receiving (reading) processes know everything about the dictionary known by the writing process at the write operation. The only remaining case to consider is when a delete by some other processor (say P_j) is concurrent with some insert by P_i. This can only happen when P_i inserts a new item y in a location previously holding another value (say x). P_i has seen the previous value x and the delete of x and has inserted a new value y. A concurrent delete occurs when P_j reads x but not the delete of x and tries, in turn, to delete x. In fact, it is possible for the delete of x by P_j to be concurrent with a long string of inserts and deletes acting on the same location. However, when P_j's delete is finally processed by P_i, it will be concurrent with the current value of the location. Since that value was written by P_i and since our resolution strategy favors writes by the owner, the delete will be rejected and the dictionary remains correct.

5 Concluding Remarks

We have argued that causal memory is better suited to DSM environments than traditional memory consistency because of the potential global synchronization required by traditional consistency when write operations are executed. We have shown that a causal DSM can be implemented efficiently and can be used to program a variety of applications. Our implementation allows several processors to cache a given location (page) and read and write operations never require communication with more than a single processor (the owner). Several applications written for atomic memory run without modification on causal memory, showing the viability of programming causal memory. We are investigating variants of causal memory in [3] and plan to implement a variant of the algorithm presented in this paper under the Clouds distributed operating system.

References

[1] S. Adve and M. Hill. Weak ordering – A new definition. In *ISCA*, pages 2–14, 1990.

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

[3] M. Ahamad, J. Burns, P. Hutto, and G. Neiger. Causal memory: A causal connecting principle. College of Computing, Georgia Tech, 1991. *Submitted to ACM PODC*.

[4] M. Ahamad, P. Hutto, and R. John. Implementing and programming causal distributed shared memory. GIT-CC-90/49, College of Computing, Georgia Tech, Oct. 1990.

[5] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM TOCS*, 5(1), Feb. 1987.

[6] G. Brown. Asynchronous multicaches. *Distributed Computing*, 4(1):31–362, 1990.

[7] M. Dubois. Delayed consistency protocols. CENG 90-21, EE-Systems Dept., U. of Southern California, July 1990.

[8] M. Fischer and A. Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *ACM PODS*, pages 70–75, 1982.

[9] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA*, pages 15–26, 1990.

[10] P. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *ICDCS*, pages 302–311, 1990.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, July 1978.

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

[13] V. Lanin and D. Shasha. Wait-free concurrent set manipulation. In *ACM PODS*, 1988.

[14] J. Lee and U. Ramachandran. Synchronization with multiprocessor caches. In *ISCA*, pages 27–39, 1990.

[15] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM TOCS*, 7(4):321–359, Nov. 1989.

[16] F. Mattern. Time and global states of distributed systems. In *Proc. 1988 Int. Workshop on Parallel and Dist. Algorithms*. North Holland, 1989.

[17] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM TOPLAS*, 8(1):142–153, January 1986.

[18] U. Ramachandran, M. Ahamad, and M. Khalidi. Coherence of distributed shared memory: Unifying synchronization and data transfer. In *ICPP*, pages 160–169, August 1989.

[19] C. Scheurich. *Access Ordering and Coherence in Shared Memory Multiprocessors*. PhD thesis, Univ. of Southern California, May 1989.

[20] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *ISCA*, pages 234–243, June 1987.

read $r_i^k(changed_i)F$. When P_i introduces this value, written by the coordinator, into its cache, all cached x_j ($j \neq i$) are invalidated. Therefore, P_i must issue read requests to all other processors when reading x_j ($j \neq i$). This results in $2(n-1)$ messages ($n-1$ requests and $n-1$ replies). After this, P_i and the coordinator communicate through the handshake bits owned by P_i, $complete_i$ and $changed_i$. Since each bit is written twice, a total of eight messages will be necessary. Thus, a total of $2n+6$ messages per processor per iteration are needed on causal memory. Now consider the same execution on atomic memory. The same number of messages are generated on atomic memory for reading the x_i and synchronizing with the coordinator ($2(n-1)+8$). However, when an owner writes x_i at the end of a phase, atomic memory requires that all cached copies be invalidated. Since every process other than P_i has a copy of x_i, this results in $n-1$ messages per processor, a cost not incurred by causal memory. Thus each phase of the synchronous linear solver requires at least $3n+5$ messages per processor when executed on atomic memory compared to $2n+6$ when executed on causal memory. This represents a substantial savings and causal memory will lead to performance gains because the communication overhead is reduced.

