Chicago Journal of Theoretical Computer Science

The MIT Press

Volume 1999, Article 9

Time Bounds for Strong and Hybrid Consistency for Arbitrary Abstract Data Types

ISSN 1073–0486. MIT Press Journals, Five Cambridge Center, Cambridge, MA 02142-1493 USA; (617)253-2889; *journals-orders@mit.edu, journals-info@mit.edu.* Published one article at a time in LaTeX source form on the Internet. Pagination varies from copy to copy. For more information and other articles see:

- *http://mitpress.mit.edu/CJTCS/*
- *http://www.cs.uchicago.edu/publications/cjtcs/*
- *ftp://mitpress.mit.edu/pub/CJTCS*
- *ftp://cs.uchicago.edu/pub/publications/cjtcs*

The *Chicago Journal of Theoretical Computer Science* is abstracted or indexed in *Research Alert,*® *SciSearch,* ® *Current Contents*®*/Engineering Computing & Technology*, and *CompuMath Citation Index.*®

The *Chicago Journal of Theoretical Computer Science* is a peer-reviewed scholarly journal in theoretical computer science. The journal is committed to providing a forum for significant results on theoretical aspects of all topics in computer science.

Time Bounds for Strong and Hybrid Consistency for Arbitrary Abstract Data Types

Martha J. Kosa
(Tennessee Technological University)
`http://www.csc.tntech.edu/~mjkosa/`
mjk9504@tntech.edu

2 September 1999

Abstract

We compare the costs of three well-known consistency guarantees (sequential consistency, linearizability, and hybrid consistency) for shared memory objects of *arbitrary* data types, generalizing previous work on specific types. We evaluate the impact of the desired consistency guarantee, the amount of synchrony among the users of the objects, and algebraic properties of a type on the worst-case time complexity of operations of the type. These algebraic properties are sufficient for proving many lower bounds and some upper bounds (even 0, meaning only local computation is required) on the worst-case times. Under certain reasonable assumptions, sequential consistency and linearizability are equally costly in perfectly synchronized systems, linearizable operations are more expensive in approximately synchronized systems than in perfectly synchronized systems, sequentially consistent operations are cheaper than linearizable operations in approximately synchronized systems, and hybrid consistency is not necessarily cheaper than sequential consistency or linearizability.

1 Introduction

1.1 Background material

In concurrent systems, processes need to share information with each other. Because of software engineering concerns, using shared data objects from arbitrary abstract data types is a popular method for organizing this information. However, processes may not have access to a physical shared memory, because they may be running on computers separated by long distances or their computer architecture may not provide a physical shared memory. They must simulate a physical shared memory with a virtual shared memory.

A consistency guarantee tells the processes using the virtual shared objects what they can expect about the values returned as the result of applying operations, even when operations are executed concurrently on the same virtual object. Researchers have defined many types of consistency guarantees—some strong, some weak, and some in between. A strong guarantee is at least as expensive to implement as a weaker guarantee, but it may be impossible (or very hard) to solve a problem by using virtual shared objects providing a weaker guarantee (see [4]).

Sequential consistency and linearizability are two strong consistency guarantees. They ensure that operations appear to have executed atomically in some sequential order that reflects the order in which operations were executed at each process. In addition, linearizability ensures that this sequential order preserves the relative ordering of all nonoverlapping operations, even those executed by different processes. Sequential consistency is a very popular consistency guarantee, used for virtual shared memories and multiprocessor caches (see [2, 5]). However, linearizability is a stronger, more intuitive guarantee, because sequential consistency provides no clues about the relative ordering of nonoverlapping operations performed by different processes. Although linearizability is more powerful than sequential consistency, it is still interesting to study both guarantees; the difference between their definitions is very slight.

Attiya and Friedman [4] investigated practical shared memory consistency models (refer to [1, 7, 9, 10]) and formally defined hybrid consistency, a generalization of those models; it systematically weakens either sequential consistency or linearizability. In hybrid consistency, there are strong and weak operations. Strong operations appear to satisfy a strong consistency guarantee, while weak operations can appear to execute in different orders at different processes as long as the relative ordering of weak and strong operations executed by the same process is preserved.

System designers should understand the costs of providing these consistency guarantees in order to choose a cost-effective consistency guarantee that best suits the needs of their applications. We focus on (distributed) message-passing implementations because they are scalable. Each process has a local memory, and all processes run a protocol to emulate a physical shared memory with a given consistency guarantee.

Attiya and Welch [5] compared the costs of implementing sequential consistency and linearizability for basic read/write objects, queues, and stacks. They measured the worst-case time for operations to complete, giving upper and lower bounds to show that the amount of process synchrony caused the cost difference between sequential consistency and linearizability to vary. With perfect synchrony, their costs are equal. However, sequential consistency is cheaper when processes are only approximately synchronized. Mavronicolas and Roth [17] continued the Attiya-Welch comparative study, improving some of their lower bounds and giving a distributed implementation of linearizable read/write objects in a system with only approximately synchronized processes. Attiya and Friedman [4] compared the cost of hybrid consistency with the costs of sequential consistency and linearizability for read/write objects, showing that hybrid consistency is cheaper when mostly weak operations are executed. Friedman [8] compared the cost of hybrid consistency with the costs of stronger guarantees for read/modify/write objects, queues, and stacks, showing that hybrid consistency is not cheaper, even when mostly weak operations are executed.

1.2 Overview of results

Instead of concentrating on specific data types, as in earlier work, we study the worst-case response times for operations of *arbitrary* abstract data types in sequentially consistent, linearizable, and hybrid consistent implementations. We show that algebraic properties of the operations of an abstract data type are sufficient for proving many lower bounds and some upper bounds on the worst-case response times. Our work generalizes and unifies previously known results, for example, [4, 5, 8, 17]. Some of these algebraic properties, such as equivalence and commutativity, were defined in Weihl's study of concurrency control and recovery in transaction systems [18]. Weihl's definitions helped us to identify interesting new algebraic properties of operations.

We now give a brief description of our results. Let d be the maximum message delay of a system of processes connected by a network, and let u be the uncertainty in the message delay ($0 \leq u < d$). This implies that the actual message delay may vary between $d - u$ and d.

In the first part of the paper, we assume that processes have perfectly synchronized clocks and constant message delays ($u = 0$). Any lower bounds can hold in systems with weaker, more realistic timing assumptions. We consider interactions among operations to determine the worst-case completion times for operations of the abstract data type to be implemented. We find several algebraic properties that cause either a single operation or a pair (collectively) to have a worst-case completion time of at least d or $2d$.

It is often the case that a particular operation or group of operations of an abstract data type is known to be used most frequently. Thus it is desirable to optimize (in terms of worst-case completion time) that operation or group of operations. We investigate conditions that allow this operation or the operations of the group to have worst-case completion times of 0 (to be *fast*), meaning that an operation can return immediately based on current local information at its invoking process, in a linearizable implementation. Making one operation fast may require another operation to be *slow*, that is, to have a worst-case completion time of $\Omega(d)$.

Operations may be classified as accessors, pure mutators, self-oblivious operations, immediately self-commuting operations, or none of the above. An *accessor* returns information about an object without changing it. A *pure mutator* changes an object without returning any information about it. The information returned by a *self-oblivious* operation does not depend on operations with the same name that are executing at nearly the same time. The information returned by an *immediately self-commuting* operation does not depend on operations with the same name that are executing simultaneously. Figure 1 shows the relationships among these classes of operations and describes the smallest possible worst-case completion time (fast or slow) of a single operation in these classes.[1]

Our general results for systems with perfectly synchronized clocks and no uncertainty in the message delay are that any single self-oblivious operation

[1]In the case of accessors and pure mutators, time is given for a group of operations.

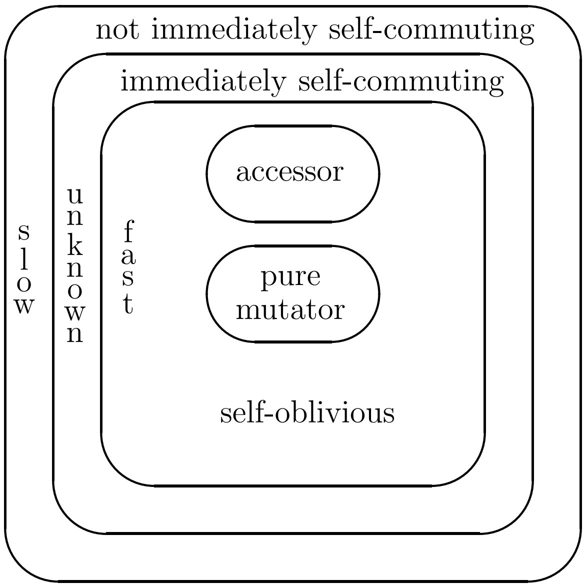


Figure 1: Operations and smallest worst-case completion times

can be made fast and that no operation that does not immediately commute with itself can be made fast. Still unknown is the smallest possible worst-case completion time for a single operation that is not self-oblivious but does immediately commute with itself. However, a large class of well-known abstract data types has self-oblivious operations.

We can use the lower and upper bounds proven to deduce that sequential consistency and linearizability are equally costly in systems with perfectly synchronized clocks when the expected completion time to perform all operations in a linearizable implementation matches the lower bound for all operations under sequential consistency. Computing the expected completion time requires knowledge of the frequency of executing each operation for the abstract data type. If $op_1, \ldots, op_n$ are the operations for an abstract data type and p_i is the frequency of execution for each op_i, where $\sum_{i=1}^{n} |p_i| = 1$, the expected completion time for a single operation is $\sum_{i=1}^{n} p_i |OP_i|$.

In the second part of the paper, we consider systems with positive uncertainty in the message delay and approximately synchronized clocks. Our goal is to find a list of algebraic properties that cause operations (which satisfy a property in the list) to have positive worst-case completion times ($\Omega(u)$) in linearizable implementations but worst-case completion times of 0 in sequentially consistent implementations. Sequential consistency is then less expensive than linearizability in this model of process synchrony.

We exhibit two properties that cause operations to have worst-case completion times of at least $u/2$ in linearizable implementations. An operation satisfying one property or a restriction of the other property can be implemented with a worst-case completion time of 0 in a sequentially consistent implementation. These properties hold for a large class of well-known abstract data types. Under these conditions, sequential consistency is cheaper than linearizability.

Finally, we consider the weaker guarantee of hybrid consistency. We use many of the algebraic properties used for sequential consistency to yield similar lower bounds for hybrid consistency, and we deduce that hybrid consistency is not necessarily cheaper than stronger guarantees.

1.3 Organization of this paper

Section 2 contains general definitions about abstract data types and definitions of the three correctness guarantees. Section 3 discusses the costs of the strong guarantees in systems with perfectly synchronized clocks, with subsections for lower bounds (sequential consistency) and upper bounds (linearizability). Section 4 discusses the costs of the strong guarantees in systems with only approximately synchronized clocks, with subsections for linearizability (lower bounds) and sequential consistency (upper bounds). Section 5 discusses the costs of hybrid consistency. We summarize our results in Section 6.

2 Definitions

A *sequential specification* (see [11]) for an abstract data type contains a set of *operations*, which are ordered pairs of call-and-response events, and a set of legal operation sequences. The legal sequences of operations reflect the semantics of the abstract data type. For example, for read/write objects, each read operation in a legal sequence returns the value written by the last preceding write operation in the sequence. The call event represents the call for the corresponding operation from the abstract data type, while the response event represents the value returned by applying the operation. We refer to call events and their matching response events separately because they may not occur atomically; they may be separated in time. For operation *op*, a call event is of the form *call*(*arglist*), where *arglist* is the argument list (possibly empty) for the operation. For example, a call event for a read operation (respectively, write operation) on a read/write object is of the form *read*() (respectively, *write*(v)). Similarly, a response event is of the form *resp*(*retlist*), where *retlist* is the list of values returned by the operation and is possibly empty. For example, a response event for a read operation (respectively, write operation) on a read/write object is of the form *ret*(v) (respectively, *ack*()). We can combine the call-and-response events to yield *op*(*arglist*)(*retlist*). In our running example, we get *read*()(v) and *write*(v)(). We can partition all operations into equivalence classes. All operations with the same name for their call events are in the same generic operation, or operation type. OP_i denotes a generic operation; op_i and op_i^j denote instantiations of OP_i (i.e., nongeneric operations).

