Causal Memory: Implementation, Programming Support and Experiences

Ranjit John Mustaque Ahamad

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280 USA
e-mail: rjohn@cc.gatech.edu

GIT-CC-93-10

Abstract

Distributed Shared Memory (DSM) has become an accepted abstraction for programming distributed systems. Although DSM simplifies the programming of distributed applications, maintaining a consistent shared memory can be expensive.

Weakly ordered systems which use synchronization information have been proposed to reduce the frequency of communication between processors. We have implemented a weakly ordered system based on the Causal memory model. We provide language and runtime support which allow programs to run efficiently on Causal memory.

Actual implementation results show a significant reduction in the number of messages when compared to a system maintaining a consistent shared memory.

Keywords: Distributed Shared Memory, Weakly ordered systems, Causal memory

1 Introduction

Distributed Shared Memory (DSM) has become an accepted abstraction for programming distributed systems. DSM simplifies programming of distributed applications since the user need not distinguish between local and remote memory operations. This simplification is at the cost of maintaining a consistent shared memory. Most DSM implementations use variants of multiprocessor cache consistency algorithms, which however perform poorly in distributed systems, where the message latencies are much higher.

A program interacts with the memory through a sequence of reads and writes. The DSM is an interface between the program and the memory which provides an ordering on the reads and writes, consistent with the memory model chosen. Ideally a distributed shared memory should provide all the consistency guarantees of a true shared memory. Lamport [23] defined a memory model called *sequential consistency* which provided such properties. Atomic memory [27] is a stronger memory model which requires that the memory maintain the real time order in which the reads and writes occur. Kai Li's Ivy system [24] is based on a writer invalidate-readers protocol which implements atomic memory. Such implementations maintain the real time order by restricting a page (which is the smallest unit of sharing) to a single writer or multiple readers at the same time.

In [13, 3] delayed invalidations and buffered writes are used to capture more efficient executions permitted by sequential consistency. Maintaining sequential consistency on a network of distributed machines, can be shown to limit performance and does not lend itself to scaling [25].

Dubois et al. [16] observe 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 only performed at synchronization points. The DASH multiprocessor [18] is a weakly ordered system which implements a memory model called Release Consistency. Release Consistency allows remote memory accesses to be propagated asynchronously, as long as they complete by the end of the critical section. Such systems guarantee sequentially consistent behavior only for programs which do not have data races [1]. The Munin system [9] implements Release Consistency in software, by delaying propagating the changes made inside a critical section till the the lock is released. Entry Consistency [10] and Lazy Release Consistency [20] reduce communication further by propagating changes only to the processor which acquires the lock.

We explore a *weakly ordered* memory system, based on Causal memory [5]. The Causal memory model requires that a read of a location return a value that is consistent with all causally related reads and writes to that location. We extend the synchronization operations and implement a programming system supporting the Causal memory model, which allows a user to exploit the benefits of a weakly ordered system. The language and runtime mechanisms in conjunction with the DSM system provides demonstrably improved overall system performance.

The goals of this paper are to show:

- Causal Memory can be efficiently implemented and is a viable architecture since most programs would port without change in the code.
- Causal memory requires far less communication compared to implementations where a consistent DSM is maintained.
- Scalable shared memory systems can be built using the Causal memory model, since global synchronization (e.g. invalidation of copies at several nodes, which requires broadcasts or atomic multicasts) is not required.

We define Causal memory in the next section and the actual implementation is described in Section 3. Section 4 discusses the programming support required to run applications efficiently on Causal memory. In Section 5 we describe how to program applications and give performance results. We talk about related work in Section 6 and conclude in Section 7.

2 Causal Memory

In [4] we define correct execution on Causal memory by the possible set of values a read to a shared location can return. We use a more generalized framework here, since this can be used to classify a range of different memory models [21].

