# Sequential Consistency versus Linearizability

(EXTENDED ABSTRACT)

Hagit Attiya*
Department of Computer Science
The Technion
Haifa 32000, Israel

Jennifer L. Welch†
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

## Abstract

The power of two well-known consistency conditions for shared memory multiprocessors, *sequential consistency* and *linearizability*, is compared. The cost measure studied is the worst-case response time in distributed implementations of virtual shared memory supporting one of the two conditions. The memory is assumed to consist of read/write objects. The worst-case response time is very sensitive to the assumptions that are made about the timing information available to the system. All the results in this paper assume that processes have clocks that run at the same rate as real time and that all message delays are in the range $[d-u, d]$ for some known constants $u$ and $d$, $0 \leq u \leq d$. If processes have perfectly synchronized clocks or if every message has delay exactly $d$, then there are linearizable implementations in which one operation (either read or write) is performed instantaneously and the response time of the other operation is $d$. These upper bounds match exactly a lower bound for sequential consistency, proved by Lipton and Sandberg, on the sum of the response times of read and write operations. If clocks are not perfectly synchronized and if message delays are variable, i.e., $u > 0$, then such a tradeoff cannot be achieved by linearizable implementations: the response time for both read and write operations is at least $\Omega(u)$. In contrast, we present sequentially consistent implementations for this weaker timing model in which one operation (either read or write) is performed instantaneously, and the worst-case response time of the other operation is $O(d)$.

## 1 Introduction

A fundamental problem in concurrent computing is how to provide programmers with a useful model of logically shared data that can be accessed atomically, without sacrificing performance. The model must specify how the data can be accessed and what guarantees are provided about the results. Shared memory is an attractive paradigm for communication among computing entities because it is familiar from the uniprocessor case, it can be considered more high level than message passing, and many of the classical solutions for synchronization problems were developed for shared memory (e.g., mutual exclusion [13]).

This problem arises in many situations at different levels of abstraction. These situations include implementing a single shared variable out of weaker shared variables, cache coherence, building multiprocessors (with both physical and distributed shared memory), and high-level applications for loosely-coupled distributed systems such as distributed file systems and transaction systems.

*Email: hagit@cs.technion.ac.il. Part of this work was performed while the author was at the Laboratory for Computer Science, MIT, supported by ONR contract N00014-85-K-0168, by NSF grants CCR-8611442 and CCR-8915206, and by DARPA contracts N00014-89-J-1988 and N00014-87-K-0825.

†Email: welch@cs.unc.edu. The work of this author was supported in part by NSF grant CCR-9010730 and an IBM Faculty Development Award.

To enhance performance (e.g., response time, availability, or fault-tolerance), many implementations employ multiple copies of the same logical piece of shared data (*caching*). Also, multiple user programs must be able to execute "concurrently," either with interleaved steps, or truly in parallel. More complications arise because at some level, each access to shared data has duration in time, from its start to its end; it is not instantaneous.

Thus, the illusion of atomic operations on single copies of objects must be supported by a *consistency mechanism*. The consistency mechanism guarantees that although operations may be executed concurrently on various copies and have some duration, they will *appear* to have executed atomically, in some sequential order that is consistent with the order seen at individual processes.[1] When this order must preserve the global (external) ordering of non-overlapping operations, this consistency guarantee is called *linearizability* ([18]);[2] otherwise, the guarantee is called *sequential consistency* ([20]). Obviously, linearizability implies sequential consistency.

Sequential consistency and linearizability are probably the two best-known consistency conditions. As the definitions of these two conditions are similar, it is important to study the relationships between them. In this paper we present a quantitative comparison of the cost to implement sequential consistency and linearizability in a non-bused distributed system. Distributed implementations are of great interest because of their ability to scale up in size. The comparison is based on time complexity — the inherent response time of the best possible distributed implementation supporting each consistency condition. That is, we present upper and lower bounds on the worst-case response time for performing an operation on an object. We concentrate on read/write objects.

We consider a collection of application programs running concurrently and communicating via virtual shared memory. The shared memory consists of a collection of *read/write objects*. The application programs are running in a distributed system consisting of a collection of nodes and a complete communication network.[3] The shared memory abstraction is implemented by a *memory consistency system* (mcs), which uses local memory at the various nodes and some protocol executed by the mcs processes (one at each node). (Nodes that are dedicated storage are modeled by nullifying the application process.) Fig. 1 illustrates a node, on which is running an application process and an mcs process. The application process sends calls to access shared data to the mcs process; the mcs process returns the responses to the application process, possibly based on messages exchanged with mcs processes on other nodes.

The correctness conditions are defined at the interface between the application processes (written by the user) and the mcs processes (supplied by the system). Thus, the mcs must provide the proper semantics when the values of the responses to calls are considered, throughout the network.

It turns out that timing information available in the model has a crucial impact on the time complexity of implementing sequential consistency and linearizability. We assume that on each node there is a real-time clock readable by the mcs process at that node, that runs at the same rate as real-time. We assume that every message incurs a delay in the interval $[d-u, d]$, for some known constants $u$ and $d$, $0 \leq u \leq d$ ($u$ stands for *uncertainty*). If $u = 0$, then the message delays are constant.

If processes have perfectly synchronized clocks and the message delays are constant, we show that the sum of the worst-case response times for a read operation and a write operation is at least $d$. The result is proved for sequential consistency, and thus, holds also for linearizability. (This formalizes and strengthens a result of Lipton and Sandberg [23].) We then show that this tradeoff is tight—it is possible to have the response time of *only one* of the operations depend on the network's latency. Specifically, we present an algorithm in which a read operation is performed instantaneously (locally), while a write operation returns within time $d$; we also present an algorithm in which the roles are reversed. These algorithms achieve linearizability, and hence, sequential consistency. (This upper bound indicates that separating sequential consistency from linearizability is not as obvious as it may seem.)

[1] This condition is similar in flavor to the notion of *serializability* from database theory ([7, 27]); however, serializability applies to *transactions* which aggregate many operations.