The notion of sequential specification implicitly assumes some initialization of the object; for example, *read*()(3) is legal for a read/write object that has 3 as its initial value. We assume the ability to explicitly initialize an object O using any sequence of operations ρ that is legal for O. Formally, the object O_ρ is a *ρ-initialized version of O* if it has the same set of operations as O, and the operation sequence σ is legal for O_ρ if and only if $\rho \circ \sigma$ is legal for O. The assumption of arbitrary explicit initialization is not unreasonable, since initialization normally occurs at the beginning of program executions.

We present two basic algebraic properties of operations from [18]. Let α and β be operation sequences. *α looks like β* if for every operation sequence γ, $\alpha \circ \gamma$ is legal only if $\beta \circ \gamma$ is legal. If α looks like β, then the user of the abstract data type never sees the result of an operation that allows the user to distinguish β from α after α is executed. If α looks like β and β looks like α, then α and β are *equivalent.* Future operations cannot distinguish between α and β.

The preceding definitions are stated in terms of sequential operations on a single object. Now we need to enhance our notation to handle multiple shared objects in concurrent systems. If e is a call event, response event, or whole operation, then $e[O, p]$ denotes that e is performed by a process p on an object O. If α is a sequence of operations, then $\alpha[O, p]$ denotes that all operations in α are performed by a process p on an object O. A sequence τ of operations for a set of objects is legal if, for each object O, the restriction of τ to operations of O, denoted $\tau|O$, is legal for O's abstract data type.

We now describe the system model.[2] A *memory consistency system* (MCS) is a set of processes P and a set of clocks $\mathcal{C}$, one for each p in P. Our assumed system consists of a collection of nodes connected by a network. An application program, a real-time clock, and an MCS process are running on each node. An MCS process can read the real-time clock residing at its node. A *clock* is a monotonically increasing function from real time to clock time (both sets are the set of real numbers). A process cannot modify the real-time clock. Processes can only obtain information about time from their clocks.

The following events can occur at the MCS process on node p:

- A *call event* occurs when the application program on node p accesses a shared object.
- A *response event* occurs when the MCS process on node p gives a response from a shared object to node p's application program.
- *Message receive events* are of the form receive(p, m, q) for all messages m and all nodes q. A message receive event occurs when the MCS process on node p receives message m from the MCS process on node q.
- *Message send events* are of the form send(p, m, q) for all messages m and all nodes q. A message send event happens when the MCS process on node p sends message m to the MCS process on node q.
- *Timer set events* are of the form timerset(p, T) for all clock times T. This means that p sets a timer to go off when its clock reads T.
- *Timer events* are of the form timer(p, T) for all clock times T. This means that a timer that was set for time T on p's clock goes off.

Call, message receive, and timer events are *interrupt events.*

[2]Our system model is the same as in [5].

An *MCS process* (or just *process*) is an automaton with a set of states, including an initial state, and a transition function. Each interrupt event causes the transition function to be applied. 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 future clock times). This means that the transition function takes as input the current state, clock time, and interrupt event and then 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 (s is the current state and s' is the new state), 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 results of p's transition function acting on s, T, and i.

A *history* of a process p with clock C is a countable sequence of steps such that the following hold:

- Steps are ordered by T, their time components, in increasing order.
- The old state in the first step is p's initial state.
- The old state of each subsequent step is the new state of the previous step.
- For the subsequence of steps with time component $T = t$, all nontimer events are ordered before any timer event, and there is at most one timer event.

An *execution* of an MCS is a set of histories, one for each process p in P with clock C_p in $\mathcal{C}$, which satisfies the following two conditions: First, for all pairs of processes p and q, every message sent from p to q is received by q, and every message received by q from p was actually sent by p (reliable message transmission and no duplicated messages). We use this one-to-one correspondence to define the *delay* of any message in an execution to be the difference between the real time of receipt and the real time of sending. Second, a timer is received by p at clock time T if and only if p has previously set a timer for T.

An execution ρ is *admissible* if the following are true:

- For every p and q, every message in ρ from p to q has delay in the range $[d-u, d]$, for fixed nonnegative integers d and u, $u \leq d$.
- For every p, at most one call at p is pending (lacks a matching response) at any given time.

Our definitions of the correctness conditions are identical to those found in [5, 4].

Given an execution σ, let $ops(\sigma)$ be the sequence of call-and-response events appearing in σ in real-time order. We need to specify a tie-breaking mechanism for ordering events that occur at the same real time t. In this ordering, the first group of events is formed by the response events that happen at time t and whose matching call events happen before time t, ordered by their process identifiers. The second group of events in the ordering is formed by the call events that happen at time t and whose matching response events happen at time t, ordered by their process identifiers with a call event immediately preceding its matching response event. The third group of events in the ordering is formed by the call events that happen at time t and whose matching response events happen after time t, ordered by their process identifiers.

We now define sequential consistency, linearizability, and hybrid consistency. These definitions all imply that every call eventually has a matching response and that call events and response events alternate at a given process (a process never has more than one pending call). If s is a sequence of operations and p is a process, then we denote the restriction of s to operations of process p by $s|p$. Given an execution ρ, a sequence of operations s is a *serialization* of ρ if it is a permutation of $ops(\rho)$.

Sequential consistency ensures that all processes agree on some ordering of the execution at the granularity of entire operations. In this ordering, the operations for each process appear in the order in which they were executed at that process. An execution ρ is *sequentially consistent* if there exists a legal serialization τ of ρ such that $ops(\rho)|p = \tau|p$ for each process p.

Like sequential consistency, linearizability ensures that all processes agree on some ordering of the execution (at the granularity of entire operations) that preserves the operation sequences of the processes. In addition, the ordering must preserve the actual timings of operations. An execution ρ is *linearizable* if there exists a legal serialization τ of ρ such that $ops(\rho)|p = \tau|p$

for each process p, and op_1 precedes op_2 in τ if the response for op_1 precedes the call for op_2 in $ops(\rho)$.

Hybrid consistency tries to give us the best of strong consistency (operations appear to execute everywhere in some fixed order) and weak consistency (there is no fixed order, admitting fast implementations). Each operation has a strong version and a weak version. We want all processes to perceive the same order of execution for all strong operations and the same relative order of execution for each pair of operations executed by the same process, where at least one operation in the pair is strong.

An execution ρ is *hybrid consistent* if there exists a serialization σ of the strong operations of ρ such that for each process p, there exists a legal sequence τ_p of operations such that the following four statements are true: (1) τ_p is a permutation of $ops(\rho)$. (2) If op_1 and op_2 are executed by the same process, op_1 precedes op_2 in ρ, and at least one of op_1 and op_2 is a strong operation, then op_1 precedes op_2 in τ_p. (3) If op_1 precedes op_2 in σ and both are strong, then op_1 precedes op_2 in τ_p. (4) We have $\tau_p|p = \rho|p$.

An MCS is a sequentially consistent (respectively, linearizable or hybrid consistent) implementation of a set of objects if any admissible execution of the MCS is sequentially consistent (respectively, linearizable or hybrid consistent).

We measure the efficiency of an implementation by the worst-case response time for any operation on any object in the set. Let O be an object and OP be a generic operation. $|OP(O)|$ is the maximum time taken by an *op* operation on O in any admissible execution. $|OP|$, the worst-case time for the generic operation OP to be completed, is the maximum of $|OP(O)|$ over all objects implemented by the MCS.

3 Perfect clocks

In this section we assume that all processes have perfectly synchronized (perfect) clocks and a constant, known message delay d. (These two concepts are equivalent; see [5]). We model constant message delay by letting the message uncertainty be $u = 0$. We give lower and upper bounds on the costs of providing sequentially consistent and linearizable implementations of virtual shared objects. These lower bounds hold under weaker, more realistic assumptions about process synchrony. We prove our lower bounds on the costs of operations in sequentially consistent implementations, implying corresponding lower bounds for linearizable implementations. Perfect clocks are necessary for the algorithms demonstrating our upper bounds. Our algorithms are primarily of theoretical interest.

Subsection 3.1 describes lower bounds on the costs of implementing single operations, pairs of operations, and larger groups of operations from abstract data types. Subsection 3.2 presents lower bounds on the costs of implementing all operations from abstract data types. Subsection 3.3 contains implementations of classes of abstract data types in which a single operation or group of operations is optimized and describes abstract data types for which the bounds in Subsection 3.1 are tight.

3.1 Lower bounds for sequential consistency for singles, pairs, and groups

We give lower bounds on the costs in sequentially consistent implementations of single operations, pairs of operations, and larger groups of operations satisfying certain algebraic properties, which we present as needed. The properties are also useful in Section 5, where we consider hybrid consistency.

If β and γ are operation sequences, then β and γ *commute* [18] if, for every operation sequence α such that $\alpha \circ \beta$ and $\alpha \circ \gamma$ are legal, $\alpha \circ \beta \circ \gamma$ and $\alpha \circ \gamma \circ \beta$ are legal and equivalent.[3]

Formally we say that two generic operations *do not commute* by negating the previous definition. OP_1 and OP_2 do not commute if there exist operation instances op_1 and op_2 and a sequence of operations α such that $\alpha \circ op_1$ and $\alpha \circ op_2$ are legal and (at least) one of the following is true:

1. $\alpha \circ op_1 \circ op_2$ is not legal.
2. $\alpha \circ op_2 \circ op_1$ is not legal.
3. $\alpha \circ op_1 \circ op_2$ does not look like $\alpha \circ op_2 \circ op_1$, which means that there exists an operation sequence γ such that $\alpha \circ op_1 \circ op_2 \circ \gamma$ is legal but $\alpha \circ op_2 \circ op_1 \circ \gamma$ is not.

[3]The concept is called *commute forward* in [18].

4. $\alpha \circ op_2 \circ op_1$ does not look like $\alpha \circ op_1 \circ op_2$, which means that there exists an operation sequence γ such that $\alpha \circ op_2 \circ op_1 \circ \gamma$ is legal but $\alpha \circ op_1 \circ op_2 \circ \gamma$ is not.

If item 1 or item 2 is true, then OP_1 and OP_2 *immediately do not commute.*[4] If item 3 or item 4 is true, then OP_1 and OP_2 *eventually do not commute.* If items 1 and 2 are true, OP_1 and OP_2 are *cyclically dependent.* Appendix A contains several examples of abstract data types with their commutativity properties.

The proofs of the theorems in this subsection are similar to each other. They show that some legal executions, each performed by a single process, may not be combined into a globally legal execution if minimum time bounds for certain operations are not obeyed. This results in a violation of sequential consistency. We justify the existence of these legal executions and our combining technique in Appendix B. With each proof, we provide a picture of the illegal global execution. In each picture, time moves from left to right, with critical times shown at the bottom, and each legal individual execution has its own line. Also, the periods of time for message transit are shown. For each operation (sequence), we show the object on which it is performed and the process performing it. Each process completes its operations based on the information available when the operations are executed. We choose starting times for operations so that the finishing times (based on the lower bounds) are too small for the processes to use the information in messages sent by other processes.

We now give a condition under which an *individual* (generic) operation of an abstract data type must be slow.