4.2 Distributed Dictionary

Consider an association table with distinct keys maintained cooperatively by a collection of processes. Simple *insert* and *delete* operations are provided along with a *lookup* operation that reports if a value has been inserted but not yet deleted. The *dictionary problem* [8] is to implement such a table without forcing processes to synchronize their operations. The correctness condition only requires that a process sees an item in the dictionary iff the item has been previously inserted and not deleted according to the operations that have been executed according to its view. A liveness condition requires that all views must eventually converge and attain consistency in the absence of further inserts and deletes. The dictionary problem was originally posed for an unreliable, asynchronous message environment. In such an environment, the view of a process P_i includes operations executed by it as well as the operations in the view of another process P_j at the time P_j sent message m, where m is the most recently sent message of P_j that has been received by P_i. In a shared memory environment, any operation in P_j's view when P_j writes a value to x is also considered to be in P_i's view after P_i reads the value of x written by P_j.

An atomic shared memory solution that maintains a single common copy of the dictionary is not interesting because the completion of a write operation of a process will force its result to be in the view of all processes even when they do not communicate with the writer process. Although this is not incorrect, the weaker correctness requirement is not exploited to provide efficient implementation of the dictionary (atomic writes may require global synchronization). In [13], Lanin and Shasha present two algorithms for a wait-free generalization of the dictionary problem in a shared-memory environment which indicate that correct solutions to the problem are complex even with stronger memory consistency in a shared memory environment.

We accept two easily maintained restrictions which were also assumed in Fischer [8]: (**R1**) Each item inserted is unique; and (**R2**) Delete requests follow their corresponding insert requests. The dictionary is maintained in a two-dimensional array *dict* with n rows (one per process) and m columns. We use the distinguished value λ to indicate that a location is "free" and that the previously held value has been deleted. Process P_i responds to $insert_i(x)$ by finding a free location in row i of *dict* and writing the new item there. By partitioning the dictionary in this way we avoid the need for synchronization when processes insert items. (We assume that m is sufficiently large to hold all items inserted by a process.) P_i responds to $lookup_i(x)$ by systematically scanning all rows of *dict* and returning true if x is found. This ensures the knowledge monotonicity property described below. Finally, a process responds to $delete_i(x)$ by scanning the entire array for x and writing λ into the location containing x if found. Notice that a process P_i may need to delete items inserted by some other process P_j. Thus a write by P_i inserting a new item into a location may be concurrent with other processes writing λ (deleting) to that location but conflicting writes may never result from concurrent inserts since only process P_i may write into row i of *dict*.

Our solution requires no synchronization around deletes even though processes (other than P_i) may write λ to a location concurrent with P_i's attempt to write a new value to the same location. Our solution guarantees correctness by requiring simply that P_i owns all locations in row i of the dictionary and that writes by the owner are always favored when resolving concurrent writes.

The correctness of the algorithm follows easily. Causal memory guarantees the view property. Clearly, each process observes its operations immediately. When P_i reads a value written by P_j (a lookup operation returns a value inserted by P_i or P_i reads a λ value written by P_j), it will also read values installed by insert or delete operations that are in the view of P_j. Thus, the solution to the dictionary problem possesses the *knowledge monotonicity* property: after each communication,

COORDINATOR	WORKER$_i$
while ($\neg done$)	**while** ($\neg done$)
wait ($\forall i\ complete_i$)	$t_i := \left(b_i - \sum_{j=1}^{i-1} \cdots\right.$
$\forall i\ complete_i := F$	$complete_i := T$
wait ($\forall i\ changed_i$)	**wait** ($\neg complete_i$)
if ($converged()$)	$x_i := t_i$
$done := T$	$changed_i := T$
$\forall i\ changed_i := F$	**wait** ($\neg changed_i$)

Figure 6: Synchronous Iterative Linear Solver

waits until all results from the previous phase have been copied to the global vector x. Then, before copying the newly computed value t_i to global x_i, each process waits until all other processes have completed computation of their new t_j (allowing the old x_i to be overwritten). Special synchronization variables such as semaphores or event counts may be used on causal memory but we prefer a simpler approach, using causal boolean shared variables (flags) and a central process to *coordinate* the remaining *worker* processes.

