Evaluation of Causal Distributed Shared Memory for Data-race-free Programs*

Ranjit John *Mustaque Ahamad*

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

GIT–CC–94/34

Abstract

Distributed Shared Memory (DSM) is becoming an accepted abstraction for programming distributed systems. Although DSM could simplify the programming of distributed applications, maintaining a consistent view of shared memory operations across processors in a distributed system can be expensive. The *causal* consistency model of DSM can allow more efficient implementations of DSM because it requires that only causally ordered memory operations be viewed in the same order at different processors. Also, *weakly ordered* systems have been proposed which advocate the use of synchronization information to reduce the frequency of communication between processors. We have implemented a system that exploits both the weaker consistency of causal memory and the synchronization information used in weakly ordered systems. Consistency is ensured by locally invalidating data that is suspected to be causally overwritten and this is only done when certain synchronization operations complete at a processor. Data-race-free programs can be developed in this system assuming that the system provided sequentially consistent memory. By implementing applications that have a variety of data sharing patterns, we show that performance close to message passing implementations of the applications can be achieved in the causal DSM system. The improved performance is due to a significant reduction in communication costs compared to a strongly consistent memory system. These results show that causal memory can meet the consistency and performance requirements of many distributed applications.

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

*This research was supported in part by NSF grant CCR-9106627 and ARPA contract N00174-93-K-0150.

1 Introduction

Distributed Shared Memory (DSM) is becoming an accepted abstraction for programming distributed systems. DSM can simplify programming of distributed applications since the user can access both local and remote information uniformly by using memory operations. This simplification is at the cost of maintaining a consistent shared memory. DSM can be implemented by using variants of multiprocessor cache consistency protocols but applications executing with such implementations of DSM may not perform well in distributed systems. The main reason for this loss in performance is the unnecessary communication that they incur compared to message passing implementations of the applications. For example, when several processors make copies of a shared data item to read it, a future write could result in communication with all these processors even when they will not access the data in the future. Such extraneous communication can be avoided in message passing systems because the application programmer determines when data needs to be exchanged between processors.

A DSM is an interface between the program and the memory system, and the memory model implemented by the DSM system defines what values can be returned when processors read shared data. Ideally, a distributed shared memory should provide all the consistency guarantees of a true shared memory. Lamport [31] defined a memory model called *sequential consistency* which requires that the execution of all processors must be equivalent to some sequential order in which memory operations are executed and read operations return values consistent with this order. The sequential order must maintain the order of memory operations issued by a particular processor (program order). In the first system that implemented DSM [32], a writer-invalidate-readers protocol was used to provide sequentially consistent DSM. Maintaining sequential consistency in a distributed system can be shown to limit performance and does not lend itself to scaling [34] due to high latencies and communication costs.

One way to reduce communication between processors in shared memory system is to guarantee consistency only at certain points in the execution of a program. This approach was first outlined by Dubois et al. [18], who observed that parallel programs define their own consistency requirements through the use of synchronization operations. Dubois et al. define a *weakly ordered* system, where synchronization operations are made explicit to the memory system and consistency maintenance is done only at synchronization points. The DASH multiprocessor [21] is a weakly ordered system that 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 a critical section (a release operation on a synchronization variable). Such systems guarantee sequentially consistent behavior only for programs that are data-race-free [1] or properly labeled [21]. 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 memory.

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 [12] implements release consistency in software by delaying propagation of the changes made inside a critical section till the release operation. Munin also identified several data sharing patterns and corresponding annotations that users can provide to reduce the cost of consistency maintenance in a DSM system. More recently, *lazy release consistency* [26] and *entry consistency* [13] memory models have been proposed, which use synchronization information to further reduce communication by propagating information about changes to shared data to only the processor that next acquires the lock.

The second approach advocated to improve the performance of shared memory systems is by weakening the consistency guarantees provided by the system. The weaker consistency does not guarantee that the execution of memory operations of all processors is equivalent to some sequential execution of these operations, as in a sequentially consistent system. Examples of weakly consistent memory systems included *pipelined RAM* (PRAM) [34], *processor consistency*[1] [4, 21, 22], and *causal* memory [5]. 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 (e.g., conservative programs for PRAM [34]), or the program must explicitly deal with the lack of strong consistency.

In this paper, we explore efficient implementations of DSM by exploiting ideas from both approaches described above — we make use of synchronization information, as advocated by the *weakly ordered* memory systems approach, for implementing causal memory [6], which is a weakly consistent memory. The causal memory model requires that a read operation return a value that is consistent with the causal order of the memory operations that are ordered before it. We extend causal memory by allowing synchronization operations and develop an implementation that exploits the fact that programs are data-race-free. We experimentally demonstrate that for a range of applications, the weaker consistency of causal memory coupled with the data-race-free nature of parallel and distributed programs, leads to a system with significantly better performance. More specifically, we show the following:

- Causal memory can be efficiently implemented and is a viable architecture for distributed programming since data-race-free programs can be programmed on it the same way as on sequentially consistent memory.
- The execution of distributed applications results in far less communication on causal memory compared to their execution on a sequentially consistent memory. Also, the execution of the applications on causal memory provides performance close to message passing systems for most applications.

[1] The DASH system implemented processor consistency only for labeled operations.

- Scalable shared memory systems can be built using the causal memory model, since global synchronization (e.g., invalidation of copies at several processors, which requires broadcasts or multicasts) is not required.

We precisely define causal memory in the next section and two different implementations are described in Section 3. In Section 4 we describe the applications with which we experimented and provide performance results. We discuss related work in Section 5 and conclude in Section 6.

2 Causal Memory

2.1 The Model

Causal memory has been defined in [6] by characterizing the possible values that could be returned when a read operation is executed by a processor. We use a more general framework here, as it allows us to easily relate causal memory to a range of memory models that have been proposed. The model is motivated by the ones used by Misra [36] and by Herlihy and Wing [23]. We define the system 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 $H = \{H_p | p \in \mathcal{P}\}$ 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_i is ordered before o_j by the program order. This defines program order to be total; it orders all operations of a given processor[2].