2.1 Formal Definition

This section formally describes the system that underlies our implementation. We use a model that is similar to those used by Misra [27] and by Herlihy and Wing [19]. We define a *system* to be a finite set of processors that interact via a shared memory that consists of a finite set of *locations*. A processor's interaction with the memory is through a series of read and write *operations* on the memory. Each such operation acts on some named location and has an associated value. For example, a write operation by processor p denoted $w_p(x)v$ stores the value v in location x; a similarly denoted read operation, $r_p(x)v$, reports that v is stored in location x. A processor initiates an operation by issuing an *invocation*. After doing so, the processor *waits* until the memory issues a *response* to the invocation. In the case of read operations, this response includes the value that was read. Upon receiving a response, the processor can continue, issuing other invocations. For any operation o, let $inv(o)$ be o's invocation and let $rsp(o)$ be its response. An *execution* *history* (or *history*) H is thus a sequence of invocation and response events. The order of events in H corresponds to the order in which they occur in the global time; this is a total order.[1] The execution of an operation corresponds to the interval in time that is bounded by its invocation and response; this is called the operation's *execution interval.* A history is *sequential* if every invocation in immediately followed by its corresponding response. In such a history, there is no overlap between the execution intervals of different operations on the memory and we can replace the events(*inv* and *rsp*) by the corresponding operation(read or write). A sequential history is *legal* if every read operation from a location returns the value of the most recent write to that location (this write is uniquely defined because the history is sequential and thus there is a total order). The definition of an execution history assumes a total ordering on invocation and response events. This can be extended to the operations comprising these events. Specifically, we say that operation o_1 precedes o_2 in H, written $o_1 \overset{H}{\rightarrow} o_2$ if $rsp(o_1)$ precedes $inv(o_2)$ in the sequence of events that H comprises. Note that order $\overset{H}{\rightarrow}$ is partial, as the execution intervals of operations may overlap. If history H has operations $o_w = w(x)v$ and $o_r = r(x)v$, then o_r reads the value written by o_w. We express this using a new relation, $\overset{H}{\mapsto}$, writing $o_w \overset{H}{\mapsto} o_r$. We call $\overset{H}{\mapsto}$ the *read-by* relation and assuming all writes have distinct values (this just simplifies the model but the general case can be easily handled), for each read operation o_r, there is a unique o_w such that $o_w \overset{H}{\mapsto} o_r$. A "happens before" relation in the sense defined by Lamport [22] can also be defined for shared memory. We denote this *causal relation* by $\overset{H}{\rightsquigarrow}$ and it combines the $\rightarrow$ relation with the *read-by* relation $\mapsto$. The relation $\overset{H}{\rightsquigarrow}$ partially orders causally related operations; two operations are related by the causal relation if any of the following hold:

[1]We assume a notion of *global time*, which linearly orders all invocation and response events and that no two events occur at precisely the same global time. Global time is used only to impose this order on events and is not, in general, available to the processors or to the subsystem that implements the shared memory.

- If o and o' are operations in H of some process p such that $o \overset{H}{\rightarrow} o'$, then $o \overset{H}{\rightsquigarrow} o'$ (program order dependence).
- if o_w and o_r are such that $o_w \overset{H}{\mapsto} o_r$, then $o_w \overset{H}{\rightsquigarrow} o_r$ (inter-processor dependence).
- if $o_1 \overset{H}{\rightsquigarrow} o_2$ and $o_2 \overset{H}{\rightsquigarrow} o_3$, then $o_1 \overset{H}{\rightsquigarrow} o_3$ (transitivity).

Given this formalism we can define a variety of memory consistency models, by putting constraints on the allowable histories for any execution. [21] defines different memory models based on this formalism. Let H_{p+w} be the history obtained from H by deleting all events of all *read* operations executed by processes other than p. A history is causal if it satisfies the following:

Causal Memory for each processor p, there is a *legal sequential* history S_p, such that, for all operations o_1 and o_2 in H_{p+w} if $o_1 \overset{H}{\rightsquigarrow} o_2$ then $o_1 \overset{S_p}{\rightarrow} o_2$.

To illustrate, Figure 1 shows an execution (the history of *inv* and *rsp* events can be easily constructed) and the corresponding *legal sequential* causal histories. 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.

