent DSM, described earlier in Section 7.1, which we will refer to as M_0. The causal version vector implementation will be referred to as M_1. We chose to compare ourselves with an implementation similar to release consistency [46] (M_2) and an implementation similar to entry consistency [8] (M_3), as these protocols were available on the system.

M_2 is a weakly ordered system which defers consistency actions to certain synchronization points. All modifications to a page are done to a shadow copy transparently to the program. Prior to exiting a synchronization region, an XOR of the shadow copy is done with the original page to generate the modifications done during the synchronization region (similar to *diffs* in [13]). These modifications are then sent to all processors caching a copy of the page. M_2 allows multiple processors to actively write-share a page, thereby avoiding the penalties due to false sharing. The implementation of M_2 is described in detail in [46].

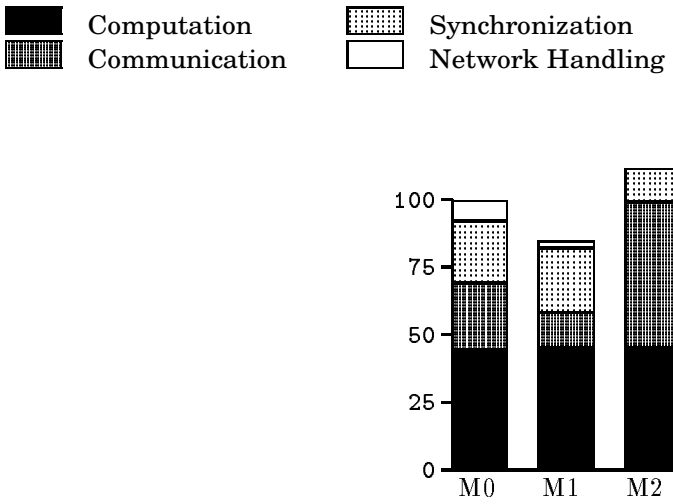
In M_3, consistency actions make use of explicit associations between shared data and the synchronization variables that control access to such data. When a processor acquires a synchronization variable, it also gets with it the data associated with the synchronization variable. M_3 is a non-page-based system and does not suffer from false sharing. M_3 is described in detail in [8].

7.6.1 Experimental Evaluation

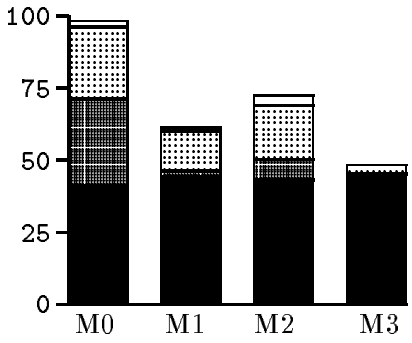
In order to understand the differences among the DSM implementations, we profiled the execution times for the three applications, CGM, TSP, and SOR, which have a substantial state sharing overhead. We chose smaller sizes for CGM and SOR so that the differences between the systems would be more pronounced (the computation component does not dominate).

The execution profiles for a representative processor is shown in Figure 32. The profile is for the case where the applications were executing on 8 processors. The completion time for the applications on the memory systems have been scaled such that the time on M_0 is 100.

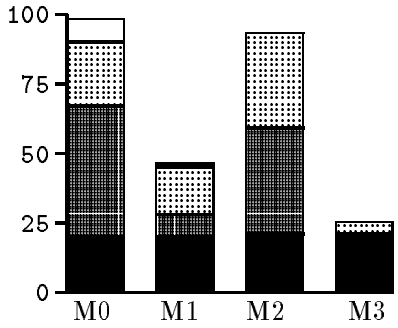
The goal of the different protocols is to reduce the amount of communication time. In the case of M_3, the modified data is transferred with the associated synchronization variable. Thus communication and synchronization are intertwined and lumped into one component, namely, the synchronization one in M_3. The causal implementation, M_1, has significantly lower communication time compared to M_0 and M_2. This is because M_1 uses local invalidations and hence consistency actions do not result in communication. In general one would expect higher communication time for M_0, owing to the increased number of page faults incurred due to invalidations on write operations. In M_2, modified data is sent by each processor only prior to exiting a synchronization region. In spite of this, M_2 displays the highest communication time of all the memory systems for CGM due to two reasons: first there is a large number of messages ($O(n^2)$) as each processor has to send its modified data to all the other processors, and second there is increased contention at the data manager (which has to propagate the diffs) due to the simultaneity of these messages. For TSP, communication time is the highest for M_0 as expected. M_1 has very