- 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 before the read.

[2]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 [21].

- Causal order: The *happens before* relation defined by Lamport [30] 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, we have developed implementation independent definitions of several memory systems [4, 29]. We now present such a characterization of causal memory.

2.2 Defining Causal Memory

Causal memory requires that values returned by read operations respect the 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. 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.

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 $o_1 \stackrel{co}{\rightarrow} o_2$ 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 1 is possible with causal memory because the corresponding views 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 program order established by each processor).

$p_1:$	$w(x)1\ w(y)2\ r(z)0$	$S_{p_1}:$	$w_1(x)1\ w_1(y)2\ r_1(z)0\ w_2(z)1$
$p_2:$	$w(z)1\ r(x)0\ r(y)2\ r(x)1$	$S_{p_2}:$	$w_2(z)1\ r_2(x)0\ w_1(x)1\ w_1(y)2\ r_2(y)2\ r_2(x)1$

(a) Two Process Execution (b) Causal Views

Figure 1: An execution which is causal but not sequentially consistent

Causal memory differs from other weakly consistent memories. Figure 2 shows an execution that is permitted by causal memory which is not allowed by processor consistent memory as implemented by the DASH multiprocessor [21]. 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 [34] 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.

$$
\begin{aligned}
p_1:&\ w(x)1\\
p_2:&\ w(x)2\\
p_3:&\ r(x)1\ r(x)2\\
p_4:&\ r(x)2\ r(x)1
\end{aligned}
$$

Figure 2: Causal but not processor consistent execution

2.3 Synchronization Operations

Causal memory has been defined using the read/write model of shared memory. In parallel applications, communication between processors also takes place via synchronization operations, which are 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. Thus, orderings between memory operations of different processors would arise due to such synchronization operations too.

Our 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).

- 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 a lock release 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_j.

- 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 $o' \stackrel{so}{\rightarrow} o_2$.

We can now extend our causal order definition 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 $o' \stackrel{eco}{\rightarrow} o_2$. The causal memory definition is extended to include the non-memory operations by using this new causal ordering[3]. The definition remains the same but we require that, if two operations o_1 and o_2 appear in a processor's view and $o_1 \stackrel{eco}{\rightarrow} o_2$, then o_1 must be ordered before o_2 in the view. Thus, orderings between memory operations in a processor's view must respect the extended causal order.

2.4 Causal Memory and Data-Race-Free Programs

The memory models motivated by the weakly ordered systems approach have demonstrated that memory system performance can be improved when synchronization operations are made visible to the hardware that implements shared memory. When programs use sufficient synchronization to control access to shared data, they need not be aware of the optimizations carried out by the hardware and can simply assume that it provides sequentially consistent memory. Adve and Hill [1] 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 when the program is executed on a sequentially consistent memory. 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. We have used this fact to show that data-race-free programs can be developed on causal memory the same way as on sequentially consistent memory. Thus, programming of this class of programs 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 for all memory operations, it can be shown that an execution of a program on causal memory is also possible on sequentially consistent memory. Thus, if the program executes correctly on sequentially consistent memory, its execution on causal memory is also correct.

Data-race-free programs also allow 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

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

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. In the example shown in Figure 3, when p_3 caches the value of y written by p_2 and reads it, the cached value of x at p_3 has become overwritten (the write to x by p_1 causally precedes the write to x by p_2), and 1 must not be returned by a future read to x by p_3.

$$
\begin{aligned}
p_1 &: \quad w(x)1 \; w(y)1 \\
p_2 &: \quad r(y)1 \; w(x)2 \; w(y)2 \\
p_3 &: \quad r(x)1 \; r(y)2
\end{aligned}
$$

Figure 3: Example that shows causal over-writing of data

When programs are known to be data-race-free, 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, $\stackrel{sc}{\rightarrow}$, is identical to the extended causal order $\stackrel{ecc}{\rightarrow}$. 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 happens before 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 algorithms we develop in the following sections make use of this fact to avoid extra processing overhead.

3 Implementing Causal Memory

We develop an implementation of causal memory for a system that consists of a set of processors connected by a network. Access to causally consistent shared data is provided by caching the data 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. To locate a data item, we introduce the notion of a *manager* as is used by Li and Hudak [32]. 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.

Similar to the implementation developed in [6], the algorithm that we develop ensures that data items cached at a processor are mutually consistent from the point of view of causal consistency. In other words, one cannot violate the consistency requirements of

causal memory by reading locally cached data. When a new data item or a new value of already cached data is added to the cache, its reading by a process at the processor can create new causal orderings between memory operations. As a result, some of the cached data values can become overwritten (the reading of an overwritten value will violate causal consistency). In the algorithm presented in [6], 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 data-race-free programs, and this allows us to develop more efficient schemes for maintaining causal consistency. Also, the implementation presented here works at the granularity of a page. This allows us to integrate consistency maintenance operations with virtual memory operations such as page faults.

The Clouds distributed operating system has been used as the test-bed for implementing causal memory. Clouds is an object based distributed operating system which provides the programmer with the notion of a globally shared memory across processors connected over a local area network. The implementation of DSM in Clouds is integrated with virtual memory. Consistency actions are performed on page faults or on protection violations. We have implemented causal memory by rewriting the existing DSM protocol on Clouds.

3.1 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, and (4) delaying consistency related actions to well defined points in a program (certain synchronization operations).

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.

We use vector timestamps to track changes in the state of shared data. Vector timestamps [35] precisely capture the causal relationships between memory operations. 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. Three operations can be performed on vector times, which 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''.

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 [11, 26] can be used to handle the merging problem but it does have copying and processing overheads, which we have found to be quite high [25]. 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 [20] 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 create any problems in our implementation of causal memory.

Finally, because we implement causal memory for data-race-free programs, we will only perform consistency maintenance operations when a synchronization variable is acquired at a processor. In particular, these operations are only performed on acquiring a lock, on returning from a barrier, and at fork and join points. We require that synchronization variables also carry vector timestamps so that the causal orderings induced by synchronization operations are reflected in the consistency maintenance operations. Below, we present the details of the implementation.