Theorem 1 *Let T be an abstract data type with a generic operation OP that immediately does not commute with itself. In any sequentially consistent implementation of objects of type T, $|OP| \geq d$.*

Proof The following proof generalizes the proof in [5] that a dequeue operation of the queue abstract data type must take at least time d. Let A be an object of type T. Let processes 1 and 2 access A. Since OP immediately does not commute with itself, there exist a sequence ρ of operations and an

[4]If op_1 and op_2 are the same instantiation of the same generic operation OP, then OP *immediately does not commute with itself.*

operation instance op such that $\rho \circ op$ is legal but $\rho \circ op \circ op$ is not legal. Suppose in contradiction that there is a sequentially consistent implementation of A_ρ for which $|OP| < d$.[5]

Figure 2 shows possible admissible executions α_1 and α_2. Since no messages are received in α_1 and α_2, replacing p_2's history in α_1 with its history in α_2 results in another admissible execution, α. By assumption, α is sequentially consistent. Thus, there is a τ that is a permutation of $ops(\alpha)$ and is legal for A_ρ. However, all possible permutations of $ops(\alpha)$ are of the form $op \circ op$, which is not legal for A_ρ. ■

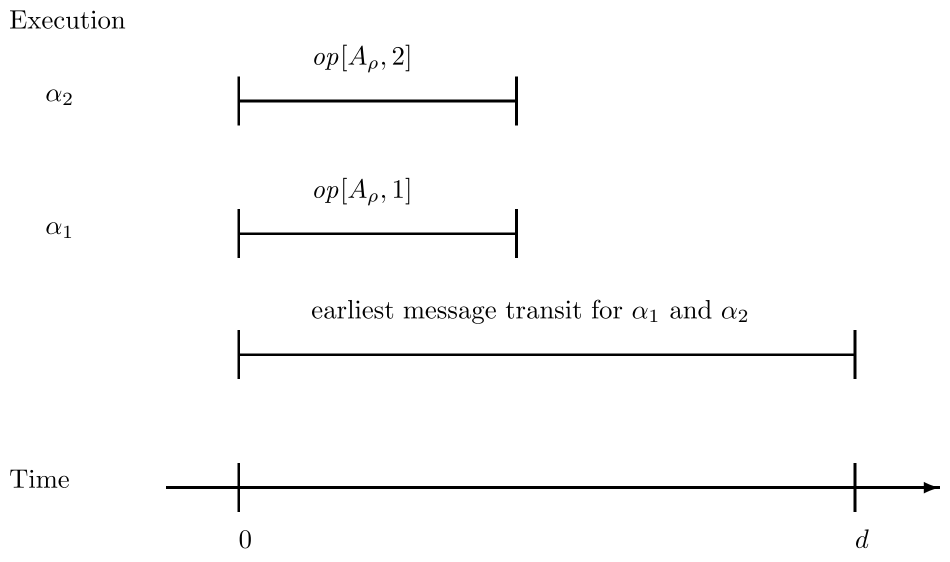


Figure 2: Counterexample for lower bound for an operation that immediately does not commute with itself (Theorem 1)

For read/write objects, $write(v)()$ immediately commutes with $write(w)()$, but the two writes eventually do not commute if $v \neq w$. Therefore, the previ-

[5]The use of the explicit initialization assumption is necessary in this proof, because we must use serialized operation sequences of the form $\rho \circ \gamma$ to prove a violation of sequential consistency, and the definition of sequential consistency does not require that ρ appear as a prefix of the serialization of the execution. The necessity of the explicit initialization assumption is an open question.

ous theorem does not apply for $|WRITE|$. In fact, [5, 17] present algorithms in which writes take less than time d.

Next we give a condition under which a *pair* of distinct (generic) operations from an abstract data type must be slow.

Theorem 2 *Let T be an abstract data type, and let OP_1 and OP_2 be distinct generic operations on T that immediately do not commute. In any sequentially consistent implementation of at least two objects of type T, $|OP_1| + |OP_2| \geq d$.*

Proof This proof generalizes the proof in [5] that $|READ| + |WRITE| \geq d$ for read/write objects. Since OP_1 and OP_2 immediately do not commute, there is a sequence of operations α and operation instances op_1 and op_2 such that $\alpha \circ op_1$ and $\alpha \circ op_2$ are legal, but (without loss of generality) $\alpha \circ op_1 \circ op_2$ is not legal. Let A and B be two objects of type T, and let processes 1 and 2 use A and B. Suppose in contradiction that there is a sequentially consistent implementation of A and B for which $|OP_1| + |OP_2| < d$.

Figure 3 shows possible admissible executions α_1 and α_2. Since no messages are received in σ_1 and σ_2 after time t, replacing process 2's history in σ_1 with its history in σ_2 results in another admissible execution, σ. Then $ops(\sigma)$ consists of the operations $op_1[A,1]$ followed by $op_2[B,1]$, and $op_1[B,2]$ followed by $op_2[A,2]$, where both pairs are preceded by $\alpha[A,1]$ and $\alpha[B,2]$.

By assumption, σ is sequentially consistent. Thus there exists a legal operation sequence τ in which the following hold:

- The operations in $\alpha[A,1]$ are followed by $op_1[A,1]$, and $op_1[A,1]$ is followed by $op_2[B,1]$.
- The operations in $\alpha[B,2]$ are followed by $op_1[B,2]$, and $op_1[B,2]$ is followed by $op_2[A,2]$.

Since $\alpha \circ op_1 \circ op_2$ is not legal, τ must have $op_1[A,1]$ follow $op_2[A,2]$. But that causes $op_2[B,1]$ to follow $op_1[B,2]$, which is not legal. ■

Cyclic dependences cause a pair of operations to be even slower than the previous lower bound.

Theorem 3 *Let T be an abstract data type with cyclically dependent generic operations OP_1 and OP_2. In any sequentially consistent implementation of an object of type T, $|OP_1| + |OP_2| \geq 2d$.*

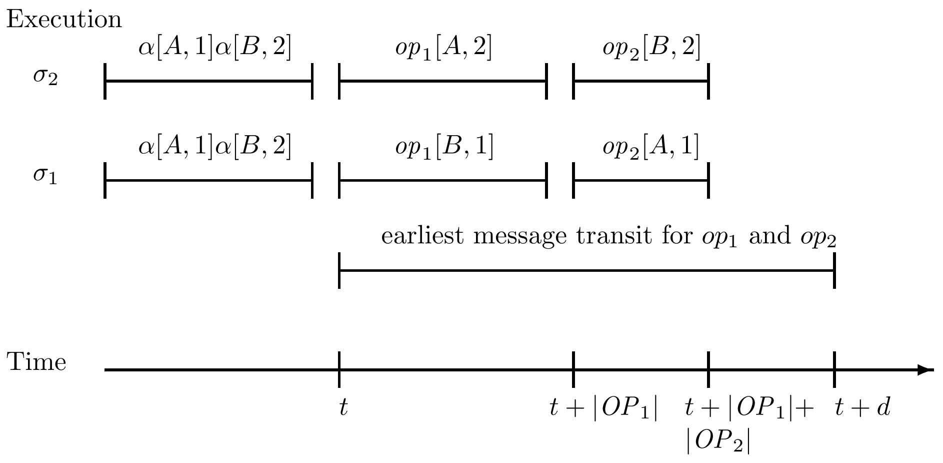


Figure 3: Counterexample for lower bound for a pair of operations that immediately do not commute (Theorem 2)

Proof Since OP_1 and OP_2 are cyclically dependent, there is a sequence of operations ρ and operation instances op_1 and op_2 such that $\rho \circ op_1$ and $\rho \circ op_2$ are legal, but $\rho \circ op_1 \circ op_2$ and $\rho \circ op_2 \circ op_1$ are not legal. Let A be an object of type T. Let processes 1 and 2 access A. Assume in contradiction that there exists a sequentially consistent implementation of A_ρ for which $|OP_1| + |OP_2| < 2d$. We can assume without loss of generality that $|OP_1| \geq |OP_2|$.

Figure 4 shows possible admissible executions α_1 and α_2. Since no messages are received in α_1 and α_2 before time d, replacing process 1's history in α_2 with its history in α_1 results in another admissible and sequentially consistent execution, α. Thus, there is a legal permutation of $ops(\alpha)$ for A_ρ. However, because of cyclic dependency, neither permutation of $ops(\alpha)$ is legal for A_ρ. ■

Dequeue and enqueue are not cyclically dependent, because an enqueue is always legal. Likewise, read and write are not cyclically dependent because a write is always legal. [5] shows that the lower bounds of Theorem 2 are tight.

We now give an example of a type, BANKACCOUNT2, with cyclically dependent operations. The object of the type consists of values denoting sav-

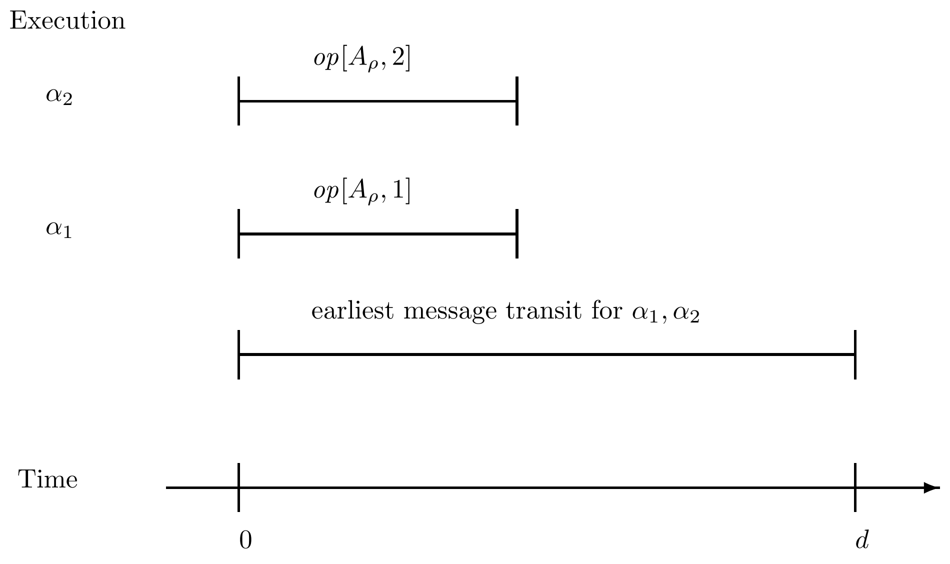


Figure 4: Counterexample for lower bound for a pair of cyclically dependent operations (Theorem 3)

ings and checking account balances. The operations are $RSUC$ and $RCUS$. $rsuc(v)(w)$ adds v to the checking account balance and returns the value of the savings account balance in w. $rcus(v)(w)$ adds v to the savings account balance and returns the value of the checking account balance in w. $RSUC$ and $RCUS$ are cyclically dependent. Initialize the checking account balance to 500 and the savings account balance to 1000. An *rsuc* instance executed alone would return 1000, and an *rcus* instance executed alone would return 500. If the *rcus* instance followed the *rsuc* instance, the *rcus* instance would not be legal, and vice versa.

We generalize cyclic dependence for larger groups of operations. If there exists an operation sequence α and operation instances $op_1, \ldots, op_n$ such that $\alpha \circ op_1$, ..., $\alpha \circ op_n$ are legal, but each operation sequence formed by α followed by a permutation of $op_1, \ldots, op_n$ is illegal, then $OP_1, \ldots, OP_n$ are *n-cyclically dependent*. The previous definition of cyclic dependence corresponds to 2-cyclic dependence.

Theorem 4 *Let T be an abstract data type with operations $OP_1, \ldots, OP_n$.*

If $OP_1, \ldots, OP_n$ are n-cyclically dependent, then one of $|OP_1|, \ldots, |OP_n|$ must be at least d in any sequentially consistent implementation of objects of type T, where at least n processes are using the objects.

Proof Let A be an object of type T. Let processes $1, \ldots, n$ access A. Since $OP_1, \ldots, OP_n$ are n-cyclically dependent, there exist an operation sequence ρ and operation instances $op_1, \ldots, op_n$ such that they can individually legally follow ρ. Assume in contradiction that there exists an implementation of A_ρ where $|OP_1| < d, \ldots, |OP_n| < d$, but they cannot all together legally follow ρ.

