# Causal Memory Meets the Consistency and Performance Needs of Distributed Applications![*]


Mustaque Ahamad[†]
Ranjit John
Prince Kohli
Gil Neiger

College of Computing
Georgia Institute of Technology
Atlanta, GA U.S.A



**Abstract**

In order to provide acceptable performance in large scale distributed systems, shared data must be cached at or close to nodes where it is accessed. Maintaining the consistency of such cached data is an important problem in distributed systems. We claim that *causal memory*, which defines consistency of shared data based on causal orderings between data accesses, provides strong enough consistency guarantees to be usable yet it allows efficient and scalable implementations. In this paper, we describe some results of our recent work that support this claim.


## 1 Introduction

Distributed systems differ from other types of systems in the support they provide for sharing state across nodes. Operating systems must provide abstractions that reduce the complexity of state sharing, yet it should be possible to implement these abstractions efficiently. A shared memory abstraction can be provided in distributed systems in software, but due to large latencies and high communication costs, efficient implementations cannot be developed because of the strong consistency guarantees of traditional shared memories. Recently several memory models have been developed both for multiprocessor and distributed systems [7, 4, 9] that weaken consistency guarantees provided by the memory system as a way to improve performance. We have explored a weakly consistent memory, called *causal memory*, which ensures that values returned by read operations are consistent with the causal orderings [5, 11] between data accesses.

We claim that the consistency guarantees provided by causal memory are most appropriate for large scale distributed systems: many applications can easily be programmed with shared data


[*]This work was supported in part by the National Science Foundation grants CCR-9106627, CCR-9301454, and ARPA contract N00174-93-K-0105.



[†]Contact author – email address: mustaq@cc.gatech.edu, phone (404) 894-2593, fax (404) 853-9378.

that is causally consistent, and there are efficient and scalable implementations of causal memory. From a theoretical perspective, causal memory is interesting because it permits non-blocking implementations; both read and write operations can return before communication completes with other nodes. For it to be useful in practice, it must provide a viable architecture for programming many distributed applications, and it must have implementations that provide better performance for these applications. Based on our recent results, we claim that causal memory can meet both of these goals. In the next section, we provide a brief summary of these results.

## 2 Programming with Causal Memory

Most distributed applications can be programmed in a system that provides causal memory assuming that the system actually provided a sequentially consistent memory. There are two main classes of such applications. First, a large class of applications employ explicit synchronization operations to ensure that a sequence of memory operations are executed atomically. Such applications are data-race-free [1], and we have proved that, for these applications, code that executes correctly with sequentially consistent memory will also execute correctly with causal memory [2].

The other application domain is the general area of computer supported cooperative work (CSCW) where shared data is accessed by asynchronously interacting users. In these applications, processes executing at different nodes share a set of objects. Copies of recently accessed shared objects should be cached locally to avoid communication with remote nodes. Naturally, the question of consistency arises between the multiple copies of shared objects. However, due to the sharing requirements of the applications, it is not necessary that changes to an object be propagated immediately to other nodes or that all users view concurrently made changes in the same order. In fact, we have found that causal memory precisely captures the consistency needs of objects shared in these applications.

## 3 Implementing Causal Memory

To verify that causal memory truly leads to performance improvements for distributed applications, we have implemented it in two different systems. The first implementation is done in the Clouds native operating system and it makes use of page fault and access violation traps to trigger consistency actions. These actions are executed by the operating system and the implementation of causal memory is integrated with the virtual memory system. Thus, consistency is provided at the level of a page. The second implementation is done at the user-level on UNIX and it uses the PVM package for inter-node communication. Page access violations are used to detect when an object is not locally cached. Consistency is maintained at the level of user defined objects rather than for pages. We use synchronization points or events such as the completion of a paragraph in a collaborative editing system to trigger the execution of consistency maintenance actions in this implementation.

The key features of both the system and user-level implementations are the use of vector timestamps to detect causal consistency of cached data and the use of local invalidations when data is suspected to be inconsistent. Thus, consistency related actions are limited to making locally cached data inaccessible and these actions are only executed when new objects are added to the local cache or certain synchronization operations are executed. As a result, invalidation or update messages are not sent to nodes that cache the data being written. Thus, causal memory provides enhanced concurrency because readers can access data concurrently with a writer without the writer having to invalidate or update their copies. Furthermore, communication is always between pair of nodes and multicast or broadcast communication is not required. To compare causal memory with other systems, we have also implemented memory systems based on several models and have a message passing interface available to us in the native operating system. The latter is used to measure the best performance that can be achieved by hand-coded message passing implementations of the applications.

## 4 Performance Results

We have used a number of parallel applications to evaluate the performance improvements made possible by causal memory. These include three applications from the NAS kernels which are embarrassingly parallel (EP), integer sort (IS), and conjugate gradient method (CGM). We also programmed matrix multiplication (MM), successive over relaxation (SOR), and traveling salesperson (TSP). Actual measurements of completion times of these applications with three systems, sequentially consistent memory, causal memory and message passing, are shown in Figure 1. For the three applications (EP, IS and MM), the completion times are similar on the three systems because of the data sharing patterns of the applications. For example, in EP, there is hardly any data sharing between processes and hence the state sharing method is unimportant.

