[10] Phillip W. Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the 10th International Conference on Distributed Computing Systems, Paris, France*, pages 302–311, 1990.

[11] Phillip W. Hutto, Mustaque Ahamad, Jim Burns, and Gil Neiger. Causal memory: A causal connecting principle. Technical report, Georgia Institute of Technology, College of Computing, 1990. *To appear.*

[12] Sacha Krakowiak. *Principles of Operating Systems.* MIT Press, 1988.

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

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

[15] Vladimir Lanin and Dennis Shasha. Wait-free concurrent set manipulation. In *Proceedings of the 1988 Symposium on Principles of Database Systems*, 1988.

[16] J. Lee and U. Ramachandran. Synchronization with multiprocessor caches. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 27–39, 1990.

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

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

[19] F. Mattern. Time and global states of distributed systems. In *Proceedings of the 1988 International Workshop on Parallel and Distributed Algorithms, Bonas, France.* North Holland, 1989.

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

[21] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, 1986.

[22] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Unifying synchronization and data transfer in maintaining coherence of distributed shared memory. Technical Report GIT-ICS-88/23, School of Information and Computer Science, Georgia Institute of Technology, June 1988.

[23] Christoph Scheurich and Michel Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 234–243. IEEE TC on Computer Architecture and ACM SIGARCH, Computer Society Press of the IEEE, June 1987.

[24] Christoph E. Scheurich. *Access Ordering and Coherence in Shared Memory Multiprocessors.* PhD thesis, University of Southern California, May 1989. (Available as TR No. CENG 89-19.).

[25] J.T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages*, 2(4):484–521, 1980.

[26] Larry Wittie and Creve Maples. MERLIN: Massively parallel heterogeneous computing. In *Proceedings of the International Conference on Parallel Processing*, 1989.

communication, 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.

An important goal of the algorithm in [8] is to improve availability by tolerating failures. In our algorithm, when the cache at each processor is large enough to hold the contents of the dictionary, failures will not affect lookup or insert operations because the causal memory implementation does not need to communicate with other processors when memory owned by the processor is written or cached values are read. However, a processor will not be able to delete a key inserted by another processor when the other processor has failed. It is possible to extend our solution to satisfy this property but we do not consider that here. In [15], Lanin and Shasha do not address failures when the problem is addressed in the context of atomic memory.

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 [11] and plan to implement a variant of the algorithm presented in this paper under the Clouds distributed operating system. We hope to produce empirical evidence for the advantages of causal DSMs in this way and to further explore the notion of weakly consistent DSMs. Our future work also includes other types of memories and their relationships.

References

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

[2] Yehuda Afek, Geoffrey Brown, and Michael Merritt. A lazy cache algorithm. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 209–223, June 1989.

[3] Mustaque Ahamad, Phillip W. Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. Technical Report GIT-CC-90/49, College of Computing, Georgia Institute of Technology, October 1990.

[4] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1), February 1987.

[5] Geoffrey M. Brown. Asynchronous multicaches. *Distributed Computing*, 4(1):31–362, 1990.

[6] Michel Dubois. Delayed consistency protocols. Technical Report CENG 90-21, Electrical Engineering-Systems Department, University of Southern California, July 1990.

[7] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, 1986.

[8] M. J. Fischer and A. Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 70–75, 1982.

[9] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

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 returns all values 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, a process P_i's view can be defined similarly by considering its operations. 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 [15], Lanin and Shasha present two algorithms for a wait-free generalization of the dictionary problem in a shared-memory environment. The algorithms presented indicate that correct solutions to the problem are complex even with stronger memory consistency in a shared memory environment.

```
$lookup_i()$ ::
for $j := 1$ to $n$
  for $k := 1$ to $m$
    $t := dict_{j,k}$
    if $(t \neq \lambda)$
      return $t$
```

```
$insert_i(x)$ ::
for $k := 1$ to $m$
  if $(dict_{i,k} = \lambda)$
    $dict_{i,k} := x$
    exit
```

```
$delete_i(x)$ ::
for $j := 1$ to $n$
  for $k := 1$ to $m$
    if $(dict_{j,k} = x)$
      $dict_{j,k} := \lambda$
      exit
```

Figure 9: The Dictionary Problem on Causal Memory

We accept two easily maintained restrictions which were also assumed in [8]: (**R1**) Each item inserted is unique; and (**R2**) Delete requests follow their corresponding insert requests. Our dictionary algorithm on causal memory is shown in Figure 9. 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 $lookup_i()$ by systematically scanning all rows of *dict* and returning any item found. Notice that this is not a "snapshot" of the array and that P_i need not immediately see values previously inserted by other processes. 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. (P_i only overwrites a location if it has been deleted. Thus this situation arises only when the deleting processes have not yet seen a delete that P_i has observed.) 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 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.

