Causal Memory: Definitions, Implementation and Programming[*]

Mustaque Ahamad
Gil Neiger
James E. Burns[†]
Prince Kohli
Georgia Institute of Technology[‡]

Phillip W. Hutto[§]

GIT–CC–93/55

September 17, 1993
Revised: July 22, 1994

Abstract

The abstraction of a shared memory is of growing importance in distributed computing systems. Traditional memory consistency ensures that all processes agree on a common order of all operations on memory. Unfortunately, providing these guarantees entails access latencies that prevent scaling to large systems. This paper weakens such guarantees by defining *causal memory*, an abstraction that ensures that processes in a system agree on the relative ordering of operations that are *causally related*. Because causal memory is *weakly consistent*, it admits more executions, and hence more concurrency, than either atomic or sequentially consistent memories. This paper provides a formal definition of causal memory and gives an implementation for message-passing systems. In addition, it describes a practical class of programs that, if developed for a strongly consistent memory, run correctly with causal memory.

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

[*] This work was supported in part by the National Science Foundation under grants CCR-8619886, CCR-8909663, CCR-9106627, and CCR-9301454. Parts of this paper appeared in S. Toueg, P. G. Spirakis, and L. Kirousis, editors, *Proceedings of the Fifth International Workshop on Distributed Algorithms*, volume 579 of *Lecture Notes on Computer Science*, pages 9–30, Springer-Verlag, October 1991.

[†] Author's current address: Bellcore, NVC 3X-114, 331 Newman Springs Road, Post Office Box 7040, Red Bank, New Jersey 07701-7040.

[‡] Authors' address: College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332-0280.

[§] Author's address: 609 Virginia Avenue NE, Atlanta, Georgia 30306.

1 Introduction

The abstraction of a shared memory is of growing importance in distributed computing systems. It allows users to program these systems without concerning themselves with the details of the underlying message-passing system. Traditionally, distributed shared memories ensure that all processes in the system agree on a common order of all operations on memory. Such guarantees are provided by sequentially consistent memory [27] and by atomic memory [28] (also called linearizable memory [20]). Unfortunately, providing these consistency guarantees entails access latencies that prevent scaling to large systems. A simple argument [10,29] can be used to show that no memory can provide strong consistency and retain low latency in systems with high message-passing delays. This tradeoff represents a significant efficiency problem since it forces applications to pay the costs of consistency even if they are highly parallel and involve little synchronization. A number of techniques [11,24] have been suggested to improve the efficiency of shared memory implementations, but all provide only partial remedies to the fundamental problem of latency and scale for strongly consistent memories.

Recent research [1,6,9,16–18,21,29] suggests that a systematic weakening of memory consistency can reduce the costs of providing consistency while maintaining a viable "target" model for programmers. Weakly consistent memories admit more executions, and hence more concurrency, than either sequentially consistent or atomic memories. This paper defines *causal memory*, an abstraction that ensures that processes in a system agree on the relative ordering of operations that are *causally related*. (Causal memory has been mentioned earlier [6,21]; however, these papers do not present careful definitions as is done here.) This paper provides a formal definition of causal memory and gives an implementation for message-passing systems. We give two classes of programs that can be developed assuming a sequentially consistent memory and that run correctly with causal memory.

Causal memory is based on Lamport's concept of *potential causality* [26]. Potential causality provides a natural ordering on events in a distributed system where processes communicate via message passing. We introduce a similar notion of causality based on reads and writes in a shared memory environment. Causal memory requires that reads return values consistent with causally related reads and writes, and we say that "reads respect the order of causally related writes." Since causality orders events only partially, reading processes may disagree on the relative ordering of concurrent writes. This provides independence between concurrent writers which reduces consistency maintenance (synchronization) costs. The idea is that the synchronization required by a program is often specified explicitly and it is not necessary for the memory to provide additional synchronization guarantees.

Causal memory is related to the ISIS causal broadcast and, thereby, to the notion of causally ordered messages [13]. Our implementation of causal memory is based on the use of vector timestamps [14,30], as is the ISIS implementation of causal broadcast. Both implementations are "non-blocking": a process may complete an operation (e.g., a write or a send) without waiting for communication with other processes. Nevertheless, causal memory is more than a collection of "locations" updated by causal broadcasts. Memory has overwrite semantics and messages have queuing semantics. A message recipient can be assured that it will eventually receive all messages that have been sent to it, but repeated reads cannot guarantee that all values written will be read. "Hidden writes", values overwritten before they are read, are always possible. Since a process may read memory locations in any order it chooses, it may read a value v_1 from location x much later than a value v_2 from location y, even when the write operation that stores v_1 in x is causally before the write of v_2 to y. In a message-passing system, such behavior would violate the required causal ordering.

We give precise characterizations of two classes of programs that run correctly with causal memory. Any execution a program in either of these classes with causal memory is actually sequentially consistent. If the program is proven correct with sequential consistent memory, then it is still correct with causal memory. One of these classes includes *data-race free* programs [1,2] that make use of explicit synchronization to prevent problems that may stem from concurrent access to shared memory.

It is far from clear that there is a "best" kind of shared memory model for use with distributed systems. Strongly consistent memories are easier to program than weak memories, but they require costly blocking implementations. Very weak memories may be implemented cheaply, but they might not be practical to program. We believe that causal memory provides a happy medium: it allows non-blocking implementations and is a useful model for a class of practical programs.

2 Shared Memory Systems

This section formally describes the system that underlies our definitions and results. We use a model derived from those used by Herlihy and Wing [20] and by Misra [33].

We define a *system* to be a finite set of processes that interact via a shared memory that consists of a finite set of *locations*. Let $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ be the set of processes. A process's interaction with the memory is through a series of read and write *operations* on the memory. Each such operation acts on some named location and has an associated value. For example, a write operation by process p_i, denoted $w_i(x)v$, stores the value v in location x; a similarly denoted read operation, $r_i(x)v$, reports to p_i that v is stored in location x.

A *local execution history* (or *local history*) of process p_i, denoted L_i, is a sequence of read and write operations. If operation o_1 precedes o_2 in L_i, we write $o_1 \underset{i}{\rightarrow} o_2$ and say that o_1 *precedes* o_2 *in program order*. An *execution history* (or *history*) $H = \langle L_1, L_2, \ldots, L_n \rangle$ is a collection of local histories, one for each process. An operation is said to be in H if it is in one of the local histories that H comprises.

Different kinds of memories are defined by considering *serializations* of certain sets of operations. If A (or, respectively, H) is a set of operations (or history), then S is a serialization of A (or H) if S is a linear sequence containing exactly the operations of A (or H) such that each read operation from a location returns the value written by the most recent preceding write to that location. (Unless otherwise stated, we assume that each location has initial value $\perp$ and that this value is returned by any read of a location with no preceding write.) *Serialization* S *respects order* $\rightarrow$ if, for any operations o_1 and o_2 in S, $o_1 \rightarrow o_2$ implies that o_1 precedes o_2 in S.

3 Earlier Memory Models

Given the formalism developed above, one can define a variety of memory consistency models. This section defines Lamport's sequential consistency [27] and the PRAM of Lipton and Sandberg [29]. The next section uses the same formalism to define causal memory.

The idea behind sequential consistency is that, although the shared memory accessed by processes may be distributed (i.e., may consist of many different modules), the processes' observations of the memory should be consistent with one that permits only sequential accesses (i.e., a single memory). History H is *sequentially consistent* if it satisfies the following:

SC: there is a serialization S of H that respects all the program orders $\xrightarrow[i]{}$.

Thus, the values returned by the read operations in H are consistent with the sequential ordering in S. If processes only communicate via the shared memory, they cannot tell, by way of their interactions with the memory, that they are not accessing a single memory. A memory is sequentially consistent if it admits only sequentially consistent histories.[1]

Recognizing that sequential consistency is costly to implement, Lipton and Sandberg developed a weaker form of memory that they called the *pipelined RAM* or *PRAM*. This memory requires only that the writes of each process be seen in program order at all other processes. Thus, each process must sequence its own operations and the writes of other processes. For this reason, we make the following definition. If H is a history and p_i is a process, let A^H_{i+w} comprise all operations in L_i and all write operations in H. A history H is PRAM if it satisfies the following:

PRAM: for each process p_i, there is a serialization S_i of A^H_{i+w} that respects all the program orders $\xrightarrow[j]{}$.

A memory is PRAM if it admits only PRAM histories.