[2] Also called *atomicity* ([17, 21, 26]) in the case of read/write objects.

[3] The assumption of a complete communication network can be omitted and is made here only for clarity of presentation.

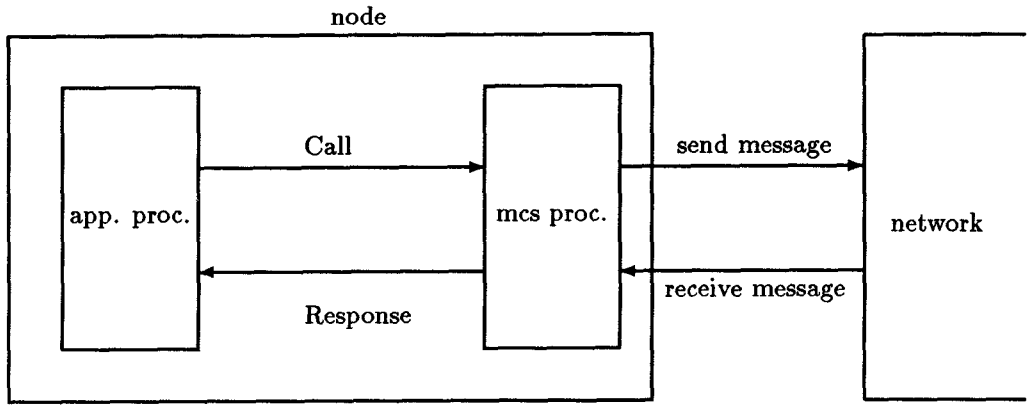



Figure 1: System Architecture

We then turn to the more realistic case of approximately synchronized clocks and uncertain message delays. We show that if linearizability is desired, neither operation can be performed instantaneously, regardless of the response time for the other operation. Specifically, we show that the worst-case response time of a read operation must be at least $u/4$ and the worst-case response time of a write operation must be at least $u/2$. (Note that $u$ can be as big as $d$.) In contrast, we present sequentially consistent implementations of read/write objects in which one operation (either read or write) is performed instantaneously (locally), and the response time of the other operation is $O(d)$. Thus, sequential consistency admits significantly more efficient implementations than linearizability, when there are significantly more operations of one type and under certain timing assumptions.

Our proofs make use of techniques from the theory of distributed systems: The lower bounds for implementations of linearizable objects are proved using *shifting* arguments, originally used in [24] for clock synchronization problems. Our efficient implementations of sequential consistency use *timestamps* in a way that was inspired by the *atomic broadcast* algorithm of [9].

Several papers have proposed sequentially consistent implementations of read/write objects, which were claimed to achieve a higher degree of concurrency (e.g., [2, 3, 6, 10, 14, 25, 29]). In particular, Afek, Brown, and Merritt ([3]) present a sequentially consistent implementation of read/write objects, for systems where processes communicate via a bus. A bus enforces global ordering on all messages delivered to the processes; such a property is not provided in a communication network. None of these papers provides an analysis of the response time of the implementations suggested (or any other complexity measure). Furthermore, none of these papers proves that similar improvements cannot be achieved for linearizability. To the best of our knowledge, this is the first time such a result is shown.

This paper addresses a simplification of the problem of memory coherence in loosely-coupled multiprocessors ([6, 10, 8, 14, 22, 25, 28, 29]). Our formal model ignores several important practical issues, e.g., limitations on the size of local memory storage, network topology, clock drift and "hot-spots". Since our lower bounds are proved in a very strong model, they clearly hold for more practical systems. We believe our algorithms can be adapted to work in more realistic systems.

## 2 Correctness Conditions

We begin with an informal description of the system model. A memory consistency system (mcs) consists of a collection of processes, one on each node of a distributed system.

Process $p$ interacts with the application program using *call* events $\text{Read}_p(X)$ and $\text{Write}_p(X, v)$ for all objects $X$ and value $v$, and *response* events $\text{Return}_p(X, v)$ (for Read) and $\text{Ack}_p(X)$ (for Write). It communicates with other processes using *message-send* and *message-receive* events. It sets timers for itself (to go off at some future clock time) and responds to them using *timer-set* and

*timer* events. The process is modeled as an automaton with states and a transition function that takes as input the current state, clock time, and a call or message-receive or timer event, and produces a new state, a set of response events, a set of message-send events, and a set of timer-set events. A *history* of a process describes what steps (i.e., applications of the transition function) the process takes at what real times; it must satisfy certain natural "consistency" conditions.

An *execution* of a set of processes is a set of histories, one for each process, satisfying the following two conditions: (1) A timer is received by $p$ at clock time $T$ if and only if $p$ has previously set a timer for $T$. (2) There is a one-to-one correspondence between the messages sent by $p$ to $q$ and the messages received by $q$ from $p$, for any processes $p$ and $q$. We use the message correspondence to define the *delay* of any message in an execution to be the real time of receipt minus the real time of sending. (The network is not explicitly modeled, although the constraints on executions imply that the network reliably delivers all messages sent.) An execution is *admissible* if the delay of every message is in the range $[d-u, d]$, for fixed nonnegative integers $d$ and $u$, $u \leq d$, and for every $p$, at any time at most one call at $p$ is *pending* (i.e., lacking a matching subsequent response).

Every object is assumed to have a *serial specification* (cf. [18]) defining a set of *operations*, which are ordered pairs of call and response events, and a set of operation sequences, which are the allowable sequences of operations on that object. In the case of a read/write object, the ordered pair of events $[\text{Read}_p(X), \text{Return}_p(X, v)]$ forms an *operation* for any $p$, $X$, and $v$, as does $[\text{Write}_p(X, v), \text{Ack}_p(X)]$. The set of operation sequences consists of all sequences in which every read operation returns the value of the latest preceding write operation (the usual read/write semantics). A sequence $\tau$ of operations for a collection of processes and objects is *legal* if, for each object $X$, the restriction of $\tau$ to operations of $X$ is in the serial specification of $X$. Given an execution $\sigma$, let $ops(\sigma)$ be the sequence of call and response events appearing in $\sigma$ in real-time order, breaking ties by ordering all response events before any call event and then using process ids.