3.2 Implementation Details

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*. When a processor accesses a page which is not locally cached, it generates an *access* violation. A *protection* violation is generated when a page with *readonly* access is written.

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.

```
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
      x.owner := self
      VT_i := inc(VT_i)
      x.VT := VT_i
```

(a) Actions Executed at p_i (i is p_i's identifier)

```
[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 4: Causal memory implementation using vector timestamps

3.2.1 Handling Page Faults

The actions executed when page faults occur at processor p_i are shown in Figure 4. 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. We explain why this is necessary in the next section. The faulting processor, on receiving the page in a DATA message, installs the page with the appropriate access rights.

3.2.2 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. Furthermore, since the causal order created by a read only orders the associated write operation before the read, the timestamp sent in a DATA message is the vector time at which the page was written, and not the current value of the sender's clock. Thus, in Figure 4, only DATA messages carry the timestamp associated with the page being sent. If the page is received as a result of a read fault, VT_i, the vector clock at p_i, is not advanced when the page is received. A clock update is not necessary when we only consider the execution of data-race-free programs. 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. We explain in the next section why VT_i is updated on write faults but not when a page is received due to a read fault.

The actions that handle the various types of faults in Figure 4 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 a processor's access to a page that it owns 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.

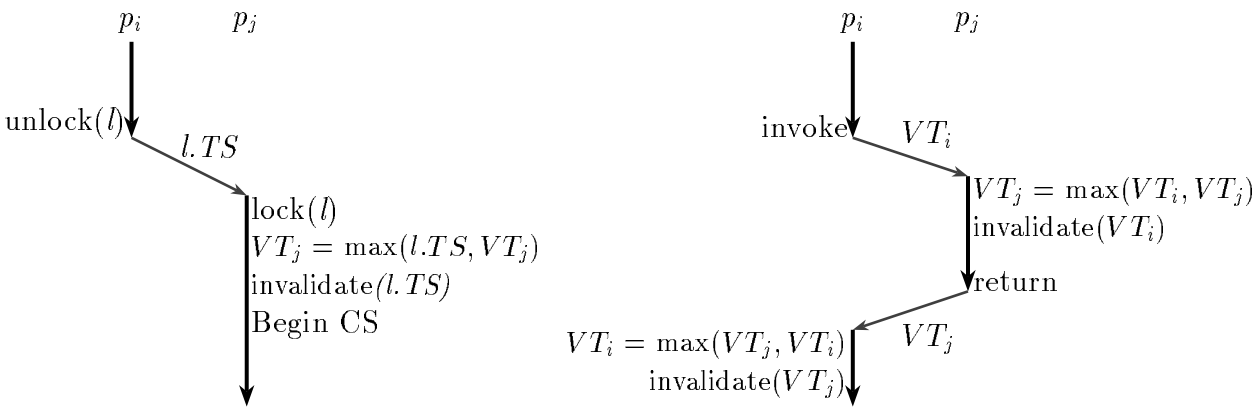
3.2.3 Synchronization Operations

Our implementation is designed to work correctly when programs are data-race-free. Thus, we need to consider the effects of synchronization 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). Examples of these operations include lock, semaphore and barrier calls. 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.

We show the acquire and release actions on a lock variable in Figure 5(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 5 is explained in the next section.

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 5(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.

The updating of clocks when acquire operations complete and the fact that programs are assumed to be data-race-free, 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



(a) Handling Synchronization

(b) Handling Invocations

Figure 5: Synchronization and Invocations

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 data-race-free 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.

In Figure 4, the write fault action does update the vector clock VT_i before generating a timestamp for the page being written. This is necessary, even when programs are data-race-free and a lock is acquired before the write is done, due to false sharing problems. We explain this using the execution shown in Figure 6. 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 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 4 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.

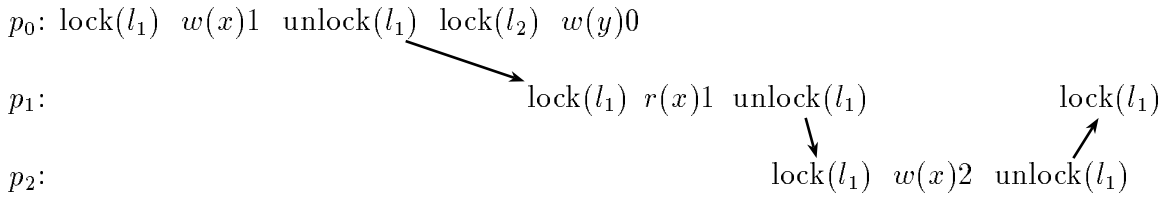


Figure 6: Example to show why clock update is necessary on write faults

3.2.4 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 $o_1 \stackrel{eco}{\rightarrow} o_2$ 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. When only data-race-free programs are considered, data consistency operations can be limited to synchronization acquires, fork-joins and barrier points in the execution of a program. This does not affect the correctness of the implementation because of the argument that with such programs, the extended causal order $\stackrel{eco}{\rightarrow}$ is the same as the order defined by the synchronization operations, $\stackrel{so}{\rightarrow}$. As a result, no new causal orderings can be created by a read operation o_r because the acquire, that preceded o_r, has already ordered the write operation that produced the data being read by o_r.

The basic consistency maintenance operation, *invalidate(timestamp TS)*, is shown in Figure 7. 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.

We claim that 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 processor.

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

Figure 7: The invalidate operator

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 data-race-free, 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.

Locally invalidating pages obviates the need for explicit invalidation messages. Performing local invalidations reduces the number of messages; it also saves the extra software overhead 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 may invalidate more pages than strictly necessary.

3.2.5 Unnecessary Invalidations

We argued that, by locally invalidating pages with lower timestamps than the timestamp received when an acquire operation is completed, we guarantee that causally overwritten data cannot be read. Such 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. The vector timestamps allow us to determine if a page is old but they do not have information that allows us to decide if a more recent version of the page exists. For instance, consider the execution shown in Figure 8. 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.

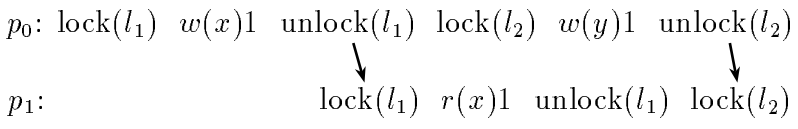


Figure 8: Unnecessary invalidation of page containing data item x

When most of the shared data accesses to a page are reads, the page could get unnecessarily invalidated quite often, leading to significant performance degradation. In many of the applications with which we have experimented, there are data that are only read or that are once written at the beginning of the execution and then only read. Such data, which does not change during the course of program execution, could get unnecessarily invalidated when the consistency maintenance operations are executed. One solution is to use an approach similar to the *annotations* used in the Munin system [12]. We provide language level support, where the user can annotate such data as *read-only* or *write-once*. Such data pages are not considered for invalidation. This annotation is used only for improving performance and is not required for correct execution.

Data which have mostly read characteristics but are not *read-only* or *write-once* would still get invalidated unnecessarily. We modify the algorithm given here 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. In the following section, we first present the modified algorithm and then discuss the performance improvements and the overhead introduced by it.

3.3 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 9, which eliminates the unnecessary invalidations, version numbers associated with pages are used to determine when the data in a page are causally overwritten[4]. 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 array*, VA_i, which stores the latest version number known to p_i of each of the shared pages. $VA_i[x.num]$ 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 VA_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.

3.3.1 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 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

[4]Version numbers were used in the Locus file system for detecting concurrent updates to a single file [37]. 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 (i is p_i's identifier)