Figure 6 shows the solver. All booleans are initially $False$ and "**wait** (B)" means "**while** ($\neg B$) *skip*". We assume n worker processes, one for each vector element. The code is easily modified so that each process computes a set of elements. As described previously, each worker process begins by assuming access to the previous (or initial) values in the global vector x and proceeds to compute and store the new value in the local variable t_i. The worker notifies the coordinator by setting $complete_i$ true (T) and then waits for the coordinator to indicate that all processes have completed by resetting $complete_i$ false (F). Once so notified, the worker copies the new value t_i to the global x_i and goes through a similar handshake before resuming the next phase of computation or terminating.

We claim the code in Figure 6 correctly solves the system $Ax = b$ on both atomic and causal memory. Notice that the x_i, and the handshake bits ($complete_i$ and $changed_i$) are the only shared global variables, and hence, the only locations that may behave non-atomically. We consider a read of some x_j ($i \neq j$) by P_i in phase k and show that the only value that may be correctly returned is the phase $k-1$ write to x_j by P_j. In other words, values from all earlier iterations are overwritten and causal memory behaves like atomic memory in this instance. Following our notation for x_i^k, we use superscripts on read and write operations to denote the phase in which the operation was performed. For example, $w_i^k(x_i)v$ denotes a write of x_i by P_i in phase k. (A process subscript 'c' denotes a read or write by the coordinator.) Consider the causal relations established between P_j's write of x_j in phase $k-1$ and P_i's read of x_j in phase k by the interactions between two worker processes (i and j) and the coordinator.

$$
\begin{array}{ll}
(1) & w_j^{k-1}(x_j)v \rightarrow w_j^{k-1}(changed_j)T \\
(2) & w_j^{k-1}(changed_j)T \rightarrow r_c^{k-1}(changed_j)T \\
(3) & r_c^{k-1}(changed_j)T \stackrel{*}{\rightarrow} w_c^{k-1}(changed_i)F \\
(4) & w_c^{k-1}(changed_i)F \rightarrow r_i^{k-1}(changed_i)F \\
(5) & r_i^{k-1}(changed_i)F \rightarrow r_i^k(x_j)v
\end{array}
$$

Above, (1) holds because the two writes are consecutive operations of P_j. (2) holds because the coordinator eventually notices that P_j has copied t_j to x_j. (3) is the key to the argument and holds because the coordinator must read *changed* true from *all* workers (j in particular) before setting *changed* false for all workers (i in particular) and initiating the next phase. (4) holds because P_i must eventually see $changed_i$ false and begin the next phase. At the beginning of phase k, P_i reads all components of x (except x_i) and, in particular, x_j, showing (5) to hold. Taken together (1)–(5) imply that $w_j^{k-1}(x_j)v \stackrel{*}{\rightarrow} r_i^k(x_j)v$. Since we assumed an arbitrary i and j this argument shows that all reads of x in the computation return the value computed in the previous iteration, the same value returned when the computation is executed on atomic memory. A similar argument holds for the handshake bits. The synchronous iterative algorithm is correct but the required synchronization is costly. It is possible to eliminate the synchronization entirely by using an *asynchronous* algorithm [4].

Programming on causal memory appears to be remarkably similar to programming on atomic memory, identical in fact, for the synchronous solver. To demonstrate execution on causal memory delivers superior performance we compare a causal and atomic DSM, both running the linear solver. We assume a comparable owner protocol for atomic memory where locations (pages) are stored at the owner and cached at other nodes. An atomic write requires that all cached copies in the system be invalidated. (In Li [15], a representative atomic DSM, a read set is maintained by the owner and invalidation messages are sent to all nodes in the read set.) In comparison, a causal write requires at most a message exchange with the owner.