Our formal definitions of sequential consistency and linearizability follow. These definitions imply that every call gets an eventual response and that calls and responses alternate at each process.

**Definition 2.1 (Sequential consistency)** *An execution $\sigma$ is* sequentially consistent *if there exists a legal sequence $\tau$ of operations such that, for each process $p$, the restriction of $ops(\sigma)$ to operations of $p$ is equal to the restriction of $\tau$ to operations of $p$.*

**Definition 2.2 (Linearizability)** *An execution $\sigma$ is* linearizable *if there exists a legal sequence $\tau$ of operations such that, for each process $p$, the restriction of $ops(\sigma)$ to operations of $p$ is equal to the restriction of $\tau$ to operations of $p$, and furthermore, whenever the response for operation $op_1$ precedes the call for operation $op_2$ in $ops(\sigma)$, then $op_1$ precedes $op_2$ in $\tau$.*

An mcs is a *sequentially consistent* implementation of a set of objects if any admissible execution of the mcs is sequentially consistent; similarly, an mcs is a *linearizable* implementation of a set of objects if any admissible execution of the mcs is sequentially consistent.

We measure the efficiency of an implementation by the worst-case response time for any operation on the object. Given a particular mcs and a read/write object $X$ implemented by it, we denote by $|W(X)|$ the maximum time taken by a write operation on $X$ and by $|R(X)|$ the maximum time taken by a read operation on $X$, in any admissible execution. Denote by $|W|$ the maximum of $|W(X)|$, and by $|R|$ the maximum of $|R(X)|$, over all objects $X$ implemented by the mcs.

# 3 Perfect Clocks

We start by considering the case in which processes have perfectly synchronized (*perfect*) clocks and message delay is constant and known.[4] We first show that the sum of the worst-case response times of read and write operations is at least $d$, even in this strong model, and even under sequential consistency. This is a formalization of a result of Lipton and Sandberg ([23]) making precise the timing assumptions made on the system. We then show that the lower bound is tight for this model by describing two algorithms that match the lower bound exactly: In the first algorithm, reads are performed instantaneously, while the worst-case response time

[4] We remark that the assumptions that processes have perfect clocks and that message delays are constant (and known) are equivalent.

for a write is $d$. In the second algorithm, writes are performed instantaneously, while the worst-case response time for a read is $d$. The algorithms actually implement linearizability, which is a stronger condition than sequential consistency.

## 3.1 Lower Bound for Sequential Consistency

We start with a formal proof of a theorem presented in [23, Theorem 1]. We show that the result holds even in highly synchronous systems, in which processes have perfect clocks and message delays are constant and known. Perfect clocks are modeled by letting $C_p(t) = t$ for all $p$. The constant message delay is modeled by letting $u = 0$; $d$ is known (and can be used by the mcs).

**Theorem 3.1 (Lipton and Sandberg)** *For any memory-consistency system that is a sequentially consistent implementation of two read/write objects $X$ and $Y$, $|W| + |R| \geq d$.*

**Proof:** Let $p$ and $q$ be two processes that access $X$ and $Y$. We prove that either $|W(X)| + |R(Y)| \geq d$ or $|W(Y)| + |R(X)| \geq d$. Assume by way of contradiction that there exists a sequentially consistent implementation of $X$ and $Y$ for which both $|W(X)| + |R(Y)| < d$ and $|W(Y)| + |R(X)| < d$. Without loss of generality, assume that 0 is the initial value of both $X$ and $Y$.

By the specification of $Y$, there is some admissible execution $\alpha_1$ such that $ops(\alpha_1)$ is

$$\text{Write}_p(X, 1)\ \text{Ack}_p(X)\ \text{Read}_p(Y)\ \text{Return}_p(Y, 0)$$

and $\text{Write}_p(X, 1)$ occurs at real time 0 and $\text{Read}_p(Y)$ occurs immediately after $\text{Ack}_p(X)$. By assumption, the real time at the end of $\alpha_1$ is less than $d$. Thus no message is received at any node during $\alpha_1$.

By the specification of $X$, there is some admissible execution $\alpha_2$ such that $ops(\alpha_2)$ is

$$\text{Write}_q(Y, 1)\ \text{Ack}_q(Y)\ \text{Read}_q(X)\ \text{Return}_q(X, 0)$$

and $\text{Write}_q(Y, 1)$ occurs at real time 0 and $\text{Read}_q(X)$ occurs immediately after $\text{Ack}_q(Y)$. By assumption, the real time at the end of $\alpha_2$ is less than $d$. Thus no message is received at any node during $\alpha_2$.

Since no message is ever received in $\alpha_1$ and $\alpha_2$, the execution $\alpha$ obtained from $\alpha_1$ by replacing $q$'s history with $q$'s history in $\alpha_2$ is admissible. Then $ops(\alpha)$ consists of the operations $[\text{Write}_p(X, 1), \text{Ack}_p(X)]$ followed by $[\text{Read}_p(Y), \text{Return}_p(Y, 0)]$, and $[\text{Write}_q(Y, 1), \text{Ack}_q(Y)]$ followed by $[\text{Read}_q(X), \text{Return}_q(X, 0)]$.

By assumption, $\alpha$ is sequentially consistent. Thus there is a legal operation sequence $\tau$ consisting of the operations $[\text{Write}_p(X, 1), \text{Ack}_p(X)]$ followed by $[\text{Read}_p(Y), \text{Return}_p(Y, 0)]$, and $[\text{Write}_q(Y, 1), \text{Ack}_q(Y)]$ followed by $[\text{Read}_q(X), \text{Return}_q(X, 0)]$. Since $\tau$ is a sequence of operations, either the read of $X$ follows the write of $X$, or the read of $Y$ follows the write of $Y$. But each possibility violates the serial specification of either $X$ or $Y$, contradicting $\tau$ being legal. ■

## 3.2 Upper Bounds for Linearizability

In this section we show that the tradeoff indicated by Theorem 3.1 is inherent, and that a sequentially consistent implementation may choose which operation to slow down. More precisely, we present an algorithm in which a read operation is instantaneous (local) while a write operation returns within time $d$; we also present an algorithm in which the roles are reversed. These algorithms actually ensure the stronger condition of linearizability. They assume that clocks are perfect and message delays are constant.