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

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 $w_j^{k-1}(x_j)v \xrightarrow{*} r_i^k(x_j)v$. 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 and reasonably efficient but the required synchronization is costly. It is possible to eliminate the synchronization entirely by using an *asynchronous* algorithm although the resulting computation may require more iterations to converge.

Programming on causal memory appears to be remarkably similar to programming on atomic memory, identical in fact, for the synchronous and asynchronous linear solvers. 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 [17], 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 $changed_i$.[4] (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 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. (This synchronization overhead is avoided in the asynchronous algorithm.) 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. Thus, since the same code can run correctly on either type of memory, using causal memory will lead to performance gains because the communication overhead is reduced.

---

[4] The first enhancement to the basic algorithm described in Section 3.3 can be used to avoid invalidations of A and b.

Iterative methods generally work in a sequence of phases. Each phase uses the results of the previous phase to generate a new, more accurate solution. Computation terminates when the result of the last phase passes a convergence test or has been computed to sufficient accuracy. In fact, many non-iterative linear solvers execute a final series of iterative improvements to eliminate accumulated numerical imprecision [21].

We consider a parallel version of the sequential iterative algorithm that solves the equation $Ax = b$ where A is an $n \times n$ matrix and b is a vector of size n. $a_{i,j}$ refers to the element of A in row i and column j and b_i (similarly x_i) refers to the ith element of vector b (x). We use superscripts to distinguish values taken on by components of x in different phases (iterations). Thus, x_i^k represents the value of the ith component of x in phase k. Given this notation, x_i^{k+1} can be computed using the following equation:

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

Notice that elements of the matrix A and vector b are inputs to the problem and, therefore, remain constant throughout all iterations. Computing x_i^{k+1}, however, requires access to all components x_i^k of the result vector 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 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[3] process to *coordinate* the remaining *worker* processes.

```
global
    $a[n, n]$     : matrix of real
    $b[n]$        : vector of real
    $x[n]$        : vector of real each initially 0
    complete      : vector of boolean each initially $F$
    changed       : vector of boolean each initially $F$
    done          : boolean initially $F$

    COORDINATOR                        WORKER$_i$

local
    converged(): returns               $t_i$: real initially 0
                 boolean
while ($\neg done$)                    while ($\neg done$)
    wait ($\forall i\ complete_i$)         $t_i := \Big(b_i - \sum_{j=1}^{i-1} a_{i,j}\, x_j$
    $\forall i\ complete_i := F$               $- \sum_{j=i+1}^{n} a_{i,j}\, x_j\Big) \Big/ a_{i,i}$
    wait ($\forall i\ changed_i$)          $complete_i := T$
    if ($converged()$)                 wait ($\neg complete_i$)
        $done := T$                        $x_i := t_i$
    $\forall i\ changed_i := F$            $changed_i := T$
                                       wait ($\neg changed_i$)
```

Figure 8: Synchronous Iterative Linear Solver

Figure 8 shows the synchronous, iterative linear solver using shared variable synchronization and a coordinator process. (The notation "**wait** (B)" for some boolean condition B is equivalent to "**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, but at a loss of clarity. 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 with the coordinator before resuming the next phase of computation or terminating. It is possible to eliminate the first synchronization handshake by having workers read alternately from x_i and t_i but we do not consider that here.

We claim the code in Figure 8 correctly solves the system $Ax = b$ on both atomic and causal memory. This follows from a simple argument. Notice that the x_i, and the handshake bits ($complete_i$ and $changed_i$)

---

[3] Decentralized (distributed) synchronization is complex even on atomic memory [12]. Such solutions are possible with causal memory but they, like their atomic memory counterparts, are complex. Furthermore, such causal memory solutions require process *participation* in the form of additional helper processes. See [11] for more details on these constructions.

consider a read miss that introduces a new value (v', VT') into the cache. The existing algorithm invalidates x if $VT < VT'$. We may avoid this invalidation even if $VT < VT'$ if $VT[i] = VT'[i]$.