Notice that both sequential consistency and PRAM require serializations that respect program order. PRAM is weaker than sequential consistency because each process may "perceive" a different serialization. While the order of two writes by a given process must be the same in all these serializations (even those for other processes), writes by different processes may appear in different orders in different serializations. Furthermore, each process's serialization does not contain the read operations of other processes, as it is not (directly) aware of these operations. Figure 1 gives an example of a PRAM history that is not sequentially consistent. This history is PRAM because the following serializations exist:

$$\begin{aligned} S_1 &= w_1(x)0;\ w_2(x)1;\ r_1(x)1 \\ S_2 &= w_2(x)1;\ w_1(x)0;\ r_2(x)0 \end{aligned}$$

We now show that the history is not sequentially consistent. Suppose that it were and

[1] A memory is *atomic* (or *linearizable*) if each history admits a serialization that not only preserves the order within the local histories but also that of any pair of operations whose executions do not overlap in real time [20,28]. The definition of such memories is beyond the scope of this paper.

$$\begin{array}{lll} p_1: & \boxed{w(x)0} & \boxed{r(x)1} \\ p_2: & \boxed{w(x)1} & \boxed{r(x)0} \end{array}$$

Figure 1: A history that is not sequentially consistent

let S be the required serialization. Inspection of L_1 shows that $w_1(x)0; w_2(x)1; r_1(x)1$ must appear in that order in S. Inspection of L_2 shows that $w_2(x)1; w_1(x)0; r_2(x)0$ must appear in that order in S. This gives a contradiction, as $w_1(x)0$ and $w_2(x)1$ must be ordered uniquely.

Slow memory given by Hutto and Ahamad [21] can also be defined using this formalism, as can processor consistency [4,16,18]. We are currently exploring the use of this formalism in the definition of other memories [25].

4 Causal Memory

We define causal memory to be intermediate between sequential consistency and PRAM. Its definition is similar to that of PRAM but is stronger because the serializations required must respect not only program order but a causality order as well. We first define causality orders.

Let $H = \langle L_1, L_2, \ldots, L_n \rangle$. A causality order of operations in H is determined by program order and a *writes-into* order that associates a write operation with each read operation (except one of a location's initial value). The writes-into order is analogous to the order in message passing systems that relates the sending of a message to its corresponding receipt. The order in message-passing systems is easier to define because, for each message receipt, there is a unique sending event. This is not the case in shared-memory systems: several write operations may write the same value to the same location, and it is not always clear which to associate with a particular read operation. (Misra simplified this situation by assuming that all writes to a location are uniquely valued.)

Because there may be multiple writes of a value to a location, there may be more than one writes-into order. A writes-into order $\mapsto$ on H is any relation with the following properties:

- if $o_1 \mapsto o_2$, then there are x and v such that $o_1 = w(x)v$ and $o_2 = r(x)v$;
- for any operation o_2, there is at most one o_1 such that $o_1 \mapsto o_2$;
- if $o_2 = r(x)v$ for some x and there is no o_1 such that $o_1 \mapsto o_2$, then $v = \perp$; that is, a read with no write must read the initial value.

A *causality order* $\leadsto$ *induced by* $\mapsto$ *for* H is a partial order that is the transitive closure of the union of the history's program order and the order $\mapsto$. In other words, $o_1 \leadsto o_2$ if and only if one of the following cases holds:

p_1:	$\boxed{w(x)0}$	$\boxed{w(x)1}$		
p_2:		$\boxed{r(x)1}$	$\boxed{w(y)2}$	
p_3:			$\boxed{r(y)2}$	$\boxed{r(x)0}$

Figure 2: A history that is not causal

- $o_1 \underset{i}{\rightarrow} o_2$ for some p_i (o_1 precedes o_2 in L_i);
- $o_1 \mapsto o_2$ (o_2 reads the value written by o_1); or
- there is some other operation o' such that $o_1 \rightsquigarrow o' \rightsquigarrow o_2$.

(If the relation $\rightsquigarrow$ is cyclic, then it is not a causality order.) If o_1 and o_2 are two operations in H such that, for causality order $\rightsquigarrow$, $o_1 \not\rightsquigarrow o_2$ and $o_2 \not\rightsquigarrow o_1$, we say that o_1 and o_2 are *concurrent with respect to* $\rightsquigarrow$.

We can now define causal memory. A history H is *causal* if it has a causality order $\rightsquigarrow$ such that

CM: for each process p_i, there is a serialization S_i of A^H_{i+w} that respects $\rightsquigarrow$.

A memory is *causal* if it admits only causal histories. Again, this is weaker than sequential consistency because each process may "perceive" a different serialization. Figure 1, given above, is causal but not sequentially consistent (it is causal because the serializations S_1 and S_2, given above, exist).

If $o_1 \underset{i}{\rightarrow} o_2$ in some p_i, then $o_1 \rightsquigarrow o_2$ for all causality orders $\rightsquigarrow$; thus, it should be clear that any causal history is also PRAM. However, not all PRAM histories are causal. Figure 2 gives an example of a history that is PRAM but not causal. It is PRAM because the following serializations, each consistent with program order, exist:

$$\begin{aligned} S_1 &= w_1(x)0;\ w_1(x)1;\ w_2(y)2; \\ S_2 &= w_1(x)0;\ w_1(x)1;\ r_2(x)1;\ w_2(y)2; \\ S_3 &= w_2(y)2;\ r_3(y)2;\ w_1(x)0;\ r_3(x)0;\ w_1(x)1. \end{aligned}$$

This history is not causal for the following reason. There is only one possible writes-into order: $w_1(x)0 \mapsto r_3(x)0$, $w_1(x)1 \mapsto r_2(x)1$, and $w_2(y)2 \mapsto r_3(y)2$. Thus, H has only one causality order, and the following relations hold: $w_1(x)0 \underset{1}{\rightarrow} w_1(x)1 \mapsto r_2(x)1 \underset{2}{\rightarrow} w_2(y)2$. Thus, $w_1(x)0 \rightsquigarrow w_1(x)1 \rightsquigarrow w_2(y)2$, and the three writes must appear in that order in all serializations. It is clear that there is no way to construct S_3 (that respects the program order in L_3) with the writes in that order so that each read (by p_3) returns the most recently written value to the location being read. Clearly, $r_3(y)2$ would have to follow $w_2(y)2$, so the only choice for S_3 is $w_1(x)0; w_1(x)1; w_2(y)2; r_3(y)2; r_3(x)0$. This is not a serialization (the last read should return 1), so the history is not causal.

5 An Implementation of Causal Memory

This section presents and proves correct an implementation of causal memory using message passing. The implementation uses an adaptation of *vector timestamps* [14,30]. It requires reliable processes and communication channels.

Each process maintains four local data structures. The first is a private copy M of the abstract shared causal memory $\mathcal{M}$. The second is a vector clock t, which is used to timestamp outgoing messages. This is a vector of natural numbers, one for each process in the system. Informally, $t[i]$ is the number of p_i's writes of which the process is aware. Two vectors can be compared by comparing their components. Vector t_1 is *less than or equal* to t_2 ($t_1 \preceq t_2$) if each of t_1's components is less than or equal to t_2's corresponding component; t_1 is *less than* t_2 ($t_1 \prec t_2$) if it is less than or equal to t_2 and is not equal to t_2. Note that "$\preceq$" is transitive. Each process also maintains two queues. The first is a first-in-first-out queue called *OutQueue*. It contains information about local writes to memory that are yet to be communicated to other processes. The second is a priority queue called *InQueue*. Each queue item includes a vector clock value, which is its *timestamp*. The queue *InQueue* is ordered by timestamp, with items with smaller timestamps appearing closer to the head. The queue is maintained so that items being added to the queue are only placed ahead of existing items whose timestamps are greater than that of the new item. That is, the new item is placed after any existing item with an equal incomparable timestamp (actually, one can show that no two items can have equal timestamps, but we do not need this fact).

The implementation for process p_i is shown in Figure 3. It consists of an initialization routine and five basic actions. Each of these actions is local and executed atomically. A *read* action is executed whenever a read of a location x is invoked by p_i. The value stored in $M[x]$ is returned to p_i. A *write* action is executed whenever a write of some value v to some location x is invoked by p_i. Process p_i increments $t[i]$, writes v to $M[x]$, and adds the tuple $\langle i, x, v, t\rangle$ to *OutQueue*; this tuple is called a *write-tuple*. Note that the read and write actions require no blocking. This is in contrast to implementations of linearizable or sequentially consistent memory; in these cases, it can be shown that some blocking is required [10,29,31].