A simple message counting argument shows the advantage of causal memory when running the synchronous linear solver. Assume that P_i owns x_i and the handshake bits $complete_i$ and $changed_i$.[2] (For simplicity assume each node in the system runs a single process. Node i runs P_i.) First consider the causal memory implementation. P_i begins phase k with the

[2] A simple enhancement to the basic algorithm can be used to avoid invalidations of A and b [4].

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

Figure 5: A Weakly Consistent Execution

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

Although our implementation may generate more invalidations than necessary, it still admits weakly consistent executions, not allowed by strongly consistent memories. The weakly consistent execution in Figure 5 is allowed both by causal memory correctness and by our implementation if P_1 is the owner of x and P_2 is the owner of y.

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

3.2 Correctness

Informally, our implementation maintains correctness by invalidating any locally cached values that could cause a violation of correctness if read. Since a cached value can be read anytime, our algorithm invalidates values that can potentially violate correctness each time a new value is introduced into the cache. Owners also perform invalidations when servicing write requests since this amounts to a potential causal interaction between the writing process and the owner.

Two observations lead to the correctness of our implementation. First, violations of causal memory correctness are always related to violations of causality. A read of x by P_i returning v can only violate correctness if P_i "knows" of some other value v' whose read or write causally follows the write of v. Second, a read of x by P_i resulting in a request to the owner (a *remote* read) can never violate correctness since the owner is guaranteed to return a value that causally follows any value of x that P_i could previously have seen. Thus, a simple strategy to maintain correctness is to force a request to the owner on every read. This strategy results in a memory that satisfies atomic correctness, not just causal correctness, but we lose all the benefits of caching. A better strategy, the one used by our implementation, is to allow a process P_i to read cached values that are concurrent with or that causally follow the value most recently introduced into the cache. Reads of any other value (not locally owned nor cached) must generate a read request to the owner. Note that subsequent remote reads might introduce values that causally precede all other cached values so this strategy allows the cache to contain values with a wide range of writestamps. See [4] for a detailed proof of the algorithm.

The basic implementation algorithm can be improved in several ways. These include scaling the unit of sharing to a page, eliminating unnecessary invalidations, and reducing the blocking of processors [4].

4 Programming

Causal memory can be easily and efficiently programmed. We show a synchronous iterative linear equation solver that works correctly on atomic memory and show that it works on causal memory as well. A simple message counting argument suggests that causal memory provides improved performance for this application. Next we consider the dictionary problem. This application has a solution on causal memory very similar to the atomic memory solution. However, a particularly elegant solution exists on causal memory when the programmer is allowed to specify a procedure for resolving concurrent writes.

4.1 Linear Solver

Very large systems of linear equations often arise in many scientific and engineering applications. Li [15] investigated such an application and found that good speedups can be obtained even on atomic DSMs. We believe that even better performance can be obtained on causal memory.

Consider a parallel iterative algorithm that solves $Ax = b$. We use x_i^{k+1} to represent the value of the ith component of x in phase $k+1$. x_i^{k+1} can be computed using the following equation: $x_i^{k+1} = \left(b_i - \sum_{j=1}^{i-1} a_{i,j}\, x_j^k - \sum_{j=i+1}^{n} a_{i,j}\, x_j^k\right) \Big/ a_{i,i}$. Note that A and b are inputs and remain constant but that computing x_i^{k+1} requires access to all x_j^k $(i \neq j)$ from the previous iteration.

A parallel implementation of the iterative method partitions the tasks of computing each new x_i among available processes. At the beginning of each iteration, a process reads the results of the previous iteration from the shared global vector x and computes and stores the new x_i in a private local variable t_i. Since processes may proceed at different rates, a *synchronous* implementation requires processes to synchronize twice per iteration. Before beginning phase $k+1$, each process

then $M_i[x]$ contains the current value of x. The remaining locations in a processor's local memory are used to *cache* copies of locations owned by other processors. We use the distinguished value $\perp$ to indicate that a node does *not* possess a cached copy of a location. If $M_i[x] = \perp$ then x is *invalid* (not cached) at P_i. Also, we say that assigning $\perp$ to a location ($M_i[x] := \perp$) *invalidates* that location at P_i. The locations owned by a processor can never be invalidated by that processor. C_i is the set of locations currently cached by processor P_i, that is, locations that are not owned by P_i and that are not invalid in M_i. Our implementation maintains correctness by invalidating cached copies that might violate causal memory correctness if read.