```
[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 9: Causal memory implementation using versioned pages

the page data will be created by the processor, the received version number of the page is incremented and the corresponding entry in VA_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 array 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 array 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 VA_i. When processor p_j next acquires this lock, the lock's timestamp is used to update VA_j. Similar to Figure 5 for vector clocks, a maximum operation is performed on each component of VA_j and the timestamp received with the lock, and the result is assigned to VA_j. Other synchronization constructs such as barriers similarly update the version arrays at the participating processors.

3.3.2 Maintaining Data Consistency

Causal consistency of shared pages is maintained by calling the *invalidate* operation, shown in Figure 10, when a lock is acquired or a barrier completed. The operation is also executed at fork and join operations. 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).

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

Figure 10: The invalidate operator for versioned pages

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.

3.4 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 array 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 array. 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 arrays. Pages that are *read-only* or *write-once* are not considered for invalidation and do not need to have version numbers associated with them.

3.5 Additional Optimizations

Our implementations of causal memory also allow several other optimizations. We briefly describe two of them here.

3.5.1 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 data-race-free.

Consider the case in which an owner processor p_1 receives a request for a page due to a read fault at processor p_2. At p_1, 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_1 would raise a protection fault on a write, which would be handled locally as p_1 is still the owner. Also, because programs are free of data races, the data in the page being written by p_1 will not be read by p_2, 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_1 writes to parts of the page for which p_1 holds an exclusive lock. Thus writes by p_1 while the page is being transmitted to p_2 will not change any data that is accessed by p_2 and hence the page need not be copied by the communication protocol. If p_2 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_1 generated a newer version since the last acquire operation of p_2. 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 [33, 17]. However, they require additional flexibility from the underlying operating system.

3.5.2 Avoiding Page Transfers on Double Faults

A *double fault* occurs when a page that is not cached locally is first read and then written [28]. 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 4 and 9, 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.

4 Evaluation of Causal Memory

To evaluate the performance benefits made possible by causal memory, we implemented both of the algorithms discussed in the previous section. To compare causal memory with a strongly consistent memory, we also implemented a DSM system that provides sequential consistency[5]. In addition, our system provides support for message passing on the same platform. The sequentially consistent DSM protocol implements a fixed manager writer-

[5]In Section 5, we address how the performance of causal memory compares with other memory systems such as release consistency, lazy release consistency and others.

invalidate-readers protocol similar to the one described by Li and Hudak [32]. We optimized it 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 [28] to avoid re-sending a page due to a double page fault. All of the DSM protocols are implemented in the Clouds operating system using low level communication mechanisms. Thus, coherence 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 level. These calls allow processes to only exchange the data they share and the data transfer is not tied to pages. 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.

We first provide brief descriptions of the applications used in the evaluation of the causal memory system. The performance measures used in the study are described next. Finally, we present and analyze the experimental results.

4.1 Applications

We have implemented a number of applications to evaluate causal memory. The applications include Embarrassingly Parallel (EP), Integer Sort (IS), and Conjugate Gradient Method (CGM) from the NASA Ames NAS kernels [9], 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.

We first provide details of the applications and then discuss how they have been programmed in the different systems.

- 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 (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. We chose a matrix size of 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. [16]. 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. Our implementation is similar to the one reported by Bal et al. [10] 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. We did use some simple annotations in the program 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.

4.2 Performance Measures

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. We present a general overview of the results and then give a detailed analysis for three of the applications.

4.3 Results

All the applications were run on SUN 3/60's connected over a 10Mbits/s ethernet[6]. The page-size, which is the unit of sharing in the memory systems, is 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 were not significantly different and we discuss this later.

Figure 11 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.

[6]We discuss the impact of faster processors and networks in Section 4.4.4.