Informally, the algorithm for fast reads and slow writes works as follows. Each process keeps a copy of all objects in its local memory. When a $\text{Read}_p(X)$ occurs, $p$ reads the value $v$ of $X$ in its local memory and immediately does a $\text{Return}_p(X, v)$. When a $\text{Write}_p(X, v)$ occurs, $p$ sends "write$(X, v)$" messages to all other processes. Then $p$ waits $d$ time units, after which it changes the value of $X$ to $v$ in its local memory and does an $\text{Ack}_p(X)$. Whenever a process receives a "write$(X, v)$" message, it changes the value of $X$ to $v$ in its local memory. (If it receives several at the same time, it "breaks ties" using sender ids, that is, it writes the value in the message from the process with the largest id and ignores the rest of the messages.) Clearly the time for every read is 0 and the time for every write is $d$, and $|W| + |R| = d$.

**Theorem 3.2** *The algorithm just described implements linearizability.*

The proof, which is omitted from this extended abstract, proceeds by explicitly constructing, for every admissible execution, a legal operation sequence satisfying the necessary conditions. In creating the operation sequence, each operation in the execution is serialized to occur at the time of its response.

The algorithm for slow reads and fast writes is similar to the previous one. Each process keeps a copy of all objects in its local memory. When a $\text{Read}_p(X)$ occurs, $p$ waits $d$ time units, after which it reads the value $v$ of $X$ in its local memory and immediately does a $\text{Return}_p(X, v)$. When a $\text{Write}_p(X, v)$ occurs, $p$ sends "write$(X, v)$" messages to all other processes (including a dummy message to itself which is delayed $d$ time units) and does an Ack immediately. Whenever a process receives a "write$(X, v)$" message, it changes the value of $X$ to $v$ in its local memory. Ties are resolved as in the previous algorithm. Clearly the time for every read is $d$ and the time for every write is 0, and $|W| + |R| = d$.

**Theorem 3.3** *The algorithm just described implements linearizability.*

The proof is the same as the proof of Theorem 3.2 except that each operation is serialized to occur at the time it is called.

# 4 Imperfect Clocks

Obviously, the assumptions of the previous section are unrealistically strong. In this section we relax them, and assume a system in which clocks run at the same rate as real time but are not initially synchronized, and in which message delays are in the range $[d - u, d]$ for some $u > 0$.

Under these assumptions, the lower bound of Theorem 3.1 still holds, but the algorithms of Theorems 3.2 and 3.3 do not work. We start by showing that for linearizability this is not a coincidence—in any linearizable implementation of a read/write object the worst-case response time of *both* read and write operations must depend on $u$, the message delay uncertainty. We then show that this is not the case for sequential consistency by presenting two algorithms, one in which reads are performed instantaneously while the worst-case response time for a write is $O(d)$, and another in which the roles are reversed. These algorithms match (within constant factors) the lower bound of Theorem 3.1.

## 4.1 Lower Bounds for Linearizability

We now show that, under reasonable assumptions about the pattern of sharing, in any linearizable implementation of an object, the worst-case time for a read is $u/4$ and the worst-case time for a write is $u/2$. The proofs of these lower bounds use the technique of *shifting*. Shifting is used to change the timing and the ordering of events in the system while preserving the local views of the processes. It was originally introduced in [24] to prove lower bounds on the precision achieved by clock synchronization algorithms. Here we describe the technique and its properties informally.

Given an execution with a certain set of clocks, if process $p$'s history is changed so that the real times at which the events occur are shifted by some amount $s$ and if $p$'s clock is shifted by the same amount, then the result is another execution in which every process still "sees" the same events happening at the same real time. The intuition is that the changes in the real times at which events happen at $p$ cannot be detected by $p$ because its clock has changed by a corresponding amount. It is possible to quantify the resulting changes to message delays in the new execution: the delay of any message to $p$ is $s$ less, the delay of any message from $p$ is $s$ more, and the delay of any message not involving $p$ has the same delay as in the original execution.

**Theorem 4.1** *Assume $X$ is a read/write object with at least two readers. Then any linearizable implementation of $X$ must have $|R(X)| \geq \frac{u}{4}$.*

**Proof:** Let $p$ and $q$ be two processes that read $X$ and $r$ be a process that writes $X$. Assume in contradiction that there is an implementation with $|R(X)| < \frac{u}{4}$. Without loss of generality, assume that the initial value of $X$ is 0. The idea of the proof is to consider an execution in which $p$ reads 0 from $X$, then $q$ and $p$ alternate reading $X$ while $r$ writes 1 to $X$, and then $q$ reads 1 from $X$. Thus there exists a read $R_1$, say by $p$, that returns 0

and is immediately followed by a read $R_2$ by $q$ that returns 1. If $q$ is shifted earlier by $u/2$, then in the resulting execution $R_2$ precedes $R_1$. Since $R_2$ returns the new value 1 and $R_1$ returns the old value 0, this contradicts linearizability.

Let $k = \lceil \frac{|W(X)|}{u} \rceil$. By the specification of $X$, there is an admissible execution $\alpha$, in which all message delays are $d - \frac{u}{2}$, consisting of the following operations (see Fig. 2(a)):

- At time $\frac{u}{4}$, $r$ does a $\text{Write}_r(X, 1)$.
- Between times $\frac{u}{4}$ and $(4k+1) \cdot \frac{u}{4}$, $r$ does an $\text{Ack}_r(X)$. (By definition of $k$, $(4k+1) \cdot \frac{u}{4} \geq \frac{u}{4} + |W(X)|$, and thus $r$'s write operation is guaranteed to finish in this interval.)
- At time $2i \cdot \frac{u}{4}$, $p$ does a $\text{Read}_p(X)$, $0 \leq i \leq 2k$.
- Between times $2i \cdot \frac{u}{4}$ and $(2i+1) \cdot \frac{u}{4}$, $p$ does a $\text{Return}_p(X, v_{2i})$, $0 \leq i \leq 2k$.
- At time $(2i+1) \cdot \frac{u}{4}$, $q$ does a $\text{Read}_q(X)$, $0 \leq i \leq 2k$.
- Between times $(2i+1) \cdot \frac{u}{4}$ and $(2i+2) \cdot \frac{u}{4}$, $q$ does a $\text{Return}_q(X, v_{2i+1})$, $0 \leq i \leq 2k$.