$P_1: \; w(x)1 \; w(y)2 \; r(z)0$
$P_2: \; w(z)1 \; r(x)0 \; r(y)2 \; r(x)1$

(a) Two Process Execution

$S_{P_1}: \; w_1(x)1 \; w_1(y)2 \; r_1(z)0 \; w_2(z)1$
$S_{P_2}: \; w_2(z)1 \; r_2(x)0 \; w_1(x)1 \; w_1(y)2 \; r_2(y)2 \; r_2(x)1$

(b) Causal Sequential Histories

Figure 1: An Execution which is Causal but not Sequentially Consistent

2.2 Atomic, SC and Causal

Atomic and sequentially consistent memories provide strong consistency guarantees, since they require that all processes "agree" on a single global sequential history that includes read and write operations of all the processes. In contrast, Causal memory does not require a global consistent view of all the operations and as a consequence, operations might appear delayed at certain processors. For instance, a processor could perform a write locally, even though readers which have cached the location could continue seeing the old value. The earliest time at which writes have to be propagated (or made visible), depends on the memory model, and has a significant influence on performance. Certain cooperative distributed applications, for example a distributed calendar service, may tolerate *stale* values. On the other hand, data used to achieve implicit synchronization may require writes to be immediately made visible. Other such examples are given in [15]. With adequate programming support, such application related information can be utilized by a causal memory system to enhance performance.

2.3 Synchronization and Forking

Causal memory has been defined using the read/write model of shared memory. But, communication between processors is not limited to the read/write operations. Programs very often contain synchronization operations for access and sequence control. Also, programs contain forks or invocations (in object based systems) for expressing concurrency. We discuss the effect of these operations on the Causal memory model.

Parallel and distributed programs, typically, define their consistency requirements through the use of synchronization operations. Synchronization is achieved through explicit language provided mechanisms, such as semaphores, locks and barriers, or implicitly by *busy-waiting*. Dubois et al. [16] define a *weakly ordered* system where dependency conditions between processes are limited to the synchronization variables. Adve & Hill [1] formalize this by defining a *happens-before* relation, which orders two operations on different processes, only if, there is a synchronization operation between them. Since a critical section guarantees that only a single process is active in it at a time and if shared data is accessed only inside a critical section, the synchronization dependences would subsume the *read-by* dependences. *Weakly ordered* systems require that the synchronization operations are made explicit to the memory system. Inter-processor dependences can be then limited to the synchronization operations [1, 20, 10].

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 *forks* "child" processes on different processors, each working on a different partition. While programming, it is assumed 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 are visible to the parent. In effect, inter-processor dependences would arise due to such operations too, and must be handled.

The formal model can be extended to include synchronization and forking events. Causal orderings would now arise between synchronization acquires and releases and also between the forking parent thread and the forked child thread.

2.4 Implementation on Clouds

The key features of the implementation of the algorithm that is described in this section include: (1) use of page fault mechanisms for accesses to shared data by processes on different nodes, (2) use of vector timestamps for maintenance of data version information, and (3) use of multiple page invalidation schemes.

The Clouds Distributed Operating System has been used as the test-bed for implementing Causal memory. Although the implementation is general, certain optimizations we discuss are in the context of an object based system and so some understanding of the computation model is required.

2.4.1 System Model

Clouds is an object based distributed operating system which provides the programmer with the notion of a globally shared memory. An object in Clouds contains, minimally, a code and a data segment. A node in the Clouds system may be a data or a compute server. Computations are carried out at the compute servers and a data server is responsible for managing the objects owned by it. A data server supplies pages on demand, for objects it owns, and is responsible for maintaining the consistency requirements. Threads are the active entities in the system and are used for expressing concurrency and may span machine boundaries by invoking an object on different nodes.

2.4.2 Basic Protocol

We have implemented Causal memory by rewriting the existing DSM protocol on Clouds. We use the page fault mechanism for sharing state between processors. Accessing a page which is not present generates an *access* violation. A *protection* violation is generated when a page with only read access is written into.