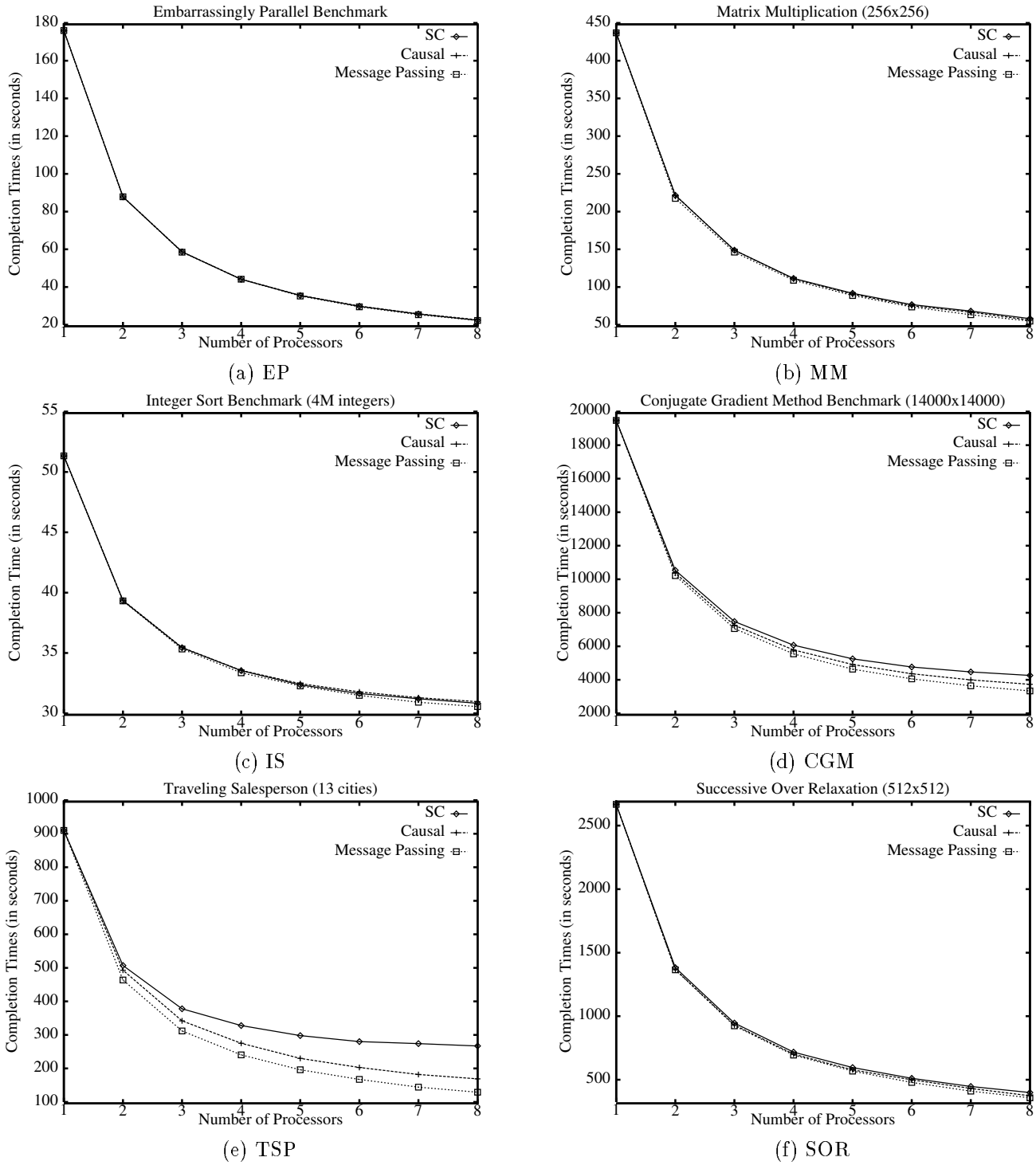
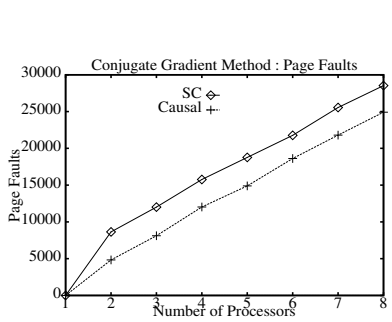
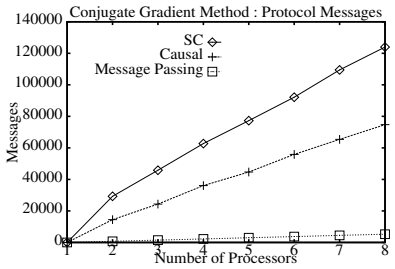


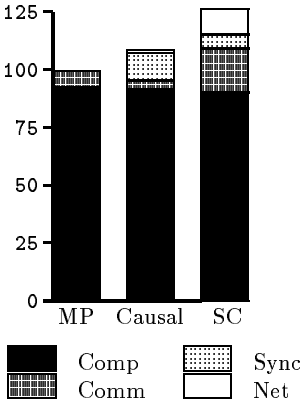
Figure 11: Execution Times



(a) Page Faults



(b) Messages



(c) Execution Profile

Figure 12: CGM analysis

3. The writes to shared data at different processors are to disjoint parts 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, we discuss these three applications in more detail in the following subsections.

4.3.1 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. We ran the program with matrices of size 14000 $\times$ 14000. The temporary arrays are of size 14000.

1. **Completion time:** Figure 11(d) compares the performance of CGM with 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 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 12(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 12(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 12(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[7] 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 13.1MB and 12.9MB, for sequentially consistent and causal memory systems, is close. Furthermore, in the message passing system, 10.5MB 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]The underlying protocol could fragment this message but we are counting only the number of times the protocol is invoked to send a message.

4.3.2 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 processor only if it has found a better tour.

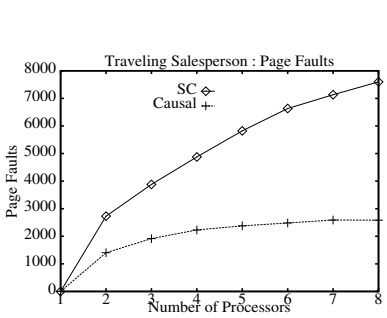
The behavior of TSP on the three systems is very different. With sequentially consistent memory, the best-tour value gets propagated immediately to all processors whenever it gets updated, since any cached value would be invalidated before the write. Causal memory allows out-of-date values of best-tour 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[8]. This was observed in the experiments that we ran. The number of nodes searched in the three cases are shown in Figure 13. The number of nodes searched by the message passing code is 7% higher than sequentially consistent memory. We discuss the various component times for TSP below.