The information in *OutQueue* must be communicated to the other processes. This is done by *send* actions, which may be performed whenever is convenient to the process but which must be performed infinitely often (that is, a process can never elect to perform no more send actions). A send action removes some nonempty prefix from *OutQueue* and sends it to all other processes. When such a message is received, p_i executes a *receive* action; it adds all the write-tuples in the message received to *InQueue* (recall that this is a priority queue based on the tuples' timestamps). The information in *InQueue* is used to update a process's view of memory. This is done by an *apply* action, which need only be performed infinitely often. The write-tuple at the head of *InQueue* can be applied if its timestamp reflects no other write of which p_i is not aware. This can be determined by comparing p_i's vector clock to the timestamp of the write; a write by p_j can be applied only if all components of its timestamp (other than the jth) are less than or equal to those of p_i's vector clock and if the jth component is exactly one more than the jth component of p_i's vector clock. When a write can be applied, it

```
/* Initialization: */
    foreach $x \in \mathcal{M}$ do
        $M[x] := \bot$
    for $j := 1$ to $n$ do
        $t[j] := 0$
    $OutQueue := \langle\rangle$
    $InQueue := \langle\rangle$

/* Read action: to read from $x$ */
    return($M[x]$)                                   /* Add $r_i(x)*$ to $L_i$ and $S_i$ */

/* Write action: to write $v$ to $x$ */
    $t[i] := t[i] + 1$
    $M[x] := v$                                      /* Add $w_i(x)v$ to $L_i$ and $S_i$ */
    enqueue $\langle i, x, v, t\rangle$ to $OutQueue$

/* Send action: executed infinitely often */
    if $OutQueue \neq \langle\rangle$ then
        let $A$ be some nonempty prefix of $OutQueue$
        remove $A$ from $OutQueue$
        send $A$ to all others

/* Receive action: upon receipt of $A$ from $p_j$ */
    foreach $\langle j, x, v, s\rangle \in A$
        enqueue $\langle j, x, v, s\rangle$ to $InQueue$

/* Apply action: executed infinitely often */
    if $InQueue \neq \langle\rangle$ then
        let $\langle j, x, v, s\rangle$ be head of $InQueue$
        if $s[k] \leq t[k]$ for all $k \neq j$ and $s[j] = t[j] + 1$ then
            remove $\langle j, x, v, s\rangle$ from $InQueue$
            $t[j] := s[j]$
            $M[x] := v$                              /* Add $w_j(x)v$ to $S_i$ */
```

Figure 3: Implementation of Causal Memory for Process p_i

is removed from *InQueue*, the corresponding component of p_i's vector clock is updated, and the new value is written to M. This means that, after the write-tuple $\langle j, x, v, s\rangle$ is applied to p_i's memory, $s \preceq t$, where t is the value of p_i's vector clock.

To facilitate the proof of correctness of the implementation, we introduce the following notation: if o is an operation of a process p_i, the *timestamp* of o, denoted $ts(o)$, is the value of p_i's vector clock immediately after o completes. Note that, for a write operation o, $ts(o)$ is the same as the timestamp included with the corresponding write-tuple. $H = \langle L_1, L_2, \ldots, L_n\rangle$ is a *history of the implementation* if each L_i is the ordered sequence of read and write operations performed by process p_i (see comments in Figure 3). Theorem 3 below shows that H is causal. The causality order $\rightsquigarrow$ used is derived from the following writes-into order $\mapsto$. If $o_2 = r_i(x)v$ is a read by p_i of some non-initial value, then $o_1 \mapsto o_2$, where o_1 is the latest write to x applied by p_i before performing o_2 (it is clear from Figure 3 that o_1 is a write of v).

The following two lemmas are used in the proof of correctness. The first asserts that the causality order $\rightsquigarrow$ is reflected in vector timestamps:

Lemma 1: *Let H be a history of the implementation and let o_1 and o_2 be two operations such that $o_1 \rightsquigarrow o_2$. Then $ts(o_1) \preceq ts(o_2)$. Furthermore, if o_2 is a write operation by p_i, $ts(o_1)[i] < ts(o_2)[i]$, so $ts(o_1) \prec ts(o_2)$.*

Proof: The proof is by induction on the structure of the order $\rightsquigarrow$. Consider three cases:

- $o_1 \underset{i}{\rightarrow} o_2$ for some p_i. Since no process ever decrements any component of its vector clock, $ts(o_1)$ must be less than or equal to $ts(o_2)$. Furthermore, if o_2 is a write operation, then p_i increments its local component during o_2, so $ts(o_1)[i] < ts(o_2)[i]$.

- $o_1 \mapsto o_2$. This means that o_1 is a write operation, say $w_i(x)v$, and o_2 is a corresponding read, say $r_j(x)v$. Note that the write-tuple associated with o_1 includes the timestamp $ts(o_1)$. By Figure 3, it is clear that p_j cannot read v from x before it applies the write to its memory. Process p_j does not apply the write until its own timestamp is greater than or equal to $ts(o_1)$ (except for $ts(o_1)[i]$, which is assigned to the i^{th} component of p_j's clock when the write is applied). Since no component of p_j's timestamp is ever decremented, it is still greater than or equal to $ts(o_1)$ when it reads v, so $ts(o_1) \preceq ts(o_2)$.

- There is some operation o' such that $o_1 \rightsquigarrow o' \rightsquigarrow o_2$. By induction, this implies that $ts(o_1) \preceq ts(o') \preceq ts(o_2)$. By the transitivity of $\preceq$, the desired result holds. If o_2 is a write by p_i, then $ts(o')[i] < ts(o_2)[i]$ by induction. Since $ts(o_1) \preceq ts(o')$ implies $ts(o_1)[i] \leq ts(o')[i]$, we have that $ts(o_1)[i] < ts(o_2)[i]$.

□

The next lemma is used to show the liveness of the implementation:

Lemma 2: *Let H be a history of the implementation and suppose that w is a write operation of process p_i. Then each process p_j eventually applies w to its memory.*

Proof: If $i = j$, the write is applied immediately; for the remainder of the proof, assume that $i \neq j$. Let $s = ts(w)$. An inspection of Figure 3 shows that, once p_i has executed w, it is always the case that one of the following holds for w: its write-tuple is in p_i's *OutQueue*, its write-tuple is in transit from p_i to p_j, its write-tuple is in p_j's *InQueue*, or p_j has applied the write. Since p_i performs send operations infinitely often and *OutQueue* is first-in-first-out, any write-tuple in *OutQueue* is eventually sent to p_j. Since channels are reliable, any write-tuple that is sent is eventually received and added to p_j's *InQueue*. We now show that p_j eventually applies any write-tuple added to *InQueue*.

Consider the time at which p_j adds the write-tuple for w, $\langle i, x, v, s\rangle$, to its *InQueue*. There are only finitely many write-tuples ahead of it at this time. Write-tuples with timestamps smaller than $ts(w)$ that can arrive in the future will also be placed ahead of $\langle i, x, v, s\rangle$ in p_j's *InQueue*. It is easy to see that there can be only finitely many such write-tuples. For this reason, we can assume by induction that, at some point in time, p_j has applied all write-tuples that are ever placed before $\langle i, x, v, s\rangle$ in p_j's *InQueue* or whose timestamps are less than s. At this point, $\langle i, x, v, s\rangle$ is at the head of p_j's *InQueue* and remains there until it is applied; we say that it is *ready to be applied by* p_j. We now show that it is indeed applied when p_j next performs an apply action.

Let t be p_j's vector clock at such a point. We must show that $t[k] \geq s[k]$ for all $k \neq j$ and that $t[j] + 1 = s[j]$. Let p_k be any process other than p_j (k could equal i). Let w' be the $s[k]$th write by p_k. This means that p_i applies w' before it performs w, which implies that $ts(w') \prec ts(w)$. Thus, p_j must order w' ahead of w in *InQueue*, which implies that, once $\langle i, x, v, s\rangle$ is ready to be applied by p_j, p_j has already applied w'. Once p_j applies w', $t[k] \geq s[k]$, as desired. Let $\widehat{w}$ be the $(s[j] \perp 1)$st write by p_i. By Lemma 1, $ts(\widehat{w}) \preceq ts(w)$. Thus, p_j must order $\widehat{w}$ ahead of w in *InQueue* and thus has already applied $\widehat{w}$. Therefore, $t[j] \geq s[j] \perp 1$. This means that p_j applies w the next time it performs an apply operation. Since p_j does this infinitely often, we conclude that p_j eventually applies this write. □

We can now prove the correctness of the implementation:

Theorem 3: *Let H be a history of the implementation. Then H is causal.*

Proof: The proof must show that, for each process p_i, there is a serialization S_i of A^H_{i+w} that respects $\rightsquigarrow$. (Recall that A^H_{i+w} is the set of all of p_i's operations and all writes in H.)