Thus in $ops(\alpha)$, $p$'s read of $v_0$ precedes $r$'s write, $q$'s read of $v_{4k+1}$ follows $r$'s write, no two read operations overlap, and the order of the values read from $X$ is $v_0, v_1, v_2, \ldots, v_{4k+1}$. By linearizability, $v_0 = 0$ and $v_{4k+1} = 1$. Thus there exists $j$, $0 \leq j \leq 4k$, such that $v_j = 0$ and $v_{j+1} = 1$. Without loss of generality, assume that $j$ is even, so that $v_j$ is the result of a read by $p$.

Define $\beta = \text{shift}(\alpha, q, \frac{u}{2})$. I.e., we shift $q$ earlier by $\frac{u}{2}$. (See Fig. 2(b).) The result is admissible since the message delays to $q$ become $d - u$, the message delays from $q$ become $d$, and the remaining message delays are unchanged.

As a result of the shifting, we have reordered read operations with respect to each other at $p$ and $q$. Specifically, in $ops(\beta)$, the order of the values read from $X$ is $v_1, v_0, v_3, v_2, \ldots, v_{j+1}, v_j, \ldots$. Thus in $\beta$ we now have $v_{j+1} = 1$ being read before $v_j = 0$, which violates linearizability. ■

**Theorem 4.2** *If $X$ is a read/write object with at least two writers, then any linearizable implementation of $X$ must have $|W(X)| \geq \frac{u}{2}$.*

The proof uses techniques similar to the proof of Theorem 4.1. It constructs an execution in which, if write operations are too short, linearizability can be violated by appropriately shifting histories.

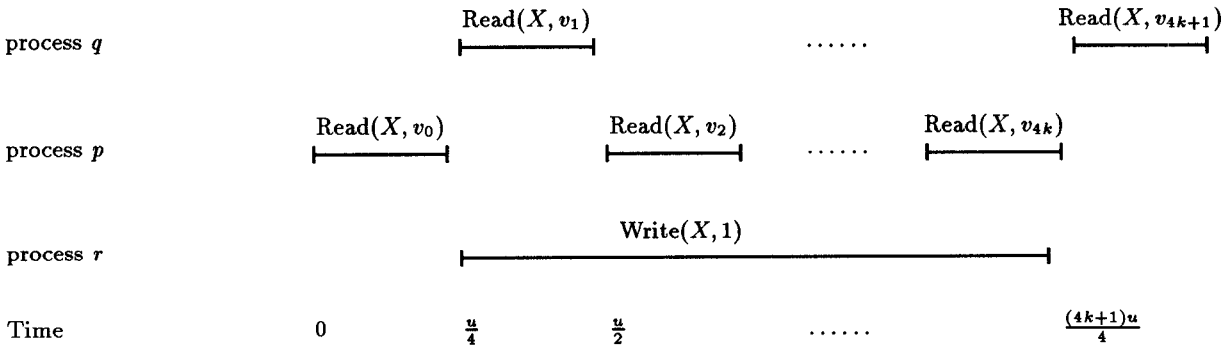
The assumptions about the number of readers and writers made in Theorems 4.1 and 4.2 are crucial to the results, since it can be shown that the algorithms from Theorems 3.2 and 3.3 are correct if there is only one reader and one writer.

## 4.2 Upper Bounds for Sequential Consistency

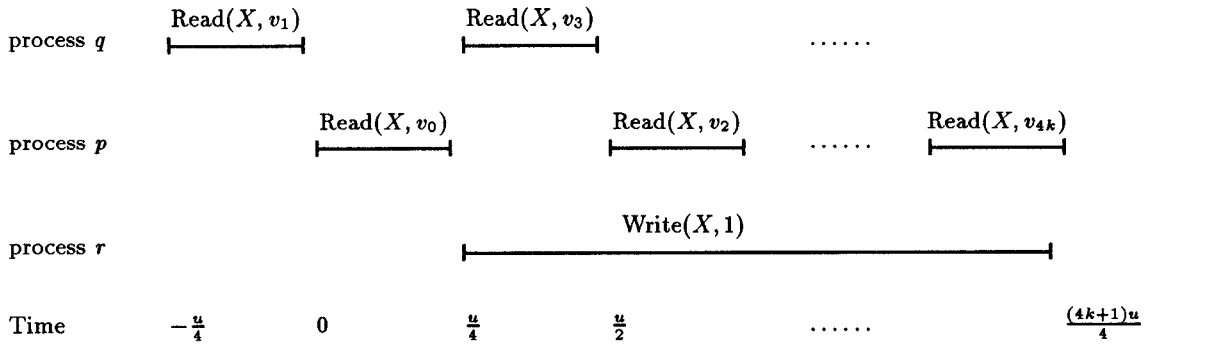
Inspecting the algorithm for fast reads (Theorem 3.2) reveals that the key point of its correctness is the fact that write updates are handled by all processes in the same order and at the same time. In order to guarantee sequential consistency, it suffices for processes to update their local copies in the same order (not necessarily at the same time). A simple way to achieve this property is for a centralized controller to collect update messages and broadcast them. This idea can be developed into two algorithms, one in which each read operation is performed instantaneously and the response time for write is $O(d)$, and another where the roles are reversed. We now present algorithms that are fully distributed and do not rely on a centralized controller. These algorithms use atomic broadcast to guarantee that all messages are delivered at the same order at all processes. Our algorithms are inspired by the atomic broadcast algorithm of Birman and Joseph [9].[5]

We start with an informal description of the algorithm for fast reads (time 0) and slow writes (times at most $6d$). Each process keeps a local copy of every object, a counter, and a set of updates that it is waiting to make to its local copies. A read returns the value of the local copy immediately. When a write comes in to $p$, $p$ requests "candidate" timestamps from all processes for this write. When a process $q$ receives a request for a candidate timestamp, it increments its counter and sets the timestamp to be the pair (counter, id). $q$ sends this timestamp to $p$ and also keeps a copy of the update marked as unready. Once $p$ receives candidate timestamps from everyone, it chooses the maximum as the final timestamp for that write and sends it to everyone.