Attempts to read a location not locally owned nor cached (a *read miss*) generate a message to the owner requesting a current copy. The requesting processor blocks until a reply is received, caches the copy received, and then completes the read. Similarly, writes to a location not owned by the writer generate a message to the owner, requesting that the write be *certified*. The writing processor blocks until a reply is received and the write is certified. Essentially, all such writes are completed cooperatively between the writing processor and the owner.

Recall that causal memory correctness is intimately related to causality. A simple *vector timestamp protocol* [16] may be used to capture precisely the evolving partial ordering of events in a distributed system, and thereby, causality. Each processor in the system maintains a separate vector timestamp VT_i that may be *incremented*, *updated*, and *compared*. Processor P_i increments VT_i by adding one to the ith component, $VT_i[i]$. $update(VT, VT')$ returns the component-wise max of the two vectors. Finally, we may compare vector timestamps. $VT < VT'$ if $\forall i : VT[i] \leq VT'[i]$ and $\exists j : VT[j] < VT'[j]$. On every write attempt, the writing processor increments its vector timestamp and associates the resulting vector time, called a *writestamp*, with the value written. Thus each location x in a processor's local memory M_i contains a *value-writestamp* pair $M_i[x] = (v, VT)$. All writes by a processor are totally ordered by these writestamps and all writestamps ever generated in the system are unique. Two writes not ordered by their associated timestamps are *concurrent*. When a processor introduces a value into its cache, it updates its vector timestamp with the writestamp associated with the value being introduced.

Identifying precisely the values that may violate correctness after a read or write of causal memory requires more overhead than we are willing to pay in our simple owner protocol. (See Hutto et al. [3].) Instead, each time a "new" value is introduced into local memory by a read or write, we invalidate all cached values that could potentially participate in a violation of causality; that is all cached values that are "older" (via the causality relation) than the value being introduced. This approach may invalidate more cached values than strictly necessary but it requires little bookkeeping overhead and ensures correctness. The algorithm is shown in Figure 4. Five procedures are presented. The first two are executed whenever P_i performs a read or write. The next two are executed by P_i on receipt of READ and WRITE requests for locations owned by P_i. The final operation, *discard*, may be performed by P_i under a variety of circumstances described below. Each operation must be executed atomically and owners must fairly alternate between issuing reads and writes and responding to READ and WRITE messages from other processors. Notice carefully the handling of writestamps of locations not locally owned. The writer increments

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

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

$[READ, x]$ ::
  receive $[READ, x]$ **from** j
  send $[R_REPLY, x, M_i[x].value, M_i[x].VT]$ **to** j

$[WRITE, x, v, VT]$ ::
  receive $[WRITE, x, v, VT]$ **from** j
  $VT_i := update(VT_i, VT)$
  $M_i[x] := (v, VT_i)$
  $\forall y \in C_i : M_i[y].VT < VT_i$
    $M_i[y] := \perp$
  send $[W_REPLY, x, v, VT_i]$ **to** j

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

Figure 4: A Simple Owner Protocol

P_1 : $w(x)2$ $w(y)2$ $w(y)3$ $r(z)5$ $w(x)4$
P_2 : $w(x)1$ $r(y)3$ $w(x)7$ $w(z)5$ $r(x)4$ $r(x)9$
P_3 : $r(z)5$ $w(x)9$

Figure 2: A Correct Execution on Causal Memory

P_1 : $w(x)5$ $w(y)3$
P_2 : $w(x)2$ $r(y)3$ $r(x)5$ $w(z)4$
P_3 : $r(z)4$ $r(x)2$