The serialization S_i for p_i is obtained simply by concatenating all writes as they are applied to p_i's memory and all reads as they occur (see comments in Figure 3). By Lemma 2, S_i includes all write operations in H, and thus all of A^H_{i+w}. S_i is a serialization because all reads and writes apply directly to p_i's copy of memory and each read thus reads the value most recently written. It remains to be seen that S_i respects $\rightsquigarrow$.

We first observe that $\rightsquigarrow$ is indeed a partial order in H. To prove this, it suffices to observe that it is acyclic by showing that $o_1 \rightsquigarrow o_2$ implies $o_2 \not\rightsquigarrow o_1$. Suppose for a contradiction that $o_1 \rightsquigarrow o_2$ and $o_2 \rightsquigarrow o_1$. By Lemma 1, this means that $ts(o_1) \preceq ts(o_2)$ and $ts(o_2) \preceq ts(o_1)$, implying that $ts(o_1) = ts(o_2)$. Lemma 1 implies that neither o_1 nor o_2 is a write operation, as this would contradict this equality. Even if o_1 and o_2 occur at the same process, it cannot be the case that each of o_1 and o_2 precede the other with respect to program order. Without loss of generality, assume that o_1 does not precede o_2 in any L_i. Since $o_1 \rightsquigarrow o_2$ and both operations are reads, there must be some write operation w such that $o_1 \rightsquigarrow w \rightsquigarrow o_2$. By Lemma 1, $ts(o_1) \prec ts(w) \preceq ts(o_2)$, implying $ts(o_1) \neq ts(o_2)$, a contradiction. We conclude that the causality order is not cyclic.

Let o_1 and o_2 be two operations in A^H_{i+w} such that $o_1 \rightsquigarrow o_2$; we must show that o_1 precedes o_2 in S_i. By Lemma 1, $ts(o_1) \leq ts(o_2)$. One of the following five cases must hold:

- Both o_1 and o_2 are operations by p_i. Since $\rightsquigarrow$ is acyclic, this means that o_1 precedes o_2 in L_i. Since p_i's operations appear in both L_i and S_i in the order in which they are performed, o_1 precedes o_2 in S_i.

- o_1 is a write by another process p_j and o_2 is an operation by p_i. An inspection of Figure 3 shows that p_i does not set its vector clock t so that $t[j] = ts(o_1)[j]$ until it applies o_1 to its local memory. Since $ts(o_2)[j] \geq ts(o_1)[j]$, o_2 can occur only after this application. This means that o_1 precedes o_2 in S_i.

- o_1 is a write by p_i and o_2 is a write by another process p_j. Since $ts(o_1) \preceq ts(o_2)$, $ts(o_1)[i] \leq ts(o_2)[i]$. This means that p_j does not execute o_2 until it has applied o_1; since p_i cannot apply o_2 before p_j and must apply o_1 before p_j, it must be that p_i applies o_1 before it applies o_2. Thus, o_1 precedes o_2 in S_i.

- o_1 is a read by p_i and o_2 is a write by another process. It is not hard to see that $o_1 \rightsquigarrow o_2$ implies that there is a write w by p_i such that $o_1 \rightsquigarrow w \rightsquigarrow o_2$. By the first case above, o_1 precedes w in S_i. By the third case above, w precedes o_2 in S_i. Thus, o_1 precedes o_2 in S_i.

- o_1 and o_2 are both writes by processes other than p_i. Suppose o_1 and o_2 are executed by processes p_j and p_k. If $j = k$, Lemma 1 implies $ts(o_1)[j] < ts(o_2)[j]$, so p_i cannot apply o_2 until it has applied o_1. Now assume that $j \neq k$ and let t be p_i's vector clock at the point when o_2 is applied. By Figure 3, $ts(o_2)[j] \leq t[j]$. Since $ts(o_1) \preceq ts(o_2)$, $ts(o_1)[j] \leq ts(o_2)[j]$. This means that p_i has already applied o_1 at this point. Thus, o_1 precedes o_2 in S_i.

In all cases, o_1 precedes o_2 in S_i, so the proof is complete. □

The implementation given in Figure 3 shows that read and write operations for causal memory can be implemented without processes experiencing any blocking. Consider the following analyses of the performance of implementations of various forms of distributed shared memory. Assume that local computation time is negligible with respect to message delays and assume that d is the worst-case message delay. Given a memory implementation, let R be the worst-case execution time for a read and W

be the worst-case execution time for a write. Attiya and Welch [10] showed that, in systems in which process clocks were not perfectly synchronized and in which there was some uncertainty with respect to message delays (e.g., some messages may take d to be delivered and others may take less), it is impossible to achieve $W = 0$ or $R = 0$ in implementations of linearizable memory (see footnote 1). Lipton and Sandberg [29] showed that, for any implementation of sequentially consistent memory, $R + W \geq d$. In contrast, our implementation of causal memory gives $R = W = 0$.

The implementation presented here is correct as long as processes and communication channels are reliable. This is a normal assumption when implementing distributed shared memory [3,9,10,29,31,32]. However, we have also developed an implementation of causal memory that is correct even in systems in which processes may fail by stopping and in which communication channels can lose messages (as long as each channel delivers infinitely many messages if infinitely many are sent) [5]. This implementation is complex and inefficient and is not presented here.

In other work [6,23], we give a more practical implementation that sacrifices the non-blocking property of the implementation presented here. The implementation also makes use of vector timestamps but associates them with pages instead of individual locations. The memory of each node is treated like a cache for some subset of the shared pages, and a page-fault occurs when an accessed page is not in the cache. This results in communication with an *owner* node, which is unique for each page. Since the owner supplies the page on a fault, this implementation also requires that writes to a page be sent to the owner. However, it is not necessary that nodes other than the owner be notified on a write operation even when they store a copy of the page. Causal memory consistency is implemented by locally invalidating pages that could potentially be causally "overwritten". Vector timestamps are used for this purpose. This implementation does require nodes to communicate before certain read or write operations can be completed and hence some memory operations may be blocking. However, we have shown [23] that this implementation provides better performance than sequentially consistent memory for several scientific applications.

6 Programming with Causal Memory

The previous section showed that causal memory may be implemented without blocking; a process's write operations can complete before other processes learn about them. To strengthen the case that causal memory is a good model of a distributed shared memory, we must also argue that it can be programmed without undue difficulty. In this section, we characterize two classes of programs; any program in these classes, if written to run correctly on sequentially consistent memory, also runs correctly in a system with causal memory. Thus, programs in these classes can be written assuming a sequentially consistent memory even for a system that provides causal memory. We show that all executions of these programs on causal memory are also possible with a sequentially consistent memory.

The existence of these classes indicates that causal memory is a viable model for programming distributed applications: if a few rules are followed, a programmer may assume that the memory is sequentially consistent, while causal memory may be used instead. Because causal memory can be implemented more efficiently, this could result in improved performance.

Section 6.1 presents some definitions and notation necessary for discussing the behavior of programs with a distributed shared memory. Section 6.2 considers first the simple but restricted class of *concurrent-write free* programs. Section 6.3 considers the more practical class of *data-race free* programs. Section 6.4 discusses other work done in proving that programs in certain classes run correctly on memories weaker than sequential consistency.

6.1 Definitions and Notation

At any time during an execution, a process is in some *local state*; this is determined by its initial state and the operations performed thus far in its local history. A process p_i runs a *local program* Π_i, which is a function from local states to *actions*; each action is either of the form $w(x)v$, indicating that value v should be written to location x, or of the form $r(x)$, indicating that the value of x should be read and returned.[2] The execution of an action is an *operation* and changes the process's local state; note that the operation associated with a read action includes the value that was read. A tuple of local programs, one for each process, is called a *program* and is usually denoted Π. H is a history of Π if all operations in H are the execution of the actions that Π would specify given the local states through which processes pass.

Recall that a history H is a tuple of local histories, L_i for each process p_i. Let $\leadsto$ be a causality order of H. We say that history $H' = \langle L'_1, L'_2, \ldots, L'_n \rangle$ is a *prefix of H with respect to* $\leadsto$ if each L'_i is a prefix of L_i and, if o is an operation in H', then all operations in H that precede o with respect to $\leadsto$ are also in H'. H' is a *proper prefix of H with respect to* $\leadsto$ if it is a prefix of H with respect to $\leadsto$ and $H' \neq H$.

6.2 Concurrent-Write Free Programs

A major advantage of using causal memory is that normal memory accesses can be implemented without blocking; processes need not synchronize with each other in performing these accesses. As a result, programs running on causal memory must do their own synchronization. One way to achieve this is to ensure that no two writes can be concurrent.

