Sequential Consistency versus Linearizability

HAGIT ATTIYA
The Technion
and
JENNIFER L. WELCH
University of North Carolina at Chapel Hill

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. Three types of shared memory objects are considered: read/write objects, FIFO queues, and stacks. If clocks are only approximately synchronized (or do not exist), then for all three object types it is shown that linearizability is more expensive than sequential consistency: we present upper bounds for sequential consistency and larger lower bounds for linearizability. We show that, for all three data types, the worst-case response time is very sensitive to the assumptions that are made about the timing information available to the system. Under the strong assumption that processes have perfectly synchronized clocks, it is shown that sequential consistency and linearizability are equally costly: we present upper bounds for linearizability and matching lower bounds for sequential consistency. The upper bounds are shown by presenting algorithms that use atomic broadcast in a modular fashion. The lower bound proofs for the approximate case use the technique of "shifting", first introduced for studying the clock synchronization problem.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; *Shared memory*; C.1.2 [**Processor Architecture**]: Multiple Data Stream Architectures—*Parallel processors*; D.3.3 [**Programming Techniques**]: Language Constructs and Features—*Abstract data types*; *Concurrent programming structures*; D.4.2 [**Operating Systems**]: Storage Management—*Distributed Memories*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Parallelism and concurrency*

General terms: Algorithms, Design

Additional Key Words and Phrases: Cache coherence, sequential consistency, linearizability, shared memory

This paper combines and unifies results that appear in preliminary form in [9] and [6]. This paper was originally submitted to *ACM Transactions on Programming Languages and Systems*, where it was reviewed and accepted for publication. It was subsequently transferred to *ACM Transactions on Computer Systems* with the agreement of the authors and the two journals, to take advantage of the shorter delay to publication for *TOCS*.
Attiya's research was supported in part by B. and G. Greenberg Research Fund (Ottawa), by Technion V.P.R. funds, and by the fund for the promotion of research at the Technion. Welch's research was supported in part by NSF grant CCR-9010730, an IBM Faculty Development Award, and an NSF Presidential Young Investigator Award. Part of work by the first author was performed while visiting DEC Cambridge Research Laboratory and 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.
Authors' addresses: H. Attiya, Department of Computer Science, The Technion, Haifa 32000, Israel; J. Welch, Department of Computer Science, Texas A&M University, College Station, TX 77843-3112.

1. INTRODUCTION

1.1 Overview

We present a quantitative comparison of the inherent costs of two well-known consistency conditions for concurrently accessed shared data—sequential consistency and linearizability. Our main conclusion is that under realistic timing assumptions it is strictly more expensive to provide linearizability than to provide sequential consistency in distributed (message-passing) implementations, where the cost measure is the worst-case time for a data access. Because their definitions are very similar, linearizability and sequential consistency are often confused, but our work shows that there can be a considerable cost to such confusion. For example, Dubois and Scheurich ([42, 19]) define a sufficient condition for sequential consistency (cf. [19, Definition 6.3]); however, this condition implies linearizability.[1] Implementing linearizability imposes more cost than is necessary to support the target condition of sequential consistency. To our knowledge, this is the first time sequential consistency is shown to be more costly than linearizability.

We also study the worst-case access time for the two conditions under more stringent timing assumptions, namely that processors have perfectly synchronized clocks. We show several lower bounds for sequential consistency in this model; these lower bounds carry over to more realistic models in which weaker assumptions hold about the clock behavior. We also present matching "counter-example" algorithms for linearizability in this model. They demonstrate that no improved lower bounds for either condition are possible without weakening the timing assumptions. The fact that sequential consistency and linearizability are equally costly in this model (for our measure) is somewhat surprising. It indicates the importance of explicitly and carefully specifying system timing assumptions and the non-triviality of separating sequential consistency from linearizability.

1.2 Detailed Description

Managing concurrent accesses to shared data by several processes is a problem that arises in many contexts, ranging from cache coherence for multiprocessors to distributed file systems and transaction systems. A consistency condition must specify what guarantees are provided about the values returned by data accesses in the presence of interleaved and/or overlapping accesses perhaps to distinct copies of a single logical data item. Two conflicting goals of a consistency condition are to be strong enough to be useful to the user and weak enough to be implemented efficiently. Sequential consistency and linearizability are two well-known consistency conditions.

Sequential consistency requires that all the data operations *appear* to have executed atomically, in some sequential order that is consistent with the order seen at individual processes.[2] When this order must also preserve the global (external) ordering of non-overlapping operations, this consistency guarantee is called

[1] Technically, the definition of [19] relies on the notion of "performing an operation", which can only be interpreted in a specific architectural model. Under a natural interpretation (e.g., as in [25]) the definition implies linearizability.
[2] This condition is similar in flavor to the notion of *serializability* from database theory ([11, 40]); however, serializability applies to *transactions* which aggregate many operations.

linearizability ([28]).[3]

Clearly linearizability is stronger than sequential consistency. As discussed in [28], linearizability has two advantages over sequential consistency. First, it is somewhat more convenient to use, especially for formal verification, because it preserves real-time ordering of operations, and hence corresponds more naturally to the intuitive notion of atomic execution of operations. Second, linearizability is compositional (or local), that is, the combination of separate linearizable implementations of two objects yields a linearizable implementation.[4] In contrast, sequential consistency is not compositional, implying that all objects must be implemented together. Consequently, development costs and the amount of synchronization needed increase, and it is harder to apply separate optimizations to different objects.

Several papers have proposed sequentially consistent implementations of read/write objects, which were claimed to achieve a high degree of concurrency (e.g., [2, 3, 10, 14, 19, 38, 42]). None of these papers proves that similar improvements cannot be achieved for linearizability and none provides an analysis of the response time of the implementations suggested (or any other complexity measure).

In contrast, our work shows the existence of a gap between the the fastest implementation of linearizability and what can be achieved for sequential consistency. To our knowledge, this is the first such quantitative result comparing the two conditions. Our results are shown in relatively abstract formal models. We believe that the correct abstraction has been done so that the results are applicable in a wide variety of contexts.

Our system model consists of a collection of application processes running concurrently and communicating via virtual shared memory. The shared memory consists of a collection of *objects*. Unlike most previous research, we consider other types of shared objects in addition to the usual read/write objects. Since read/write objects do not provide an expressive and convenient abstraction for concurrent programming (cf. [27]), many multiprocessors now support more powerful concurrent objects, e.g., FIFO queues, stacks, Test&Set and Fetch&Add ([14, 26]). We study FIFO queues and stacks; our results easily extend to Test&Set and Fetch&Add.

The application processes are running in a distributed system consisting of a collection of nodes and a communication network. The network need not be fully connected physically, but it must be possible for every node to communicate with every other node. 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 can be modeled by nullifying the application process.) Fig. 1 illustrates a node, on which an application process and an MCS process are running. 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 responses must be consistent with the particular consistency condition that is being implemented. Thus the consistency condition is defined at the interface between the application process and the MCS.

[3] Also called *atomicity* ([27, 33, 39]) in the case of read/write objects.

[4] We use the term *implementation* in its usual meaning in the semantics literature, of satisfying a certain specification.

Figure 1 should go on this page.

Fig. 1. System Architecture

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. 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. Thus d is the worst-case delay in the network over all pairs of processes and u is the worst-case uncertainty. Our lower bounds are given in terms of d and u. Our upper bounds are given in terms of d (u happens not to enter in); however all our algorithms also work in asynchronous systems where d and u are unknown (or even unbounded), since these constants appear nowhere in the code.

We have chosen to focus on the distinction between performing a data operation locally at a process, based on its local state, and performing an operation that requires communication between processes before it can return to the user. We model this by assuming 0 time for local processing and $d > 0$ time for the worst-case communication cost in the system. This is a reasonable approximation, as in many systems the time required for communication far outweighs the time for local computation.

We start with the case when the process clocks are only approximately synchronized and there is uncertainty in the message delay.[5] Under this assumption, for all three object types, there are gaps between the upper bounds for sequentially consistent implementations and the lower bounds for linearizable implementations. We show that there are operations that can be done instantaneously (i.e., locally) in sequentially consistent implementations but that require $\Omega(u)$ time in linearizable implementations (note that u can be as large as d). In particular:

- For read/write objects:
  - for linearizability, the worst-case time for a read is at least $u/4$,
  - for linearizability, the worst-case time for a write is at least $u/2$,
  - for sequential consistency, there is an algorithm that guarantees time 0 for a read and time $2d$ for a write, and another that guarantees the reverse.
- For FIFO queues:
  - for linearizability, the worst-case time for an enqueue is at least $u/2$,
  - for sequential consistency, there is an algorithm that guarantees time 0 for an enqueue and time $2d$ for a dequeue.
- The situation for stacks is analogous to that for FIFO queues, with "pop" playing the role of "dequeue" and "push" the role of "enqueue".

Thus, under these timing assumptions, linearizability is more expensive to implement than sequential consistency, when there are significantly more operations of one type.

We then consider the stronger model when processes' clocks are perfectly synchronized. In this case we show several strong lower bounds for sequential consistency. We also give matching upper bounds for linearizability. In particular:

[5] If there is no uncertainty in the message delay, then clocks can be perfectly synchronized.

—For read/write objects:
  —For sequential consistency, the sum of the worst-case response times for a read operation and a write operation is at least d.
  —For linearizability, there is an algorithm in which a read operation is performed instantaneously (locally), while a write operation returns within time d; also there is an algorithm in which the roles are reversed.

  The above lower bound formalizes and strengthens a result of Lipton and Sandberg ([35]). The two matching algorithms show that the lower bound tradeoff is tight—it is possible to have the response time of *only one* of the operations depend on the network's latency.

—For FIFO queues:
  —for sequential consistency, the worst-case response time for a dequeue operation is at least d,
  —for linearizability, there is an algorithm in which an enqueue operation returns instantaneously, while a dequeue operation returns within time d.

—For stacks, as in the case of imperfect clocks, the results are analogous to those for FIFO queues, with "pop" playing the role of "dequeue" and "push" the role of "enqueue".