The data and compute server actions are described in Figure 2. The procedures described are performed either at page faults or on receipt of a message. Each procedure is executed atomically.

The implementation we describe allows a single writer to coexist with multiple readers. False sharing might create the problem of multiple processors writing to different parts of the same page. With language support which allows such data to be put on different pages, this problem can be alleviated to some extent. But false sharing cannot be avoided altogether, especially in cases where dynamic data partitioning is done. The implementation can be extended to allow multiple writers by having the data server merge the changes. The relative costs of merging to restricting to a single writer has still to be studied. It must be noted that false sharing where there is a single writer and multiple readers is tolerated. A page shuttles only when multiple writers write to data on the same page.

C_i is the set of shared data pages which are cached at processor P_i. Each page has the fields *owner*, *keeper* and *access*. The *owner* field for a page x identifies its data server. The *keeper* field indicates the node which has the latest copy of the page; if it is ϕ the data server supplies the page. The *access* field specifies the access privileges to the page on the local processor.

We use vector timestamps[26] to capture the evolving causal relationships. A vector timestamp is associated with each page and each processor P_i carries a timestamp VT_i. Three operations can be performed on vector timestamps, which are described below:

- *increment*: inc(VT) by processor P_i increments VT by adding one to its ith component, $VT[i]$.
- *update*: update(VT, VT') returns the component wise maximum of the two vectors.
- *comparison*: $VT < VT'$ returns true if all components of VT are less than or equal to VT' and there is at least one component which is less.

Notice the handling of timestamps. The same page cached on different processors may have different timestamps. The page timestamp reflects the updates to the page that the processor has

```
$ReadAccessFault$ ::
    send $[READ\_REQ, y]$ to $y.owner$
    recv$[R\_REPLY, y, VT']$ from $y.owner$
    $y.access := read$
    $y.VT := VT'$
    $VT_i := update(VT_i, VT')$
    $\forall x \in C_i : x.VT < VT' \wedge x.access = read$
        local_invalidate(x)

$WriteAccessFault$ ::
$ProtectionFault$ ::
    send $[WRITE\_REQ, y]$ to $y.owner$
    recv $[W\_REPLY, y, VT']$ from $y.owner$
    $y.access := write$
    $y.VT := VT'$
    $VT_i := update(VT_i, VT')$
    $VT_i := inc(VT_i)$
    $\forall x \in C_i : x.VT < VT' \wedge x.access = read$
        local_invalidate(x)

$[FORWARD\_REQ, y, j]$ ::
    recv $[FORWARD\_REQ, y]$ from $y.owner$
    local_invalidate(y)
    send $[F\_REPLY, y, VT_i]$ to $j$

$[FWD\_COPY\_REQ, y, j]$ ::
    recv $[FWD\_COPY\_REQ, y, j]$ from $y.owner$
    $VT_i := inc(VT_i)$
    send $[FC\_REPLY, y, VT_i]$ to $j$
```

(a) Compute Server Actions

```
$[READ\_REQ, x]$ ::
    recv $[READ\_REQ, x]$ from $i$
    if $(x.keeper = \phi)$
        send $[REPLY, x, x.VT]$ to $i$
    else
        send $[FWD\_COPY, x, i]$ to $x.keeper$

$[WRITE\_REQ, x]$ ::
    recv $[WRITE\_REQ, x]$ from $i$
    if $(x.keeper = \phi)$
        send $[REPLY, x, x.VT]$ to $i$
    else
    send $[FORWARD, x, i]$ to $x.keeper$
    $x.keeper := i$
```

(b) Data Server Actions

Figure 2: Causal Implementation

seen. A processor's timestamp is *incremented* on a writefault and if it gets a request to forward a copy of a page. This is sufficient to order the writes. Comparing page timestamps allows us to determine if the modifications to shared data happened concurrently, or have to be ordered. At a page fault, we locally *invalidate* any cached pages that contain values that could potentially violate causal memory correctness if read. The *local_invalidate* function is responsible for invalidating all mappings to a page, so that any future access to data in the page would generate an access fault.