Figure 3: Causal Broadcasting is Not Causal Memory

$w(x)v$, the value v is live *for o ($v \in \alpha(o)$) if either:*

1. *o' is concurrent[1] with o. That is, $o' \stackrel{*}{\not\rightarrow} o \stackrel{*}{\not\rightarrow} o'$.*
2. *Or, o' precedes o with no intervening read or write of x with value v'. That is, $o' \stackrel{*}{\rightarrow} o$ but there is no $o'' = a(x)v'$ where $o' \stackrel{*}{\rightarrow} o'' \stackrel{*}{\rightarrow} o$ and $a = r$ or $a = w$.*

Definition 2 (Causal Memory) *An execution on causal memory is* correct *if the value returned by each read operation in the execution is* live *for that read. That is, for each $o = r(x)v$, $v \in \alpha(o)$.*

The three process sequences in Figure 2 depict a correct execution on causal memory since each read returns a live value (a value in $\alpha(o)$ as defined above). In this and all following examples we assume initial writes to all locations of the value 0. Consider the read of z by P_1. To determine if this is a correct read we must determine the causal relationships between $r_1(z)5$ and all writes of z in the execution.

By inspecting Figure 2 we can see that $r_1(z)5$ is concurrent with $w_2(z)5$. Since $w_2(z)5$ is the only write to z in this execution, other than the initial write of 0, we conclude that $\alpha(r_1(z)5) = \{0, 5\}$, and that the read $r_1(z)5$ is correct. $r_3(z)5$ is shown correct by the same argument. By a similar argument we conclude that $\alpha(r_2(y)3) = \{0, 2, 3\}$ and that the read $r_2(y)3$ is likewise correct. Finally, we consider the two reads of x by P_2. Both $w_1(x)2$ and $w_2(x)1$ (and the initial write of 0) are overwritten by P_2's write of 7 to x and so do not appear in $\alpha(r_2(x)4)$. However, both $w_2(x)4$ and $w_3(x)9$ remain concurrent to $r_2(x)4$. Thus, $\alpha(r_2(x)4) = \{4, 7, 9\}$ and $r_2(x)4$ is correct. P_2's read $r_2(x)4$ serves notice that P_2's earlier write to x has now been overwritten. Thus P_2's second read of x may correctly return only 4 or 9. Since it returns 9, the entire execution is shown correct on causal memory.

So far we have required only that correct reads of causal memory select *some* value in the set $\alpha(o)$. It is possible to further refine the definition of causal memory and specify a *policy* for selecting among alternatives in α. Such a policy can be used formally to guarantee various liveness and fairness properties of causal memory. Also, in Section 4, we will see that allowing the programmer to select among such policies can significantly simplify programming of some applications.

As we have said, causal memory is closely related to the ISIS causal broadcast and, thereby, to the notion of causally ordered messages. But causal memory is more than a collection of "locations" updated by causal broadcasts. There are significant differences in the two models.

One way to relate the two models is to assume that each processor has a copy of the memory (a cache) and writes are sent as broadcast messages to all processors. When a message arrives at a processor, it updates its memory by storing the value contained in the message into the appropriate location. A read by the processor returns the value in its memory. It may seem that when the message delivery order preserves causality (for example by using the causal broadcast protocol of ISIS) the values returned by read operations will satisfy the requirements of causal memory. This, however, is not true. The execution in Figure 3 is not allowed by causal memory but is possible when writes are treated as causal broadcasts. Since $w(x)5$ and $w(x)2$ are concurrent, their messages may arrive in different orders at P_2 and P_3. If $w(x)5$ arrives at P_2 after $w(x)2$ the values in Figure 3 will be returned, but 2 is not in $\alpha(r(x)2)$.

3 Implementation

In this section we show how to implement a causal DSM using only local memory accesses and reliable, ordered message passing between any two processors. We discuss several enhancements to the basic algorithm in the full paper [4].