Thus we need to assume that clocks are imperfect in order to separate sequential consistency from linearizability, indicating that this separation is not as obvious as it may seem and depends on delicate timing assumptions.

Section 2 presents our definitions and reviews a technique called "shifting" used in our lower bounds. Section 3 covers the imperfect clock case. There is a subsection for each object type; each subsection consists of the lower bound(s) for linearizability followed by the upper bound(s) for sequential consistency. Section 4 considers the case of perfect clocks. Again there is one subsection for each of the three object types; each subsection consists of the lower bound(s) for sequential consistency followed by the upper bound(s) for linearizability. We conclude in Section 5 with a discussion of our results, describe work that followed the original version of this paper, and suggest avenues for further research.

2. PRELIMINARIES

2.1 Objects

Every shared object is assumed to have a *serial specification* (cf. [28]) 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. A sequence τ of operations for a collection of objects is *legal* if, for each object O, the restriction of τ to operations of O, denoted $\tau|O$, is in the serial specification of O.

In the case of a read/write object X, the ordered pair of events [$\text{Read}_p(X)$, $\text{Ret}_p(X, v)$] forms a Read operation for any process p and value v, and [$\text{Write}_p(X, v)$, $\text{Ack}_p(X)$] forms a Write operation. 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).[6]

[6] The specifications used in this paper are operational. It is possible to give algebraic (axiomatic) specifications (cf. [28]); operational specifications are used here for simplicity.

In the case of a FIFO queue Q, the ordered pair of events [$\mathrm{Deq}_p(Q)$, $\mathrm{Ret}_p(Q, v)$] forms a Deq operation for any process p and value v, and [$\mathrm{Enq}_p(Q, v)$, $\mathrm{Ack}_p(Q)$] forms an Enq operation. The set of operation sequences consists of all sequences that obey the usual FIFO queue semantics. That is, with a sequence of operations we associate a sequence of *queue states*, starting with an initial empty state and continuing with a state for each operation representing the state of the queue *after* the operation. Each enqueue operation adds an item to the end of the queue, and each dequeue operation removes an item from the head of the queue, or returns $\perp$ if the queue is empty.

The specification of a stack S is similar to the specification of a queue: [$\mathrm{Pop}_p(S)$, $\mathrm{Ret}_p(S, v)$] forms a Pop operation for any process p and value v, and [$\mathrm{Push}_p(S, v)$, $\mathrm{Ack}_p(S)$] forms a Push operation. The set of operation sequences consists of all sequences that obey the usual last-in-first-out stack semantics.

2.2 System Model

We assume a system consisting of a collection of nodes connected via a communication network. On each node there is an application process, a memory-consistency system (MCS) process, and a real-time clock readable by the MCS process at that node. Formally, a *clock* is a monotonically increasing function from $\Re$ (real time) to $\Re$ (clock time).[7] The clock cannot be modified by the process. Processes do not have access to the real time; each process obtains its only information about time from its clock.

Below we list and informally explain the *events* that can occur at the MCS process on node p. (The name p is also used for the MCS process on node p.)

Call events:	the application process on node p wants to access a shared object.
Response events:	the MCS process on node p is providing a response from a shared object to the application process on node p.
Message receive events:	$\mathrm{receive}(p, m, q)$ for all messages m and nodes q: the MCS process on node p receives message m from the MCS process on node q.
Message send events:	$\mathrm{send}(p, m, q)$ for all messages m and MCS processes q: the MCS process on node p sends message m to the MCS process on node q.
Timer set events:	$\mathrm{timerset}(p, T)$ for all clock times T: p sets a timer to go off when its clock reads T.
Timer events:	$\mathrm{timer}(p, T)$ for all clock times T: a timer that was set for time T on p's clock goes off.

The call, message-receive, and timer events are *interrupt events*.

An *MCS process* (or simply *process*) is an automaton with a (possibly infinite) set of states, including an initial state, and a transition function. Each interrupt event causes an application of the transition function. The transition function is a function from states, clock times, and interrupt events to states, sets of response events, sets of message-send events, and sets of timer-set events (for subsequent

[7] $\Re$ denotes the real numbers.

clock times). That is, the transition function takes as input the current state, clock time, and interrupt event (which is the receipt of a call from the application process, or the receipt of a message from another node, or a timer going off), and produces a new state, a set of response events for the application process, a set of messages to be sent, and a set of timers to be set for the future.

A *step* of p is a tuple (s, T, i, s', R, M, S), where s and s' are states, T is a clock time, i is an interrupt event, R is a set of response events, M is a set of message-send events, S is a set of timer-set events, and s', R, M, and S are the result of p's transition function acting on s, T, and i.

A *history* of a process p with clock C is a mapping from $\Re$ (real time) to finite sequences of steps such that

(1) for each real time t, there is only a finite number of times $t' < t$ such that the corresponding sequence of steps is nonempty (thus the concatenation of all the sequences in real-time order is a sequence);
(2) the old state in the first step is p's initial state;
(3) the old state of each subsequent step is the new state of the previous step;
(4) for each real time t, the clock time component of every step in the corresponding sequence is equal to $C(t)$; and
(5) for each real time t, in the corresponding sequence all non-timer events are ordered before any timer event and there is at most one timer event.

A *memory-consistency system (MCS)* is a set of processes P together with a set of clocks $\mathcal{C}$, one for each p in P. An *execution* of an MCS is a set of histories, one for each process p in P with clock C_p in $\mathcal{C}$, satisfying the following two conditions: (1) 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. (2) A timer is received by p at clock time T if and only if p has previously set a timer for T. (The network is not explicitly modeled, although the constraints on executions given below imply that the network reliably delivers all messages sent.)

Execution σ is *admissible* if the following conditions hold:

(1) For every p and q, every message in σ from p to q has its delay in the range $[d - u, d]$, for fixed nonnegative integers d and u, $u \leq d$. (This is a restriction on the network.)
(2) For every p, at most one call at p is pending at a time. (This is a restriction on the application process.)

Note that the last condition allows each application process to have at most one call outstanding at any time. This outlaws pipelining or prefetching.

2.3 Correctness Conditions

Given an execution σ, let $ops(\sigma)$ be the sequence of call and response events appearing in σ in real-time order, breaking ties for each real time t as follows. First order all response events for time t whose matching call events occur before time t, using process ids to break any remaining ties. Then order all operations whose call and response both occur at time t. Preserve the relative ordering of operations for each process and break any remaining ties with process ids. Finally, order all call events for time t whose matching response events occur after time t, using process ids to break any remaining ties.

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. Given a sequence s of operations and a process p, we denote by $s|p$ the restriction of s to operations of p.

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

Definition 2.2. (Linearizability) *An execution σ is* linearizable *if there exists a legal sequence τ of operations such that τ is a permutation of $ops(\sigma)$, for each process p, $ops(\sigma)|p$ is equal to $\tau|p$, 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 τ.*

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 linearizable.

We measure the efficiency of an implementation by the worst-case response time for any operation on any object in the set. Given a particular MCS, an object O implemented by it, and an operation P on O, we denote by $|P(O)|$ the maximum time taken by a P operation on O in any admissible execution. We denote by $|P|$ the maximum of $|P(O)|$ over all objects O implemented by the MCS.

2.4 Shifting

A basic technique we use in our lower bound proofs (in Sections 3.1.1 and 3.2.1) is *shifting*, originally introduced in [36] to prove lower bounds on the precision achieved by clock synchronization algorithms. Shifting is used to change the timing and the ordering of events in the system while preserving the local views of the processes.

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.

More precisely, the *view* of process p in history π of p with clock C is the concatenation of the sequences of steps in π, in real-time order. The real times of occurrence are not represented in the view. Two histories, one of process p with clock C and the other of process p with clock C', are *equivalent* if the view of p is the same in both histories. Two executions, execution σ of system $(P, \mathcal{C})$ and execution σ' of $(P, \mathcal{C}')$, are *equivalent* if for each process p, the component histories for p in σ and σ' are equivalent. Thus, the executions are indistinguishable to the processes. Only an outside observer who has access to the real time can tell them apart.

Given history π of process p with clock C, and real number s, a new history $\pi' = \mathit{shift}(\pi, s)$ is defined by $\pi'(t) = \pi(t + s)$ for all t. That is, all tuples are shifted earlier in π' by s if s is positive, and later by $|s|$ if s is negative. Given a clock C and real number s, a new clock $C' = \mathit{shift}(C, s)$ is defined by $C'(t) = C(t) + s$ for all t. That is, the clock is shifted forward by s if s is positive, and backward by $|s|$ if s is negative.

The following lemma observes that shifting a history of process p and p's clock by the same amount produces another history.

Lemma 2.1. *Let π be a history of process p with clock C, and let s be a real number. Then $\mathit{shift}(\pi, s)$ is a history of p with clock $\mathit{shift}(C, s)$.*

Given execution σ of system $(P, \mathcal{C})$, and real number s, a new execution $\sigma' = \mathit{shift}(\sigma, p, s)$ is defined by replacing π, p's history in σ, by $\mathit{shift}(\pi, s)$, and by retaining the same correspondence between sends and receives of messages. (Technically, the correspondence is redefined so that a pairing in σ that involves the event for p at time t, in σ' involves the event for p at time $t - s$.) All tuples for process p are shifted by s, but no others are altered. Given a set of clocks $\mathcal{C} = \{C_q : q \in P\}$, and real number s, a new set of clocks $\mathcal{C}' = \mathit{shift}(\mathcal{C}, p, s)$, is defined by replacing clock C_p by clock $\mathit{shift}(C_p, s)$. Process p's clock is shifted forward by s, but no other clocks are altered.

The following lemma observes that shifting one process' history and clock by the same amount in an execution results in another execution that is equivalent to the original.

Lemma 2.2. (**Lundelius and Lynch** [36]) *Let σ be an execution of system $(P, \mathcal{C})$, p a process, and s a real number. Let $\mathcal{C}' = \mathit{shift}(\mathcal{C}, p, s)$ and $\sigma' = \mathit{shift}(\sigma, p, s)$. Then σ' is an execution of $(P, \mathcal{C}')$, and σ' is equivalent to σ.*

The following lemma quantifies how message delays change when an execution is shifted. Notice that the result of shifting an admissible execution is not necessarily admissible.