In CGM, TSP and SOR, there is significant sharing between processes. For example, in CGM, each iteration has two phases. In the first phase, processes generate new data values which are used by a single process in the second phase. Processes share a work queue and a tour value in TSP to implement a branch-and-bound technique for finding the best tour. The red/black algorithm that we implemented for SOR has nearest neighbor data sharing pattern. In all these applications, it can be seen that the completion times for causal memory are lower than sequentially consistent memory. In particular, causal memory provides 5 to 50% reduction in completion times compared to sequentially consistent memory implemented using a variant of the invalidation based protocol used in Ivy [12]. We also measured various components of the completion time of an application which include actual user computation time, communication time, and synchronization time. Communication time is what a process spends waiting for a page fault to be serviced or to receive a message. This component of completion time depends on the underlying state sharing technique and our results show that causal memory reduced it significantly. For example, compared to sequentially consistent memory, this decrease ranged from 20 to 85 % for the applications. Also, we have observed that, because the causal memory implementations do not require global synchronization (such as invalidation of all copies of a page), their scalability is better. The performance improvements relative to sequentially consistent memory are possible without any change in the way the applications are programmed.

Message passing implementations of the applications do provide better performance (5 to 30 % lower completion time) but the implementations are very different. The difference in the execution times between causal memory and message passing are more due to the page-based implementation of causal memory. In our system, page size is 8K bytes. Since in message passing implementations only the data that needs to be shared is transferred whereas the minimal unit of transfer is a page in causal memory, in some applications the differences in the amount of data transferred are significant. For example, for TSP, message passing only transmits 18K bytes compared to 20 Mbytes in causal memory. Our user-level implementation of causal memory is not page-based which provides further performance improvements. For TSP, the difference in completion times for causal memory and message passing implementations reduces from 23% to 10% when the application is run with the user-level implementation.

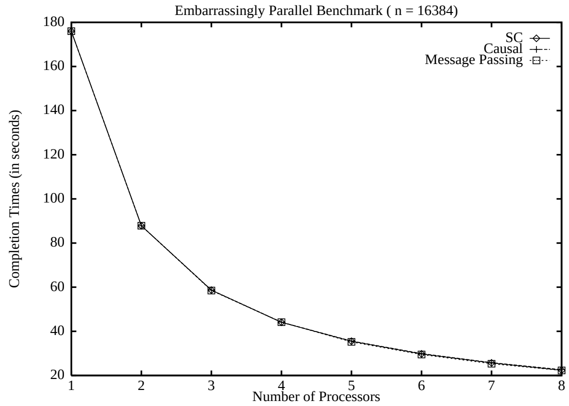
We have compared the performance of the applications with sequentially consistent memory and message passing systems. A natural question is how causal memory compares with systems based on release consistency (RC), lazy release consistency (LRC) or entry consistency (EC). RC is implemented in the Munin system [6] but it also requires that users provide annotations that specify the sharing patterns for a set of data items. These annotations are used to determine the best coherence protocol for the set of data items. LRC is implemented by the TreadMarks system [10] and Midway implements EC [4].

In addition to the sequentially consistent memory system, we have also compared causal memory with a memory system derived from the RC approach [8]. We found that causal memory outperforms RC because of two main reasons. First, although RC allows the system to asynchronously send coherence related messages, it still requires that all nodes that have copies of a data item be notified when the data item is written. Also, like Munin, our implementation of RC used the *diff* mechanism to handle concurrent writing to a page due to false sharing. The processing overheads associated with the *diff* operations were found to be high.

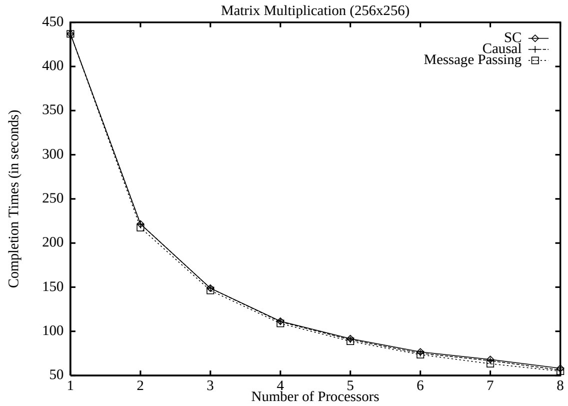
It can be argued that our implementation of causal memory provides performance competitive with LRC for programs that have explicit synchronization. In particular, the local invalidations at a node when it acquires a synchronization variable (e.g., lock) are similar to the consistency actions done by LRC at the node that next acquires the synchronization variable. There are differences in what exact information is communicated in the two systems but the fact that coherence actions are performed only at the nodes where the data is going to be accessed next result in a similar number of messages and we expect the performance of causal memory and LRC to be similar. EC has an advantage over both causal memory and LRC due to the explicit associations between data and synchronization variables. Thus, when a lock is transferred, EC only sends changes to data that will be accessed in the critical section controlled by the lock. On the other hand, LRC and causal memory must transfer information about changes to all shared data items because no such associations are assumed. We compared causal memory with a memory system that is similar to the Midway implementation of EC. Although this system does provide somewhat better performance than causal memory (difference between completion times are similar to the differences between message passing and causal memory), it requires programmer assistance.