Let H be a history with causality order $\leadsto$. H is *concurrent-write free with respect to* $\leadsto$ if there are no two write operations w_1 and w_2 in H that are concurrent with respect to $\leadsto$. Program Π is *concurrent-write free* if, for all histories H of Π and all causality orders $\leadsto$ of H, if H has a serialization that respects $\leadsto$ (note that this implies that H is sequentially consistent), then H is concurrent-write free with respect to $\leadsto$. Note that the concurrent-write freedom of a program is only a statement about its sequentially consistent histories. An example of a concurrent-write free program is

[2] These actions should not be confused with the implementation actions described in Figure 3.

x, y, and z are shared variables, initially 0;
a, b, c, and d are local variables

```
process $p_1$:
    $x := 1$
    $y := 1$

process $p_2$:
    repeat
        $a := y$
    until $a = 1$
    $z := 1$

process $p_3$:
    $b := y$;
    repeat
        $c := z$
    until $c = 1$
    $d := x$
```

Figure 4: A Concurrent-Write Free Program

given in Figure 4.[3] It is concurrent-write free because, in any execution of the program, the three writes to global variables must be related as follows by any causality order $\rightsquigarrow$: $w_1(x)1 \rightsquigarrow w_1(y)1 \rightsquigarrow w_2(z)1$. (The read of y by p_3 is not relevant to the concurrent-write freedom of the program. It serves to make the program not data-race free; see below.)

Let H be a causal history, let $\rightsquigarrow$ be a causality order that proves H is causal, and suppose that H is concurrent-write free with respect to $\rightsquigarrow$. For each process p_i, let S_i be the serialization of A^H_{i+w} that respects $\rightsquigarrow$ (see Section 4). From $\rightsquigarrow$, define a *strong causality order*, denoted $\Rightarrow$, as follows: $o_1 \Rightarrow o_2$ if and only if one of the following cases holds:

- $o_1 \rightsquigarrow o_2$;
- o_1 is a read by process p_i, o_2 is a write, and o_1 precedes o_2 in S_i; or
- there is some other operation o' such that $o_1 \Rightarrow o' \Rightarrow o_2$.

The idea behind $\Rightarrow$ is that it extends $\rightsquigarrow$ by ordering a read after any writes that causally precede it and before all other writes. It is not hard to see that, if H is concurrent-write free with respect to $\rightsquigarrow$, then the associated $\Rightarrow$ is acyclic; in particular, if $o_1 \Rightarrow o_2$, then $o_2 \not\Rightarrow o_1$. Furthermore, for any operation in such a history, there are only finitely many operations that precede it with respect to $\Rightarrow$.

The following theorem shows that concurrent-write free programs produce only sequentially consistent executions when run on causal memory:

Theorem 4: *If* Π *is concurrent-write free, then all histories of* Π *with causal memory are sequentially consistent.*

Proof: The proof is by induction on the structure of causal histories of Π. Specifically, let H be a finite causal history of Π and let $\rightsquigarrow$ be a causality order that proves that H is causal. (The proof for infinite H follows.) We will prove the following for H given

[3] In this figure and in Figure 5, an assignment to a shared variable indicates a write action. An assignment with a shared variable on the right side indicates a read action.

that it holds for all proper prefixes of H with respect to $\rightsquigarrow$: H is concurrent-write free with respect to $\rightsquigarrow$ and has a serialization that respects $\rightsquigarrow$.

To show that H is concurrent-write free with respect to $\rightsquigarrow$, assume for a contradiction that w_1 and w_2 are two concurrent writes in H. Clearly, w_1 and w_2 are executed by different processes, so assume that w_1 is performed by p_j and w_2 by p_k, where $j \neq k$. Define $H' = \langle L'_1, L'_2, \ldots, L'_n \rangle$ by letting L'_i be the subsequence of L_i containing all operations that precede either w_1 or w_2 with respect to $\rightsquigarrow$. If either w_1 or w_2 appears in H', one precedes the other with respect to $\rightsquigarrow$, giving a contradiction. Assume instead that neither operation appears in H'; this means L'_j includes p_j's operations up to but not including w_1 and that the same holds for L'_k, p_k, and w_2. Clearly, H' is a proper prefix of H with respect to $\rightsquigarrow$; by inductive hypothesis, H' is concurrent-write free with respect to $\rightsquigarrow$ and has a serialization that respects $\rightsquigarrow$. Now define $\widehat{H} = \langle \widehat{L}_1, \widehat{L}_2, \ldots, \widehat{L}_n \rangle$ by

$$\begin{array}{rcll} \widehat{L}_j & = & L'_j; w_1; & \\ \widehat{L}_k & = & L'_k; w_2; & \\ \widehat{L}_i & = & L'_i & \text{if } i \notin \{j,k\}. \end{array}$$

$\widehat{H}$ is also a (not necessarily proper) prefix of H with respect to $\rightsquigarrow$ and is thus an execution of Π. Let S' be a serialization of H' that respects $\rightsquigarrow$. This implies that $S'; w_1; w_2$ is a serialization of $\widehat{H}$ that also respects $\rightsquigarrow$. Since $\widehat{H}$ is a history of Π and Π is concurrent-write free, $\widehat{H}$ is concurrent-write free with respect to $\rightsquigarrow$. This means that w_1 and w_2 cannot be concurrent with respect to $\rightsquigarrow$, giving the desired contradiction.

We now show that H has a serialization that respects $\rightsquigarrow$. As noted above, the order $\Rightarrow$ is acyclic. Since H is finite, we can choose an operation o in H such that for no o' in H does $o \Rightarrow o'$ hold. Let $\overline{H}$ be identical to H but excluding o. $\overline{H}$ is a proper prefix of H with respect to $\rightsquigarrow$ and, by the inductive hypothesis, has a serialization $\overline{S}$ that respects $\rightsquigarrow$. Clearly, $\overline{S}; o$ respects $\rightsquigarrow$; if it did not, either $\overline{S}$ would not respect $\rightsquigarrow$ or there would be an operation o' in $\overline{H}$ such that $o \rightsquigarrow o'$, which contradicts the definition of o. We will now prove that $\overline{S}; o$ is a serialization of H, proving that H is sequentially consistent.

Assume for a contradiction that $\overline{S}; o$ is not a serialization of H. This means that o is some read operation $r_i(x)v$. Recall that H is causal; let S_i be the serialization of A^H_{i+w} that respects $\rightsquigarrow$. There are two possibilities:

- There is some write to x in H. All such writes precede o in S_i: any write that does not do so will follow o with respect to $\Rightarrow$, contradicting the definition of o. Let w be the latest write to x in S_i. Since S_i is a linearization, w writes the value v. Since $\overline{S}; o$ is not a serialization, there must be some write w' to x of another value after w in $\overline{S}$. Since H is concurrent-write free with respect to $\rightsquigarrow$, there are two possibilities:
  - $w' \rightsquigarrow w$. Since $\overline{S}$ respects $\rightsquigarrow$, this means that w' precedes w in $\overline{S}$, contradicting the definition of w'.
  - $w \rightsquigarrow w'$. This means that w must precede w' in S_i, contradicting the definition of w.

- There is no write to x in H. This implies that there can be no write to x in S_i either. Since $\overline{S};o$ is not a serialization, it must be that $v \neq \bot$. This means that S_i cannot be a serialization either, which is a contradiction.

Since all cases lead to contradictions, we conclude that $\overline{S};o$ is a serialization of H that respects $\leadsto$. This implies that H is sequentially consistent.

This theorem also holds if H is an infinite causal history of Π. Let $\leadsto$ be a causality order that proves that H is causal. We first prove that H is concurrent-write free with respect to $\leadsto$. If not, let w_1 and w_2 be two writes in H that are concurrent with respect to $\leadsto$. Let H' be the shortest prefix of H that includes w_1 and w_2. Note that H' is causal and that $\leadsto$ is a causality order that proves it. It is easy to see that H' is finite; by the above, H' is concurrent-write free with respect to $\leadsto$. This implies that w_1 and w_2 are related by $\leadsto$, giving the desired contradiction. Let $\Rightarrow$ be the strong causality order for H derived from $\leadsto$. We know that $\Rightarrow$ is acyclic and that any operation in H has a finite number of predecessors with respect to $\Rightarrow$. Define an infinite sequence $(H_0, H_1, \ldots)$ of finite prefixes of H with respect to $\leadsto$, each having all previous ones as proper prefixes with respect to $\leadsto$, as follows. H_0 is the empty history. H_{i+1} includes H_i plus one operation o such that all operations in H that precede o with respect to $\Rightarrow$ appear in H_i (the operations o can be chosen "fairly" so that every operation in H appears in some H_i). Given this construction, there can be no operation in H_i that follows o with respect to $\Rightarrow$. An inspection of the proof above shows that the serializations S_i of the prefixes H_i respect $\leadsto$ and are such that, for all i, S_i is prefix of S_{i+1}. This means that $\lim_{i\to\infty} S_i$ is well-defined and is thus a serialization of H. This shows that H is sequentially consistent. □

6.3 Data-Race Free Programs

While concurrent-write free programs run correctly with causal memory, they form a very restricted class and allow very little concurrency. In this section, we define the more practical class of data-race free programs and show that they also run correctly with causal memory. Alternative definitions have been given elsewhere [1,2,16].