[5] Birman and Joseph credit Skeen for the original idea, which is based on *two-phase commit*.




(a) The execution $\alpha$.




(b) The execution $\beta$.

Figure 2: Executions used in the proof of Theorem 4.1.

$p$ also sets a timer to go off $4d$ time later,[6] when it can be sure that every process has received the final timestamp and updated its local copy. When the timer goes off, $p$ Acks the write. When $q$ receives a final timestamp, it updates the timestamp for that write, marks it as ready, and sorts all the pending updates by timestamp. Then it does the updates in order of increasing timestamps until hitting an unready update. $q$ also updates its counter to be at least as large as the counter in the final timestamp just received.

The algorithm uses the following data types:

timestamp — (integer, id) (break ties with processor ids);

write — record with fields:

obj : name of an object (object to be written),
val : value of obj (value to be written),
uid : timestamp (unique id of this write request, assigned by initiator),
ts : timestamp (candidate or final),
ready : boolean (have final timestamp?),
cands : set of timestamp (candidate timestamps, only used by initiator).

The state of each process consists of the following variables:

count : integer, initially 0 (generates successive integers for creating timestamps);

updates : set of write, initially empty (set of updates waiting to be made to local copies);

pending-write : name of an object (write is pending on this object);

[6] The algorithm can be made completely asynchronous by replacing the timer with explicit acknowledgements; this will increase the time complexity of a write to $7d$.

```
Read_p(X):
  generate Return_p(X, v),
    where v is the value of p's copy of X

Write_p(X, v):
  count := count + 1
  pending-write := X
  add (X, v,(count,p),(count,p),false,{(count,p)})
    to updates
  send REQ-TS(X, v,(count,p)) to all processes
    (except self)

receive REQ-TS(X, v, u) from q:
  count := count + 1
  add (X, v, u,(count,p),false,∅) to updates
  send CAND-TS(X, v, u,(count,p)) to q

receive CAND-TS(X, v, u, T) from q:
  let E be the entry in updates with
    E.obj = X, E.val = v, and E.uid = u
  add T to E.cands
  if |E.cands| = n then
      send MAX-TS(X, v, u, max(E.cands))
        to all processes (including self)
      set timer for current time +4d
  endif

receive MAX-TS(X, v, u, (i, r)) from q:
  let E be the entry in updates with
    E.obj = X, E.val = v, and E.uid = u
  count := max(count,i)
  E.ts := (i, r)
  E.ready := true
  while E', element in updates with smallest ts,
    is ready do
      write E'.val to local copy of E'.obj
      remove E' from updates
  endwhile

Timer_p:
  generate Ack_p(pending-write)
```

Figure 3: The transition function.

copy of every object $X$, initially equal to its initial value.

Each process also knows $n$, the total number of processes, and $d$, the maximum message delay. The transition function of process $p$ appears in Fig. 3.

We first sketch the proof. To show sequential consistency, we must demonstrate, for any admissible execution $\sigma$, a sequential order for all operations in $\sigma$ such that the order at each process is preserved and each read returns the value of the latest write. The operations are ordered by first ordering all writes in final timestamp order, and then placing each read, say on object $X$ at process $p$, after the latest of (1) the previous operation for $p$, and (2) the write that generated the latest update to $p$'s copy of $X$ preceding the read's return. The resulting sequence respects the order at each process by construction and because of the way timestamps are assigned. Showing that the sequence satisfies the specification of read-write objects depends on two facts: (1) that updates are done at each process in final timestamp order, and (2) that if a read operation follows a write operation at any process $p$, then $p$ reads its local copy for the read after it updates its local copy for the write.

**Lemma 4.3** *Let $\sigma$ be any admissible execution of the algorithm. Then every write operation in $\sigma$ is given a unique final timestamp.*

**Lemma 4.4** *Let $\sigma$ be any admissible execution of the algorithm. Then the final timestamps assigned to write operations in $\sigma$ form a total order.*

**Lemma 4.5** *Let $\sigma$ be any admissible execution of the algorithm. Then for any process $p$, $p$'s local copies of the objects take on all the values contained in writes and the updates are done in timestamp order.*

**Proof:** The final timestamp order of the writes in $\sigma$ is uniquely defined, by Lemmas 4.3 and 4.4. Clearly every write is eventually assigned a final timestamp, which is at least as large as all its candidate timestamps.

First we show that the update associated with every write is made at every process. Consider the set of writes whose updates are not made at all processes; let $W$ be the write in this set with the smallest final timestamp and let $p$ be a process where $W$'s update is not made. Let $t$ be the time when $p$ receives $W$'s final timestamp. Since $p$ increments count to be at least as large as the count in $W$'s final timestamp, every write that is added to $p$'s updates set subsequently has a timestamp larger than $W$'s. Let $W'$ be any write in $p$'s

updates set at time $t$ whose timestamp is less than $W$'s. If $W'$ is not ready, then eventually it will be. If $W'$'s final timestamp is greater than $W$'s, then it cannot block $W$'s update at $p$. If $W'$'s final timestamp is less than $W$'s, then by the choice of $W$, its update is eventually done at $p$, after which it does not block $W$'s update at $p$. Thus eventually nothing prevents $W$'s update from being made at $p$.