Lemma 2.3. (**Lundelius and Lynch** [36]) *Let σ be an execution of system $(P, \mathcal{C})$, p a process, and s a real number. Let $\mathcal{C}' = \mathit{shift}(\mathcal{C}, p, s)$ and $\sigma' = \mathit{shift}(\sigma, p, s)$. Make the obvious correspondence between messages in σ and in σ'. Suppose x is the delay of message m from process q to process r in σ. Then the delay of m in σ' is x if $q \neq p$ and $r \neq p$, $x - s$ if $r = p$, and $x + s$ if $q = p$.*

3. IMPERFECT CLOCKS

We start by assuming 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 $u > 0$.

We show that in this model there is a gap between the upper bounds for sequential consistency and the lower bounds for linearizability, for all three object types. The lower bounds state that the worst-case time for a read is at least $u/4$ and the worst-case time for a write, an enqueue, or a push is at least $u/2$. Recall that u is the uncertainty in the message delay and can be as large as d. In contrast, we describe sequentially consistent algorithms that implement these operations instantaneously.

Intuitively, the algorithms are similar to a snoopy write-through cache protocol with a write-broadcast policy for bused systems. The main idea is that in order to guarantee sequential consistency, it suffices for processes to update their local copies in the same order. (This is provided immediately in bused systems.) A simple way to achieve this property is for a centralized controller to collect update messages and broadcast them. Using atomic broadcast it is possible to translate this idea into algorithms that are fully distributed and do not rely on a centralized controller. The algorithms do not rely on timing information and also work in an asynchronous system.

Atomic broadcast ([13]) is a communication primitive which guarantees that every message sent using the primitive is received at every process, all messages are delivered in the same order at all processes, and two messages sent by the same process are delivered in the same order they were sent. Our implementations are described in a modular way so that they will work with any atomic broadcast algorithm (e.g., [13, 16, 23]). The interface to the primitive consists of two operations, ABC-send(m) to broadcast a message m (possibly consisting of several fields) and ABC-receive(m) to receive a message m. In analyzing our implementations, we assume there is a known bound, h, on the time that the atomic broadcast primitive takes to deliver a message to all processes. Each of our implementations has one fast operation, which takes time 0, and one slow operation, which takes time h. In Appendix A we describe and prove correct a fast atomic broadcast algorithm with $h = 2d$. By using this algorithm in our implementations, we obtain implementations in which slow operations take time $2d = O(d)$.

3.1 Read/Write Objects

We show in Section 3.1.1 that in any linearizable implementation of a read/write object, the worst-case response time of *both* read and write operations must depend on u. We then present in Section 3.1.2 two sequentially consistent algorithms for read/write objects, 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.

3.1.1 *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, described in Section 2.4.

Theorem 3.1. *Assume X is a read/write object with at least two readers and a distinct writer. Then any linearizable implementation of X must have $|Read(X)| \geq \frac{u}{4}$.*

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 $|Read(X)| < \frac{u}{4}$. 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 R_2 precedes R_1 in the resulting execution. Since R_2 returns the new value 1 and R_1 returns the old value 0, this contradicts linearizability.

Figure 2 should go on this page.

Figure 2

Let $k = \lceil \frac{|Write(X)|}{u} \rceil$. By the specification of X, there is an admissible execution α, in which all message delays are $d - \frac{u}{2}$, consisting of the following operations (see Fig. 2(a))[8]:

—At time $\frac{u}{4}$, r does a $\mathrm{Write}_r(X, 1)$.

—Between times $\frac{u}{4}$ and $(4k+1) \cdot \frac{u}{4}$, r does an $\mathrm{Ack}_r(X)$. (By definition of k, $(4k+1) \cdot \frac{u}{4} \geq \frac{u}{4} + |Write(X)|$, and thus r's write operation is guaranteed to finish in this interval.)

—At time $2i \cdot \frac{u}{4}$, p does a $\mathrm{Read}_p(X)$, $0 \leq i \leq 2k$.

—Between times $2i \cdot \frac{u}{4}$ and $(2i+1) \cdot \frac{u}{4}$, p does a $\mathrm{Ret}_p(X, v_{2i})$, $0 \leq i \leq 2k$.

—At time $(2i+1) \cdot \frac{u}{4}$, q does a $\mathrm{Read}_q(X)$, $0 \leq i \leq 2k$.

—Between times $(2i+1) \cdot \frac{u}{4}$ and $(2i+2) \cdot \frac{u}{4}$, q does a $\mathrm{Ret}_q(X, v_{2i+1})$, $0 \leq i \leq 2k$.

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, ..., 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 $\beta = \mathrm{shift}(\alpha, q, \frac{u}{2})$; i.e., we shift q earlier by $\frac{u}{2}$. (See Fig. 2(b) The result is admissible, since by Lemma 2.3 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, ..., v_{j+1}, v_j, ... Thus in β we now have $v_{j+1} = 1$ being read before $v_j = 0$, which violates linearizability. □

Theorem 3.2. *If X is a read/write object with at least two writers and a distinct reader, then any linearizable implementation of X must have* $|Write(X)| \geq \frac{u}{2}$.

The proof uses techniques similar to the proof of Theorem 3.1. It constructs an execution in which, if write operations are too short, linearizability can be violated by appropriately shifting histories.

Proof. Let p and q be two processes that write X and r be a process that reads X. Assume in contradiction that there is an implementation with $|Write(X)| < \frac{u}{2}$. Without loss of generality, assume that the initial value of X is 0. By the specification of X, there is an admissible execution α such that

—$ops(\alpha)$ is $\mathrm{Write}_p(X, 1)$ $\mathrm{Ack}_p(X)$ $\mathrm{Write}_q(X, 2)$ $\mathrm{Ack}_q(X)$ $\mathrm{Read}_r(X)$ $\mathrm{Ret}_r(X, 2)$;

—$\mathrm{Write}_p(X, 1)$ occurs at time 0, $\mathrm{Write}_q(X, 2)$ occurs at time $\frac{u}{2}$, and $\mathrm{Read}_r(X)$ occurs at time u; and

[8]In the figures, time runs from left to right, and each line represents events at one process. Important time points are marked at the bottom.

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

Write_p(X, v):
  ABC-send(X, v)