Let H be a history with causality order $\leadsto$. Two operations o_1 and o_2 in H *compete with respect to* $\leadsto$ if both access the same location, at least one is a write, and they are concurrent with respect to $\leadsto$. H is *data-race free with respect to* $\leadsto$ if it contains no pair of operations that compete with respect to $\leadsto$. A history data-race free with respect to $\leadsto$ has the property that all writes to a given location are linearly ordered with respect to $\leadsto$. Program Π is *data-race free* if, for all histories H of Π and all causality orders $\leadsto$ of H, if H has a serialization that respects $\leadsto$ (note that this implies that H is sequentially consistent), then H is data-race free with respect to $\leadsto$. Note that the data-race freedom of a program is only a statement about its sequentially consistent histories.

Previous definitions of data-race free programs were quite different from ours. These definitions were for systems with normal data operations (reads and writes) and special *synchronization operations*. Any competing operations in a sequentially consistent execution of a data-race free program must be separated (by a kind of causality) by synchronization operations. It is not hard to see that our definition is a generalization of this to systems in which there need not be synchronization operations with specified semantics. Sections 6.3.1 and 6.3.2 below give two ways in which data-race free programs (using our definition) may be derived. The class of data-race free programs should not be confused with the memory models DRF0 [1] and DRF1 [2].

The following theorem shows that data-race free programs produce only sequentially consistent executions when run on causal memory:

Theorem 5: *If* Π *is data-race free, then all histories of* Π *with causal memory are sequentially consistent.*

Proof: The proof is by induction on the structure of causal histories of Π. Specifically, let H be a finite causal history of Π and let $\rightsquigarrow$ be a causality order that proves that H is causal. (The proof for infinite histories follows.) We will prove the following for H given that it holds for all proper prefixes of H with respect to $\rightsquigarrow$: H is data-race free with respect to $\rightsquigarrow$ and has a serialization that respects $\rightsquigarrow$.

To show that H is data-race free with respect to $\rightsquigarrow$, assume for a contradiction that o_1 and o_2 are two operations in H that compete with respect to $\rightsquigarrow$. We can assume by induction that there is no operation o in H such that either (1) $o \rightsquigarrow o_1$ holds and o and o_2 compete with respect to $\rightsquigarrow$ or (2) $o \rightsquigarrow o_2$ holds and o and o_1 compete with respect to $\rightsquigarrow$. If o_1 and o_2 are both reads, both are performed by the same process, or they are to different locations, then they do not compete. Assume, therefore, that o_1 and o_2 are concurrent with respect to $\rightsquigarrow$, o_1 is a write to x performed by p_j and o_2 is an operation on x performed by p_k, where $j \neq k$. Define $H' = \langle L'_1, L'_2, \ldots, L'_n \rangle$ by letting L'_i be the subsequence of L_i containing all operations that precede either o_1 or o_2 with respect to $\rightsquigarrow$. If either o_1 or o_2 appears in H', they are related by $\rightsquigarrow$, and we are done. Assume instead that that neither operation appears in H'; this means that L'_j includes p_j's operations up to but not including o_1 and that the same holds for L'_k, p_k, and o_2. Clearly, H' is a proper prefix of H with respect to $\rightsquigarrow$; by inductive hypothesis, H' is data-race free with respect to $\rightsquigarrow$ and has a serialization that respects $\rightsquigarrow$. Now define $\widehat{H} = \langle \widehat{L}_1, \widehat{L}_2, \ldots, \widehat{L}_n \rangle$ by

$$\begin{array}{rcll} \widehat{L}_j & = & L'_j; o_1; & \\ \widehat{L}_k & = & L'_k; o_2; & \\ \widehat{L}_i & = & L'_i & \text{if } i \notin \{j, k\}. \end{array}$$

$\widehat{H}$ is also a (not necessarily proper) prefix of H with respect to $\rightsquigarrow$ and is thus an execution of Π. Let S' be a serialization of H' that respects $\rightsquigarrow$. We will now prove that $\widehat{H}$ has a serialization that respects $\rightsquigarrow$. By the data-race freedom of Π, this will imply that o_1 and o_2 do not compete with respect to $\rightsquigarrow$, giving the desired contradiction.

If o_1 and o_2 are both write operations, then $S'; o_1; o_2$ is a serialization of $\widehat{H}$ that respects $\rightsquigarrow$. Assume instead that o_2 is a read (o_1 was already assumed to be a write). If o_2 returns the value that o_1 writes, then $S'; o_1; o_2$ is a serialization of $\widehat{H}$ that respects $\rightsquigarrow$. Suppose instead that o_2 returns a different value. There are two possible cases:

- S' contains a write to x and o_2 returns the value written by the last such write. In this case, $S'; o_2; o_1$ is a serialization of $\widehat{H}$ that respects $\rightsquigarrow$.

- S' contains a write to x and o_2 does not return the value written by the last such write w. Because H' is data-race free with respect to $\rightsquigarrow$, all of its writes to x are totally ordered by $\rightsquigarrow$. Since S' respects $\rightsquigarrow$, all other writes to x precede w with respect to $\rightsquigarrow$. Recall that H is causal and that o_2 is performed by p_k; let S_k be the serialization of A^H_{k+w} that respects $\rightsquigarrow$. By the above, all other writes to x must precede w in S_k. Since o_2 does not return the value written by w, it must also precede w in S_k. Since S_k respects $\rightsquigarrow$, $w \not\rightsquigarrow o_2$. Since H' contains only operations that causally precede o_1 or o_2 and w appears in H', it must be that $w \rightsquigarrow o_1$. Consider now two sub-cases:

  – $o_2 \not\rightsquigarrow w$. This implies that w and o_2 are concurrent with respect to $\rightsquigarrow$ and thus compete with respect to $\rightsquigarrow$. This means that w contradicts the assumption that there is no operation causally preceding o_1 that competes with o_2 with respect to $\rightsquigarrow$.

  – $o_2 \rightsquigarrow w$. This implies $o_2 \rightsquigarrow o_1$, contradicting the fact that o_1 and o_2 are concurrent with respect to $\rightsquigarrow$.

  Thus, this case leads to a contradiction.

- S' contains no writes to x. Since no writes to x causally precede o_2, that operation must return the initial value $\perp$. In this case, $S'; o_2; o_1$ is a serialization of $\widehat{H}$ that respects $\rightsquigarrow$.

We have shown that all non-contradictory cases lead to serializations of $\widehat{H}$ that respects $\rightsquigarrow$. Since $\widehat{H}$ is a history of Π and Π is data-race free, $\widehat{H}$ is data-race free with respect to $\rightsquigarrow$. This means that o_1 and o_2 cannot compete with respect to $\rightsquigarrow$, giving the desired contradiction.

We now show that H has a serialization that respects $\rightsquigarrow$. Since H is finite and $\rightsquigarrow$ is acyclic, we can choose an operation o in H such that for no o' in H does $o \rightsquigarrow o'$ hold. Let $\overline{H}$ be the same as H but excluding o. $\overline{H}$ is a proper prefix of H with respect to $\rightsquigarrow$ and, by the inductive hypothesis, has a serialization $\overline{S}$ that respects $\rightsquigarrow$. Clearly, $\overline{S}; o$ respects $\rightsquigarrow$ if it did not, either $\overline{S}$ would not respect $\rightsquigarrow$ or there would be an operation o' in $\overline{H}$ such that $o \rightsquigarrow o'$, which contradicts the definition of o. We will now prove that $\overline{S}; o$ is a serialization of H, proving that H is sequentially consistent.

Assume for a contradiction that $\overline{S}; o$ is not a serialization of H. This means that o is some read operation $r_i(x)v$. Recall that H is causal; let S_i be the serialization of A^H_{i+w} that respects $\rightsquigarrow$. There are two possibilities:

- There is some write to x in H. All such writes precede o with respect to $\rightsquigarrow$: any write that does not so either competes with o with respect to $\rightsquigarrow$ (contradicting the data-race freedom of H with respect to $\rightsquigarrow$) or follows o with respect to $\rightsquigarrow$ (contradicting the definition of o). Thus, all writes to x precede o in S_i. Let w be the latest such write. Since S_i is a linearization, w writes the value v. Since $\overline{S}; o$ is not a serialization, there must be some write w' to x of another value after w in $\overline{S}$. Since H is data-race free with respect to $\rightsquigarrow$, w and w' are related by $\rightsquigarrow$ and there are two possibilities:

  - $w' \rightsquigarrow w$. Since $\overline{S}$ respects $\rightsquigarrow$, this means that w' precedes w in $\overline{S}$, contradicting the definition of w'.
  - $w \rightsquigarrow w'$. This means that w must precede w' in S_i, contradicting the definition of w.