2.4.3 Handling Local Invalidations

Locally invalidating pages obviates the need for explicit invalidation messages. Performing local invalidations does not just reduce the number of messages but also does not have the extra software overheads of network interrupts and context switches, associated with invalidation messages. The saving in the number of messages is at the cost of the extra computation required to determine and invalidate causally older pages. Also, we might invalidate more pages than strictly necessary. This is because, not all pages that are older (w.r.t timestamps) are outdated. For instance, consider the execution in figure 3.

$$\begin{array}{lll} P_1: & w(x)1 & w(y)1 \\ P_2: & r(x)1 & r(y)1 \end{array}$$

Figure 3: Page containing x will get invalidated

Assume that x and y are on different pages. When P_2 reads the value y, the page containing x would get invalidated even though it holds the current value of x.

We allow user provided information to reduce unnecessary invalidations and also to reduce the frequency of doing the local invalidations. We have considered the following approaches that allow application specific knowledge to be used to control the invalidation operation. We further elaborate on how this is achieved in the next section on programming support.

- User annotations - Pages which are only written into during initialization and never again, will not have their timestamps incremented. Such pages would get unnecessarily invalidated at each pagefault. The Munin system[9] has identified various classes of data access patterns which are characteristic of shared memory programs. Typical data access patterns identified are *Private*, *WriteOnce* and *WriteShared* among others. We provide language mechanisms, described in the next section, for a user to pass this information to the memory system. Data which is *Private* or *WriteOnce* is never invalidated.

- synchronization based - for programs which are "properly" synchronized, invalidates need only be done when a process acquires a lock. For this, synchronization mechanisms have been extended to work on causal memory, and data used for implicit synchronization has to be made explicit to the memory system.

- Object information - Context switching between processes, also introduces the problem where cached pages not related to the computation could get invalidated. The data segment of an object is directly accessed by the code in the code segment, and not by any other object. When timestamps are compared, only pages with lower timestamps belonging to the object being accessed are discarded.

Another approach is to perform the invalidations periodically to ensure that updates will eventually get propagated. The user can specify a time interval after which the page is unmapped. It is useful for programming asynchronous algorithms [8] and certain distributed applications which tolerate stale values.

2.5 A note on Scalability

The implementation we sketched requires at the maximum, three messages to satisfy a pagefault request. Unlike traditional implementations of atomic DSM, Causal memory does not require expensive multicasts or broadcasts. This bound on the number of messages allows scalable shared memory systems to be built. But it must be noted that with each page a vector timestamp has to be propagated. Since the timestamp is of the order of the number of processors, the size of the messages increases with increasing the number of messages. Also to do local invalidates, our implementation considers all pages in the cache for pages which are causally older. As cache sizes grow this overhead could be significant. Currently we are investigating ways to reduce this cost.

3 Programming Support

Programming on Causal memory requires language and runtime support. The Clouds Distributed-C++ [7] compiler and the runtime system have been extended to achieve the following.

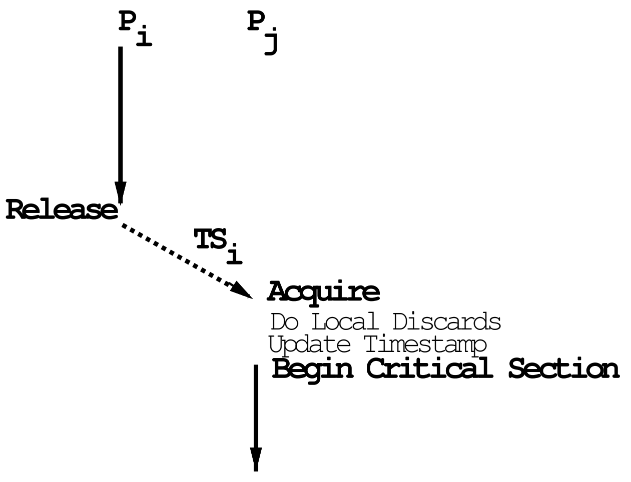
- The language provides the ability to define multiple user data segments within an object. A segment is an aggregate of data with similar access patterns.
- The synchronization operations have been extended to carry a vector timestamp, which allow consistency preserving operations to be done at synchronization points.