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

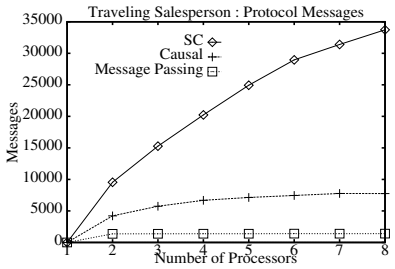
Figure 13: Nodes visited for TSP

1. **Completion time:** Figure 11(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. 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. There is another reason that message passing provides better completion time. Both the memory systems suffer because of the mismatch

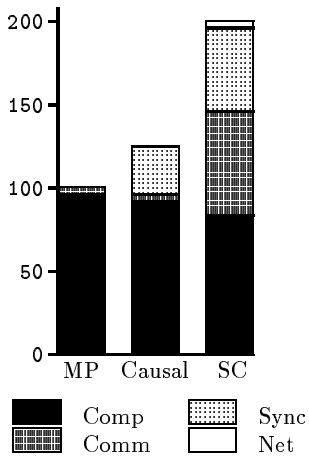
[8]We can make the sender transmit new best-tour values as soon as they arrive but this does not solve the problem. The other processors must receive these values and due to the asynchronous nature of when these values arrive, one cannot code points in the program where a *msgreceive* should be executed.



(a) Page Faults



(b) Messages



(c) Execution Profile

Figure 14: TSP analysis

between the amount of shared data and the page size, which is the unit of coherence. We discuss this when we consider the number of messages and the amount of data transferred.

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 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 14(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.

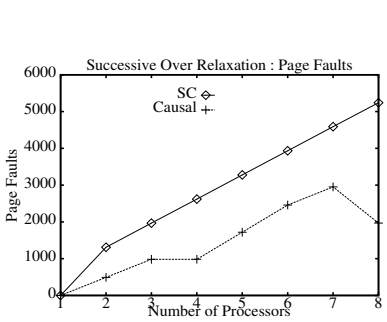
As mentioned earlier, in both memory systems, the unit of coherence is a page and hence 8K bytes are transferred even when two integers (current-job and best-tour) are shared. This explains why the message passing system sends around 18 Kbytes of data, whereas the sequentially consistent and causal memory systems send 48.8MB and 20.2MB of data respectively. As seen in Figure 14, avoiding unnecessary data transfer does result in better completion time for message passing because it reduces the communication time significantly.

4.3.3 SOR

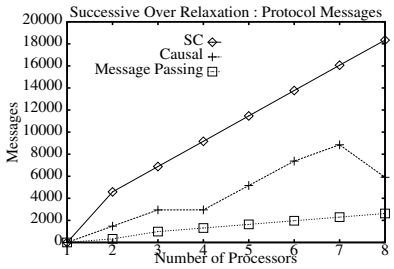
Data Sharing Characteristics: The data sharing pattern in SOR is quite different from the applications discussed so far. The computation consists of a sequence of iterations. Each iteration consists of two phases, an odd phase and an even phase, which are separated by barriers. Each processor computes 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 11(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 15(c), the computation time dominates the completion time and, since it is the same in all three systems, the completion 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 mismatch between the data granularity (512 elements of floats — 2048 bytes) and the large page size (8192 bytes).

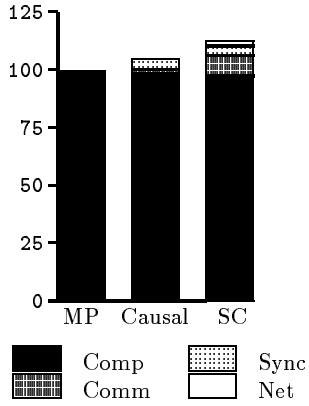
2. **Page faults:** Figure 15(a) shows the number of page faults on the two memory systems. SOR on causal memory takes 62% fewer page faults compared to the sequentially consistent memory when the application is run on 8 processors. This is for two reasons. First, because of the producer-consumer nature of data sharing, a



(a) Page Faults



(b) Messages



(c) Execution Profile

Figure 15: SOR analysis

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. We use Figure 16 to explain this difference. In the first phase of the iteration, because of the order in

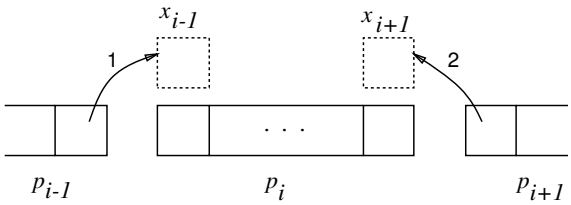


Figure 16: Data sharing for SOR

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 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.

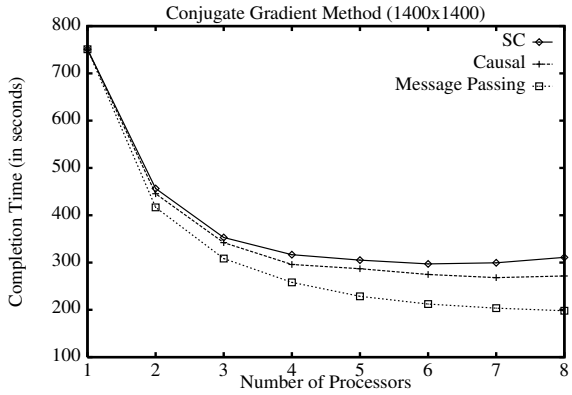
3. **Messages:** Figure 15(b) compares the number of messages sent by the three systems while executing SOR. Causal memory sends 68% fewer messages than sequentially consistent memory because of fewer page faults and the fact that invalidations are local. The amounts of data transferred in the messages in sequentially consistent, causal and message passing systems are 20.6MB, 15.4MB and 5.1MB respectively. The memory systems send more data due to the large page size as explained earlier.