2. *Reducing the Blocking of Processors:* In the algorithm, a writing processor blocks awaiting a response from the owner. Such processors may continue execution if they read only cached values. Subsequent writes or read misses by the same processor must force blocking until a response to the previous write is received. A further optimization allows writes to locations stored by the same owner to proceed without blocking since messages between processors are ordered. Thus, blocking may be reduced whenever writes are followed by cache reads or writes to locations of the same owner.

3. *Scaling the Unit of Sharing:* We can associate timestamps with pages instead of individual locations. It is possible for a processor to write several locations in a page but the page must be sent to the owner before writing another page or a read miss. Since the page could be written by several processors concurrently, the owner must apply their writes to its copy separately (it cannot just overwrite its copy with the page received from another processor). For example, consider page p that is owned by processor P_1. Assume that P_2 and P_3 add p to their caches as the result of read misses and then write locations x and y respectively. When P_1 receives the page first from P_2 and then from P_3 when they execute the write operations, P_1 must not overwrite x when P_3's write is received. Thus, P_1 should selectively update the locations that are written by the sender processor. Notice that since a single timestamp is used for all locations in a page, timestamps for a particular value appear to advance when writes are done to other locations in the page. This does not effect correctness but causes more invalidations.

4. *Handling of Concurrent Writes:* Currently the owner's existing value is always overwritten whenever a new value is received. If this new value represents a concurrent write it is possible for the owner to choose among the two values based on some resolution policy. In Section 4 we show how programmers can elegantly exploit such policies.

5. *Propagating New Values:* Currently, processors request new values of cached locations only after invalidations or discards. Owners can easily keep information about processors caching locations and propagate new values to caching processors. Such propagation occur asynchronously.

Several of the above improvements represent policies that may be advantageous for particular applications or classes of applications. For example, propagation of new values is useful when processors repeatedly read values. Similarly, page size also has an effect on invalidations since timestamps for locations appear to advance faster, read or write misses due to false invalidations must be considered. We illustrate in the next section that it is useful to choose the value written by the owner processor when concurrent write operations are executed for a location.

4 Programming

In this section we show that causal memory can be easily programmed, using techniques similar to those used to program atomic shared memory. We begin by showing both synchronous and asynchronous iterative programs for solving a system of linear equations. These programs, written for atomic memory, run without change on causal memory when processes synchronize using shared variables. We give a simple message counting argument that shows executions on causal memory to be more efficient. 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. We believe that these examples, coupled with the efficient implementation of causal memory shown in the previous section, demonstrate causal memory to be a particularly attractive target architecture for DSM systems.

4.1 Linear Solver

Very large systems of linear equations often arise in many scientific and engineering applications. Parallel solutions to these problems are useful and appropriate, often providing the only tractable solutions possible. Iterative methods such as the Gauss-Seidel algorithm are particularly amenable to parallelization. Li [17] investigated such an application and found that good speedups can be obtained even on atomic DSMs. (Solutions to other numerical algorithms in DSM environments are discussed in [26].) We believe that even better performance can be obtained on causal memory.

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

Figure 7: A Weakly Consistent Execution Allowed by the Causal Memory Implementation

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 7 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 [3] for a detailed proof of the algorithm.

Notice that our approach embodies a *sufficient* but not *necessary* condition for correctness. Our condition may be overly pessimistic in some cases but it provides a simple, easily implemented rule which guarantees that no read may return an overwritten (causally prior) value. As an example of such a case, consider the process histories $P_1: \; w(x)1 \; w(y)2$ and $P_2: \; r(x)1 \; r(y)2$ and assume that P_1 owns both x and y. If neither location is initially cached at P_2 then our algorithm will invalidate x in P_2's cache when y is introduced even though x's value has not been overwritten.

3.3 Improvements to the Basic Algorithm

The basic implementation algorithm can be improved in several ways. Since it is structured in terms of access to individual locations, it is not very practical in distributed systems where it is costly to transmit small amounts of data over the network. However, we show that the unit of sharing could be scaled up to a page. Similarly, the blocking of processes when they send new values to the owner processors could be reduced. We briefly discuss some of these improvements.