Figure 5 shows admissible executions $\alpha_1, \ldots, \alpha_n$. Since no messages are received in $\alpha_1, \ldots, \alpha_n$, for each $i > 1$, replacing p_i's history in α_1 with its history in α_i results in another admissible and sequentially consistent execution, α. Thus, there is a τ that is a permutation of $ops(\alpha)$ and is legal for A_ρ. However, no possible permutation of $ops(\alpha)$ is legal for A_ρ. ∎

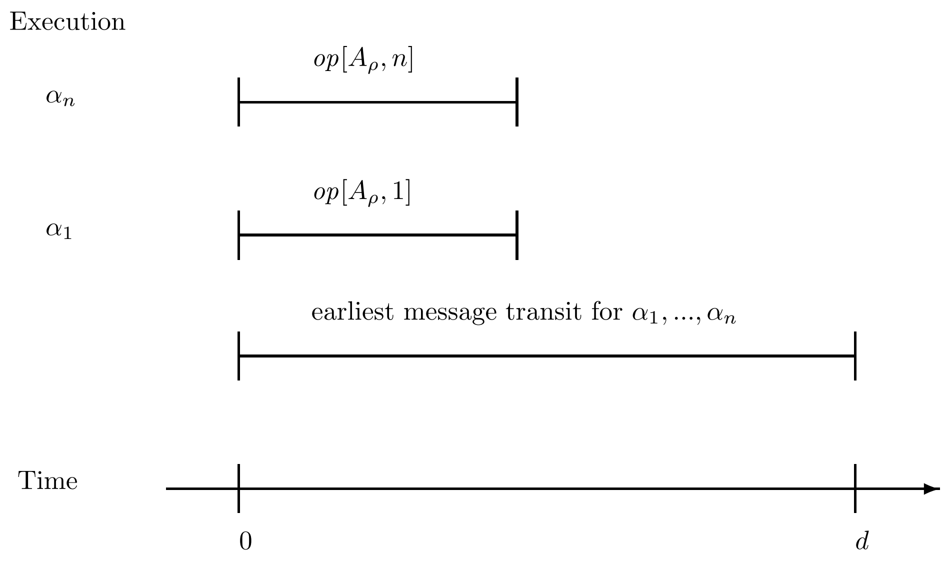


Figure 5: Counterexample for lower bound for n-cyclically dependent operations (Theorem 4)

We now give an example of a type, BANKACCOUNT3, with 3-cyclically dependent operations. The object of the type consists of values denoting savings, checking, and loan account balances. The operations are $RSUC$, $RCUL$, and $RLUS$. $rsuc(v)(w)$ adds v to the checking account balance and returns the value of the savings account balance in w. $rcul(v)(w)$ adds v to the loan account balance and returns the value of the checking account balance in w. $rlus(v)(w)$ adds v to the savings account balance and returns the value of the loan account balance in w. Initialize the savings account balance to 500, the checking account balance to 1000, and the loan account balance to 1500. An *rsuc* instance executed alone would return 500, an *rcul* instance executed alone would return 1000, and an *rlus* instance would return 1500. In any permutation of all three instances, at least one instance would be illegal if some account balance has changed.

The number of accessing processes is important. If only two processes access objects of type BANKACCOUNT3, there exists an implementation of BANKACCOUNT3 where $|RSUC| = d/2$, $|RCUL| = d/2$, and $|RLUS| = d/2$, matching the lower bound of $3d/2$ given in Theorem 8. The implementation described in Theorem 15 suffices.

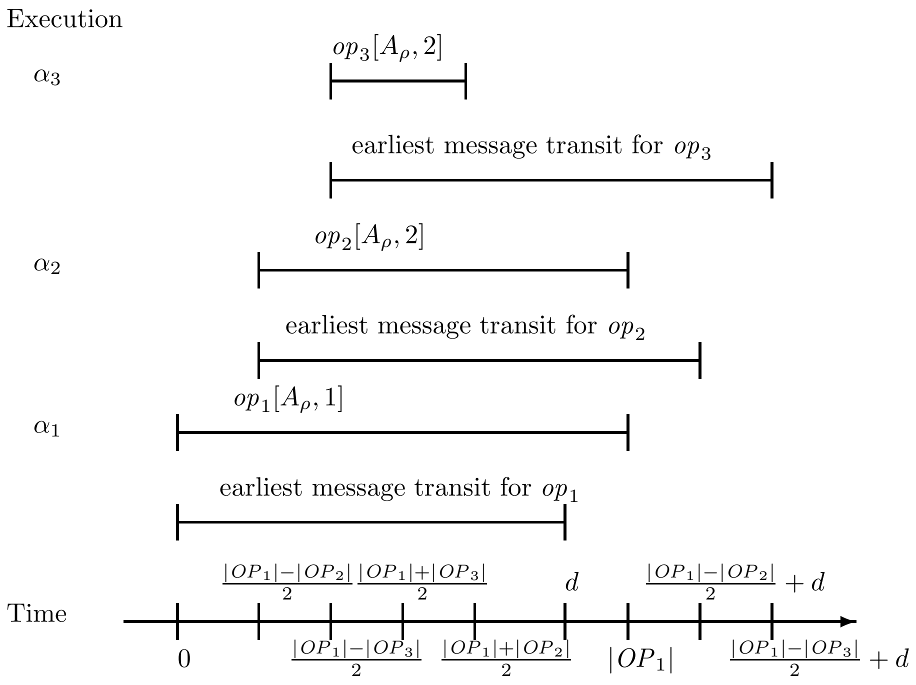
The next theorem gives a larger lower bound for 3-cyclic operations.

Theorem 5 *If T is an abstract data type with operations OP_1, OP_2, and OP_3 that are 3-cyclic, then $|OP_1| + |OP_2| + |OP_3| \geq 2d$ in any sequentially consistent implementation of objects of type T, where there are at least three accessing processes.*

Proof By Theorem 4, at least one of $|OP_1|$, $|OP_2|$, and $|OP_3|$ must be at least d. Without loss of generality, assume that $|OP_1| \geq d$. Assume in contradiction that $|OP_2| + |OP_3| < d$. We consider A_ρ, the ρ-initialized version of A.

Figure 3.1 shows admissible executions α_1, α_2, and α_3. Since no messages are received in α_1, α_2, and α_3 before time d, replacing process 1's history in α_3 with its history in α_1 and replacing process 2's history in α_3 with its history in α_2 results in another admissible and sequentially consistent execution α. Thus, there is a legal permutation of $ops(\alpha)$ for A_ρ. However, no possible permutation of $ops(\alpha)$ is legal for A_ρ. ■

We now define a condition causing either a pair of operations or an individual operation in a trio of operations to be slow. A generic operation OP is *noninterleavable with respect to OP_1 preceding OP_2* if there exist operation



Execution
$op_3[A_\rho, 2]$
α_3
earliest message transit for op_3
$op_2[A_\rho, 2]$
α_2
earliest message transit for op_2
$op_1[A_\rho, 1]$
α_1
earliest message transit for op_1
$\frac{|OP_1|-|OP_2|}{2}$
$\frac{|OP_1|+|OP_3|}{2}$
d
$\frac{|OP_1|-|OP_2|}{2}+d$
Time
0
$\frac{|OP_1|-|OP_3|}{2}$
$\frac{|OP_1|+|OP_2|}{2}$
$|OP_1|$
$\frac{|OP_1|-|OP_3|}{2}+d$

Figure 6: Counterexample for lower bound for 3-cyclically dependent operations (Theorem 5)

sequence ρ and operation instances op, op_1, and op_2 such that $\rho \circ op$ and $\rho \circ op_1 \circ op_2$ are legal, but none of $\rho \circ op \circ op_1 \circ op_2$, $\rho \circ op_1 \circ op \circ op_2$, and $\rho \circ op_1 \circ op_2 \circ op$ is legal.

Theorem 6 *Let T be an abstract data type, and let OP_1, OP_2, and OP_3 be generic operations of T such that OP_3 is noninterleavable with respect to OP_1 preceding OP_2. Then in any sequentially consistent implementation of objects of type T, $|OP_1| + |OP_2| \geq d$ or $|OP_3| \geq d$.*

Proof Since OP_3 is noninterleavable with respect to OP_1 preceding OP_2, there exists an operation sequence ρ and operation instances op_1, op_2, and op_3 such that $\rho \circ op_3$ and $\rho \circ op_1 \circ op_2$ are legal, but none of $\rho \circ op_3 \circ op_1 \circ op_2$, $\rho \circ op_1 \circ op_3 \circ op_2$, and $\rho \circ op_1 \circ op_2 \circ op_3$ is legal.

Let A be an object of type T. Let processes 1 and 2 access A. Assume in contradiction that there exists a sequentially consistent implementation of A_ρ for which $|OP_1| + |OP_2| < d$ and $|OP_3| < d$.

Figure 7 shows admissible executions α_1 and α_2. Since no messages are received in α_1 and α_2, replacing process 1's history in α_2 with its history in α_1 results in another admissible and sequentially consistent execution, α. Thus, there is a permutation of $ops(\alpha)$, τ, which preserves the order of process 1's operations in α and is legal for A_ρ. However, because of noninterleavability, none of the three permutations of $ops(\alpha)$ that satisfy the order of process 1's operations is legal for A_ρ. ■

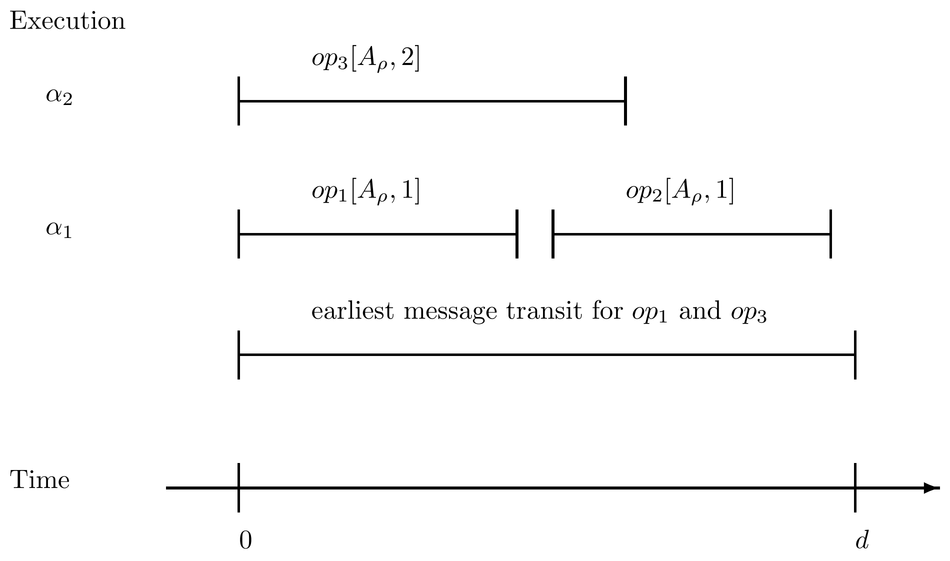


Figure 7: Counterexample for noninterleavability lower bound (Theorem 6)

Consider our BANKACCOUNT2 objects, and define the additional operations UC (which adds a certain amount to the checking account balance) and RS (which reads and returns the value of the savings account balance). $RCUS$ is noninterleavable with respect to UC preceding RS. Initialize the checking account balance to 500 and the savings account balance to 1000. An *rcus* instance executed alone would return 500, and an *rs* instance following

Table 1: Corollaries for Subsection 3.1

Operations	Type/table references	Theorems	Citations
DEQ	Table 5	1	[5]
POP	Table 6	1	[5]
DEL	Table 7	1	
WITHDRAW	Table 8	1	
READ, WRITE	Read/write objects	2	[5, 15]
Any two distinct generic operations	Tables 5–8	2	
RSUC, RCUS	BANKACCOUNT2	3	
RSUC, RCUL, RLUS	BANKACCOUNT3	4, 5	
RCUS, UC, RS	BANKACCOUNT2	6	

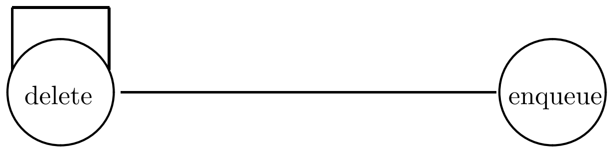
a *uc* instance would return 1000. Including all three instances would make either the *rcus* or the *rs* illegal, depending on where the *rcus* instance was placed.

Table 1 displays types applicable to the theorems in this subsection. For each row of the table, the operation from the first column, which is from the type or table reference listed in the second column, meets the hypotheses for the theorem(s) listed in the third column. Any references to a specific lower bound corresponding to the given operation are presented in the fourth column.