- On object invocations and returns, the thread of control carries with it a vector timestamp which indicates the modifications to shared data it is aware of.

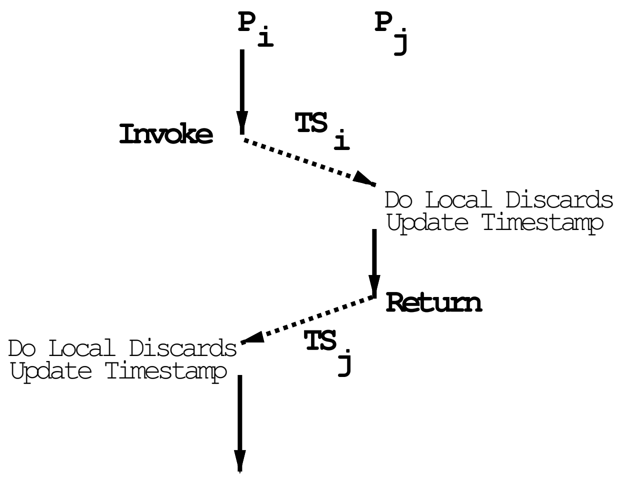
3.1 User Defined Segments

Segments provide the means to partition data and group data structures which have similar access patterns. The keyword **segment** specifies the smallest logical unit of consistency maintenance. Segments can be tagged as *Static* or *Causal*. In contrast to Munin [9], this labeling for ordinary data does not imply multiple protocols. We use this information to reduce the unnecessary invalidations that would otherwise occur. Data which is used privately by a processor, globally initialized data and data which is written only once during an initialization phase is tagged as *Static*. Data structures which exhibit a general read and write sharing pattern are labeled *Causal*. Only data labeled as *Causal* are considered for invalidation during the local invalidate operation. For programs which are data-race-free, the data must be one of *Static* or *Causal*.

We provide another annotation, *Transient*, which is useful for programming asynchronous applications and certain distributed cooperative applications which are non data-race-free. *Transient* is used where the writes need not get immediately propagated. Currently our implementation periodically locally invalidates *Transient* data which is cached in the read mode. This ensures that the writes will eventually get propagated. In the next section, we program applications to illustrate how these annotations are used.



(a) Handling Synchronization



(b) Handling Invocations

Figure 4: Synchronization and Invocations

3.2 User Synchronization

Explicit synchronization mechanisms have been modified so that programs run correctly on Causal memory. The semaphore acquire and release actions are shown in Figure 4. Each semaphore has an associated vector timestamp. On a release operation the processor P_i's timestamp is assigned to the semaphore. The processor P_j which acquires the semaphore uses this timestamp to invalidate causally older pages in its cache, updates its timestamp and only then executes the critical section. By updating its timestamp, the processor P_j ensures that P_i's changes will be visible to it, when it page faults.

Read-write locks are implemented similarly. Barriers are implemented by having a barrier server do an update of the timestamps of every thread at the barrier. This new timestamp is transmitted with the barrier release and every node performs the local invalidate operation using the timestamp.

3.3 Forks and Invocations

Forking is a mechanism for achieving parallelism in parallel and distributed systems. In object based systems this is achieved through object invocations on different nodes. Object invocations on Causal memory are handled as shown in Figure 4. When processor P_i invokes an object on P_j, the processor P_j uses P_i's timestamp to invalidate causally older pages and updates its own timestamp before executing code. This ensures that if P_j faults on a page, it will see the changes done by P_i. Similarly, at a fork-join or invocation return, the processor P_i's timestamp has to be updated to ensure that the modifications made by P_j fall in its causal history. Asynchronous invocations can be handled similarly by locally invalidating pages and updating the timestamp, when the results are *claimed.*

4 Writing Programs

In our experience, we have found it convenient to develop programs in two phases. Programs are written without using the keyword *segment.* If data is not annotated, a centralized manager invalidation protocol similar to the one described in [24] is used. Once the program is shown to run correctly, we annotate the data.