ABC-receive(X, v) from q:
  set local copy of X to v
  if q = p then generate Ack_p(X) endif
```

Fig. 3. Sequentially consistent fast read algorithm.

—the message delays in α are d from p to q, $d-u$ from q to p, and $d-\frac{u}{2}$ for all other ordered pairs of processes.

Let $\beta = \mathit{shift}(\mathit{shift}(\alpha, p, -\frac{u}{2}), q, \frac{u}{2})$; i.e., we shift p later by $\frac{u}{2}$ and q earlier by $\frac{u}{2}$. The result is still an admissible execution, since by Lemma 2.3 the delay of a message from p or to q becomes $d-u$, the delay of a message from q or to p becomes d, and all other delays are unchanged.

But $ops(\beta)$ is

$$\mathrm{Write}_q(X,2)\ \mathrm{Ack}_q(X)\ \mathrm{Write}_p(X,1)\ \mathrm{Ack}_p(X)\ \mathrm{Read}_r(X)\ \mathrm{Ret}_r(X,2)$$

which violates linearizability, because r's read should return 1, not 2. □

The assumptions about the number of readers and writers made in Theorems 3.1 and 3.2 are crucial to the results, since it can be shown that the algorithms from Theorems 4.2 and 4.3 are correct if there is only one reader and one writer.

3.1.2 *Upper Bounds for Sequential Consistency*

Fast Reads. We start with the algorithm for fast reads (time 0) and slow writes (time at most h).

In the algorithm, each process keeps a local copy of every object. A read returns the value of the local copy immediately. When a write comes in to p, p sends an atomic broadcast containing the name of the object to be updated and the value to be written; but it does not yet generate an Ack for the write operation. When an update message is delivered to a process q, q writes the new value to its local copy of the object. If the update message was originated by q, then q generates an Ack and the (unique pending) write operation returns.

More precisely, the state of each process consists of a copy of every object, initially equal to its initial value. The transition function of process p appears in Fig. 3.

To prove the correctness of the algorithm, we first show:

Lemma 3.3. *For every admissible execution and every process p, p's local copies take on all the values contained in write operations, all updates occur in the same order at each process, and this order preserves the order of write operations on a per-process basis.*

Proof. By the code, an ABC-send is done exactly once for each write operation. By the guarantees of the atomic broadcast, each process receives exactly one message for each write operation, these messages are received in the same order at each process, and this order respects the order of sending on a per-process basis. □

Call the total order of Lemma 3.3 the "Abcast order".

Lemma 3.4. *For every admissible execution, every process p, and all objects X and Y, 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.*

Proof. The lemma is true because W does not end until its update is performed at its initiator. □

Theorem 3.5. *There exists a sequentially consistent implementation of read/write objects with $|Read| = 0$ and $|Write| = h$.*

Proof. Consider the algorithm just presented. Clearly the time for any read is 0. The time for any write is the time for the initiator's ABC-send to be received by the initiator, which is at most h. The remainder of the proof is devoted to showing sequential consistency. Fix admissible execution σ.

Define the sequence of operations τ as follows. Order the writes in σ in Abcast order. Now we explain where to insert the reads. We proceed in order from the beginning of σ. $[\mathrm{Read}_p(X), \mathrm{Ret}_p(X, v)]$ 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 of p's local copy of X preceding the generation of the $\mathrm{Ret}_p(X, v)$. (Break ties using process ids; e.g., if every process reads some object before any process writes any object, then τ begins with p_1's read, followed by p_2's read, etc.)

We must show $ops(\sigma)|p = \tau|p$ for all processes p. Fix some process p. The relative ordering of two reads in $ops(\sigma)|p$ is the same in $\tau|p$ by definition of τ. The relative ordering of two writes in $ops(\sigma)|p$ is the same in $\tau|p$ by Lemma 3.3. Suppose in $ops(\sigma)|p$ that read R follows write W. By definition of τ, R comes after W in τ.

Suppose in $ops(\sigma)|p$ that read R precedes write W. Suppose in contradiction that R comes after W in τ. Then in σ there is some read $R' = [\mathrm{Read}_p(X), \mathrm{Ret}_p(X, v)]$ and some write $W' = [\mathrm{Write}_q(X, v), \mathrm{Ack}_q(X)]$ such that (1) R' equals R or occurs before R in σ, (2) W' equals W or follows W in the Abcast order, and (3) W' spawns the latest update to p's copy of X that precedes R''s read. But in σ, R' finishes before W starts. Since updates are performed in σ in Abcast order (Lemma 3.3), R' cannot see W''s update, a contradiction.

We must show τ is legal. Consider read $R = [\mathrm{Read}_p(X), \mathrm{Ret}_p(X, v)]$ in τ. Let W be the write in σ that spawns the latest update to p's copy of X preceding R's read of p's copy of X. Clearly $W = [\mathrm{Write}_q(X, v), \mathrm{Ack}_q(X)]$ for some q. (If there is no such W, then consider an imaginary write at the beginning of σ.) By the definition of τ, R follows W in τ. We must show that no other write to X falls in between W and R in τ. Suppose in contradiction that $W' = [\mathrm{Write}_r(X, w), \mathrm{Ack}_r(X)]$ does. Then by Lemma 3.3, the update for W' follows the update for W at every process in σ.

Case 1: $r = p$. Since τ preserves the order of operations at p, W' precedes R in σ. Since the update for W' follows the update for W in σ, R sees W''s update, not W's, contradicting the choice of W.

Case 2: $r \neq p$. By definition of τ, there is some operation in $ops(\sigma)|p$ that, in τ, precedes R and follows W' (otherwise R would not follow W'). Let O be the first such operation.

Suppose O is a write to some object Y. By Lemma 3.4, O's update to p's copy of Y precedes R's read of p's copy of X. Since updates are done in Abcast order, the update for W' occurs at p before the update for O, and thus before R's read, contradicting the choice of W.

Suppose O is a read. By the definition of τ, O is a read of X, and W''s update to p's copy of X is the latest one preceding O's read (otherwise O would not follow W'). Since updates are done in Abcast order, the value from W' supersedes the value from W, contradicting the choice of W. $\square$

Theorem 3.1 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 ABC-send for the write occurs at time t. It arrives at process r at time t and at process q at time $t + h$. Meanwhile, r performs a read at time t and gets the new value 1, while q performs a read at time $t + h/2$ 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.

Fast Writes. We now discuss the algorithm that ensures sequential consistency with fast writes (time 0) and slow reads (time at most h). When a Read(X) comes in to p, if p has no pending updates (to any object, not just X) that it initiated, then it Returns the current value of its copy of X. Otherwise, it waits for all pending writes to complete and then returns. This is done by maintaining a count of the pending writes and waiting for it to be zero. When a Write(X) comes in to p, it is handled very similarly to the other algorithm; however, it is Acked immediately. Effectively, the algorithm pipelines write updates generated at the same process. Specifically, the state of each process consists of the following variables:

—num : integer, initially 0 (number of pending updates initiated by this process),
—copy of every object, initially equal to its initial value.

The transition function of process p appears in Fig. 4.

Theorem 3.6. *There exists a sequentially consistent implementation of read/write objects with* $|Read| = h$ *and* $|Write| = 0$.

Proof. Consider the algorithm just presented. Clearly every write takes 0 time. The worst-case time for a read occurs if the return must wait for the initiator to receive its own ABC-send for a pending write. This takes at most h time. The structure of the proof of sequential consistency is identical to that in the proof of Theorem 3.5. We just need a new proof for Lemma 3.4. Lemma 3.4 is still true for this algorithm because when a Read occurs at p, if any update initiated by p is still waiting, then the Return is delayed until the latest such update is performed. $\square$

Theorem 3.2 implies that this algorithm does not guarantee linearizability. An explicit scenario is easy to construct as well.

3.2 FIFO Queues

We show in Section 3.2.1 that in any linearizable implementation of a FIFO queue, the worst-case response time of an enqueue operation must depend on u. We then present in Section 3.2.2 a sequentially consistent implementation in which enqueue operations return instantaneously while the worst-case response time for a dequeue operation is h.

```
Read_p(X):
  if num = 0 then
      generate Ret_p(X, v), where v is the value of p's copy of X
  endif

Write_p(X, v):
  num := num + 1
  ABC-send(X, v)
  generate Ack_p(X)

ABC-receive(X, v, i) from q:
  set local copy of X to v
  if p = q then
      num := num − 1
      if num = 0 then
          generate Ret_p(X, v), where v is the value of p's copy of X
      endif
  endif
```

Fig. 4. Sequentially consistent fast write algorithm.

3.2.1 *Lower Bound for Linearizability.* We show that in any linearizable implementation of a FIFO queue the worst-case time for an enqueue is $u/2$ (assuming that at least two processes can enqueue to the same FIFO queue). The proof uses the technique of shifting, described in Section 2.4.

Theorem 3.7. *If Q is a FIFO queue with at least two enqueuers and a distinct dequeuer, then any linearizable implementation of Q must have $|Enq(Q)| \geq \frac{u}{2}$.*

Proof. Let p and q be two processes that can enqueue to Q and r be a process that dequeues from Q. Assume in contradiction that there is an implementation with $|Enq(Q)| < \frac{u}{2}$. Initially, Q is empty. By the specification of Q, there is an admissible execution α such that

—$ops(\alpha)$ is $\mathrm{Enq}_p(Q,1)$ $\mathrm{Ack}_p(Q)$ $\mathrm{Enq}_q(Q,2)$ $\mathrm{Ack}_q(Q)$ $\mathrm{Deq}_r(Q)$ $\mathrm{Ret}_r(Q,1)$;

—$\mathrm{Enq}_p(Q,1)$ occurs at time 0, $\mathrm{Enq}_q(Q,2)$ occurs at time $\frac{u}{2}$, and $\mathrm{Deq}_r(Q)$ occurs at time u; and

—the message delays in α are d from p to q, $d-u$ from q to p, and $d-\frac{u}{2}$ for all other ordered pairs of processes.

Let $\beta = shift(shift(\alpha, p, -\frac{u}{2}), q, \frac{u}{2})$; i.e., we shift p later by $\frac{u}{2}$ and q earlier by $\frac{u}{2}$. The result is still an admissible execution, since by Lemma 2.3 the delay of a message from p or to q becomes $d-u$, the delay of a message from q or to p becomes d, and all other delays are unchanged. But $ops(\beta)$ is

$$\mathrm{Enq}_q(Q,2)\ \mathrm{Ack}_q(Q)\ \mathrm{Enq}_p(Q,1)\ \mathrm{Ack}_p(Q)\ \mathrm{Deq}_r(Q)\ \mathrm{Ret}_r(Q,1)$$

which violates linearizability, because r's dequeue should return 2, not 1 (by the FIFO property). □

The assumption about the number of enqueuers made in Theorem 3.7 is crucial to the results, since it can be shown that the algorithm of Theorem 4.6 is correct if there is only one enqueuer.

```
Enq_p(Q, v):
  ABC-send(Q, v, "enq")
  generate Ack_p(Q)

Deq_p(Q):
  ABC-send(Q, "deq")