3.1 The Basic Algorithm

The shared causal memory is partitioned among the processors in the system. The locations assigned to a processor are *owned* by that processor. Each processor P_i has a local memory M_i indexed by location names (addresses) in the causal memory namespace $\mathcal{N}$. The locations owned by a processor are always stored in the local memory of that processor so that, if $i = owner(x)$

[1] This condition illuminates our exclusion of the reads-from relation established by o itself when defining $\alpha(o)$. If this reads-from relation is also considered then $o = r(x)v$ never reads from a write concurrent with o since a read causally follows its corresponding write *by definition*.

similar to our causal memory was introduced independently by Scheurich in his 1989 thesis [19]. He identifies *causal correctness* as a property enjoyed by various restrictions of sequentially consistent memory, although he does not consider a causally correct memory interesting in its own right.

Our goals in this paper are to show that:

- Causal memory can be efficiently implemented in a DSM environment. We demonstrate an implementation algorithm that requires significantly less synchronization than comparable implementations of strongly consistent memory.
- Causal memory provides a reasonable and viable target architecture for programming distributed applications. We demonstrate solutions on causal memory to the dictionary problem, and program an iterative linear equation solver.
- Causal DSM potentially provides improved overall system performance. Since similar code may be used to program applications on both atomic and causal memories, and since implementations of causal memory are more efficient than implementations of strongly consistent memory, we argue that programming with causal memory leads to improved performance.

Section 2 defines causal memory and its implementation is given in Section 3. Section 4 discusses programming with causal memory. In Section 5 we offer concluding remarks.

2 Causal Memory

Consider a system of n processes that interact by sharing causal memory. A process is defined by the sequence of *operations* it performs. An operation $o = a(x)v$ acts on *location* x with *value* v. A *write* operation $w(x)v$ associates the value v with location x. A *read* operation $r(x)v$ reports on this association. When necessary we include a subscript on operations to identify the process performing the operation. Thus, $w_3(x)1$ is a write of x by process P_3. To simplify notation, we assume all writes are unique (easily implemented by associating a timestamp with writes). Thus, each read can be identified with the unique write that it "reads from." Note that several reads (possibly by distinct processes) may read from the same write.

We relate reads and writes using the *potential causality* relation. Two simple rules capture the causality relation (denoted $\rightarrow$). First, if o and o' are two successive operations by the same process, then $o \rightarrow o'$ and we say that o *causally precedes* o' or that o *happens before* o'. Second, if the read operation o_r reads from the write operation o_w then $o_w \rightarrow o_r$. (Recall that we require all writes to be unique.) Thus causality is a combination of the *program order* relation and the *reads-from* relation on operations. Finally, we assume that all locations are initialized by writes of a distinguished value that precede all operations in any process sequence.

$$\begin{aligned} P_1 &: \quad w(x)1\ w(y)2\ r(y)2\ r(x)1 \\ P_2 &: \quad w(z)1\ r(y)2\ r(x)1 \end{aligned}$$

Figure 1: Example of Causal Relations

The transitive closure of $\rightarrow$ (denoted $\stackrel{*}{\rightarrow}$) is also a useful notion. If o *transitively precedes* o' ($o \stackrel{*}{\rightarrow} o'$) then there is some sequence of operations $o_1, \ldots, o_k$ where $o_i \rightarrow o_{i+1}$ for all integers $0 < i < k$ and where $o = o_1$ and $o' = o_k$. Two operations not related by $\stackrel{*}{\rightarrow}$ are said to be *concurrent*. In the example execution in Figure 1 the writes of x and z are concurrent and $w(x)1 \stackrel{*}{\rightarrow} r_1(y)2$. Notice that a read may *establish* causality by relating a read and write that are otherwise concurrent or a read may simply *confirm* causality by reading from a write already related by program order. Thus, $r_2(y)2$ establishes causality by reading from $w(y)2$ while $r_1(x)1$ confirms the ordering $w(x)1 \stackrel{*}{\rightarrow} r_1(x)1$ already established by program order.

Informally, causal memory obeys the register property so that reads return the value of the "most recent write" where "most recent" is determined by causality. However, since $\stackrel{*}{\rightarrow}$ may only partially order writes on causal memory (writes on atomic or sequentially consistent memory are *totally ordered*) there may be no single most recent write. Thus, causal reads routinely select their return value from among a *set* of correct values. Call the values in this set *live* for read operation o. Then $\alpha(o)$ denotes the values live for read operation o. Values that causally precede the values (writes) in this set are said to have been *overwritten*.