Now we show that at each process $p$, updates are made in final timestamp order. Suppose in contradiction that the final timestamp of write $W_1$ is less than the final timestamp of write $W_2$, but $p$ performs $W_2$'s update before $W_1$'s. When $p$ performs $W_2$'s update, it cannot yet have an entry for $W_1$, because otherwise it would either block (if $W_1$ was not ready) or perform $W_1$'s update before $W_2$'s. But then $p$'s candidate timestamp for $W_1$ would be greater than $W_2$'s final timestamp, since $p$'s count is increased when MAX-TS is received, implying that $W_1$'s final timestamp is greater than $W_2$'s. ■

**Lemma 4.6** *Let $\sigma$ be any admissible execution of the algorithm and $p$ be any process. If $W_1$ precedes $W_2$ in $ops(\sigma)|p$[7], then the final timestamp of $W_1$ is less than the final timestamp of $W_2$.*

**Lemma 4.7** *Let $\sigma$ be any admissible execution of the algorithm and $p$ be any process. If read $R$ of object $Y$ follows write $W$ to object $X$ in $ops(\sigma)|p$, then $R$'s read of $p$'s local copy of $Y$ follows $W$'s write of $p$'s local copy of $X$.*

**Theorem 4.8** *This algorithm ensures sequential consistency with $|R| = 0$ and $|W| = 6d$.*

**Proof:** (Sketch) Clearly the time for any read is 0. The time for any write is the time for the REQ-TS messages to be received, the subsequent CAND-TS messages to be received, and the $4d$ timeout to expire, which is at most $6d$.

We now show sequential consistency. Fix some admissible execution $\sigma$. We define a legal sequence of operations $\tau$, such that for every process $p$, $ops(\sigma)|p = \tau|p$. In $\tau$, we order the writes in $\sigma$ by final timestamps. To insert the reads, we proceed in order from the beginning of $\sigma$. [$\text{Read}_p(X)$, $\text{Return}_p(X, v)$] goes immediately after the latest of (1) the previous operation for $p$ (either read or write, on any object), and (2) the write that spawned the latest update to $p$'s local copy of $X$ preceding the generation of the $\text{Return}_p(X, v)$. (Break ties using process ids.)

We must show $ops(\sigma)|p = \tau|p$ for all processes $p$. Two reads are ordered correctly by definition of $\tau$. Two writes are ordered correctly by Lemma 4.6. The interesting case is when a read $R$ precedes write $W$, in $ops(\sigma)|p$. Suppose in contradiction that $R$ comes after $W$ in $\tau$. Then in $\sigma$ there is some read $R' = [\text{Read}_p(X), \text{Return}_p(X, v)]$ and some write $W' = [\text{Write}_q(X, v), \text{Ack}_q(X)]$ such that (1) $R'$ occurs before $R$ in $\sigma$, (2) the final timestamp of $W'$ is greater than the final timestamp of $W$, and (3) $W'$ spawns the latest update to $p$'s copy of $X$ that precedes $R'$'s read. But $W'$ must have already received its final timestamp before $R'$'s read occurs, which means before $W$ starts. But then the timestamp of $W$ would be greater than the timestamp of $W'$, which is a contradiction.

To show $\tau$ is legal, first note that for every read $R$ of $X$ by $p$, the write $W$, whose update to $p$'s local copy of $X$ provides the value returned, follows $R$. Lemmas 4.5 and 4.7 and the definition of $\tau$ can be used to prove that no other write falls between $W$ and $R$ in $\tau$. ■

Theorem 4.1 implies that this algorithm does not guarantee linearizability. We can also explicitly construct an admissible execution that violates linearizability as follows. The initial value of $X$ is 0. Process $p$ writes 1 to $X$. The final timestamp for the write is sent at time $t$. It arrives at process $r$ at time $t$ and at process $q$ at time $t + d$. Meanwhile, $r$ performs a read at time $t$ and gets the new value 1, while $q$ performs a read at time $t + d/2$ and gets the old value 0. No permutation of these operations can both conform to the read/write specification and preserve the relative real-time orderings of all non-overlapping operations.

We now discuss the algorithm that ensures sequential consistency with fast writes (time 0) and slow reads (time at most $3d$). (Its detailed code and proof of correctness are omitted from this abstract.) This algorithm is similar to the previous algorithm. When a Read($X$) comes in to $p$, if $p$ has no updates (to any object, not just $X$) that it initiated waiting to be made, then it Returns the current value of its copy of $X$. Otherwise, it marks the waiting update (that it initiated) with the largest timestamp

[7] $ops(\sigma)|p$ is the restriction of $ops(\sigma)$ to the operations of $p$.

and Returns once this update is made. When a Write($X$) comes in to $p$, it is handled very similarly to the other algorithm; however, it is Acked immediately. Since a process $p$ may be handling several writes at a time, it is important that $q$ respond to timestamp requests from $p$ in the correct order.[8] Effectively, the algorithm pipelines write updates generated at the same process. We have:

**Theorem 4.9** *The algorithm just described implements sequential consistency with* $|R| = 3d$ *and* $|W| = 0$.

The structure of the proof is the same as for the previous algorithm, while making concession to the fact that the writes are acknowledged immediately and that reads are sometimes delayed.

Theorem 4.2 implies that this algorithm does not guarantee linearizability. It is also not difficult to construct an explicit scenario.

# 5 Conclusions and Further Research

The impact of the correctness guarantee on the efficiency of supporting it was studied under various timing assumptions. Although we still do not have a complete picture of this problem, our results indicate that supporting sequential consistency can be more cost-effective than supporting linearizability, for read/write objects and under certain timing assumptions. Two other conclusions can be drawn from our results: First, perfect clocks admit more efficient implementations, and thus it may be worthwhile to provide such clocks. Second, knowing in advance the sharing patterns of the object (i.e., how many processes read it and how many processes write it) results in faster implementations. Thus, the mcs can benefit from having the application program (the user) supply "hints" about the sharing patterns of the object.