1. *Eliminating Unnecessary Invalidations:* By using both an owner protocol and vector timestamps to implement causal memory we have access to valuable information about the state of memory that can be exploited to reduce unnecessary invalidations. If we require owners to increment their timestamps when servicing write requests we can tell if a location owned by processor P_i has been modified by inspecting the ith component of an incoming vector timestamp. Assume that the cached value of x (owned by P_i) at P_j is (v, VT). Now

rate vector timestamp VT_i. Vector timestamps may be *incremented*, *updated*, and *compared*. Processor P_i increments VT_i by adding one to the ith component, $VT_i[i]$. (Notationally, $increment(VT_i)$ returns a new vector timestamp VT' where $\forall j \neq i : VT'[j] = VT_i[j]$ and $VT'[i] = VT_i[i] + 1$.) $update(VT, VT')$ 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 $\exists j : VT[j] < VT'[j]$. Given these operations, the vector timestamp protocol that captures causality is easily explained. 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 $M_i[x] = (v, VT)$. (We may access the elements of this pair using the notation $M_i[x].value$ and $M_i[x].VT$.) 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 th 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 [11] for an implementation that does pay this cost.) 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. To do this, we invalidate 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 full algorithm that implements this scheme is shown in Figure 6. 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 the local timestamp and sends this to the owner, along with the value being written. The owner's local vector timestamp is then updated

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

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

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

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

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

Figure 6: A Simple Owner Protocol Implementing 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 4: Causal Broadcasting is Not Causal Memory

$P_1: \ w(x)1 \ r(x)2 \ w(y)3$
$P_2: \ w(x)2 \ r(x)1 \ w(z)4$
$P_3: \ r(y)3 \ r(z)4 \ r(x)?$

Figure 5: (Strict) Causal Memory is Not Prefix-Extendible

sages 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. Consider the program execution in Figure 4 which is not allowed by causal memory but is possible when the write operations are treated as broadcast messages. Since $w(x)5$ and $w(x)2$ are concurrent, the messages corresponding to them could arrive in different orders at the processes. In particular, if the two messages arrive in different orders at P_2 and P_3 (at P_2 $w(x)5$ arrives after $w(x)2$) then the values shown in Figure 4 will be returned, where $r(x)2$ returns value 2 which is not in $\alpha(r(x)2)$. In fact, it is possible to obtain the execution in Figure 5 in which α for the last read operation of P_3 is empty. Thus, in the absence of other writes, there is no value that can be returned by this read operation of P_3 which is contrary to the fact that there should always be a value that could be read. Clearly, a correct implementation of causal memory must disallow the execution of Figure 5 and must do more than simply delivering written values in an order that preserves causality.

3 Implementation

In this section we show how to implement a causal distributed shared memory using only local memory accesses and reliable, ordered message passing between any two processors. After describing our implementation, we then discuss several possible enhancements of the basic algorithm.

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 $i = owner(x)$ 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. We require that messages to and from owners are reliable and point-to-point ordered. That is, messages sent by P_i to P_j are guaranteed to be delivered to P_j in the order in which they were sent by P_i, but no guarantees are made on the relative order in which P_j receives messages from distinct processors.

We have now explained all of the mechanics of the implementation but have not yet revealed how correctness is maintained. Recall that causal memory correctness is intimately related to causality. A simple *vector timestamp protocol* [19] may be used to capture precisely the evolving partial ordering of events in a distributed system, and thereby, causality. A vector timestamp is simply a vector of scalar timestamps, one per processor. Each processor in the system maintains a sepa-