When defining $\alpha(o)$ for read operation $o = r(x)v$ by process P_i, we will consider all the causal relationships in the execution *except* the reads-from ordering established by o itself. The write $o' = w(x)v$ may causally precede, follow, or be concurrent with o. Writes that causally follow o are *never* live for o while writes concurrent with o are *always* live for o. Writes that precede o *may* be live for o if they have not been overwritten. If two related writes transitively precede o ($w(x)v \stackrel{*}{\rightarrow} w(x)v' \stackrel{*}{\rightarrow} o$) then clearly the earlier write of v is overwritten by the write of v'. However, an intervening read operation $r(x)v'$ serves notice that v has been overwritten and is sufficient to eliminate v from $\alpha(o)$ as well.

Definition 1 (Live Values) *Given $o = r(x)$ and $o' =$*

Implementing and Programming Causal Distributed Shared Memory*

Mustaque Ahamad *Phillip W. Hutto* *Ranjit John*

College of Computing, Ga Tech
Atlanta, Georgia 30332-0280 USA
Email: {mustaq, pwh, rjohn}@cc.gatech.edu

Abstract

Causal memory is a weakly consistent memory in which reads are required to return the value of the most recent write based on the causal ordering of read and write operations. We present a simple owner protocol for implementing a causal distributed shared memory (DSM) and argue that our implementation is more efficient than comparable coherent DSM implementations. Moreover, we show that writing programs for causal memory is no more difficult than writing programs for atomic shared memory. We believe that causal memory is an attractive target architecture for DSM systems.

1 Introduction

Distributed shared memory is an attractive abstraction because it allows processes uniform access to local and remote information. This uniformity of access simplifies programming, eliminating the need for separate mechanisms to access local state and remote state. However, consistent distributed shared memory (DSM) can be difficult to implement efficiently. Most DSM implementations to date use variants of multiprocessor cache consistency algorithms that perform poorly in high latency distributed systems. Weakly consistent memories allow implementations better suited to the high latencies encountered in distributed systems.

Traditionally, a shared memory is correct if reads return the value of the "most recent write" to the location being read. Atomic memory satisfies this "register property" by regarding reads and writes as *operation intervals* on a global time line and requiring that operations "take effect" at some point within the operation interval [17]. Under this model, each operation corresponds to a distinct point (operations may not "take effect" simultaneously) on the global time line and, for any read operation, the most recent write is well-defined. While the order of overlapping writes may not be determined until a subsequent read operation "chooses" which write is the most recent, the resulting execution must still obey the register property. Sequential consistency [12] is a weakening of atomic memory that relaxes the requirement that operations take effect during their operation intervals. Several researchers [20, 2, 6] have sought to exploit the considerable flexibility provided by sequential consistency over atomic memory yet the requirement that sequentially consistent executions appear "as if" they obey the register property is severely restrictive.

Existing implementations of consistent (atomic) DSM [15, 18] require frequent, expensive global synchronization leading to inefficiency and problems of scale. Researchers in the architecture community have also begun to question the wisdom of always maintaining strong consistency [1, 14, 9, 7]. Recent work [10] has suggested that the *principled weakening* of consistency may solve problems of latency and scale and still provide a reasonable programming model.

We explore a type of weakly consistent memory introduced in [10] that we call *causal memory*. (A formal study of causal memory is presented in [3] where the memory discussed in this paper is called *strict* causal memory.) Informally, causal memory requires that reads return values consistent with all causally related reads and writes of that same location. We say that "reads respect the order of causally related writes." Causal memory does not require all writes of a single location to be totally ordered; several processors may write a location concurrently and independently, without synchronizing. Subsequent readers may disagree on the relative ordering of these concurrent writes. Causal memory is based on Lamport's concept of *potential causality* [11]. We introduce a similar notion of causality based on reads and writes in a shared memory environment. Causal memory is also closely related to the ISIS *causal broadcast* introduced in [5]. A notion

*Funded by NSF grants CCR-8619886 and CCR-8806358.