ABC-receive(Q, v, "enq") from q:
  enqueue v on local copy of Q

ABC-receive(Q, "deq") from q:
  val := dequeue local copy of Q
  if p = q then generate Ret_p(Q, val) endif
```

Fig. 5. Sequentially consistent fast enqueue algorithm.

3.2.2 *Upper Bound for Sequential Consistency.* Informally, the algorithm works as follow. Each process keeps a local copy of every object. When a request to enqueue v to Q comes in to p, p broadcasts an update message with the object name, the operation name, and the value to be enqueued to all processes. The operation returns immediately. When a request to dequeue from Q comes in to p, p broadcasts an update message with the object name and the operation name. It does not generate a response.

When an update message (either "deq" or "enq") is delivered to a process it handles it by performing the appropriate change (enqueue or dequeue) to the local copy of the object. If the update is a dequeue by the same process, the dequeue operation that is currently waiting returns the value that was dequeued from the local copy. (Note that by well-formedness, there is only one pending dequeue operation for a given process.)
In more detail, the state of each process consists of the following variables:

—copy of every object, initially equal to its initial value

—val : value (of a queue element)

The transition function of process p appears in Fig. 5.

To prove correctness of the algorithm we show:

Lemma 3.8. *In every admissible execution, all updates are done exactly once at each local copy, updates are done in the same order at each process, and this order preserves the per-process order.*

Theorem 3.9. *There exists a sequentially consistent implementation of FIFO queues with* $|Enq| = 0$ *and* $|Deq| = h$.

Proof. Consider the algorithm just presented. Clearly, the time for an enqueue is 0 and the time for a dequeue is at most h. The remainder of the proof is devoted to showing sequential consistency. Fix some admissible execution σ.

Define the sequence of operations τ as follows: Order the operations in σ by Abcast order. From Lemma 3.8 it follows that operations by p are ordered in τ as they were ordered in σ, and thus $ops(\sigma)|p = \tau|p$, for all processes p. It remains to show that τ is legal, i.e., that for every FIFO queue Q, $\tau|Q$ is in the serial specification of Q. Pick any Q and consider $\tau|Q = op_1 op_2 \ldots$. Suppose op_i is $[\mathrm{Deq}_p(Q), \mathrm{Ret}_p(Q, v)]$. Since the local updates at p occur in Abcast order (Lemma 3.8), updates at p to the local copy of Q occur in the same order as in τ, and the claim follows. $\square$

Theorem 3.7 implies that this algorithm does not guarantee linearizability. It is also possible to construct an explicit scenario which violates linearizability.

3.3 Stacks

These results are analogous to those for FIFO queues with Pop in place of Deq and Push in place of Enq.

Theorem 3.10. *If S is a stack with at least two pushers and a distinct popper, then for any linearizable implementation of S, $|Push(S)| \geq \frac{u}{2}$.*

Theorem 3.11. *There exists a sequentially consistent implementation of stacks with $|Push| = 0$ and $|Pop| = h$.*

4. PERFECT CLOCKS

In the previous section, we have shown a gap between the cost of implementing sequential consistency and the cost of implementing linearizability. This separation hinged on the assumption that clocks are not initially synchronized. In this section, we show that this assumption is necessary by considering the case in which processes have perfectly synchronized (*perfect*) clocks and message delay is constant and known.[9] Another contribution of these results is that our lower bounds for this model also hold *a fortiori* in more realistic models. Perfect clocks are modeled by letting $C_p(t) = t$ for all p and t. The constant message delay is modeled by letting $u = 0$; d is known and can be used by the MCS.

For each of the three object types, we first prove lower bounds on the worst-case response time for sequentially consistent implementations. Since sequential consistency is a weaker condition than linearizability, these bounds also hold for linearizable implementations. Then we present algorithms that achieve linearizability, and hence sequential consistency, with worst-case response times matching the lower bounds. Section 4.1 considers read/write objects, Section 4.2 considers FIFO queues, and Section 4.3 considers stacks.

4.1 Read/Write Objects

We show in Section 4.1.1 that for sequential consistency the sum of the worst-case response times of read and write operations is at least d, even in this strong model. This is a formalization of a result of Lipton and Sandberg ([35, Theorem 1]), making precise the timing assumptions made on the system. We then show in Section 4.1.2

[9] The assumptions that processes have perfect clocks and that message delays are constant (and known) are equivalent. If one assumes that clocks are not necessarily synchronized perfectly (but run at the rate of real time) and that the message delay is constant and known, then a simple algorithm suffices to synchronize the clocks perfectly. If one assumes that clocks are perfectly synchronized and that there is a known upper bound d on message delays, then constant message delays can be easily simulated by timestamping each message with the clock time of the sender and having each recipient delay any message that arrives with delay smaller than d until the delay is exactly d.

that the lower bound is tight for this model by describing two linearizable algorithms that match the lower bound exactly: In the first algorithm, reads are performed instantaneously, while the worst-case response time for a write is d. In the second algorithm, writes are performed instantaneously, while the worst-case response time for a read is d.

4.1.1 *Lower Bounds for Sequential Consistency*

THEOREM 4.1. (LIPTON AND SANDBERG [35]) *For any memory-consistency system that is a sequentially consistent implementation of two read/write objects X and Y, $|Write| + |Read| \geq d$.*

PROOF. Let p and q be two processes that access X and Y. Assume by way of contradiction that there exists a sequentially consistent implementation of X and Y for which both $|Write(X)| + |Read(Y)| < d$ and $|Write(Y)| + |Read(X)| < d$. 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 α_1 such that $ops(\alpha_1)$ is

$$\mathrm{Write}_p(X, 1)\ \mathrm{Ack}_p(X)\ \mathrm{Read}_p(Y)\ \mathrm{Ret}_p(Y, 0)$$

and $\mathrm{Write}_p(X, 1)$ occurs at real time 0 and $\mathrm{Read}_p(Y)$ occurs immediately after $\mathrm{Ack}_p(X)$. By assumption, the real time at the end of α_1 is less than d. Thus no message is received at any node during α_1.

By the specification of X, there is some admissible execution α_2 such that $ops(\alpha_2)$ is

$$\mathrm{Write}_q(Y, 1)\ \mathrm{Ack}_q(Y)\ \mathrm{Read}_q(X)\ \mathrm{Ret}_q(X, 0)$$

and $\mathrm{Write}_q(Y, 1)$ occurs at real time 0 and $\mathrm{Read}_q(X)$ occurs immediately after $\mathrm{Ack}_q(Y)$. By assumption, the real time at the end of α_2 is less than d. Thus no message is received at any node during α_2.

Since no message is ever received in α_1 and α_2, the execution α obtained from α_1 by replacing q's history with q's history in α_2 is admissible. Then $ops(\alpha)$ consists of the operations [$\mathrm{Write}_p(X, 1)$, $\mathrm{Ack}_p(X)$] followed by [$\mathrm{Read}_p(Y)$, $\mathrm{Ret}_p(Y, 0)$], and [$\mathrm{Write}_q(Y, 1)$, $\mathrm{Ack}_q(Y)$] followed by [$\mathrm{Read}_q(X)$, $\mathrm{Ret}_q(X, 0)$].

By assumption, α is sequentially consistent. Thus there is a legal operation sequence τ consisting of the operations [$\mathrm{Write}_p(X, 1)$, $\mathrm{Ack}_p(X)$] followed by [$\mathrm{Read}_p(Y)$, $\mathrm{Ret}_p(Y, 0)$], and [$\mathrm{Write}_q(Y, 1)$, $\mathrm{Ack}_q(Y)$] followed by [$\mathrm{Read}_q(X)$, $\mathrm{Ret}_q(X, 0)$]. Since τ 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 τ being legal. □

4.1.2 *Upper Bounds for Linearizability.* In this section we show that the tradeoff suggested by Theorem 4.1 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.

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 $\mathrm{Read}_p(X)$ occurs, p reads the value v of X in its local memory and immediately does a $\mathrm{Ret}_p(X, v)$. When a

$\mathrm{Write}_p(X, v)$ occurs, p sends "write(X, v)" messages to all other processes. Then p waits d time, after which it changes the value of X to v in its local memory and does an $\mathrm{Ack}_p(X)$. 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.)

Theorem 4.2. *There exists a linearizable implementation of read/write objects with $|Read| = 0$ and $|Write| = d$.*

Proof. Consider the algorithm just described. Clearly the time for every read is 0 and the time for every write is d.

Let σ be an admissible execution of this algorithm. For each operation in σ, say that it *occurs* at the real time when its response happens. Let τ be the sequence of operations in σ ordered by time of occurrence, breaking ties with process ids. Clearly $\sigma|p$ is equal to $\tau|p$ for all p, and the order of non-overlapping operations is preserved.

It remains to show that τ is legal, i.e., that for every object X, $\tau|X$ is in the serial specification of X. Since X is a read/write object, we must show that every Read Returns the value written by the latest preceding Write (and if there is no such Write, then it returns the initial value).

Pick any X and consider $\tau|X = op_1 op_2 \ldots$. Suppose op_i is $[\mathrm{Read}_p(X), \mathrm{Ret}_p(X, v)]$ and op_i occurs at time t in σ.

Case 1: No Write precedes op_i in τ. By the definition of τ, no Write is Acked before op_i starts. Since the Ack for a Write happens at the same time that every process updates its local copy of X, the Read reads the initial value for X and Returns that value.

Case 2: Some $\mathrm{Write}_p(X, v)$ is the latest Write preceding op_i in τ. By the definition of τ, this Write is Acked before op_i starts, but no other Write is Acked before op_i starts. Since the Ack for a Write happens at the same time that every process updates its local copy of X, the Read reads v for the value of X and Returns that value. $\square$

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 $\mathrm{Read}_p(X)$ occurs, p waits d time, after which it reads the value v of X in its local memory and immediately does a $\mathrm{Ret}_p(X, v)$. When a $\mathrm{Write}_p(X, v)$ occurs, p sends "write(X, v)" messages to all other processes (including a dummy message to itself which is delayed d time) 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.

Theorem 4.3. *There exists a linearizable implementation of read/write objects with $|Read| = d$ and $|Write| = 0$.*

Proof. Consider the algorithm just described. Clearly the time for every read is d and the time for every write is 0.

Let σ be an admissible execution of this algorithm. For each operation in σ, say that it *occurs* at the real time when its call happens. Let τ be the sequence of operations in σ ordered by time of occurrence, breaking ties with process ids.

Clearly $\sigma|p$ is equal to $\tau|p$ for all p, and the order of non-overlapping operations is preserved.

It remains to show that τ is legal, i.e., that for every object X, $\tau|X$ is in the serial specification of X. Since X is a read-write object, we must show that every Read Returns the value written by the latest preceding Write (and if there is no such Write, then it returns the initial value).

Pick any X and consider $\tau|X = op_1 op_2 \ldots$. Suppose op_i is $[\text{Read}_p(X), \text{Ret}_p(X, v)]$ and op_i occurs at time t in σ.

Case 1: No Write precedes op_i in t. By the definition of τ, no Write starts before op_i starts. Since the local changes occur d time after the Write starts and the Read reads the local memory d time after the Read starts, it reads the local memory before any change is made to it. Thus the Read returns the initial value.

Case 2: Some $\text{Write}_p(X, v)$ is the latest Write preceding op_i in τ. Essentially the same argument as in Case 1 works. □

4.2 FIFO Queues

We show in Section 4.2.1 that for sequential consistency the worst-case response time of a dequeue operation is at least d, even when clocks are perfectly synchronized and message delays are constant. We then show in Section 4.2.2 that this lower bound is tight for this model by describing a linearizable algorithm that matches the lower bound exactly: enqueues are performed instantaneously, while dequeues take time d.

4.2.1 *Lower Bound for Sequential Consistency*

Theorem 4.4. *For any sequentially consistent implementation of a FIFO queue Q, $|Deq(Q)| \geq d$.*

Proof. Let p and q be two processes that access Q. Assume by way of contradiction that there exists a sequentially consistent implementation of Q for which $|Deq(Q)| < d$. Let $T = |Deq(Q)|$. By definition, the queue Q is initially empty.[10] By the specification of Q, there is some admissible execution α'_1 such that $ops(\alpha'_1)$ is

$$\text{Enq}_q(Q,1)\ \text{Ack}_q(Q)\ \text{Deq}_p(Q)\ \text{Ret}_p(Q,v_1)\ \ldots \text{Deq}_p(Q)\ \text{Ret}_p(Q,v_i)\ \ldots$$