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

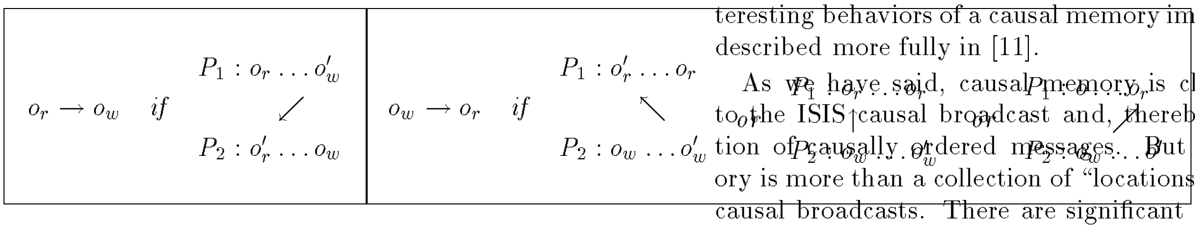

Figure 3: Relating o_w and o_r

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 (including the initial write of 0). In general, a read and write by different processes can only be related in the ways shown in Figure 3. A read o_r by P_1 can only precede a write o_w by P_2 if there is some write o'_w by P_1 following o_r that precedes (perhaps transitively) some read o'_r by P_2 that, in turn, precedes o_w. o_r can only follow o_w by one of the similar relations shown.

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. In fact, $\alpha(r_1(z)5) = \alpha(r_3(z)5)$. By a similar argument we conclude that $\alpha(r_2(y)3) = \{0, 2, 3\}$ ($r_2(y)3$ is concurrent with *both* $w_1(y)2$ and $w_1(y)3$) 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. In both cases, such policies can introduce subtle and interesting behaviors of a causal memory implementation described more fully in [11].

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. First, memory has overwrite semantics and messages have queueing semantics. There are several implications of this. A message recipient can be assured that repeated receives will provide all the messages that have been sent. But repeated reads can not guarantee that all values written will be read. "Dead writes," values overwritten before they are read, are always possible. Stated positively, reads may occur at the reader's discretion. A message recipient is forced to receive all earlier messages in order to receive a later, interesting message. A related property of memory allows writes to different locations to be read in an order different from the order written (or, perhaps, not at all!). Such a behavior would violate causal ordering in a message system. Another difference between the models is that the same value may be read repeatedly by a single process whereas, in a message system, receives are destructive, removing the message from the queue. Another way of saying this is that causal memory is fundamentally *non-blocking*,[2] reads may continue indefinitely without further intervention by other processes. Message system are inherently blocking, requiring cooperation between the sending and receiving processes.

In message passing, if several messages arrive at a process, they are queued until processed. On the other hand, it is only necessary to maintain a single value for a memory location. 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 mes-

---

[2] A *cache-based implementation* of causal memory may introduce blocking in order to fetch a copy of some location which has not been previously accessed from that location's owner, but the definition of causal memory does not ever require blocking of an implementation.

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

Figure 1: Example of Causal Relations

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 may relate reads and writes using a shared memory analog of Lamport's [13] *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.

The transitive closure of $\rightarrow$ (denoted $\xrightarrow{*}$) is also a useful notion. If o *transitively precedes* o' ($o \xrightarrow{*} o'$) 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 $\xrightarrow{*}$ are said to be *concurrent*. In the example execution in Figure 1 the writes of x and z are concurrent and $w(x)1 \xrightarrow{*} r_1(y)2$. 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 $w(x)1 \xrightarrow{*} r_1(x)1$ already established by program order. The read $r_2(x)1$ also confirms the ordering $w(x)1 \xrightarrow{*} r_1(x)1$ established by the intervening read $r_2(y)2$.

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 $\xrightarrow{*}$ 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. Informally, we consider the program order and reads-from relations P_i participates in *before* o returns a value, excluding the reads-from relation established by the execution of o. Given a write operation $o' = w(x)v$, how might o' be related to o (excluding the reads-from relation established by o as described above)? The write o' 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 ($w(x)v \xrightarrow{*} w(x)v' \xrightarrow{*} o$) 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 a read operation $o = r(x)$ and a write operation $o' = w(x)v$, the value v is* live *for o ($v \in \alpha(o)$) if either:*

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

Given α we may now state the definition of causal memory.

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 read operation $o = r(x)v$, $v \in \alpha(o)$.*

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 guing that no consistent memory can retain low latency and scale. Large-scale multiprocessors often avoid the cost of consistency by allowing occasional inconsistent executions. For example, neither the IBM RP3 nor the NYU Ultracomputer [25] provide strong consistency in hardware. These early efforts focused on providing software mechanisms for *hiding* the occasional inconsistency of the hardware. Lipton and Sandberg's PRAM [18] is also of this character. Researchers in the architecture community have also begun to question the wisdom of always maintaining strong consistency [1, 16, 9, 6]. More recent work [10] has suggested that the *principled weakening* of consistency may solve problems of latency and scale and still provide a reasonable target architecture for programmers. Weakly consistent memories have the additional advantage of making consistency requirements explicit that remain implicit in programs on strongly consistent memory.