# 5 Conclusions

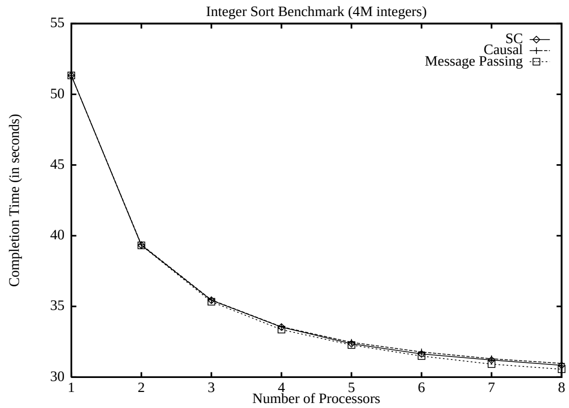
The experimental results we have obtained so far strongly suggest that causal memory provides the correct level of consistency for managing distributed shared state. Its consistency guarantees are strong enough for building many practical distributed applications and, at the same time, it allows



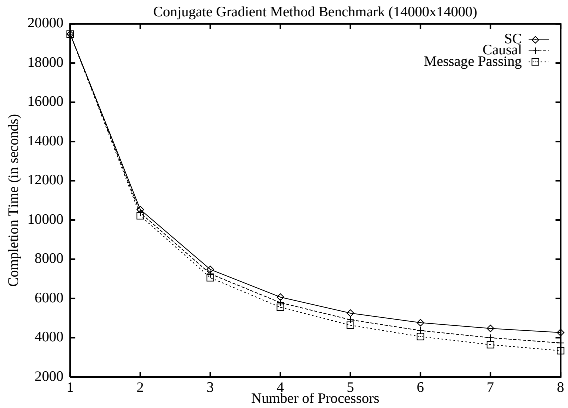

(a) EP



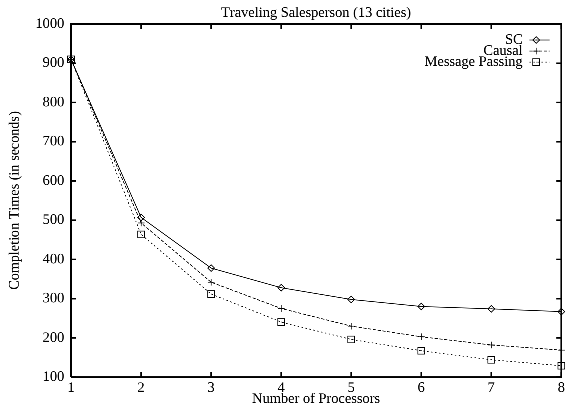

(b) MM



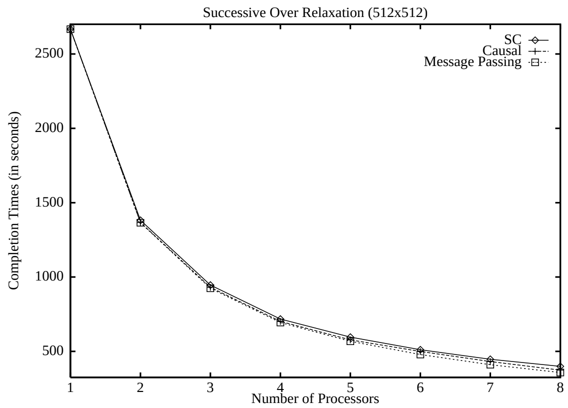

(c) IS




(d) CGM




(e) TSP




(f) SOR

Figure 1: Completion Times

more efficient and scalable implementations. Furthermore, it is similar to lazy release consistency [9] and entry consistency [4] in that it can exploit information about synchronization operations in performing consistency maintenance. Additionally, it allows cooperative applications to be programmed where changes to shared state are propagated asynchronously. These observations and the more scalable nature of the implementations of causal memory lead us to believe that causal memory provides the right abstraction for sharing state in large scale distributed systems.

# References


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

[2] Mustaque Ahamad, Gil Neiger, Jim Burns, Prince Kohli and Phillip Hutto. Causal Memory: Definitions, Implementation and Programming. Georgia Tech Technical Report 93/54. 1993.

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

[4] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, September 1991.

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

[6] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. *ACM Symposium on Operating Systems Principles*, October 1991.

[7] Kourosh Gharchorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[8] Ranjit John, Mustaque Ahamad, Umakishore Ramachandran, Ajay Mohindra and R. Ananthanarayanan. An evaluation of state sharing techniques in distributed operating systems. Technical Report GIT-CC-93-73, College of Computing, Georgia Institute of Technology.

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

[10] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, January 1994.

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

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