$\text{Enq}_q(Q,1)$ occurs at real time 0 and $\text{Ack}_p(Q)$ occurs at time t; the first $\text{Deq}_p(Q)$ occurs at time t, while the jth $\text{Deq}_p(Q)$ occurs at time $t+(j-1)T$ (see Figure 6(a)). Consider now the infinite sequence $v_1, \ldots, v_i, \ldots$. It is possible that many of them are $\perp$; however, since only a finite number of Deq operations can be serialized before the Enq operation, we have:

Lemma 4.5. *There exists some i such that $v_i \neq \perp$.*

Fix this particular i, and note that $v_i = 1$ and, for all j, $1 \leq j < i$, $v_j = \perp$. Let α_1 be α'_1 truncated after the ith Deq operation by p. More precisely, $ops(\alpha_1)$ is

$$\text{Enq}_q(Q,1)\ \text{Ack}_q(Q)\ \text{Deq}_p(Q)\ \text{Ret}_p(Q,\perp)\ \ldots \text{Deq}_p(Q)\ \text{Ret}_p(Q,\perp)\ \text{Deq}_p(Q)\ \text{Ret}_p(Q,1)$$

[10] If we allow queues to be initially non-empty, the proof of the lower bound becomes much simpler; we leave the details to the interested reader.

Figure 6 should go on this page.

Figure 6

$\mathrm{Enq}_q(Q,1)$ occurs at real time 0 and $\mathrm{Ack}_q(Q)$ occurs at time t; the first $\mathrm{Deq}_p(Q)$ occurs at time t, while the ith $\mathrm{Deq}_p(Q)$ occurs at time $t+(i-1)T$ (see Figure 6(b)). It is clear that the v_j's are exactly as in α'_1. By assumption, the real time at the end of α is less than $t+(i-1)T+d$. Thus, no message sent after $t+(i-1)T$ is received during α_1.

We now consider the execution where the ith (and last) dequeue by p is replaced with a dequeue by q. More precisely, by the specification of Q, there is some admissible execution α_2 such that $ops(\alpha_2)$ is

$$\mathrm{Enq}_q(Q,1)\ \mathrm{Ack}_q(Q)\ \mathrm{Deq}_p(Q)\ \mathrm{Ret}_p(Q,\perp)\ldots\mathrm{Deq}_p(Q)\ \mathrm{Ret}_p(Q,\perp)\ \mathrm{Deq}_q(Q)\\ \mathrm{Ret}_q(Q,u)$$

$\mathrm{Enq}_q(Q,1)$ occurs at real time 0 and $\mathrm{Ack}_q(Q)$ occurs at time t; the first $\mathrm{Deq}_p(Q)$ occurs at time t, while the $(i-1)$st $\mathrm{Deq}_p(Q)$ occurs at time $t+(i-2)T$, and Deq_q occurs at time $t+(i-1)T$ (see Figure 6(c)). Since α_2 is sequentially consistent, it follows that $u=1$. By assumption, the real time at the end of α_2 is less than $t+(i-1)T+d$. Thus, no message sent after $t+(i-1)T$ is received during α_2.

Consider now an execution α obtained from α_1 by replacing q's history with q's history in α_2. No message sent after time $t+(i-1)T$ is ever received in α_1 or α_2, and α_1 and α_2 are identical until time $t+(i-1)T$. This implies that α is admissible. Then $ops(\alpha)$ is

$$\mathrm{Enq}_q(Q,1)\ \mathrm{Ack}_q(Q)\ \mathrm{Deq}_p(Q)\ \mathrm{Ret}_p(Q,\perp)\ldots\mathrm{Deq}_p(Q)\ \mathrm{Ret}_p(Q,1)\ \mathrm{Deq}_q(Q)\\ \mathrm{Ret}_q(Q,1)$$

(see Figure 6(d)). By assumption, α is sequentially consistent. Thus, there is a legal sequence τ, which is a permutation of the above operations. However, in τ the element "1" is enqueued once but dequeued twice, a contradiction. □

4.2.2 *Upper Bound for Linearizability.* In this section we show that the lower bound given in Theorem 4.4 is tight for the model with perfect clocks. Specifically, we present an algorithm in which an enqueue operation returns instantaneously, while a dequeue operation returns within time d. The algorithm ensures the stronger condition of linearizability.

The algorithm works as follows. Each process keeps a copy of all queues in its local memory. When an $\mathrm{Enq}_p(Q,v)$ occurs, p sends "enqueue(Q,v)" messages to all other processes (including a message to itself which is delayed d time) and does an Ack immediately. When a $\mathrm{Deq}_p(Q)$ occurs, p sends "dequeue(Q)" messages to all other processes (including a message to itself which is delayed d time). After waiting d time, p handles its own message and does a $\mathrm{Ret}_p(Q,v)$. Whenever a process receives an "enqueue(Q,v)" or "dequeue(Q)" message, it makes the appropriate update to the copy of Q in its local memory. (If it receives several messages at the same time, it "breaks ties" using sender ids, that is, it handles them by increasing order of process ids.)

Theorem 4.6. *There exists a linearizable implementation of FIFO queues with* $|Enq|=0$ *and* $|Deq|=d$.

In the proof, we serialize each operation to occur d time after it is called. Since all processes update their local copies at these serialization times, the claim follows.

Proof. Consider the algorithm just described. Clearly $|Enq| = 0$ and $|Deq| = d$.

Let σ be an admissible execution of this algorithm. For each operation in σ, say that it *occurs* at time d after the real time when its call happens. Let τ be the sequence of operations in σ ordered by time of occurrence, breaking ties with process ids. Clearly $\sigma|p$ is equal to $\tau|p$ for all p, and the order of non-overlapping operations is preserved.

It remains to show that τ is legal, i.e., that for every object Q, $\tau|Q$ is in the serial specification of Q. Pick any Q and consider $\tau|Q = op_1 op_2 \ldots$. Suppose op_i is $[\mathrm{Deq}_p(Q), \mathrm{Ret}_p(Q, v)]$. Because message delay is fixed, updates at p to the local copy of Q occur in the same order as in τ, and the claim follows. □

4.3 Stacks

The results for stacks are analogous to those for FIFO queues, with POP playing the role of DEQ and PUSH the role of ENQ.

Theorem 4.7. *For any sequentially consistent implementation of a stack S, $|Pop(S)| \geq d$.*

Theorem 4.8. *There exists a linearizable implementation of stacks with $|Push| = 0$ and $|Pop| = d$.*

5. CONCLUSIONS AND FURTHER RESEARCH

We have presented a quantitative comparison of the data access time for two well-known consistency conditions for concurrently accessed shared data—sequential consistency and linearizability. Our results indicate that supporting sequential consistency can be more cost-effective than supporting linearizability, for certain object types and under certain timing assumptions. Our results also show that a very precise definition of the guarantees provided is important since seemingly minor differences in the definitions result in significant differences in the inherent efficiency of implementing them. 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.

Our work is closely related to the design of cache consistency schemes that guarantee sequential consistency ([14, 15, 17, 21, 32]). Our implementations use ideas similar to those previously used in cache coherence protocols (cf. [5]). In a sense, we have snooping protocols (without a bus) with a *write broadcast* policy (cf. [29, pp. 467–469]). We believe our ideas can be modified to accommodate a *write invalidate* policy; however, this will slow down the reads.

Our results can be extended to obtain bounds on the response time of implementing other objects, e.g., Test&Set registers, under sequential consistency and linearizability. Further work in this direction is currently underway [22, 31].

The modular usage of atomic broadcast in our implementations of sequential consistency admits several extensions. For example, a bus provides an easy mechanism for atomic broadcast by enforcing a global ordering on all messages delivered to the processes. Afek, Brown, and Merritt ([3]) present a sequentially consistent implementation of read/write objects, for systems where processes communicate via a bus. It might be possible to improve and simplify the correctness proof of the algorithm in [3] using this observation. Also, atomic broadcast algorithms can be made *fault-tolerant*. This can help in the design of memory consistency systems that can sustain failures of some of the processes. In general, the issue of fault-tolerance is rarely addressed in the current research on memory consistency. As multiprocessors scale up and the probability of failure increase, this will become an important concern.

Following the original appearance of our results [9], Mavronicolas and Roth have shown that the tradeoff $|W| + |R| = d$ for sequential consistency is indeed continuous, and there are algorithms that achieve all intermediate values [37]. Furthermore, they have shown that the $\Omega(u)$ bounds we prove for linearizability in the approximate clocks model are tight, by extending our algorithms for the perfect clocks model. It is known that u depends on how closely the clocks in the system are synchronized. Since closely synchronized clocks admit more efficient implementations of linearizability it may be worthwhile to provide such clocks.

Recently, several non-global conditions that are weaker than sequential consistency have been suggested, e.g., weak ordering ([20, 12, 1]), release consistency ([24, 25]), pipelined memory ([35]), slow memory ([30]), causal memory ([4]), loosely coherent memory ([10]), and the definitions in [17] and [41]. 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. Results in this direction appear in [7, 8].

The cost measure we have chosen to analyze is response time, but it is clear that efficiency in general, and response time in particular, are not the only criteria for evaluating consistency guarantees. Other important quantitative measures are amount of local processing and level of message traffic. It has been suggested that it may be possible to implement linearizability *more* cheaply than sequential consistency with regard to these measures, since linearizability is a local property and sequential consistency is not. More qualitative properties such as the ease of designing, verifying, programming, and debugging algorithms using such shared memories are also very important.

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". It will be interesting to understand how these issues influence the bounds.

As multiprocessor systems become larger, distributed implementations of shared virtual memory are becoming more common since 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. (For instance, our work makes use of shifting and atomic broadcast techniques from the theoretical and practical distributed computing literature.) We hope our work contributes toward a more solid ground for this interaction.

ACKNOWLEDGMENTS

The authors thank Sarita Adve, Roy Friedman, Mark Hill, and Rick Zucker for helpful comments on an earlier version of this paper. We especially thank Martha Kosa for a careful reading. The comments of the anonymous referees helped us

improve the presentation.

APPENDIX A. ATOMIC BROADCAST

The atomic broadcast algorithm employed by our algorithms is based on assigning timestamps to messages. Each process maintains a local timestamp (counter) and a vector with (conservative) estimates of the timestamps of all other processes. A process keeps a timestamp bigger than or equal to the timestamps of all the other processes (according to its estimates). Upon a request to broadcast a message, the message is tagged with the requester's current timestamp. Each process maintains a set of messages that are waiting to be delivered. A message with timestamp x is delivered only when the process is certain that all other messages with timestamp $\leq x$ have arrived at it. This is done by waiting to learn that all processes have increased their timestamp to be at least $x + 1$.[11] Once it learns that all processes have increased their timestamps beyond x, the process handles all pending messages with timestamps less than or equal to x, in order, breaking ties using process ids.

More precisely, to broadcast a message m, p sends a message (t_p,m) to all processes (including itself), where t_p is p's current timestamp. It then increases its own timestamp by one, and returns. When a process q receives a message with timestamp t_p from p, it saves it in a list of pending messages, sorted by timestamp and process id. It then increases its timestamp to be at least as large as $t_p + 1$ and sends a timestamp increase message "timestamp(t_q, q)".

When a process receives a timestamp increase message, it updates the timestamp entry for the sender, and checks to see if there are any pending messages whose timestamp is strictly less than all processes' timestamps (saved in its local vector). These messages are delivered in increasing timestamp order, breaking ties using process ids.

The algorithm uses the following data types:

timestamp = integer
message = record with fields
    mess : string (message to be delivered)
    ts : timestamp (assigned by initiator)
    id : process id (id of initiator)

Each process knows n, the total number of processes.
The state of each process consists of the following components:

ts : array[1..n] of integer, all initially 0
    (estimate (from below) timestamps of all processes)
pending : set of message, initially empty
    (set of message waiting to be delivered)

The transition function of process p appears in Fig. 7.

To show that this algorithm implements atomic broadcast, we must show, for any admissible execution, that messages are delivered at the same order to all processes. The ordering of messages is done by timestamps (breaking ties with process ids). The resulting sequence respects the order at each process by construction and because of the way timestamps are assigned.

[11] For simplicity, the algorithm presented here assumes FIFO channels. This assumption can be removed if sequence numbers are employed.