(a) CGM



(b) TSP



(c) SOR

Figure 32: Execution profiles (scaled to 100 for M_0)

low communication time because it works with old tour values and hence updating the tour value does not result in communication. The communication time for M_2 is higher than M_1 but significantly lower than M_0 even though the best tour value has to be sent to all the other processors. This is because the size of shared data is very small (e.g., 4 bytes for best tour) and M_2 sends only the *diffs* thus resulting in very small messages. Unlike CGM, SOR requires only near neighbor communication. Thus we would have expected that M_2 would incur significantly less communication compared to M_0 for SOR. However, we observe that the communication times for both systems are very close for SOR. This is because every alternate location in a page gets modified in each iteration thus negating any advantage of using diffs for reducing the size of the messages. M_1 has much lower communication time since it allows a single writer to be concurrent with multiple readers thus handling the write-read false sharing which is inherent in the SOR application.

The synchronization time component accounts for the time a process spends while waiting for a synchronization operation to complete. This time not only depends on the cost of communicating with a remote synchronization server but also on contention for a synchronization variable (e.g., a *lock*). Contention for the lock depends on server load and the time a process is inside the critical section (which could be quite different for the protocols due to the process page faulting inside the critical section). M_3, has the lowest synchronization time due to lower contention since processes never page fault while holding locks. M_1 has the extra overhead of doing the local invalidations on acquiring a synchronization variable. The large synchronization component for M_2 in SOR is because of contention at the manager processor which propagates the diffs. Thus a process arriving at the barrier has to wait until all other processors have propagated their changes.

The execution of CGM and SOR are not data dependent and, as a result, all four DSM systems have the same computation time for these applications. TSP execution is data dependent since the best tour value is used to prune bad tours. In M_0, a new value of best tour propagates to other processors as soon as the new value is written. In $M_1 - M_3$, processors could continue to cache an old value of the best tour variable even though some processor has found a better tour. As a result, the computation time for these systems is more than M_0 as they evaluate certain redundant tours.

M_0 has the highest network handling time for the applications. This is expected since it sends out invalidation messages. In M_2, the update messages are mostly received when the receiver is also waiting for its modifications to be propagated (as in the case of CGM and SOR where all the processes arrive at the barriers around the same time). Thus, the time spent in processing the message is not accounted in the network handling time.

M_3 provides performance very close to that possible through message passing. But M_3 complicates the programmers task by requiring that the programmer explicitly associate data with synchronization variables. Also, associating data with synchronization constructs like barriers, which are used for sequence control, seems non-intuitive. In contrast, M_1 requires very little additional help from the programmer and performs fairly close to M_3 considering the small sizes of the applications.

7.7 Concluding Remarks

In this chapter, we did a detailed experimental evaluation of the causal memory implementations and compared it with a sequentially consistent DSM system and a message passing system. To compare their performance, we ran six

different applications on the systems. The causal implementation reduced the communication overhead for three of the applications by as much as 70% – 90% compared to the sequentially consistent DSM. The performance of five of the applications running on causal memory came within 12% of the execution time on a message passing system.

Further, we compared the causal implementation to two other weakly ordered DSM implementations which were available on the Clouds operating system. The results from these experiments show that the causal implementation allows good performance with very little additional help from the programmer.

Chapter 8

Conclusions and Future Work

It is generally believed that DSM's provide a simple abstraction to program applications on distributed systems. However, performance of DSM systems has not matched the performance of message passing systems, where the programmer has explicit control over data placement and data movement. Traditional DSM implementations adapted a cache consistency protocol developed for shared-memory multiprocessors and implemented it in software on a network of workstations. The basis of this dissertation is that a distributed system differs too significantly from a shared-memory multiprocessor for such an approach to work well. This dissertation explored a weakly consistent memory model called causal memory. This chapter concludes the dissertation and offers suggestions for future research.

8.1 Conclusions

This dissertation shows that weakening the consistency allows scalable and high performance DSM systems to be built without adversely affecting programming. Causal DSM can be easily programmed because most applications can be developed assuming a sequentially consistent memory model. We show that if a program is data-race-free, its execution would be correct on causal memory. Further, if a program has data races, it would execute correctly

if there are no concurrent write operations. The only programs which we found that do not execute correctly are software solutions to the mutual exclusion problem which require the stronger guarantees of sequential consistency. Typical applications rarely exhibit the data sharing patterns of these mutual exclusion algorithms.