---

[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*.

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 [11] where the memory discussed in this paper is called *strict* causal memory.) Informally, causal memory requires that the value returned by a read of causal memory must be 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. In fact, multiple processors can write a location concurrently and independently, without having to synchronize with each other. Subsequent readers may disagree on the relative ordering of these concurrent writes. Causal memory is based on the concept of *potential causality* introduced by Lamport in [13]. Lamport's 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 is also closely related to the ISIS *causal broadcast* introduced in [4]. Causal broadcasts ensure that message delivery respects the *causal ordering* of broadcast events. Causal memory provides similar guarantees but in an environment with memory instead of message semantics.

A notion similar to our causal memory was introduced independently by Scheurich in his 1989 thesis [24]. He identifies *causal correctness* as a property enjoyed by various restrictions of sequentially consistent memory although he does not consider a causally correct memory to be interesting in its own right. Also his statement of causal correctness is informal compared to our own and does not distinguish the logical variants (flickering, strict, etc.) that we discuss elsewhere in depth [11]. Nonetheless, Scheurich's work is well-informed and valuable, particularly as a study of memory consistency in tightly-coupled multiprocessors. His causal correctness is a refinement of the earlier notion of *strong ordering* introduced by Dubois, Scheurich and Briggs in the seminal work [7]. This work introduced the authors and many other researchers to the topic of shared memory consistency.

Our goals in this paper are to show that:

- Causal memory can be efficiently implemented in a DSM environment. We demonstrate implementation algorithms that require 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 (strongly consistent) and causal (weakly consistent) memories, and since implementations of causal memory are clearly more efficient than implementations of strongly consistent memory, we argue that programming with causal memory leads to improved performance.

The rest of the paper is structured as follows. Section 2 defines causal memory and shows sample executions. Our base implementation algorithm is presented in Section 3 along with several possible enhancements. Section 4 discusses programming causal memory and presents solutions to several applications. 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

Implementing and Programming Causal Distributed Shared Memory*

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

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280 USA
Phone: (404) 894-2593

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 show that our implementation is more efficient than comparable coherent DSM implementations. Possible enhancements and a proof of correctness for the implementation are included. Moreover, we show that writing programs for causal memory is no more difficult than writing programs for atomic shared memory. We present several applications on causal memory including an iterative linear equation solver and a distributed dictionary and argue that causal memory provides improved performance for these applications. We believe that causal memory is an attractive target architecture for DSM systems.

1 Introduction

The shared memory abstraction in distributed systems is attractive because it allows processes to access local and remote information in a transparent manner. This uniformity of access simplifies the programming of distributed applications, eliminating the need for separate mechanisms to access local state (reads and writes of memory) and remote state (messages or remote procedure calls). Moreover, many find programming shared memory systems less cumbersome and more intuitive than using explicit message passing. 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 are poorly matched to the facilities and characteristics of distributed systems. For example, caching of recently accessed locations is a successful optimization in multiprocessors largely because cached values can be inexpensively invalidated to maintain consistency. Such invalidations are expensive in a distributed system and require atomic (ordered) multicasts to all nodes caching copies of an invalid value. Weakly consistent memories allow implementations better suited to the hardware environment provided in distributed systems than strongly consistent implementations derived from tightly-coupled multiprocessor environments.

The traditional notion of memory correctness requires reads to 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 [20]. 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 [14] is a weakening of atomic memory that relaxes the requirement that operations take effect during their operation intervals. Several researchers ([23, 2, 5]) 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 ([17, 22]) require frequent, expensive global synchronization (e.g. invalidations of copies at several nodes) leading to inefficiency and problems of scale. The costly problem of maintaining strong memory consistency is not unique to distributed systems, however. Lipton and Sandberg [18] make this point forcefully ar-

---

*This work was funded by NSF grants CCR-8619886 and CCR-8806358.