```
ABC-send_p(m):
  send (ts[p],m) to all processes
  ts[p] := ts[p] + 1

receive (t,m) from q:
  add (m,t,q) to pending
  if t + 1 > ts[p] then
          ts[p] := t + 1
          send timestamp(ts[p]) to all processes
  endif

receive timestamp(t) from q:
  ts[q] := t
  repeat
          let E be element with smallest (ts,id) pair in pending
          if for some q, ts[q] ≤ E.ts then exit
          deliver E.m { this is the ABC-receive }
          remove E from pending
  endrepeat
```

Fig. 7. Atomic broadcast algorithm.

More formally, fix some admissible execution σ of the algorithm. The next lemma follows immediately from the code.

Lemma A.1. *Let p be any process. Then every message broadcast by p in σ is given a unique timestamp in increasing order.*

This immediately implies:

Lemma A.2. *The timestamps assigned to messages in σ, together with process ids, form a total order.*

This total order is called *timestamp order*.

Lemma A.3. *Let p be any process. Then all messages are delivered to p in σ in timestamp order.*

Proof. Let (t_1, q_1) be the timestamp of the message m_1, and let (t_2, q_2) be the timestamp of the message m_2. Suppose, by way of contradiction, that $(t_1, q_1) < (t_2, q_2)$ but m_2 was delivered to p before m_1.

When m_2 is delivered to p, it cannot yet have the message m_1 in pending, because otherwise it would deliver it before m_2. By the code, in order to deliver m_2, it must be that $ts_p[q_1] > t_2$. But then p must have received a timestamp message from q_1 with a timestamp $t \geq t_2 + 1$. Since $(t_1, q_1) < (t_2, q_2)$ it must be that $t_1 \leq t_2$, and hence $t > t_1$. By the code, the message m_1 was sent before the timestamp message. But then the FIFO property of the communication system implies that p has already received m_1. A contradiction. $\square$

The next lemma guarantees that each message is delivered within time $2d$ from the initiation of the operation.

Lemma A.4. *If process p broadcasts a message m, then m is delivered at each process within time at most $2d$ in σ.*

Proof. Assume p broadcasts m at time T, with timestamp x. By time $T+d$ all processes will get the message (x,m), and will set their timestamps to be at least $x+1$, sending a timestamp increase message to all other processes, if necessary. Thus, by time $T+2d$, all processes will have in their timestamp vectors values that are strictly larger than x, and will deliver m. □

Lemmas A.3 and A.4 prove the following theorem.

Theorem A.5. *The algorithm in Fig. 7 is an atomic broadcast algorithm with $h = 2d$.*

REFERENCES

1. S. Adve and M. Hill, "Weak Ordering—A New Definition," *Proc. 17th Int. Symp. on Computer Architecture,* 1990, pp. 2–14.
2. S. Adve and M. Hill, "Implementing Sequential Consistency in Cache-Based Systems," *Proc. Int. Conf. on Parallel Processing,* 1990, pp. I-47–50.
3. Y. Afek, G. Brown, and M. Merritt. "Lazy Caching," *ACM Trans. on Programming Languages and Systems,* Vol. 15, No. 1 (January 1993), pp. 182–205.
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. J. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. on Computer Systems,* Vol. 4, No. 4 (November 1986), pp. 273–298.
6. H. Attiya, "Implementing FIFO Queues and Stacks," *Proc. 5th Int. Workshop on Distributed Algorithms,* 1991, pp. 80–94, Lecture Notes in Computer Science #579, Springer-Verlag. Also: Technical Report #672, Department of Computer Science, The Technion, Haifa, May 1991.
7. H. Attiya and R. Friedman, "A Consistency Condition for High-Performance Multiprocessors," *Proc. 24th ACM Symp. on Theory of Computing,* Victoria, B.C., May 1992, pp. 679–690. Also: Technical Report #719, Department of Computer Science, The Technion, Haifa, June 1991.
8. H. Attiya, S. Chaudhuri, R. Friedman and J. L. Welch, "Non-Sequential Consistency Conditions for Shared Memory," *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures,* 1993, to appear.
9. H. Attiya and J. L. Welch, "Sequential Consistency versus Linearizability," *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures,* 1991, pp. 304–315.
10. J. Bennett, J. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proc. 2nd ACM Symp. on Principles and Practice of Parallel Processing,* 1990, pp. 168–176.
11. P. Bernstein, V. Hadzilacos, and H. Goodman, *Concurrency Control and Recovery in Database Systems,* Addison-Wesley, Reading, MA, 1987.
12. R. Bisiani, A. Nowatzyk, and M. Ravishankar, "Coherent Shared Memory on a Distributed Memory Machine," *Proc. Int. Conf. on Parallel Processing,* 1989, pp. I-133–141.
13. K. Birman and T. Joseph, "Reliable Communication in the Presence of Failures," *ACM Trans. on Computer Systems,* vol. 5, No. 1 (February 1987), pp. 47–76.