The key to building a scalable software DSM system is to reduce the number of messages required to maintain consistency of data. We showed that in the causal implementations, the number of messages generated to maintain consistency of data is independent of the number of processors in the system. The number of messages is reduced at the cost of some longer length messages and more processing to determine causally over-written pages. This tradeoff is appropriate considering the rate at which processor speeds and network bandwidths are increasing.

Two implementations of a causally consistent memory were outlined. These were actually implemented in the kernel of the Clouds distributed operating system. To compare the performance of causal memory, we implemented two other systems - a sequentially consistent DSM and a message passing system. Experimental evaluations confirm our earlier objectives that causal consistency provides a viable base for efficient implementations of DSM. The causal protocol reduced the communication cost by as much as 70 – 90% compared to the sequentially consistent DSM for applications which had a significant state-sharing overhead.

8.2 Future Work

In this section, we discuss enhancements to the implementations that have been described in the earlier chapters and future directions for research.

8.2.1 Synchronization

In this thesis, we concentrated on reducing the time spent by an application on communication. Synchronization was achieved in our experiments using a simple central synchronization server. In all the applications that we tested, the performance difference on the causal DSM compared to the message passing implementation was primarily because of the synchronization overhead. We plan to implement a distributed synchronization server which would reduce the contention compared to the central server solution. Also, if more knowledge is available about the applications, the synchronization modes can be weakened. For instance, assume that we have a program which has no false sharing. In this case, we can grant a write lock request even when there are other processes with pending read locks. This is possible since we allow a single writer to co-exist with readers. The writing process would be writing to a copy of the data in its cache while the reading processes are reading the older version of the data.

8.2.2 Fine Granular Sharing

The system that we did our experiments on, supported a page size of 8192 bytes. The mismatch between this large page size and the actual shared data size lead to much extraneous communication for some of the applications. Support for multiple page sizes is one way of reducing the amount of data transmitted. Several CPU's today provide support for multiple page sizes but operating systems have not made this feature available to the users as it would complicate several operating system components. Another approach is to have the smarts in the compiler to support multiple page sizes in software. We propose to investigate these issues further.

8.2.3 Hardware Support

Another area where this work can be extended is to look at the hardware support which could reduce the cost of consistency maintenance. Specifically the directory maintenance which was done by a static manager could be done by a global physically addressed memory. A network processor could handle the network messages and providing it with direct memory access to the physical memory at the processor would eliminate the network handling overhead. Experience with more applications is required to see if the network handling costs justify the cost of the extra hardware.

8.2.4 Size of Timestamps

The size of the timestamp could limit the scalability of the causal protocols. In the vector timestamp protocol, the timestamp is the size of the number of processors in the system. Much research has already been done to control the size of the timestamps [57]. It needs to be seen whether these techniques can be used in our implementation. The size of the version vector is proportional to the size of the shared address space. The maximum size of the version vector that we encountered for the applications, after annotating program data, was 512 bytes. We need to get experience with more applications to get an understanding of the typical sizes of the version vector.

Bibliography

[1] SUN RPC reference manual. Sun Microsystems Ltd., Mountain View., 1985.

[2] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[3] Divyakant Agrawal, Manhoi Choy, Hong V. Leong, and Ambuj K. Singh. Mixed consistency: A model for parallel programming. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, 1994.

[4] Mustaque Ahamad, Rida Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, 1993.

[5] Mustaque Ahamad, James E. Burns, Phillip W. Hutto, and Gil Neiger. Causal memory. In *Proceedings of the 5th International Workshop on Distributed Algorithms*, October 1991.

[6] Mustaque Ahamad, Phillip W. Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, May 1991.

[7] Mustaque Ahamad, Gil Neiger, Prince Kohli, James E. Burns, and Phillip W. Hutto. Causal memory: Definitions, implementation and programming. Technical Report GIT-CC-93-55, College of Computing, Georgia Institute of Technology, 1993.

[8] R. Ananthanarayanan. *High Performance Distributed Shared Memory*. PhD thesis, Georgia Institute of Technology, In Preparation.

[9] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(4), August 1994.

[10] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS parallel benchmarks. Technical Report Report RNR-91-002, NAS Systems Division, Applied Research Branch, NASA Ames Research Center, January 1991.

[11] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Experience with distributed programming in Orca. In *International Conference on Computer Languages*, 1990.

[12] Gérard M. Baudet. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*. PhD thesis, Carnegie-Mellon University, April 1978.

[13] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, May 1990.

[14] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In

Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming, March 1990.

[15] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, September 1991.

[16] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, August 1991.

[17] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th Symposium on Operating System Principles*, 1989.

[18] D. Cheriton. The V distributed sytem. *Communication of the ACM*, March 1988.