4.4 Discussion

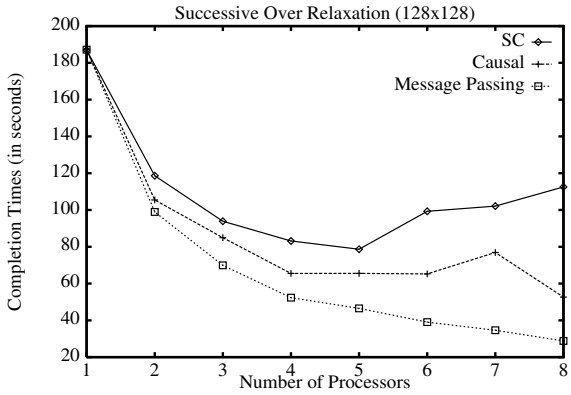
In our results, we see that causal memory performs better than sequentially consistent memory system. This is due to two primary reasons: it tolerates false sharing between readers and a writer, and it sends fewer messages. We discuss both of these issues and also comment on performance of causal memory when vector timestamps instead of version numbers are used in its implementation. We compare causal memory with other memory systems in Section 5.

4.4.1 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 17(a)



(a) CGM



(b) SOR

Figure 17: Effect of False Sharing

shows the completion times for CGM when the problem size is 1400 (the result discussed earlier were for size 14000). We chose this size 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.

We also ran SOR with grid size 128×128, which results in only 8 pages of shared data (the earlier results were for 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 17(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.

4.4.2 Causal Memory Implementations

We ran all the applications with both implementations of causal memory that have been discussed in this paper. 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 18.

	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 18: Comparing the two implementations

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

4.4.3 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 12, 14, and 15 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 5), 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 [41] 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.

4.4.4 Impact of Faster Processors and Network

Our 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 [40]. The memory systems will also become more competitive with message passing in architectures that have smaller page sizes or support multiple page sizes.

5 Related Work

Many approaches for building DSM systems have been proposed. In his pioneering work, Li adapted a multiprocessor cache coherence protocol to develop a software based implementation of shared memory in a distributed system. Although several improvements have been suggested to this approach [20, 28], exploiting synchronization information and weaker consistency for shared memory represent two significant advancements in handling the performance problems associated with DSM systems. In this section, we compare the causal memory system presented in this paper to systems that make use of synchronization information in maintaining coherence. We also relate causal memory to other weakly consistent memory systems.

5.1 Memory Models

The use of synchronization information in coherence maintenance was first advocated by Dubois and Scheurich [18] for multiprocessor systems. They argued that only the execution of synchronization operations needs to be sequentially consistent as long as coherence actions for other memory operations complete by the time a following synchronization operation is executed. Thus, on a write to a shared data item, invalidation messages need not be delivered and processed by all processors that have the data cached, before the writing processor is allowed to issue the next memory request. These invalidations must complete

before a following synchronization operation is completed. This approach can provide improved performance because execution of a non-synchronization operation is not delayed until communication completes between processors.

Release consistency (RC), proposed by Gharachorloo et. al., further refined the weakly ordered approach by dividing synchronization operations in two types: acquire and release [21]. It is only necessary that coherence actions are performed before a release operation is completed. In the version of RC where synchronization operations are sequentially consistent, it was shown that with proper labeling of memory operations (acquire or release synchronization operations or ordinary operations), one can develop programs on a RC system assuming that it provides sequential consistency. Thus, the optimizations made possible by RC in implementing memory consistency do not increase the complexity of programming. Adve and Hill [1], also developed a similar approach and introduced the notion of data-race-free programs. Hybrid consistency [8] and buffered consistency [38] are examples of other memory models that are based on classifying synchronization operations and defining coherence relative to them.

Lazy release consistency (LRC) [26] and entry consistency (EC) [13], both of which were proposed after RC, make even more aggressive use of synchronization in performing coherence actions. Although their implementations differ, the key idea used by both is that changes to shared data are made inside critical sections controlled by synchronization variables (e.g., locks), and such changes only need to be propagated to the processor that next enters the critical section. In contrast, RC requires that when shared data is modified in a critical section, all processors caching the data item be sent a request to invalidate or update its cached value before the release operation completes. The determination of which processor next acquires a synchronization variable is done at runtime. Thus, in LRC and EC, coherence actions can be delayed at a processor until it succeeds in acquiring a synchronization variable.

RC, LRC, EC and other similar systems maintain a sequentially consistent memory interface for programmers as long as their applications are properly synchronized (e.g., are data-race-free). Other memory models have been proposed that make the weaker consistency of a memory system visible to the programmers. Examples of these include pipelined RAM (PRAM) [34], processor consistency (PC) [22, 21], and causal and slow memories [5, 24]. Programmers must either show that programs are not affected by the weaker consistency (e.g., conservative programs with respect to PRAM [34]) or must include code in their applications that deals with such weak consistency. It is argued that better performance can be achieved in these systems because strong consistency is not provided for any set of memory operations.

The causal memory system that we explore in this paper bears similarities to both approaches: providing weak consistency and exploiting synchronization. Weak consistency is provided because processors are not guaranteed a single view or serial order of all memory operations as is done in a sequentially consistent memory system. By assuming that programs are data-race-free, we exploit synchronization information because it ensures that memory operations on non-synchronization data do not create additional causal orderings.

This observation allows important optimizations in the implementation of our system which are discussed next.

5.2 DSM Systems

Several DSM systems have been implemented. Ivy, the first DSM system, implemented sequentially consistent memory by using a writer-invalidates-readers protocol. In Mirage [20], pinning was implemented to avoid certain problems in the Ivy protocol. Munin [11], implemented a family of protocols, including the first software implementation of RC. Munin showed that performance competitive with hand coded message passing programs can be achieved if data sharing patterns of an application can be identified and appropriate consistency protocols can be associated with shared data items. User specified annotations were used in Munin for this purpose. LRC is implemented in the TreadMarks system [27] and the Midway system implements EC. In LRC, writes to shared data are propagated when locks are transferred between processors. To ensure that the processor that acquires the lock next receives all changes to shared data that were known to the processor that released the lock, causal dependencies are recorded using vector timestamps and a history based mechanism is used to determine what data modifications have to be transmitted with the lock transfers. LRC handles false sharing by allowing multiple concurrent writers for a page but a *diff* mechanism is used to merge changes made by concurrent writes. The implementation of EC in the Midway system guarantees optimal data transfer. This is achieved by two separate mechanisms. First, like LRC, modifications to shared data are transmitted with transfer of synchronization variables. Second, Midway requires programmer to establish associations between shared data items and synchronization variables. This allows it to transfer changes to only the data items that are associated with the synchronization variable being transferred. LRC, on the other hand, must include information about changes to all shared data.