14. W. Brantley, K. McAuliffe, and J. Weiss, "RP3 Processor-Memory Element," *Proc. Int. Conf. on Parallel Processing,* 1985, pp. 782–789.
15. L. M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. on Computers,* Vol. C-27, No. 12 (December 1978), pp. 1112–1118.
16. J. Chang and N. F. Maxemchuk, "Reliable Broadcast Protocols," *ACM Trans. on Computer Systems,* Vol. 2, No. 3, pp. 251-273.
17. W. W. Collier, "Architectures for Systems of Parallel Processes," IBM TR 00.3253, Poughkeepsie, NY, January 1984.
18. E. W. Dijkstra, "Hierarchical Ordering Of Sequential Processes," *Acta Informatica,* Vol. 1 (1971), pp. 115–138.
19. 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.

20. M. Dubois, C. Scheurich, and F. A. Briggs, "Memory Access Buffering in Multiprocessors," *Proc. 13th Int. Symp. on Computer Architecture,* June 1986, pp. 434–442.
21. M. Dubois, C. Scheurich, and F. A. Briggs, "Synchronization, Coherence and Event Ordering in Multiprocessors," *IEEE Computer,* Vol. 21, No. 2 (June 1986), pp. 9–21.
22. R. Friedman, "Implementing Hybrid Consistency with High-Level Synchronization Operations" *Proc. 12th ACM Symp. on Principles of Distributed Computing,* 1993, to appear.
23. H. Garcia-Molina and A. Spauster, "Message Ordering in a Multicast Environment," *Proc. Int. Conf. on Distributed Computing Systems,* 1989, pp. 354-361.
24. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennessey, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proc. 17th Int. Symp. on Computer Architecture,* 1990, pp. 15-26.
25. P. Gibbons, M. Merritt and K. Gharachorloo, "Proving Sequential Consistency of High-Performance Shared Memories," *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures,* July 1991, pp. 292–303.
26. A. Gottlieb, R. Grishman, C. K. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, "The NYU Ultracomputer—Designing a MIMD Shared-Memory Parallel Computer," *IEEE Trans. on Computers,* Vol. C-32, No. 3, February 1983, pp. 175–189.
27. M. Herlihy, "Wait-Free Implementations of Concurrent Objects," *Proc. ACM Symp. on Principles of Distributed Computing*, 1988, pp. 276–290.
28. M. Herlihy and J. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. on Programming Languages and Systems,* Vol. 12, No. 3 (July 1990), pp. 463–492.
29. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann Publishing, 1990.
30. 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.
31. M. Kosa, *Consistency Conditions for Concurrent Shared Objects: Upper and Lower Bounds,* Ph.D. dissertation, Department of Computer Science, University of North Carolina, Chapel Hill, NC. To appear 1993.
32. L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Trans. on Computers,* Vol. C-28, No. 9 (September 1979), pp. 690–691.
33. L. Lamport, "On Interprocess Communication. Parts I and II," *Distributed Computing,* Vol. 1, No. 2 (1986), pp. 77–101.
34. K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. on Computer Systems,* Vol. 7, No. 4 (November 1989), pp. 321–359.
35. R. Lipton and J. Sandberg, *PRAM: A Scalable Shared Memory,* TR CS-TR-180-88, Princeton University, September 1988.
36. J. Lundelius and N. Lynch, "An Upper and Lower Bound for Clock Synchronization," *Information and Control,* Vol. 62, Nos. 2/3 (August/September 1984), pp. 190–204.
37. M. Mavronicolas and D. Roth, "Efficient, Strongly Consistent Implementations of Shared Memory," *Proc. 6th Int. Workshop on Dist. Algorithms,* 1992, pp. 346–361. Lecture Notes in Computer Science #647, Springer Verlag.
38. S. Min and J. Baer, "A Timestamp-Based Cache Coherence Scheme," *Proc. Int. Conf. on Parallel Processing,* 1989, pp. I-23–32.
39. J. Misra, "Axioms for Memory Access in Asynchronous Hardware Systems," *ACM Trans. on Programming Languages and Systems,* Vol. 8, No. 1 (January 1986), pp. 142–153.
40. C. Papadimitriou, *The Theory of Concurrency Control,* Computer Science Press, Rockville, MD, 1986.
41. U. Ramachandran, M. Ahamad, and M. Y. Khalidi, "Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer," *Proc. Int. Conf. on Parallel Processing,* 1989, pp. II-160–169.
42. C. Scheurich and M. Dubois, "Correct Memory Operation of Cache-Based Multiprocessors," *Proc. 14th Int. Symp. on Computer Architecture,* 1987, pp. 234–243.

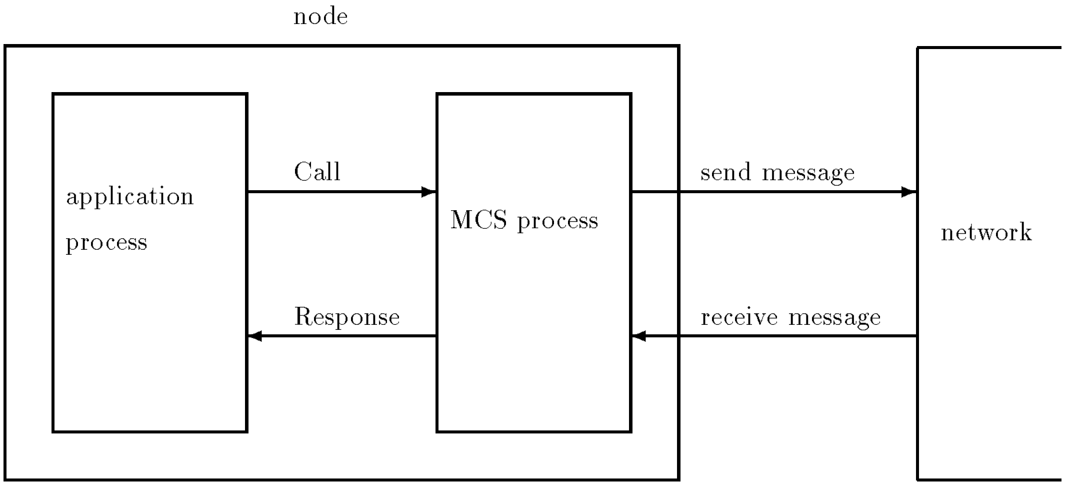
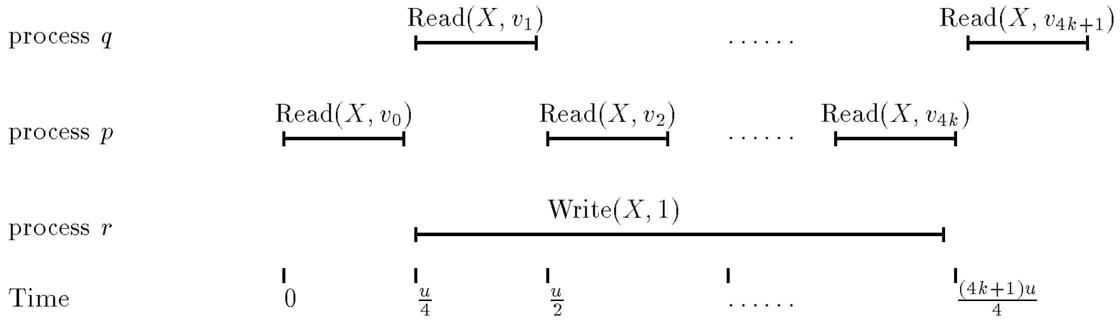
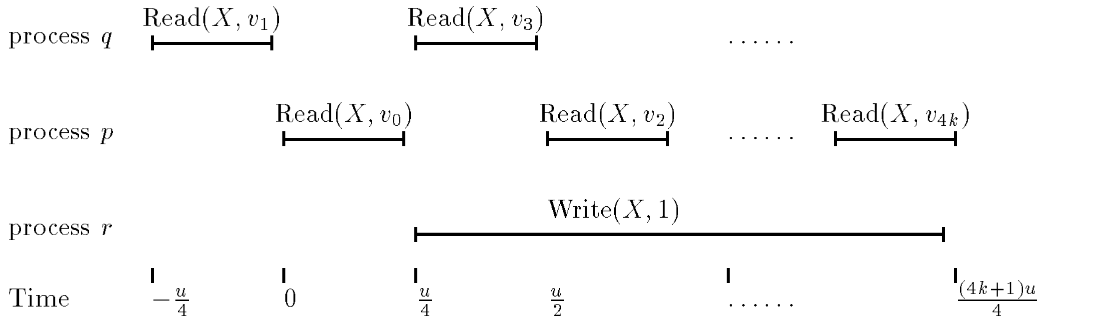

node
application
process
Call
Response
MCS process
send message
receive message
network

Figure 1.

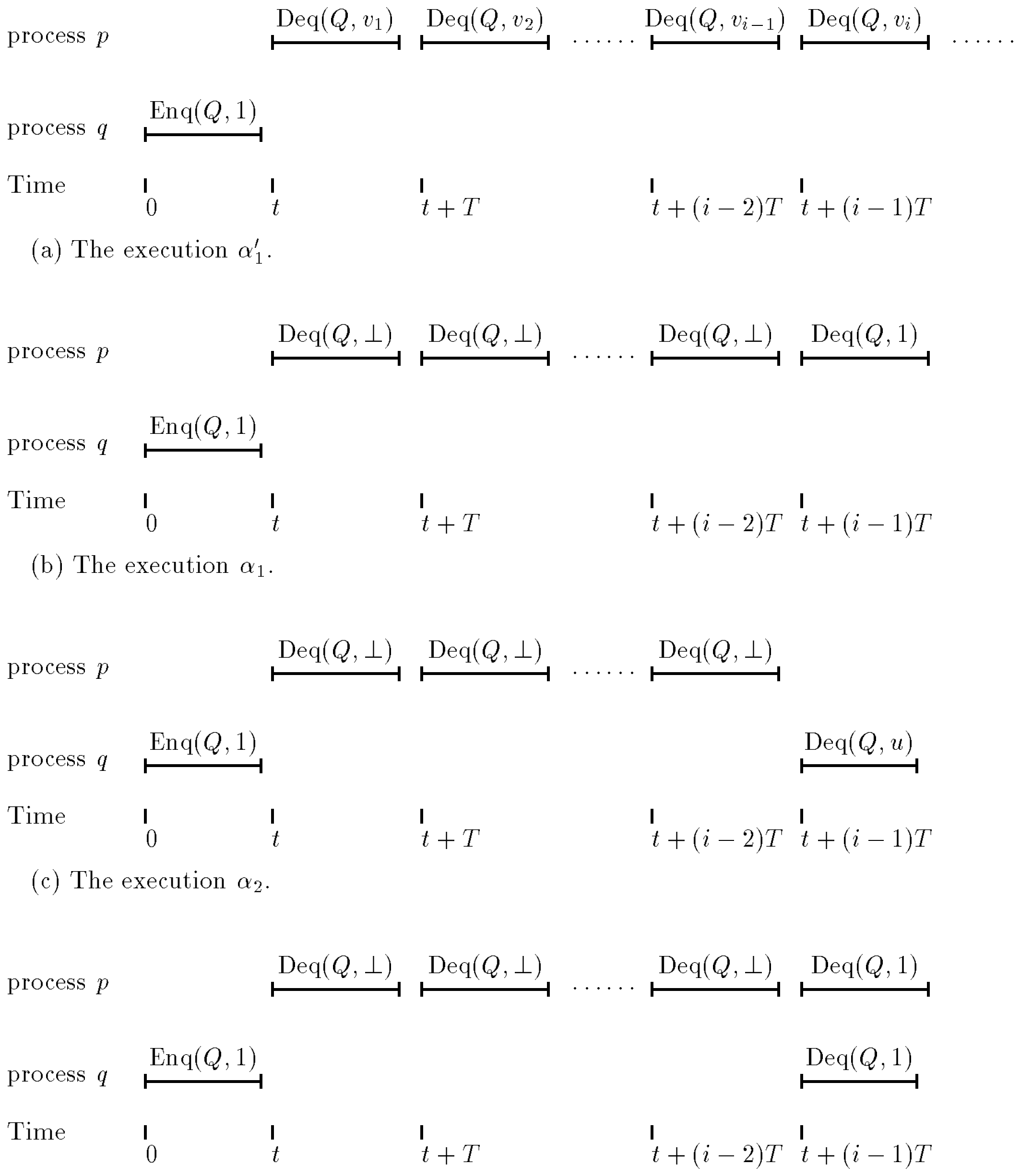


(a) The execution α.



(b) The execution β.

Figure 2. Executions used in the proof of Theorem 3.1.



(a) The execution α_1'.

(b) The execution α_1.

(c) The execution α_2.

(d) The execution α.

Figure 6. Executions used in the proof of Theorem 4.4.