Our work leaves open many interesting questions. Obviously, it is desirable to narrow the gaps between our upper and lower bounds. It will be interesting to understand how practical issues such as local memory size and clock drift influence the bounds. We have studied only read/write objects, although our definitions can be extended in a straightforward way to apply to other data objects. It will be very interesting to obtain bounds on the response time of implementing other objects, e.g., FIFO Queues and Test-and-Set registers, under sequential consistency and linearizability. Preliminary results in this direction appear in [5]. The cost measure we have chosen to analyze is response time, but there are other interesting measures, including throughput and network congestion.

The problem that we have studied is closely related to the problem of designing cache consistency schemes in which some sort of global ordering must be imposed on the operations ([10, 11, 12, 16, 20]). Our results show that making the definitions of these orderings more precise is important since seemingly minor differences in the definitions result in significant differences in the inherent efficiency of implementing them. Recently, several non-global conditions that are weaker than sequential consistency have been suggested, e.g., weak ordering ([15, 8, 1]), pipelined memory ([23]), slow memory ([19]), causal memory ([4]), loosely coherent memory ([6]), and the definitions in [12] and [28]. It would be interesting to investigate the inherent efficiency of supporting these consistency guarantees. In order to do so, crisp and precise definitions of these conditions are needed.

It is clear that efficiency, in general, and response time, in particular, are not the only criteria for evaluating consistency guarantees. In particular, the ease of designing, verifying, programming, and debugging algorithms using such shared memories is very important.

As multiprocessor systems become larger, distributed implementations of shared virtual memory are becoming more common. (Truly shared memories, or even buses, cannot be used in systems with a large number of processors.) Such implementations and their evaluation relate issues concerning multiprocessor architecture, programming language design, software engineering, and the theory of concurrent systems. We hope our work contributes toward a more solid ground for this interaction.

**Acknowledgements:** The authors thank Sarita Adve, Roy Friedman, Mark Hill, and Rick Zucker for helpful comments on an earlier version of this paper.

[8] For simplicity, we assume FIFO channels, but this assumption can be removed if sequence numbers are employed.

# References


[1] S. Adve and M. Hill, "Weak Ordering—A New Definition," *Proc. 17th ISCA*, 1990, pp. 2–14.

[2] S. Adve and M. Hill, "Implementing Sequential Consistency in Cache-Based Systems," *Proc. ICPP*, 1990.

[3] Y. Afek, G. Brown, and M. Merritt. "A Lazy Cache Algorithm," *Proc. 1st SPAA*, 1989, pp. 209–222.

[4] M. Ahamad, P. Hutto, and R. John, *Implementing and Programming Causal Distributed Shared Memory*, TR GIT-CC-90-49, Georgia Inst. of Tech., December 1990.

[5] H. Attiya, "Implementing FIFO Queues and Stacks," in preparation.

[6] J. Bennett, J. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proc. 2nd PPoPP*, 1990, pp. 168–176.

[7] P. Bernstein, V. Hadzilacos, and H. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, MA, 1987.

[8] R. Bisiani, A. Nowatzyk, and M. Ravishankar, "Coherent Shared Memory on a Distributed Memory Machine," *Proc. ICPP*, 1989, pp. I-133–141.

[9] K. Birman and T. Joseph, "Reliable Communication in the Presence of Failures," *TOCS*, vol. 5, no. 1, pp. 47–76.

[10] W. Brantley, K. McAuliffe, and J. Weiss, "RP3 Processor-Memory Element," *Proc. ICPP*, 1985, pp. 782–789.

[11] L. M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. on Computers*, vol. C-27, no. 12, pp. 1112–1118.

[12] W. W. Collier, "Architectures for Systems of Parallel Processes," IBM TR 00.3253, Poughkeepsie, NY, January 1984.

[13] E. W. Dijkstra, "Hierarchical Ordering Of Sequential Processes," *Acta Informatica*, 1971, pp. 115–138.

[14] M. Dubois and C. Scheurich, "Memory Access Dependencies in Shared-Memory Multiprocessors", *IEEE Trans. on Software Engineering*, vol. 16, no. 6 (June 1990), pp. 660–673.

[15] M. Dubois, C. Scheurich, and F. A. Briggs, "Memory Access Buffering in Multiprocessors," *Proc. 13th ISCA*, June 1986, pp. 434–442.

[16] M. Dubois, C. Scheurich, and F. A. Briggs, "Synchronization, Coherence and Event Ordering in Multiprocessors," *IEEE Computer*, vol. 21, no. 2, pp. 9–21.

[17] M. Herlihy, "Wait-Free Implementations of Concurrent Objects," *Proc. 7th PODC*, 1988, pp. 276–290.

[18] M. Herlihy and J. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *TOPLAS*, vol. 12, no. 3, pp. 463–492.

[19] P. Hutto and M. Ahamad, *Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories*, TR GIT-ICS-89/39, Georgia Inst. of Tech., October 1989.

[20] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Trans. on Computers*, vol. C-28, no. 9, pp. 690–691.

[21] L. Lamport, "On Interprocess Communication. Parts I and II," *Distributed Computing*, vol. 1, no. 2 (1986), pp. 77–101.

[22] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *TOCS*, vol. 7, no. 4, pp. 321–359.

[23] R. Lipton and J. Sandberg, *PRAM: A Scalable Shared Memory*, TR CS-TR-180-88, Princeton University, September 1988.

[24] J. Lundelius and N. Lynch, "An Upper and Lower Bound for Clock Synchronization," *Information and Control*, vol. 62, Nos. 2/3, pp. 190–204.

[25] S. Min and J. Baer, "A Timestamp-Based Cache Coherence Scheme," *Proc. ICPP*, 1989, pp. I-23–32.

[26] J. Misra, "Axioms for Memory Access in Asynchronous Hardware Systems," *TOPLAS*, vol. 8, no. 1, pp. 142–153.

[27] C. Papadimitriou, *The Theory of Concurrency Control*, Computer Science Press, Rockville, MD, 1986.

[28] U. Ramachandran, M. Ahamad, and M. Y. Khalidi, "Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer," *Proc. ICPP*, 1989, pp. II-160–169.

[29] C. Scheurich and M. Dubois, "Correct Memory Operation of Cache-Based Multiprocessors," *Proc. 14th ISCA*, 1987, pp. 234–243.