In this section, we program an iterative linear equation solver and a distributed calendar service. These programs are simple enough to illustrate how the data annotation is done. In [6], we use NAS kernel applications, a traveling salesperson program and a matrix multiplication to compare performance on Causal memory with other implementations of DSM.

4.1 Linear Equation Solver on Causal Memory

Large systems of linear equations often arise in many scientific and engineering applications. Li [24] investigated such an application and found that good speedups can be obtained on atomic DSM's. We show here that even better performance can be obtained on Causal memory.

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 parallel implementation of the iterative method partitions the task of computing the vector x among available processors. At the beginning of each iteration, each processor reads the results from the previous iteration and computes the new vector x_i which it stores in a private local vector. Processors synchronize twice in each iteration, once to ensure that the new values have been computed and then to store these values into the vector x.

Figure 5 shows the code for the linear solver. The vector t_i is local to processor P_i; the function *converged* checks for convergence after each iteration. We used a 128×128 size matrix. A parent thread initializes the matrix A and the vector b and then creates the workers on different nodes. The array x is partitioned so that there is no false sharing.

We compare its performance on Causal memory with an atomic DSM implementation. The atomic DSM implementation uses a centralized manager writer-invalidate-readers protocol [24]. Multicast communication is not used for invalidating readers caching a *dirty* page; rather separate messages need to be sent. The data partitioning was done identically in both cases. The computation was done on Sun 3/60's. In Figure 6 we show the reduction in completion time and messages obtained on Causal memory. Also, since Causal memory allows a writer to co-exist with multiple readers there are fewer pagefaults. In particular, for the 3 processor case, Causal memory has 50 % fewer pagefaults than atomic memory.

As we increase the number of workers, it can be analytically shown, that the reduction in

```
object linear_solver {
  segment  { float A[MAX_SIZE][MAX_SIZE]; } [Static];
  segment  { float b[MAX_SIZE]; } [Static];
  segment  { float x[MAX_SIZE]; } [Causal];

  main()
  {
  initialize A & b;
  for(i = 1; i < num_of_nodes; i++) {
    create_process(worker, node[i], i, dim, num_of_nodes);
  }

  worker(int i, int n, int num_of_nodes)
  {
  while (¬converged())
    $t_i := \left(b_i - \sum_{j=1}^{i-1} a_{i,j}\, x_j^k - \sum_{j=i+1}^{n} a_{i,j}\, x_j^k\right) \Big/ a_{i,i}$
    barrier();
    $x_i := t_i$
    barrier();
  }
}
```

Figure 5: Iterative Linear Equation Solver

#Workers	2	3	4	5	6
%Reduction in messages	66.67	57.14	52.63	50.00	48.27
%Reduction in time	18.08	22.74	30.11	35.21	37.99

Figure 6: Performance Results

messages would approach 40 percent. It must, however, be noted that because of the small problem size, the communication cost dominates the completion time. In practice, the problem sizes would be much larger and the computation cost would dominate. In these cases, although the reduction in the number of messages would be still significant, its effect on the total completion time will be reduced.

4.2 Distributed Calendar

Consider the problem of implementing a distributed calendar service [14]. Every user in the system maintains an appointment calendar, which can be browsed by others. In addition, 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. The code for such a service is shown in figure 7.

Access to a user's calendar is protected by a read-write lock. This is required since we allow users to modify and browse any other person's calendar. *Browse_calendar* allows a user to look at anyone's appointment calendar. The function *update_calendar* permits one to add 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 flag is of type *Transient*, since updates need not be immediately propagated.