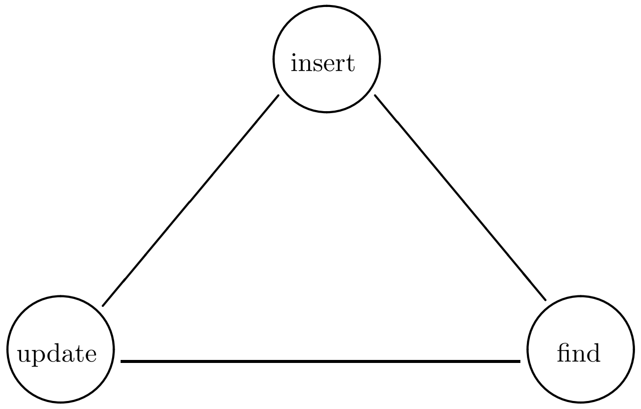
3.2 Lower bounds for sequential consistency for all operations of a type

We now use the previous results and the structure of the commutativity properties to deduce lower bounds on the worst-case time complexity for *all* operations of abstract data types in sequentially consistent implementations.

An alternate way to represent the commutativity properties of an abstract data type T is to use a *commutativity graph* $CG(T)$, with the generic operations as nodes. An edge exists between two nodes if their corresponding operations immediately do not commute. A loop exists at a node if the corresponding operation immediately does not commute with itself. Figure 8 shows some examples of commutativity graphs.



(a) Commutativity graph for queue operations



(b) Commutativity graph for reference-count set operations

Figure 8: Examples of commutativity graphs

$NC(T)$ is the subset of nodes in $CG(T)$ such that each node's corresponding operation immediately does not commute with itself. $RCG(T)$ (the reduced commutativity graph for T) is the subgraph of $CG(T)$ formed by deleting all nodes in $NC(T)$ and their incident edges. $Maxdom(RCG(T))$ is a subgraph of $RCG(T)$ formed by a maximum independent edge dominating set of $RCG(T)$. Notice that a maximum independent edge dominating set of a graph is a largest subset of edges of the graph such that distinct edges in the subset do not have nodes in common and all other edges in the graph have a node in common with one of the edges in the set. It is clear that the size of a maximum independent edge dominating set is at most half the number of nodes in the graph.

This result gives lower bounds on the costs of implementing abstract data types with general commutativity graphs.

Theorem 7 *Let T have operations $OP_1, OP_2, \ldots, OP_n$ and commutativity graph $CG(T)$. In any sequentially consistent implementation of T,*

$$\sum_{i=1}^{n} |OP_i| \geq (|NC(T)| + |Maxdom(RCG(T))|)d.$$

Proof If OP_i immediately does not commute with itself, then $|OP_i| \geq d$ by Theorem 1. Thus, $\sum_{OP_i \in NC(T)} |OP_i| \geq |NC(T)|d$. Let (OP_i, OP_j) be an edge in $Maxdom(RCG(T))$. By the definition of $Maxdom(RCG(T))$, OP_i and OP_j immediately do not commute. By Theorem 2, $|OP_i| + |OP_j| \geq d$. Adding these inequalities yields the desired result. ■

We now give a lower bound on the time required for all operations of an abstract data type with a *clique* in its commutativity graph.

Theorem 8 *Let T have operations $OP_1, OP_2, \ldots, OP_n$ such that for all $i \in \{1, \ldots, n\}$, OP_i immediately does not commute with OP_j if $i \neq j$. In any sequentially consistent implementation of T, $\sum_{i=1}^{n} |OP_i| \geq |NC(T)|d + (n - |NC(T)|)d/2$.*

Proof Let $s = |NC(T)|$. Let $OP_{i_1}, \ldots, OP_{i_s}$ be the operations that immediately do not commute with themselves. By Theorem 1, $|OP_{i_k}| \geq d$ for all k in $\{1, \ldots, s\}$. Thus, $\sum_{k=1}^{s} |OP_{i_k}| \geq sd$. Let $OP_{j_1}, \ldots, OP_{j_{n-s}}$ be the remaining operations. We can assume without loss of generality that $t = n - s > 2$, because Theorem 2 handles the $t = 2$ case. Since OP_{j_a} and OP_{j_b} do not commute for all a, b in $\{1, \ldots, t\}$ where $a \neq b$, by Theorem 2, we get a system of $C(t, 2) = t(t-1)/2$ inequalities of the form $|OP_{j_a}| + |OP_{j_b}| \geq d$, where $a \neq b$.[6]

We want to minimize $\sum_{k=1}^{t} |OP_{j_k}|$ given the above constraints and the constraints $|OP_{j_k}| \geq 0$ for all $k \in \{1, \ldots, t\}$. We want to minimize the sum,

[6] $C(t, 2)$ means the number of ways to choose two distinct items from a set of t items.

because we want the the sum of each $|OP_{j_a}|$ and $|OP_{j_b}|$ in the global sum to be as close as possible to d from the corresponding individual constraint. If the system of equations of the form $|OP_{j_a}| + |OP_{j_b}| = d$ has a solution, the solution would yield a lower bound for $\sum_{k=1}^{t} |OP_{j_k}|$, because each $|OP_{j_k}|$ is nonnegative. Let us use the system of equations. Without loss of generality, we consider the following:

- $|OP_{j_1}| + |OP_{j_2}| = d.$
- $|OP_{j_1}| + |OP_{j_3}| = d.$
- $|OP_{j_2}| + |OP_{j_3}| = d.$

We can subtract the third equation from the second to yield the equation $|OP_{j_1}| - |OP_{j_2}| = 0$. Adding the new equation to the first yields $2|OP_{j_1}| = d$. We now get that $|OP_{j_1}| = d/2$. It easily follows that $|OP_{j_k}| = d/2$ for all $k \in \{1, \ldots, t\}$. Thus, $\sum_{k=1}^{t} |OP_i| = \sum_{k=1}^{s} |OP_{i_k}| + \sum_{k=1}^{t} |OP_{j_k}| \geq (n-s)d/2$, as desired. ■

Most of our lower bounds do not restrict the costs of individual operations, provided that certain sums of costs exceed the lower bounds. Let us consider implementations of abstract data types where each operation takes either time 0 or time d. Thus each implementation corresponds to a labeling of the commutativity graph where each node is labeled with a 0 or a d, a *0-d assignment*. The total cost of the implementation equals d multiplied by the number of d's in the labeling. We now show that the the minimum number of d's required in a labeling is the size of a minimum vertex cover (VC).[7]

Theorem 9 *Let T have operations $OP_1, \ldots, OP_n$. In any sequentially consistent implementation of T corresponding to a 0-d assignment,*

$$\sum_{i=1}^{n} |OP_i| \geq |VC(CG(T))|d.$$

Proof Assume in contradiction that there exists a sequentially consistent implementation of T corresponding to a 0-d assignment such that

$$\sum_{i=1}^{n} |OP_i| < |VC(CG(T))|d.$$

[7] A minimum vertex cover is a smallest subset of nodes of a graph such that each edge in the graph has an endpoint contained in the subset.

Table 2: Corollaries for Subsection 3.2

Type/Table reference	Theorems	$\lvert Maxdom \rvert$	$\lvert NC \rvert$	n	$\lvert VC \rvert$
Table 9	7, 9	1			2
Table 9 + delete operation	8		2	5	

If (OP_i, OP_j) is an edge in $CG(T)$, then OP_i and OP_j immediately do not commute by the definition of $CG(T)$. Thus $|OP_i| + |OP_j| \geq d$ by Theorem 2, implying that at least one of OP_i and OP_j must be labeled with d in the 0-d assignment. Let S be the set of operations labeled with d. $|S| < |VC(CG(T))|$. Since $VC(CG(T))$ is a smallest set of nodes that cover the edges of $CG(T)$, S does not cover the edges. Thus there exists an edge (OP_i, OP_j) in $CG(T)$ such that OP_i and OP_j are not in S, that is, they are labeled with 0 in the 0-d assignment. By the definition of $CG(T)$, OP_i and OP_j immediately do not commute, but $|OP_i| + |OP_j| = 0$, violating Theorem 2. ■

Table 2 displays types applicable to the theorems in this subsection. For each row of the table, the type referenced in the first column meets the hypotheses of the theorem(s) listed in the second column, with commutativity graph properties as shown in the remaining columns.

3.3 Upper bounds for linearizability

We know various algebraic properties of operations to prove lower bounds on their worst-case completion times in linearizable implementations. Now we show that the lower bounds are tight. First we explain how certain classes of operations and individual operations can be made fast. Then we present types where the sum of the worst-case completion times for all operations is optimized. Finally we discuss the effects of the results on the relative costs of sequential consistency and linearizability.

3.3.1 Optimizing a class of operations or a single operation

The worst-case completion time of an operation is at least d if it immediately does not commute with itself. Given an operation that immediately commutes with itself, can we implement it so that it performs only local computation, for which the time is assumed to be negligible compared to the message delay in the communication network?

We demonstrate three algebraic properties allowing an operation or group of operations to be made fast. Making a single operation from an abstract data type fast has proved to be a nontrivial task. Although we have not quite developed an exact characterization of when a single operation can be made fast, we have determined a property (self-obliviousness) that is sufficiently general in allowing a single operation to be made fast. Any operation that immediately does not commute with itself cannot be made fast. If an operation is noninterleavable with respect to a pair of operations, then some slowdown must occur (either the individual operation or the pair). A lack of self-obliviousness is related to both of these properties. If an operation immediately does not commute with itself, then it is not self-oblivious. If an operation is not self-oblivious, then it may be noninterleavable with respect to another operation and itself.

We now define more properties of generic operations. Although the properties are noted in the object-oriented programming literature, we are not aware of any formal definitions like ours. Let α and β be arbitrary operation sequences. If the legality of $\alpha \circ \mathit{aop} \circ \beta$ implies the legality of $\alpha \circ \beta$ for any instance *aop* of generic operation AOP and if the legality of $\alpha \circ \beta$ implies the legality of $\alpha \circ \mathit{aop}^* \circ \beta$ for some instance *aop* of generic operation AOP (where op^* denotes 0 or more copies of *op*), then AOP is an *accessor*. Informally, an accessor does not change an object's state. *FIND* for reference-count sets (Table 9) is an accessor. MOP is a *mutator* if there exist operation sequences α and β such that $\alpha \circ \mathit{mop} \circ \beta$ is legal but $\alpha \circ \beta$ is not legal for some instance *mop* of MOP. Informally, a mutator changes an object's state, and this change can be detected. *INS* and *UP* for reference-count sets are mutators.

For any abstract data type, we can always implement accessor operations such that they only perform local computation. Other operations take time d in the implementation.

Theorem 10 *Let T be an abstract data type. If T has generic accessor operations $AOP_1, \ldots, AOP_n$, then there exists a linearizable implementation of objects of type T where $|AOP_1| = \cdots = |AOP_n| = 0$ and $|MOP_1| = \cdots = |MOP_m| = d$ for all other generic operations $MOP_1, \ldots, MOP_m$.*

Proof We exhibit an implementation where $|AOP_1| = \cdots = |AOP_n| = 0$

and $|MOP_1| = \cdots = |MOP_m| = d$. Each process keeps a copy of all objects in its local memory. When an aop_i is invoked at process p, p performs the operation locally and returns the result from the operation. When a mop_j operation on object X is invoked at process p, p sends a message $DoMop_j(X)$ with the argument list for the operation to all processes (including itself), waits d time, and returns the result of performing the operation. When a process receives any form of *DoMop* message, it performs the operation on X in its local memory. If the message was sent by that process, it saves the result so that the process can return it. We can break ties in the following way: $DoMop_k$ is handled before $DoMop_l$ if $k < l$, and we use process identifiers to break any remaining ties.

Our correctness proof is very similar to a proof in [5] for read/write objects. Let ρ be an admissible execution. We systematically construct the desired τ. Each operation in ρ *occurs* at the time of its response. Let τ be the sequence of operations in ρ ordered by the times of occurrence, breaking ties by placing *mop* operations before *aop* operations, aop_k before aop_l if $k < l$, mop_k before mop_l if $k < l$, and using process identifiers to break any remaining ties. By construction, $\rho|p = \tau|p$ for all p, and τ preserves the relative ordering of nonoverlapping operations.

We now must show that τ is legal or that, for each object X, $\tau|X$ is in the sequential specification of T. Consider an accessor operation. It returns based on its local state. Its local state reflects all changes made by mutators occurring up to the time of the accessor in ρ. Thus the accessor operation returns a legal value list in τ.