Our implementation of causal memory differs from Ivy and the Munin implementation of RC because processors that cache a data item being written are not notified either on the write or when a release operation on a synchronization variable is executed. Since data-race-free programs allow us to perform consistency related actions only when an acquire operation completes at a processor, the operation of our implementation resembles the TreadMarks and Midway systems that implement LRC and EC, respectively. However, not only do we differ in how we arrive at this particular implementation of causal memory, there are also several significant operational differences. We use synchronization to propagate causal dependencies whereas TreadMarks uses it to identify the processors at which data must be made consistent. TreadMarks handles multiple writers by creating page copies and by using *diff* operations. We have found that the processing overhead associated with *diff* operations can be significant [25].

The other difference between causal memory implementation and TreadMarks is in the amount of state which is maintained to track pages that have been actually modified. TreadMarks tries to identify exactly the pages which have been modified and invalidates

only those pages. Although this leads to fewer invalidations/updates, the state grows rapidly in size and garbage collection is required to limit it. Our approach with vector clocks is rather pessimistic, in that we invalidate any page which has a lower timestamp even though the page may not have been modified. This may lead to more messages but the additional complexity of maintaining write records and garbage collection is avoided. In the version array implementation, we avoid unnecessary invalidations but more data may be sent when synchronization variables are transferred between processors. LRC could also perform some unnecessary invalidations (or updates) due to false sharing. Note that LRC only maintains the information that a certain page has been modified in a particular synchronization interval. However, due to false sharing, another processor p_i could read the page after it has been modified. But if p_i now acquired the synchronization variable, its page would still get invalidated (or updated) even though it would have the current copy of the page.

The Midway implementation uses compile-time support to track updates to shared data. It can send with a synchronization variable the exact set of data items that will be accessed because of the explicit associations. Because of this, it does not need to transfer information about updates to other data items as is done in TreadMarks or with causal memory.

We have compared the performance of applications running on causal memory with an Ivy like protocol that implements sequentially consistent memory and a message passing system. A natural question is how it compares with the implementations of RC, LRC or EC. We have also implemented RC and our results show that causal memory performs better than RC [25]. This is because RC does not reduce the number of messages; invalidation or update messages have to be sent to all processors caching the modified data items when a release operation is executed. Causal memory does not send any messages on a release operation. In fact, we found that the causal memory implementation performed better than the RC implementation for all of the applications we described.

We feel that despite the many differences in the operation of their implementations, the performance of causal memory will be close to LRC and EC. This is because synchronization is used to provide memory coherence in a similar way in these systems. The most recent implementation of LRC includes many optimizations (including latency reduction techniques) which could provide somewhat better performance. Such optimizations, which may require programmer assistance, can also be used with causal memory. EC has an advantage over both causal and LRC because of the explicit associations it requires. These associations allow it to determine the exact set of data to be transmitted with little processing overhead. Thus, it could provide better performance but requires that programmers do additional work to specify the associations. Since its performance must be between message passing and causal memory, we believe the differences in these systems will not be significant.

Other implementations of DSM systems have also been reported. Boyer [15] describes an implementation of causal DSM on Mach using external pagers. Simple message counting arguments are presented to show its superior performance over conventional atomic DSM.

There have been several hardware implementations of weakly ordered systems and are described in [21, 2, 39, 14, 19].

6 Concluding Remarks

We have presented a DSM system based on causal memory and have given two implementations for it. These implementations make use of the optimizations made possible by weakly ordered as well as weakly consistent systems. Since these are the two fundamental techniques that have been proposed for improving the performance of DSM systems, we feel that the causal memory system that we present does provide a highly efficient implementation of shared memory in a distributed system. By actually implementing this system along with a sequentially consistent memory system and message passing, we are able to evaluate it experimentally. We used six applications to capture a wide range of data sharing patterns. These applications can be programmed on causal memory the same way as on sequentially consistent memory. Our results show that causal memory does lead to improvement in performance. In particular, it significantly reduces the number of messages exchanged between processors when the applications are executed. We believe that our results support our claim that causal memory is easy to use and can provide performance close to message passing systems.

Future research must address many problems to further evaluate causal memory. For example, it needs to be evaluated with faster processors and higher speed network than the one we had available in our study. Furthermore, a user level implementation can make the implementations accessible to a wide range of researchers. As the implementations of LRC and EC become mature, it is necessary that causal memory be implemented along with these memory systems using a uniform set of mechanisms. This will allow us to evaluate the relative strengths and weaknesses of these systems.

We used scientific applications to evaluate the memory systems. Such applications do not capture data sharing patterns of many distributed applications. For example, applications that support cooperation between asynchronous users (e.g., collaborative editors), have very different kind of state sharing requirements. In future, we plan to develop causal memory support for such applications.

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] Divyakant Agrawal, Manhoi Choy, Hong Va 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] Hagit Attiya and Roy Friedman. A correctness condition for high-performance multiprocessors. Extended Abstract, Department of Computer Science, The Technion,Haifa, Israel, November 1991.

[9] 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.

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

[11] 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.

[12] John K. Bennett, John B. Carter, and Willy Zwaenpoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming*, pages 168–176, March 1990.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

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

[24] Phil W. Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enhance concurrency in distrbuted shared memories. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 302–311, 1990.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, c-28(9):690–691, September 1979.

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

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

[34] 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.

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

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

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

[38] Gautam Shah and Umakishore Ramachandran. Towards exploiting the architectural features of beehive. In *International Workshop on Scalable Shared Memory Multiprocessors*, May 1993.

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

[40] 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.

[41] 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.