The local invalidations of causally outdated pages are carried out at lock acquires, as described previously. In figure 8 we compare the message cost due to our approach and contrast it with the cost due to atomic DSM's. We have assumed that each user's calendar data would not require

```
object calendar {
    segment  { calendar_type calendar[MAX_USERS]; } [Causal];
    segment  { lock mutex[MAX_USERS]; } [Causal];
    segment  { boolean changed[MAX_USERS]; } [Transient];


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 7: Shared Memory Distributed Calendar

Operations	browse	update	set_up_meeting	daemon
Max Messages(Atomic)	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 8: Message counts for different memory models

storage more than a page. This assumption just simplifies the message count analysis.

Causal memory requires just 3 messages to service a page fault. On the other hand, the number of messages on atomic memory depends on the number of concurrent readers. The two values for the *listener_daemon* case correspond to the cases where the *if* statement is false or true (in which case, there is a write to the flag).

5 Related Work

Boyer [12] describes an implementation of causal DSM on Mach using external pagers. Simple message counting arguments are presented to show its superior performance to conventional atomic DSM's.

Munin [9] implements release consistency in software by delaying propagating the changes made inside a critical section, till the lock is released. Modifications to shared data is tracked by using the page fault mechanism. *Dirty* pages are compared with a copy of the original page and the modified data is propagated to all processors.

Lazy Release Consistency [20] tracks the causal dependencies between writes, acquires and releases and propagates the writes by piggy-backing the modified data on lock transfer messages.

LRC uses a history based mechanism to record the modified data which has to be transmitted with the lock transfers.

Entry Consistency [10] requires that all shared data be associated with a synchronization variable. When a processor acquires a synchronization variable, only modified data associated with the synchronization variable need to be transferred. Compile time support is used to track data items which have been modified. Extra programming effort may be required to identify just the right data to be associated with a synchronization variable.

There have been several hardware implementations of weakly ordered systems and are described in [18, 2, 28, 11, 17].

6 Conclusions

We have shown that a causal DSM can be efficiently implemented. With adequate programming support, causal DSM significantly reduces the frequency of communication between processors. Also, scalable systems can be built since accesses to data do not require expensive global synchronization. Performance on systems with slow networks and fast processors can be especially significant. We are currently investigating other applications which promise better performance on causal DSM.

References

[1] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. Technical Report 1051, University of Wisconsin, Madison, September 1991.

[3] Y. Afek, G. Brown, and M. Merritt. A lazy cache algorithm. In *Proceedings of the SPAA*, pages 209–223, June 1989.

[4] 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.

[5] 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.

[6] R. Ananthanarayan, Ranjit John, Ajay Mohindra, Mustaque Ahamad, and Umakishore Ramachandran. An evaluation of state sharing techniques in distributed operating systems. Technical report, College of Computing, Georgia Institute of Technology, Atlanta, April 1993.

[7] R. Ananthanarayanan. Clouds C++ reference manual. Technical Report GIT-CC-91-07, College of Computing, Georgia Institute of Technology, 1991.

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

[9] 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*, pages 125–135, May 1990.

[10] 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.

[11] R. Bisiani and M. Ravishankar. PLUS: A distributed shared memory system. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 115–124, June 1990.

[12] Fabienne Boyer. A causal distributed shared memory based on external pagers. In *Proceedings of the 2nd Usenix Mach Symposium*, November 1991.

[13] G. Brown. Asynchronous multicaches. *Distributed Computing*, 4(1):331–362, 1990 1990.

[14] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs.* The MIT Press, Cambridge, Mass.

[15] D. R. Cheriton. Problem-oriented shared memory: A decentralized approach to distributed systems design. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 190–197, May 1986.

[16] 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.

[17] Michel Dubois, Jin Chin Wang, Luiz A. Barroso, Kangwoo Lee, and Yung-Syau Chen. Delayed consistency and its effects on the miss rate of parallel programs. Technical report, Dept. of Electrical Engineering Systems, USC, April 1991.

[18] Kourosh Gharchorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared memory multipro-

cessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[19] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objec ts. *ACM Transactions on Programming Languages*, 12(3):463–492, July 1990.

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

[21] 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.

[22] Leslie Lamport. Time, clocks and the ordering of events. *Communications of the ACM*, 21(7):558–565, July 1978.

[23] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *ACM TOCS*, c-28(9):690–691, September 1979.

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

[25] 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.

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

[27] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142–153, January 1986.

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