Consider a mutator operation that returns a value list. It returns based on its local state at its response time in ρ. Its local state reflects all changes made by mutators occurring up to the time of the mutator. Let the sequence of mutators occurring up to the time of the mutator in ρ be α. The mutator is legal after α, which is a subsequence of τ. Any accessors interleaved with α in τ do not affect the legality of the mutator. Thus the mutator operation returns a legal value list in τ. ■

We now define a restriction of mutator operations. If the legality of arbitrary operation sequence α implies the legality of $\alpha \circ mop$ for any instance *mop* of generic mutator operation *MOP*, then *MOP* is a *pure mutator*. Informally, the return value lists of pure mutators do not depend on the states of the objects on which they are invoked. A push operation for an unbounded stack is a pure mutator.

For any abstract data type, we can make pure mutator operations fast. Other operations take time d in the implementation.

Theorem 11 *Let T be an abstract data type. If T has generic pure mutator operations $MOP_1, \ldots, MOP_n$, then there exists a linearizable implementation of objects of type T where $|MOP_1| = \cdots = |MOP_n| = 0$ and $|OP_1| = \cdots = |OP_m| = d$ for all other generic operations $OP_1, \ldots, OP_m$.*

Proof We exhibit an implementation where $|MOP_1| = \cdots = |MOP_n| = 0$ and all other operations $(OP_1, \ldots, OP_m)$ take time d. Each process keeps a copy of all objects in its local memory. When a mop_i is invoked at process p, p sends a message $DoMop_i(X)$ with the argument list for the operation to all processes (including itself) and returns immediately. When a process receives any form of *DoMop* message, it performs the operation on X in its local memory. When an op_j on object X is invoked at process p, p sends a message $DoOp_j(X)$ with the argument list for the operation to all processes (including itself), waits d time, and returns the result of performing the operation. When a process receives any form of *DoOp* message, it performs the operation on X in its local memory. If the message was sent by that process, it saves the result so that the process can return it.

We can break ties in the following way: *DoMop* messages are handled before *DoOp* messages, $DoMop_k$ is handled before $DoMop_l$ if $k < l$, $DoOp_k$ is handled before $DoOp_l$ if $k < l$, and we use process identifiers to break any remaining ties.

The correctness proof is very similar to a proof in [5] for read/write objects. Let ρ be an admissible execution. We systematically construct the desired τ. Each operation in ρ *occurs* time d after the time of its call. Let τ be the sequence of operations in ρ ordered by the times of occurrence, breaking ties by placing *mop* operations before *op* operations, mop_k before mop_l if $k < l$, op_k before op_l if $k < l$, and by using process identifiers to break any remaining ties. By construction, $\rho|p = \tau|p$ for all p, and τ preserves the relative ordering of nonoverlapping operations.

We now must show that τ is legal or that, for each object X, $\tau|X$ is in the sequential specification of T. All *mop* operations are legal because they are pure mutators. Consider an op_k that returns a value list. It returns based on its local state at its response time in ρ. Its local state reflects all changes made by operations occurring up to the time of the op_k in ρ. Thus the op_k returns a legal value list in τ. ■

Corollary 12 *There exists a linearizable implementation of increment-half objects (Table 10), where* $|READ| = d$, $|INC| = 0$, *and* $|HALF| = 0$.

Accessors and pure mutators can be (separately) optimized. Can we optimize a self-commuting operation that both accesses and modifies the states of the objects on which it is invoked?

We must be careful, because two fast operation instances do not know about each other if they are executed less than time d apart. To optimize generic operation FOP, we show it is sufficient for FOP to be *self-oblivious*. Intuitively, this means that a *fop* operation instance does not indirectly affect another *fop* operation instance.

Let $\alpha_1, \alpha_2, \ldots$ be operation sequences. FOP is self-oblivious if, whenever $\alpha_1 \circ fop^1, \alpha_1 \circ \alpha_2 \circ fop^2, \ldots, \alpha_1 \circ \alpha_2 \circ \cdots \circ \alpha_n \circ fop^n, \ldots$ are legal, there exists an instantiation of return values for the operations in α_i for each $i \geq 1$, creating new sequences of operations α'_i for each $i \geq 1$, such that $\alpha'_1 \circ fop^1 \circ \alpha'_2 \circ fop^2 \circ \cdots \circ \alpha'_n \circ fop^n \cdots$ is legal. It is assumed that no α_i contains a *fop* instance. An operation that immediately does not commute with itself is not self-oblivious, because the α's can be empty, with the possible exception of α'_1.

What kinds of operations are self-oblivious? Accessors are self-oblivious because they do not change object state information. A pure mutator is self-oblivious because it is always legal after a legal sequence of operations. However, determining whether an arbitrary operation is self-oblivious involves the semantics for all operations of its type. An operation that immediately commutes with itself is self-oblivious if all other operations of its type do not perform conditional updates (whether and how to perform updates are based on object state information).

For read/write objects, reads and writes are self-oblivious. Reads are self-oblivious because they are accessors. We now show how writes satisfy the self-oblivious definition. Each write in an operation sequence is legal. If $\alpha_1 \circ write^1, \alpha_1 \circ \alpha_2 \circ write^2, \ldots, \alpha_1 \circ \alpha_2 \cdots \alpha_n \circ write^n, \ldots$ are legal, then $\alpha_1 \circ write^1 \circ \alpha'_2 \circ write^2 \cdots \alpha'_n \circ write^n$ is legal, where the return value for the reads in α'_i for $i \geq 2$ is the value written by $write^{i-1}$.

For unbounded queues and stacks, enqueues and pushes are self-oblivious. We now show how enqueues satisfy the self-oblivious definition. The argument for pushes is analogous. If $\alpha_1 \circ \mathit{enqueue}^1, \alpha_1 \circ \alpha_2 \circ \mathit{enqueue}^2, \ldots, \alpha_1 \circ \alpha_2 \cdots \alpha_n \circ \mathit{enqueue}^n, \ldots$ are legal, then $\alpha_1 \circ \mathit{enqueue}^1 \circ \alpha'_2 \circ \mathit{enqueue}^2 \cdots \alpha'_n \circ \mathit{enqueue}^n$ is legal, where the return value for the first dequeue returning $\perp$ in α_i for $i \geq 2$ is changed in α'_i to the value enqueued by $\mathit{enqueue}^{i-1}$.

For BANKACCOUNT2 objects, $RSUC$ and $RCUS$ are self-oblivious, because each immediately commutes with itself and does not perform conditional updates. We now show how $RSUC$ satisfies the self-oblivious definition. The argument for $RCUS$ is analogous. An $RSUC$ operation in an operation sequence is legal if the value returned is the value written by the last $RCUS$ preceding the $RSUC$ in the sequence. If $\alpha_1 \circ \mathit{rcus}^1, \alpha_1 \circ \alpha_2 \circ \mathit{rcus}^2, \ldots, \alpha_1 \circ \alpha_2 \cdots \alpha_n \circ \mathit{rcus}^n, \ldots$ are legal, then $\alpha_1 \circ \mathit{rcus}^1 \circ \alpha'_2 \circ \mathit{rcus}^2 \cdots \alpha'_n \circ \mathit{rcus}^n$ is legal, where the return value for the $RSUC$ operations in α'_i for $i \geq 2$ is the value written by rcus^{i-1}.

Theorem 13 *Let T be an abstract data type with a self-oblivious generic operation SELFOP. There exists a linearizable implementation of objects of type T where $|SELFOP| = 0$ and $|OP| = 2d$ for all other generic operations OP.*

We instantiate the algorithm in Figure 9 to yield our implementation of objects of type T. Each assignment statement is executed locally. Each process keeps both an actual copy and a scratch copy of each object. In addition, each process maintains an ordered set of message slots, where each slot is indexed by a time. A slot holds messages that are either sent or received at a given time.

1. au: In Figure 9, Is everything italic that should be? Also, check capitalization and punctuation. Put periods at ends of sentences?

When a *selfop* operation is invoked, the invoking process sends a message about the operation to all other processes, determines the return value list (possibly by updating its scratch copy of the object based on messages it has received), and returns. When an instance of another generic operation is invoked, the invoking process sends a message about the operation to all other processes.

Messages about *selfop* have a higher priority than messages about other operations. If a message about a self-oblivious operation is received at time t, then it is placed in slot $t - d$. If any other message is received at time t, then it is placed in slot t.

```
$fastop(args)[X, p]$:
$scratch_X := X$
Handle all unhandled message in slots up to (current time)
about $X$ in message slot order,
updating $scratch_X$ as necessary
Determine return value based on $scratch_X$
Send handle-fastop($X$, $args, p$) to all processes
$fastopret(returnvalue)[X, p]$

$otherop(args)[X, p]$:
send handle-otherop($X$, $args, p$) to all processes

receive a message:
if it is a handle-fastop, then
  Insert in slot (current time $- d$), tiebreaking after other
  message types and then by process id
else /* handle-otherop message */
  Insert in slot (current time) with a fixed tiebreaking
  order among message types and then by process id
  timerset($p, d$)

when timer goes off:
for each unhandled message in slots up to (current time $- d$),
considered in message slot order, do:
  Use it to update the actual copy of its object
  if it is not a handle-fastop message, then
  /* it is of the form handle-otherop($X$, $args, q$) */
    if $p = q$, then
      Determine return value based on $X$
      $otheropret(return\ value)[X, p]$
  Mark message as handled
```

Figure 9: Algorithm for optimizing one operation: Code for process p

If a received message is not about a *selfop* operation, then a timer for d later is set. When a timer goes off, all messages are handled that are in slots indexed by time up to d before the current time, updating actual copies of objects as necessary. At this time, if the process has a pending operation for which its message has been handled, the process completes its pending operation. The timers and slots are also used to give *selfop* higher priority than other operations because of its need to complete quickly.

Proof We must prove that the implementation guarantees linearizable executions. Let ρ be an admissible execution. To form τ, we place all operations in order according to their message slots and, if their message slots contain multiple messages, according to their positions in the message slots.

First, we show that τ preserves the relative ordering of nonoverlapping operations in ρ. Suppose that op_1 precedes op_2 in ρ; that is, t_1, the finishing time for op_1, is less than t_2, the starting time for op_2. We must analyze the ordering based on the classification of op_1 and op_2. Because op_1 may or may not be a *selfop* operation and op_2 may or may not be a *selfop* operation, we have four cases to check:

a. op_1 and op_2 are *selfop* operations. The starting time for op_1 is t_1. op_1's message slot is $(t_1+d)-d = t_1$, and op_2's message slot is $(t_2+d)-d = t_2$. We have $t_1 < t_2$, and thus op_1 is correctly placed before op_2 in τ.

b. op_1 is a *selfop* operation, but op_2 is not a *selfop* operation. As before, op_1's message slot is t_1, and op_2's message slot is $t_2 + d$. We have $t_1 < t_2$, and so $t_1 < t_2 + d$. Thus, op_1 is correctly placed before op_2 in τ.

c. op_1 are op_2 are not *selfop* operations. The starting time for op_1 is $t_1 - 2d$. op_1's message slot is $(t_1 - 2d) + d = t_1 - d$, and op_2' s message slot is $t_2 + d$. We have $t_1 < t_2$, and so $t_1 - d < t_2 + d$. Thus, op_1 is correctly placed before op_2 in τ.

d. op_1 is not a *selfop* operation, but op_2 is. As before, op_1's message slot is $t_1 - d$, and op_2's message slot is $(t_2 + d) - d = t_2$. We have $t_1 < t_2$, and so $t_1 - d < t_2$. Thus, op_1 is correctly placed before op_2 in τ.

We now prove the legality of $\tau = op_1 \circ op_2 \circ \cdots \circ op_n \cdots$ by induction. We must first show that op_1 is legal. Suppose op_1 was performed on object X.

If op_1 is a *selfop* operation, then op_1's return value is based on the sequence of operations that are executed on $scratch_X$ at op_1's process by the time op_1 is invoked in ρ. Since op_1 is the first operation placed, it is in the earliest time slot. Thus we have an empty sequence of operations that are executed on $scratch_X$ at op_1's process by the time op_1 is invoked in ρ. op_1 is legal, because the return value is based on the semantics of op_1's abstract data type. If op_1 is not a *selfop* operation, then op_1's return value is based on the sequence of operations that are executed at op_1's process in ρ by the time op_1 returns in ρ. As before, op_1 is legal.