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

[20] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, 1993.

[21] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, June 1986.

[22] L. J. Fidge. Timestamp in message passing systems that preserves partial ordering. In *11th Australian Computer Conference*, February 1988.

[23] B. D. Fleisch and G. J. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the ACM Symposium on Operating System Principles*, 1989.

[24] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[25] P. Gibbons, M. Merritt, and K. Gharachorloo. Proving sequential consistency of high-performance shared memories. In *3rd ACM Symposium on Parallel Algorithms and Architectures*, 1991.

[26] James R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, University of Wisconsin, Madison, February 1991.

[27] R. Guerraoui, B. Garbinato, and K. R. Mazouni. The GARF library of DSM consistency models. In *6th SIGOPS European Workshop on "Matching Operating Systems to Application Needs"*, September 1994.

[28] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *1993 Summer Usenix conference*, June 1993.

[29] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages*, 12(3), July 1990.

[30] Phil W. Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the International Conference on Distributed Computing Systems*, 1990.

[31] Ranjit John, Mustaque Ahamad, Umakishore Ramachandran, R. Ananthanarayan, and Ajay Mohindra. An evaluation of state sharing techniques in distributed operating systems. Technical Report GIT-CC-93-73, College of Computing, Georgia Institute of Technology, Atlanta, 1993.

[32] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *7th ACM Symposium on Principles of Distributed Computing*, 1988.

[33] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, 1992.

[34] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, January 1994.

[35] R. E. Kessler and M. Livny. An analysis of distributed shared memory algorithms. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, 1989.

[36] Prince Kohli, Gil Neiger, and Mustaque Ahamad. A characterization of scalable shared memories. In *Proceedings of the 22nd International Conference of Parallel Processing*, August 1993.

[37] Leslie Lamport. Time, clocks and the ordering of events. *Communications of the ACM*, 21(7), July 1978.

[38] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, c-28(9), September 1979.

[39] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM TOCS*, 7(4), November 1989.

[40] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, 1993.

[41] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, September 1988.

[42] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *11th ACM Symposium on Operating System Principles*, November 1987.

[43] B. Liskov and R. Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. In *5th ACM Symposium on Principles of Distributed Computing*, 1986.

[44] F. Mattern. Time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, 1989.

[45] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1), January 1986.

[46] Ajay Mohindra. *Issues in the Design of Distributed Shared Memory Systems*. PhD thesis, Georgia Institute of Technology, 1993.

[47] S. J. Mullender and A. S. Tanenbaum. The design of a capability based distributed operating system. *The Computer Journal*, 29(4), March 1986.

[48] B. J. Nelson. Remote procedure call. Technical Report CSL-81-9, Xerox Palo Alto Research Center, 1981.

[49] D. S. Parker. Detection of mutual consistency in distributed systems. *IEEE Transactions on Software Engineering*, May 1983.

[50] U. Ramachandran, Mustaque Ahamad, and Y. Khalidi. Coherence of distributed shared memory: Unifying synchronization and data transfer. In *Proceedings of the International Conference on Parallel Processing*, August 1989.

[51] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. CHORUS distributed operating system. *Computing Systems*, 1(4), 1988.

[52] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, June 1987.

[53] A. Schiper, J. Eggli, and A. Sandoz. A new algorithm to implement causal ordering. In *3rd International Workshop on Distributed Algorithms*, 1988.

[54] Abraham Silberschatz and James L. Peterson. *Operating System Concepts*. Addison-Wesley, 1988.

[55] SUN. *The SPARC Architecture Manual*. Sun Microsystems Inc., No. 800-199-12, Version 8, January 1991.

[56] Chandramohan A. Thekkath and Henry M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2), May 1993.

[57] Gene T. J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, 1984.

Vita

Ranjit John was born in Allepey, Kerala, India on September 12, 1966. He graduated from St. Xaviers High School in Bokaro, India in 1984 where he won the National Talent Search scholarship to pursue undergraduate studies. With the scholarship, he attended the Birla Institute of Technology and Science at Pilani where he was admitted for a degree in Physics. After a year, he added Computer Science as a second major. In 1985, he wrote his first computer program which was on punch cards for an IBM 1130. By the time the 1130 was sold as scrap, his first program still had not compiled.

In 1989, he joined the PhD program in Computer Science at the Georgia Institute of Technology in Atlanta. During this time he worked on improving his squash game which paid off when he reached the semi-finals of the Georgia Tech squash tournament in 1994.

The author received an MS in Computer Science from Georgia Institute of Technology in 1991.