- There is no write to x in H. This implies that there can be no write to x in S_i either. Since $\overline{S}; o$ is not a serialization, it must be that $v \neq \perp$. This means that S_i cannot be a serialization either, which is a contradiction.

Since all cases lead to contradictions, we conclude that $\overline{S}; o$ is a serialization of H that respects $\rightsquigarrow$. This implies that H is sequentially consistent.

This theorem also holds when H is an infinite causal history of Π. Let $\rightsquigarrow$ be a causality order that proves that H is causal. We first prove that H is data-race free with respect to $\rightsquigarrow$. If not, let o_1 and o_2 be two operations in H that compete with respect to $\rightsquigarrow$. Let H' be the shortest prefix of H that includes o_1 and o_2. Note that H' is causal and that $\rightsquigarrow$ is a causality order that proves it. It is easy to see that H' is finite; by the above, H' is data-race free with respect to $\rightsquigarrow$. This implies that o_1 and o_2 do not compete with respect to $\rightsquigarrow$, giving the desired contradiction. We define an infinite sequence $(H_0, H_1, \ldots)$ of finite prefixes of H with respect to $\rightsquigarrow$, each having all previous ones as proper prefixes with respect to $\rightsquigarrow$, as follows. H_0 is the empty history. H_{i+1} includes H_i plus one operation o such that all operations in H that precede o with respect to $\Rightarrow$ appear in H_i (the operations o can be chosen "fairly" so that every operation in H appears in some H_i). Given this construction, there can be no operation in H_i that follows o with respect to $\rightsquigarrow$. An inspection of the proof above shows that the serializations S_i of the prefixes H_i respect $\rightsquigarrow$ and are such that, for all i, S_i is prefix of S_{i+1}. This means that $\lim_{i\to\infty} S_i$ is well-defined and is thus a serialization of H. This shows that H is sequentially consistent. $\Box$

(Theorem 5 also follows from an independently derived result of Singh's [35].)

The classes of data-race free and concurrent-write free programs are incomparable. For example, consider the concurrent-write free program given in Figure 4. It is *not* data-race free. In an execution in which p_3 reads y before p_1 writes it, these two operations are concurrent and thus compete. On the other hand, the data-race free program given in Figure 5 is not concurrent-write free. In any given iteration, the variables $x[i]$ may all be written concurrently.

Despite this incomparability, the class of data-race free programs contains many more programs that are of practical use. The following subsections demonstrate two ways of obtaining data-race free programs. Both of these sections require some kind of blocking, the first through the use of programmer-specified busy-waiting and the second through the augmentation of causal memory with semaphores. This use of blocking does not eliminate the advantages gained by our non-blocking implementation of causal memory. Blocking is required for any kind of synchronization, and data-race free programs require a programmer to do explicit synchronization. The advantage of causal memory is that it requires such blocking *only* when explicit synchronization is required. It does not require blocking for ordinary memory operations.

complete$[1..n]$ and *changed*$[1..n]$ are shared variables, initially 0;
done is a shared variable, initially *false*;
$x[1..n]$ are shared variables, initially 0;
$A[1..n, 1..n]$ and $b[1..n]$ are shared constants;
$t[1..n]$ are local variables, $t[i]$ local to p_i;
converged is an external routine that evaluates convergence

```
process $p_0$:
    while not $done$
        for $i := 1$ to $n$
            await($complete[i] = 1$)
        for $i := 1$ to $n$
            $complete[i] := 0$
        for $i := 1$ to $n$
            await($changed[i] = 1$)
        $done := converged(A, x, b)$
        for $i := 1$ to $n$
            $changed[i] := 0$
```

```
process $p_i$:
    while not $done$
        $t[i] := \left(b[i] \perp \sum_{j=1}^{i-1} A[i,j]\, x[j] \perp \sum_{j=i+1}^{n} A[i,j]\, x[j]\right) \Big/ A[i,i]$
        $complete[i] := 1$
        await($complete[i] = 0$)
        $x[i] := t[i]$
        $changed[i] := 1$
        await($changed[i]) = 0$
```

Figure 5: Synchronous Iterative Linear Solver on Causal Memory

6.3.1 Programs with await Statements

One common way to synchronize processes' actions is by blocking a process until some desired condition becomes true. To capture this in our program model, we allow a program to specify an action of the form **await**$(x = v)$; in process histories, we will denote this by $a(x)v$. This blocks the process until the desired condition is true, that is, until the shared variable x takes on the value v. It can be implemented by simple read actions as follows:

```
repeat
    $a := x$
until $a = v$
```

However, we consider an **await** as a single read that appears in a local history only once each time it is invoked (any preceding reads of other values do not). Thus, a writes-into order $\mapsto$ relates to $a(x)v$ $w(x)v$ and not any writes of other values read before the **await** completes. It is not hard to see that Theorem 5 continues to apply when **await** statements are added to the model. (Singh [35] also augments the usual memory operations with **await** operations.)

Many programs use **await** statements to synchronize the access to shared variables. For example, they can be used to effect barrier synchronization to control access to certain data. An example is given in Figure 5. The example is a synchronous iterative linear equation solver that solves $Ax = b$, where A is a known

$n \times n$ matrix, b is a known vector, and x is the vector that is to contain the solution. The solver operates in a series of *phases*: in each phase, process p_i computes a new value for the solution component $x[i]$. If we use $x^k[i]$ to represent the value of the ith component of x in phase k, then new values are computed as follows: $x^{k+1}[i] = \left(b[i] \perp \sum_{j=1}^{i-1} A[i,j]\, x^k[j] \perp \sum_{j=i+1}^{n} A[i,j]\, x^k[j]\right) \Big/ A[i,i]$. Thus, the computing of $x^{k+1}[i]$ requires access to all $x^k[j]$ (for $j \neq i$) from the previous iteration. The process p_i $(1 \leq i \leq n)$ computes $x[i]$. The process p_0 tests for convergence and synchronizes each worker twice per iteration using a barrier technique: before reading the various $x[j]$ from phase k and before writing $x[i]$ for phase $k+1$. (By making the array t shared and having workers read alternately from $x[i]$ and $t[i]$, we could eliminate the first synchronization.)

The program in Figure 5 is easily shown to be correct with sequential consistency. It is also not hard to see that it is data-race free. Access to $x[i]$ is controlled by $complete[i]$ and $changed[i]$. Suppose for example that p_j $(j \neq i)$ reads the kth iteration value v from $x[i]$ and let v' be the $(k+1)$st iteration value of $x[i]$. It is not hard to see that the following causal chain must exists (for any writes-into order $\mapsto$) in the $k+1$st iteration:

$$\begin{gathered} r_j(x[i])v \underset{j}{\rightarrow} w_j(complete[j])1 \mapsto a_0(complete[j])1 \underset{0}{\rightarrow} \\ w_0(complete[i])0 \mapsto a_i(complete[i])0 \underset{i}{\rightarrow} w_i(x[i])v'. \end{gathered}$$

Thus, these two accesses to $x[i]$ do not compete to any $\leadsto$; similar arguments show that there are no competing accesses in any execution of the program; thus, the program is data-race free. Theorem 5 now implies it runs correctly on causal memory. In fact, it runs faster with causal memory than with sequential consistency [22].

While the program presented in Figure 5 requires a centralized coordinator, there also exists a fully distributed solution [15]. This solution is also data-race free and thus runs correctly with causal memory.

6.3.2 Programs with Semaphores

While **await** statements allow for barrier synchronization such as that used in Figure 5 above, they do not suffice for implementing other kinds of synchronization, such as critical sections. Recall that **await** statements can be implemented with a "spinning read." However, it has been shown that the mutual exclusion necessary for implementing critical sections cannot be realized with causal memory without cooperation [9]; for example, Peterson's algorithm [34] for mutual exclusion will not run correctly with causal memory.

Mutual exclusion can be implemented with special synchronization primitives such as *semaphores.* A semaphore is a variable holding a non-negative integer that supports two operations: V, which atomically increments the value, and P, which atomically decrements it. If the semaphore's value is zero, then a P operation is blocked until the semaphore becomes positive.