The induction step comes next. Suppose that $op_1 \circ \cdots \circ op_k$ is legal. We must prove that $op_1 \circ \cdots \circ op_k \circ op_{k+1}$ is legal. We have two cases to consider. Either op_{k+1} is a *selfop* operation, or it is not. Suppose that op_{k+1} is performed on object X.

Case 1. If op_{k+1} is a *selfop* operation, then its return value is based on the sequence of operations that are executed on $scratch_X$ at op_{k+1}'s process by the time op_{k+1} is invoked in ρ. We must now analyze the structure of $op_1 \circ \cdots \circ op_k$ to continue. $op_1 \circ \cdots \circ op_k$ has no *selfop* operation, or the sequence has at least one *selfop* operation.

Case 1.1. If $op_1 \circ \cdots \circ op_k$ has no *selfop* operation, then the message slots of $op_1, \ldots, op_k$ are earlier than or at the same time as op_{k+1}'s message slot. Thus, $op_1, \ldots, op_k$ have been performed, or their messages have been received at op_{k+1}'s process, implying that they have been performed on $scratch_X$ (if they use X at all). $scratch_X$ reflects the work done by $op_1, \ldots, op_k$. Thus, op_{k+1} is legal, because it was performed based on a legal sequence of operations and the semantics of op_{k+1}'s abstract data type.

Case 1.2. Let op_l be the last *selfop* operation that starts before op_{k+1}. Either op_l starts more than time d before op_{k+1} or op_l starts less than time d before op_{k+1}.

Case 1.2.1. If op_l is performed on X, then $scratch_X$ reflects the change to X, implying that op_{k+1} is legal.

Case 1.2.2. Consider the operations that start less than time d before op_{k+1} and are placed before op_{k+1} in τ. The time slot for any operation op that is not a *selfop* is $t_{op} + d$, where t_{op} is the starting time for op. The time slot for op_{k+1} is $t_{op_{k+1}}$, its starting time, since op_{k+1} is a *selfop*. Because $t_{op_{k+1}} - t_{op} < d$, we have $t_{op_{k+1}} < t_{op} + d$. Thus, op is placed after op_{k+1} in τ. This means that any operation starting less than time d before op_{k+1} and placed before op_{k+1} in τ is a *selfop*. In particular, op_k is a *selfop*. Let

op_f be the first *selfop* starting less than time d before op_{k+1} and placed before op_{k+1}. $op_f, \ldots, op_{k+1}$ are all *selfop* operations. By the induction hypothesis, $op_1 \cdots op_k$ is legal. Let $\alpha_1 = op_1 \cdots op_{f-1}$. $\alpha_1 \circ op_f$ is legal, $\alpha_1 \circ op_{f+1}$ is legal, $\ldots, \alpha_1 \cdot op_k$ is legal, and $\alpha_1 \circ op_{k+1}$ is legal, because the processes performing $op_f, \ldots, op_k, op_{k+1}$ have received information about all operations in α_1. Each operation is performed based on a legal sequence of operations and the semantics of the operation's abstract data type. We now use the self-oblivious property. Let each of $\alpha_2, \ldots, \alpha_{k-f+2}$ be the empty sequence. The largest subscript on an α_i is $k-f+2$, because we need to place $op_f, \ldots, op_{k+1}$, a total of $k-f+2$ operations. $\alpha_1 \circ op_f \circ \alpha_2 \circ op_{f+1} \cdots \alpha_{k-f+1} \circ op_k \circ \alpha_{k-f+2} \circ op_{k+1}$ is legal by the definition.

Case 2. If op_{k+1} is not a *selfop* operation, then its return value is based on the sequence of operations that execute at op_{k+1}'s process in ρ by the time op_{k+1} returns in ρ. This sequence is a prefix of τ, because the execution order of operations is the message slot ordering. By the induction hypothesis, the sequence is legal. Since op_{k+1} is executed based on a legal sequence of operations and the semantics of op_{k+1}'s abstract data type, it is legal. ■

In our implementation optimizing a self-oblivious operation, all nonoptimized operations have a worst-case response time of $2d$. Our implementation assumes that the optimized operation has cyclic dependences with all other operations. Thus, for certain abstract data types, there are gaps between our upper bound and the lower bounds.

3.3.2 Types having a tight lower bound for all operations

We now exhibit some abstract data types with implementations in which the total time for *all* operations matches the lower bounds from Subsection 3.2. Minimizing the total time required for all operations may help when the frequencies of invoking each operation are approximately equal.

A pure modify-read (PMR) object is a variable X that can be read or modified by a pure mutator operation. This is a generalization of the pseudo–read-modify-write (PRMW) object, because the pure mutator operation may be such that it eventually does not commute with itself. (A PRMW object is a variable that can be read, written, or modified by a pure mutator operation that is a commutative arithmetic operation; see [3].)

There exists a linearizable implementation of PMR objects with the total worst-case response time for all operations matching the lower bound on the total worst-case response time for all operations. In the implementation, the pure mutators take time 0, and the read operation takes time d.

Theorem 14 *For any set of PMR objects with operations READ and pure mutator operations $MOP_1, \ldots, MOP_n$, there exists a linearizable implementation of the set with $|READ| = d$ and $|MOP_1| = \cdots = |MOP_n| = 0$.*

Proof Since all nonread operations are pure mutators, we can instantiate the implementation described in Theorem 11 to yield an implementation achieving the desired time bounds. ■

An n-component array object is an array of n items with each item capable of being read and/or written by some operation. For example, $R1W2R3(v_2)(v_1, v_3)$ reads v_1 from item 1, writes v_2 to item 2, and reads v_3 from item 3. Thus, we combine the call and response events, as described in Section 2. Any operation in which the items being read and the items being written are disjoint sets is easily seen to be immediately self-commuting, and any operation that is not immediately self-commuting must read and write the same item. If OP_1 reads item i and OP_2 writes item i, then OP_1 and OP_2 immediately do not commute.

We now exhibit some implementations of component array data types without n-cyclic dependences that match the lower bounds given earlier.

Theorem 15 *For any component array data type T with immediately self-commuting operations $OP_1, \ldots, OP_n$ and no k-cyclic dependences (for each $k \geq 2$), there exists a linearizable implementation of T such that $\sum_{i=1}^{n} |OP_i| = nd/2$.*

Proof We exhibit an implementation where $|OP_1| = \cdots = |OP_n| = d/2$. We need our assumption of no k-cyclic dependences here because of Theorem 3. Each process keeps a copy of all objects in its local memory. When an op_i on object X is invoked at process p, p sends a message $DoOp_i(X)$ with the argument list for the operation to all processes (including itself), waits $d/2$ time, and returns the result of performing the operation. When a process receives any form of $DoOp$ message, it performs the operation on the appropriate object in its local memory. We can break ties in the following way: $DoOp_l$ is handled before $DoOp_m$ if op_l reads from a component that

op_m writes. We use process identifiers to break any remaining ties. The assumption about no k-cyclic dependences always allows ties to be broken.

We now prove that this algorithm is correct. Let ρ be an admissible execution. We systematically construct τ. Each operation that does not write occurs at its response time. Every other operation occurs $d/2$ time after its response time.

Let τ be the sequence of operations in ρ ordered by times of occurrence. We break ties by placing op_l before op_m if op_l reads a component that op_m writes and by using process identifiers to break any remaining ties. By construction, $\rho|p = \tau|p$ for all p, and τ preserves the relative ordering of nonoverlapping operations.

We must now show that $\tau = op_1 \circ op_2 \circ \cdots$ is legal. Consider op_i, which returns based on the local state of its process $d/2$ time after its invocation time, t. The local state reflects all changes made by other operations starting by time $t - d/2$. Any operation starting after time $t - d/2$ does not appear in τ before op_i by construction unless it does not write. Therefore, it does not affect the legality of op_i. ■

If OP_i and OP_j immediately do not commute for all $i \neq j$, then our upper bound matches the lower bound from Theorem 8.

The following abstract data type satisfies the hypothesis of Theorem 15. Consider a 2-component array object with operations $R1R2$, $W1R2$, and $W2$. The abstract data type has a 3-clique as its commutativity graph. Each operation is immediately self-commuting. No cyclic dependences exist. This type is similar to BANKACCOUNT2.

We can optimize implementations of certain data types with 0-d assignments if they have no n-cyclic dependences.

Theorem 16 *For any component array data type T with immediately self-commuting operations $OP_1, \ldots, OP_n$ and no k-cyclic dependences (for each $k \geq 2$), there exists a linearizable implementation of T such that $\sum_{i=1}^{n} |OP_i| = |VC(T)|d$.*

Proof Assume that $VC(T) = \{OP_1, \ldots, OP_{|VC(T)|}\}$. We exhibit an implementation where $|OP_1| = \cdots = |OP_{|VC(T)|}| = d$ and $|OP_{|VC(T)|+1}| = \cdots = |OP_n| = 0$. By the definitions of vertex cover and commutativity graph, OP_i and OP_j immediately commute for all i, j in $\{|VC(T) + 1|, \ldots, n\}$.

Each process keeps a copy of all objects in its local memory. When an op_i on object X is invoked at process p, p sends a message $DoOp_i(X)$ with the argument list for the operation to all processes (including itself). If $i > |VC(T)|$, the process immediately returns the result of the operation (reading the components that should be read). If $i \leq |VC(T)|$, the process waits d time and then returns the result of performing the operation. When a process receives any form of $DoOp$ message, it performs the operation on the appropriate object in its local memory. We can break ties in the following way: $DoOp_i$ is handled before $DoOp_j$ if $OP_i \notin VC(T)$ and $OP_j \in VC(T)$. Otherwise, $DoOp_i$ is handled before $DoOp_j$ if $i < j$. We use process identifiers to break any remaining ties.

We now prove that this algorithm is correct. Let ρ be an admissible execution. We systematically construct τ. Each operation occurs at its response time. Let τ be the sequence of operations in ρ ordered by times of occurrence, breaking ties by placing op_k before op_l and using the same method as in message handling. By construction, $\rho|p = \tau|p$ for all p, and τ preserves the relative ordering of nonoverlapping operations.

We must now show that τ is legal. We have $\tau = op_{1i_1} \circ op_{2i_2} \circ \cdots$, where ki_k means that the kth operation in τ is an op_{i_k} operation. In op_{ki_k}, if $i \leq |VC(T)|$, then the operation returns based on the local state of its process d time after its invocation time, t. The local state reflects all changes made by other operations starting by time t. Any operation starting after time t does not appear in τ before op_{ki_k} by construction unless it immediately commutes with op_{ki_k}. Therefore, it does not affect the legality of op_{ki_k}. If $i > |VC(T)|$, then the operation returns based on the local state of its process at its invocation time, t. The local state reflects all changes made by other operations starting by time $t-d$. Any operation starting after time $t-d$ does not appear in τ before op_{ki_k} by construction unless it immediately commutes with op_{ki_k}. Therefore, it does not affect the legality of op_{ki_k}. ■

If T's commutativity graph has a small vertex cover ($|VC(T)| < n/2$), then the implementation described in the proof of Theorem 16 is faster than that in the proof of Theorem 15 with respect to the total worst-case time complexity for all operations.

The following abstract data type satisfies the hypothesis of Theorem 16. Consider a 3-component array object with operations $W1$, $R2$, $R3$, $R1W2W3$, and $R1R2W3$. Figure 10 shows the commutativity graph for the abstract data type. The set $\{R1W2W3, R1R2W3\}$ forms a vertex cover for the graph. Each operation is immediately self-commuting. No cyclic dependences exist. This type is similar to BANKACCOUNT3.