It is possible to add semaphores to our definition of causal memory; call the result *extended causal memory.* Note that every operation on a semaphore reads and then writes the semaphore (e.g., a V operation first reads the semaphore and then writes an incremented value). Because of this, all operations on a semaphore are causally related, meaning that there can be no competing accesses to a semaphore. This implies that, in an execution with extended causal memory, all operations on a semaphore appear in the same order to all processes. An implementation of extended causal memory would require blocking and is beyond the scope of this paper. It is not hard to see that Theorem 5 applies to extended causal memory.

Semaphores can be useful in synchronization. For example, the program in Figure 5 can be modified to use semaphores. Let the arrays *complete* and *changed* be of semaphores, and let each write to an array element be a V operation and each **await** statement be a P operation. The program remains correct and data-race free.

Semaphores can also be used to implement critical sections. With each critical section is associated a semaphore with initial value 1. A process invokes P on the semaphore before entering a critical section and invokes V on the same semaphore upon leaving.

6.4 Other Work

Other researchers have considered different programming models and the correctness of programs in those models on memories weaker than sequential consistency.

Gibbons, Merritt, and Gharachorloo considered the DASH system's *RCsc* version of *release consistency* [17]. This is a "mixed" memory model in that it allows programmers to specify (or "label") whether operations are "weak" or "strong". In this case, the strong operations are sequentially consistent, whereas weak operations are ordered based on when they are invoked relative to the strong operations. A program is *properly labeled* if there are no data races among the weak operations. Gibbons et al. showed that, when run on RCsc, properly labeled programs admit only sequentially consistent executions. Attiya et al. showed a similar result for a different mixed memory model, called *hybrid consistency* [8]. They also proved that only sequentially consistent executions are obtained if either all writes or all reads are labeled as strong. Our results contrast with both of these in that we do not require a memory model that allows strong (sequentially consistent) operations (except in Section 6.3.2).

Singh [35] independently considered programming models for purely weak consistency models such as causal memory. His work classifies programs based on the types of executions they permit with a weaker form of memory. Our work differs from his in that we classify programs based on how they execute with sequentially consistency and then prove properties about their execution on causal memory. We believe that this is a potentially more productive approach, as it is easier for programmers to reason about the behavior of programs with sequential consistency.

Heddaya and Sinha [19] considered a variety of weaker forms of memory, including slow memory [21]. They showed that all programs in the class of *totally asynchronous iterative algorithms* [12] run correctly on slow memory (and, therefore, on causal memory). We note here that the class of *synchronous iterative algorithms* is a broader class and not all of these programs run correctly with slow memory. However, these programs are data-race free (Figure 5 gives an example) and run correctly with causal memory.

7 Discussion

We have presented a new model of distributed shared memory called *causal memory*. We defined it formally using a simple framework that allows it to be compared easily with other memory models. We exhibited a message-based implementation of causal memory. Finally, we formally characterized two classes of programs that run correctly with causal memory, assuming that they do so under sequential consistency.

Our formal analysis shows causal memory to lie between sequential consistency (a strong memory) and PRAM (a weak one). This suggests that it may be powerful enough to program easily (like strong memories) but at the same time allow inexpensive implementations (like weak memories). These are borne out by the results in Sections 5 and 6.

Our implementation of causal memory is *non-blocking*; a process can always complete a read or a write operation immediately, without having to communicate with other processes. All communication can take place in the background between memory accesses. It is important to note that this implementation, like the definition of causal memory, lies between sequential consistency and PRAM; it allows histories that are not sequentially consistent but no PRAM histories that are not causal.

Section 6 shows that all concurrent-write free and data-race free programs will run correctly on causal memory. These are programs in which data accesses are controlled using explicit synchronization. Such synchronization is often necessary even for distributed programs designed to run with sequentially consistent memory. For example, the synchronization in the program in Figure 5 is necessary even with stronger memories, yet the program runs correctly with causal memory. By requiring only that the programmer explicitly specify the synchronization needed, we allow him or her to use a form of memory that can be implemented much more efficiently.

All these facts show that causal memory has the potential to be an important model for distributed shared memory systems. To realize this potential, we are currently exploring implementations of causal memory in actual distributed systems [6]. These are more practical than the theoretically motivated implementation given in Section 5, basing communication on entire pages rather than single variables (see the discussion at the end of that section). In the future, we plan to benchmark these implementations to better compare causal memory with other intermediate memory models.

Acknowledgements

We were greatly helped in the development of Section 6 by discussions with Ambuj K. Singh. Dr. Singh and Subodh Kumar both provided comments on earlier versions of this paper.

References

[1] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *Proceedings of the Seventeenth 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. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[3] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.

[4] Mustaque Ahamad, Rida A. Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *Proceedings of the Fifth Symposium on Parallel Algorithms and Architectures*, pages 251–260. ACM Press, June 1993. A full version appears a Technical Report 92/34, College of Computing, Georgia Institute of Technology.

[5] Mustaque Ahamad, James E. Burns, Phillip W. Hutto, and Gil Neiger. Causal memory. In S. Toueg, P. G. Spirakis, and L. Kirousis, editors, *Proceedings of the Fifth International Workshop on Distributed Algorithms*, volume 579 of *Lecture Notes on Computer Science*, pages 9–30. Springer-Verlag, October 1991. A revised and expanded version exists [7].

[6] Mustaque Ahamad, Phillip W. Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. In *Proceedings of the Eleventh International Conference on Distributed Computing*, pages 274–281, May 1991.

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

[8] Hagit Attiya, Soma Chaudhuri, Roy Friedman, and Jennifer L. Welch. Shared memory consistency conditions for non-sequential execution: Definitions and programming strategies. In *Proceedings of the Fifth Symposium on Parallel Algorithms and Architectures*, pages 241–250. ACM Press, June 1993.

[9] Hagit Attiya and Roy Friedman. A correctness condition for high performance multiprocessors. In *Proceedings of the Twenty-Fourth ACM Symposium on Theory of Computing*, pages 679–690. ACM Press, May 1992.

[10] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

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

[12] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[13] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[14] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, August 1991.

[15] Roy Friedman, 1991. Personal communication.

[16] 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 Seventeenth International Symposium on Computer Architecture*, pages 15–26, May 1990.

[17] Phillip B. Gibbons, Michael Merritt, and Kourosh Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Proceedings of the Third Symposium on Parallel Algorithms and Architectures*, pages 292–303. ACM Press, July 1991.

[18] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, March 1989.

[19] Abdelsalam Heddaya and Himanshu Sinha. Computing with non-coherent shared memory. Technical Report 93-007, Computer Science Department, Boston University, June 1993.

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

[21] Phillip W. Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the Tenth International Conference on Distributed Computing Systems*, May 1990. A complete version appears as Technical Report 89/39, School of Information and Computer Science, Georgia Institute of Technology.

[22] Ranjit John. *Implementing and Programming Weakly Consistent Memories.* Ph.D. dissertation, Georgia Institute of Technology, 1994.

[23] Ranjit John and Mustaque Ahamad. Implementation and evaluation of causal memory for data race free programs. Technical Report 94/30, College of Computing, Georgia Institute of Technology, July 1994.

[24] R. E. Kessler and M. Livny. An analysis of distributed shared memory algorithms. In *Proceedings of the Ninth International Conference on Distributed Computing*, pages 498–505, June 1989.

[25] Prince Kohli, Gil Neiger, and Mustaque Ahamad. A characterization of scalable shared memories. In *Proceedings of the Twenty-Second International Conference on Parallel Processing*, pages I-332–I-335, August 1993.

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

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

[28] Leslie Lamport. On interprocess communication; part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

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

[30] Friedemann Mattern. Virtual time and global states of distributed systems. In Michel Cosnard, Patrice Quinton, Yves Robert, and Michel Raynal, editors, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. North-Holland, October 1988.

[31] Marios Mavronicolas and Dan Roth. Sequential consistency and linearizability: Read/write objects. In *Proceedings of the Twenty-Ninth Annual Allerton Conference on Communication, Control, and Computing*, pages 683–692, October 1991. A revised version appears as Technical Report 28-91, Aiken Computation Laboratory, Harvard University, June 1992 under the title "Linearizable Read/Write Objects".

[32] Marios Mavronicolas and Dan Roth. Efficient, strongly consistent implementations of shared memory. In A. Segall and S. Zaks, editors, *Proceedings of the Sixth International Workshop on Distributed Algorithms*, number 647 in Lecture Notes on Computer Science, pages 346–361. Springer-Verlag, November 1992.

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

[34] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[35] Ambuj K. Singh. A framework for programming using non-atomic variables. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 133–140. IEEE Computer Society Press, April 1994.