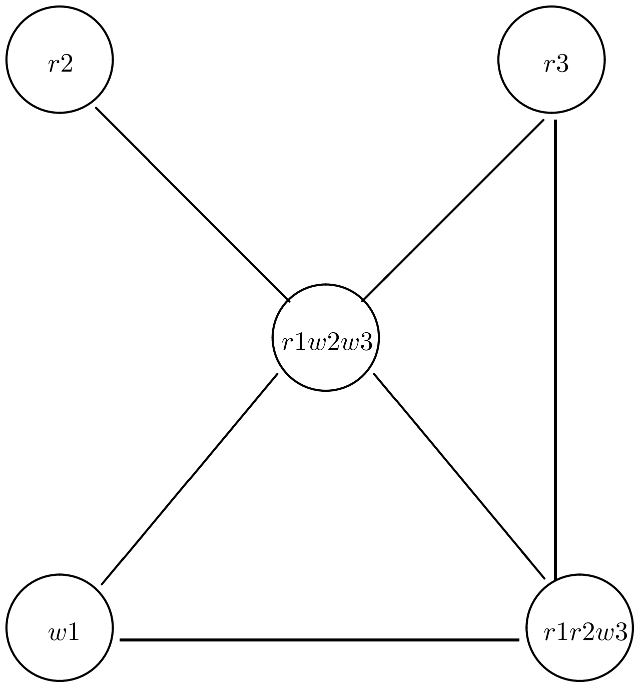


Figure 10: Example commutativity graph for Theorem 16

Theorems 15 and 16 hold for abstract data types without n-cyclic dependences. However, some abstract data types do have n-cyclic dependences. Improving the bounds for implementations of those types is much more complicated than improving the bounds for implementations of types without n-cyclic dependences. We show a tight bound for component array data types with three immediately self-commuting operations and a 3-cyclic dependence.

Theorem 17 *If T is a component array data type with operations OP_1, OP_2, and OP_3 having a 3-cyclic dependence, where OP_1 is immediately self-commuting and has no cyclic dependences with OP_2 and OP_3, then there exists a linearizable implementation of T such that $\sum_{i=1}^{3} |OP_i| = 2d$.*

Proof We exhibit an implementation where $|OP_1| = 0$, $|OP_2| = d$, and $|OP_3| = d$. $|OP_1| + |OP_2| + |OP_3| \geq 2d$ by Theorem 5. At least one of $|OP_1|, |OP_2|$, and $|OP_3|$ must be at least d by Theorem 6. The assumption that OP_1 is immediately self-commuting is necessary because of Theorem 1. Each process keeps a copy of all objects in its local memory. When an op_1 is invoked at process p, p sends a message $DoOp_1(X)$ with the argument list for the operation to all processes (including itself) and returns immediately. When a process receives any form of $DoOp_1$ message, it performs the operation on X in its local memory. When an op_j ($j = 2$ or 3) on object X is invoked at process p, p sends a message $DoOp_j(X)$ with the argument list for the operation to all processes (including itself), waits d time, and returns the result of performing the operation. When a process receives any form of $DoOp_j$ ($j = 2$ or 3) message, it performs the operation on X in its local memory. We can break ties in the following way: $DoOp_i$ messages are handled before $DoOp_j$ messages if $i < j$, and we use process identifiers to break any remaining ties.

We now prove that this algorithm is correct. Let ρ be an admissible execution. We systematically construct the desired τ. Each operation in ρ occurs time d after the time of its call. Let τ be the sequence of operations in ρ ordered by the times of occurrence, breaking ties by placing op_k before op_l if $k < l$, and using process identifiers to break any remaining ties. By construction, $\rho|p = \tau|p$ for all p, and τ preserves the relative ordering of nonoverlapping operations.

We now must show that τ is legal, or that for each object X, $\tau|X$ is in the sequential specification of T. Consider an op_k that returns a value list. It returns based on the local state of its process at its response time in ρ. The local state reflects all changes made by operations occurring up to the time of the op_k in ρ. Thus the op_k returns a legal value list in τ. ■

The BANKACCOUNT3 abstract data type satisfies the hypothesis of Theorem 17.

3.3.3 Relative costs

Let us summarize this section's findings. We know several algebraic properties of operations causing individual operations or groups of operations satisfying one of those properties to have a worst-case response time of $\Omega(d)$.

Let L_1 be the list of properties. We have linearizable implementations of classes of abstract data types in which the worst-case response time for one operation or a group of operations is optimized (the worst-case response time is 0), provided that the operation or group satisfies one of three algebraic properties. Let L_2 be the list of three properties. If not satisfying any property in L_2 implies satisfying a property in L_1, then we could say that sequential consistency and linearizability are equally costly. Unfortunately, L_1 and L_2 do not quite satisfy that desired relationship. However, if the operation or group of operations to be optimized is known to be invoked most frequently, then sequential consistency and linearizability are equally costly for those abstract data types.

In addition, now we have linearizable implementations of certain classes of abstract data types with the total worst-case completion time for all operations matching the sequential consistency lower bound for all operations. This implies that sequential consistency and linearizability are equally costly for these classes.

4 Imperfect clocks

In this section we assume that clocks run at the same rate as real time but are not initially synchronized, and that message delays are in the range $[d-u, d]$ for some $u > 0$. We prove that many operations in linearizable implementations of virtual shared objects must have worst-case response times that are $\Omega(u)$, under reasonable assumptions about the amount of sharing of objects by processes. We show that for many abstract data types, we can provide operations with worst-case time complexities of 0 in sequentially consistent implementations. This shows that linearizability can be more expensive than sequential consistency under our assumptions about process synchrony. Subsection 4.1 contains our lower bounds on the costs of operations under linearizability, and Subsection 4.2 contains our sequentially consistent implementations of classes of abstract data types.

To prove our lower bounds, we use shifting techniques introduced in [16] to prove lower bounds on the precision achieved by clock synchronization algorithms. These techniques are used in [5] to prove $\Omega(u)$ worst-case response times for read, write, and enqueue operations. Shifting changes the timing and ordering of system events without changing the local view of each process.

In 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 also shifted by amount s, then the result is another execution in which every process "sees" the same events happening at the same real times. Because its clock has changed by the same amount, p cannot detect the changes in the real times at which events occur.

The *view* of process p in history π of p with clock C is the concatenation of the sequences of steps in π, arranged in real-time order. The view does not contain the real times when the steps occurred. History h of process p with clock C and history h' of process p with clock C' are *equivalent* if p's view is the same in both histories. Execution ρ of the system $(P, \mathcal{C})$ and execution ρ' of the system $(P, \mathcal{C}')$ are equivalent if the component histories for p in ρ and ρ' are equivalent for all p in P. This means that the processes cannot tell the difference between the two executions.

Given a history π of a 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. This means that all tuples are shifted earlier in π' by s if $s > 0$ and later in π' by $-s$ if $s < 0$. Given a clock C and a real number s, a new clock $C' = \mathit{shift}(C, s)$ is defined by $C'(t) = C(t) + s$ for all t. This means that the clock is shifted forward by s if $s > 0$ and backward by s if $s < 0$. Shifting a history of process p and p's clock by the same amount produces another history.

Lemma 18 ([16]) *Let π be a history of a 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 an execution ρ of a system $(P, \mathcal{C})$ and a real number s, we define a new execution $\rho' = \mathit{shift}(\rho, p, s)$ by replacing π (p's history in ρ) with $\mathit{shift}(\pi, s)$ and by retaining the same correspondence between sends and receives of messages. (We redefine the correspondence so that a pairing in ρ that involves p's event at time t involves p's event at time $t - s$ in ρ'.) All tuples for a process p are shifted by s, but no other tuples are changed.

Given a set of clocks $\mathcal{C} = \{C_q\}_{q \in P}$ and a real number s, we define a new set of clocks $\mathcal{C}' = \mathit{shift}(\mathcal{C}, p, s)$ by replacing C_p with $\mathit{shift}(C_p, s)$. Process p's clock is shifted forward by s, but no other clocks are changed. Shifting one process's history and its clock by the same amount in an execution results in an execution that is equivalent to the original.

Lemma 19 ([16]) *Let ρ be an execution of system $(P, \mathcal{C})$, let p be a process, and let s be a real number. Let $\mathcal{C}' = \mathit{shift}(\mathcal{C}, p, s)$ and $\rho' = \mathit{shift}(\rho, p, s)$. Then ρ' is an execution of $(P, \mathcal{C}')$, and ρ' is equivalent to ρ.*

The following lemma tells us the amount of change encountered in message delays when an execution is shifted. Shifting an admissible execution may produce a nonadmissible execution.

Lemma 20 ([16]) *Let ρ be an execution of the system $(P, \mathcal{C})$, let p be a process, and let s be a real number. Let $\mathcal{C}' = \mathit{shift}(\mathcal{C}, p, s)$ and $\rho' = \mathit{shift}(\rho, p, s)$. Suppose x is the delay of message m from process q to process r in ρ. Then the delay of m in ρ' is x if $q \neq p$ and $r \neq p$, $x - s$ if $r = p$, and $x + s$ if $q = p$.*

4.1 Lower bounds for linearizability

We now give two algebraic properties of operations that cause individual operations to have a worst-case time complexity of $\Omega(u)$ in linearizable implementations of objects of their abstract data types, under reasonable assumptions about the amount of sharing of objects by processes.

This first result is that an operation must have worst-case time complexity of $\Omega(u)$ if the legality of some sequence depends on the order in which two instances of the operation are executed.

Theorem 21 *Let T be an abstract data type with an operation OP such that $\rho \circ op^1 \circ op^2$ does not look like $\rho \circ op^2 \circ op^1$ for some operation sequence ρ and some operation instances op^1 and op^2. For any linearizable implementation of an object of type T in a system with at least three different processes, $|OP| \geq u/2$.*

Proof We generalize the proofs in [5] that a write for a read/write object and an enqueue for a queue must take time at least $u/2$. Since $\rho \circ op^1 \circ op^2$

does not look like $\rho \circ op^2 \circ op^1$, there exists an operation sequence γ such that $\rho \circ op^1 \circ op^2 \circ \gamma$ is legal but $\rho \circ op^2 \circ op^1 \circ \gamma$ is not legal.

Let A be an object of T. Let p_1 and p_2 be two processes that modify A, and let p_3 be a process that performs operations on A. Assume in contradiction that there is a linearizable implementation of A for which $|OP| < u/2$. By the specification of A and Lemma 31, there is an admissible execution α such that the following hold:

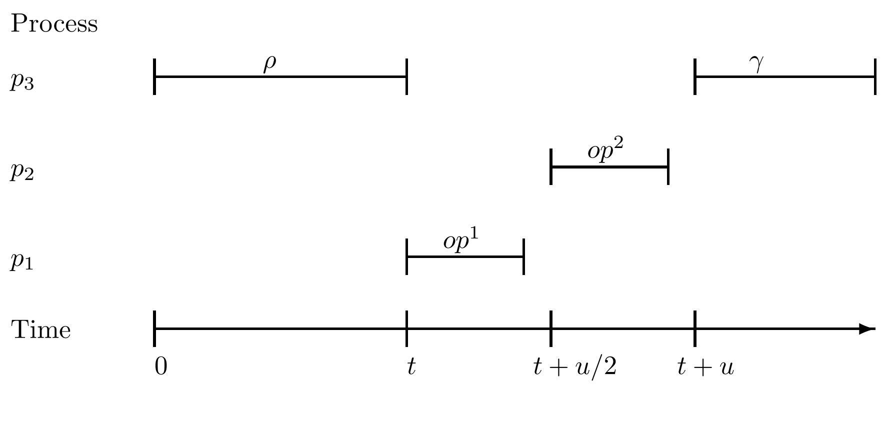
- $ops(\alpha)$ is $\rho[A, p_3] \circ op^1[A, p_1] \cdot op^2[A, p_2] \circ \gamma[A, p_3]$.
- $\rho[A, p_3]$ starts at time 0 and ends at time t, $op^1[A, p_1]$ starts at time t, $op^2[A, p_2]$ starts at time $t + u/2$, and $\gamma[A, p_3]$ starts at time $t + u$.
- The message delays in α are d from p_1 to p_2, $d - u$ from p_2 to p_1, and $d - u/2$ for all other ordered pairs of processes.

Let $\beta = shift(shift(\alpha, p_1, -u/2), p_2, u/2)$ (shift p_1 later by $u/2$ and p_2 earlier by $u/2$). We have admissible β because, by Lemma 20, the delay of a message from p_1 or to p_2 is $d - u$, the delay of a message from p_2 or to p_1 is d, and all other delays are unchanged. But the linearization of $ops(\beta)$ is $\rho[A, p_3] \circ op^2[A, p_2] \circ op^1[A, p_1] \circ \gamma[A, p_3]$, which is not legal. ■

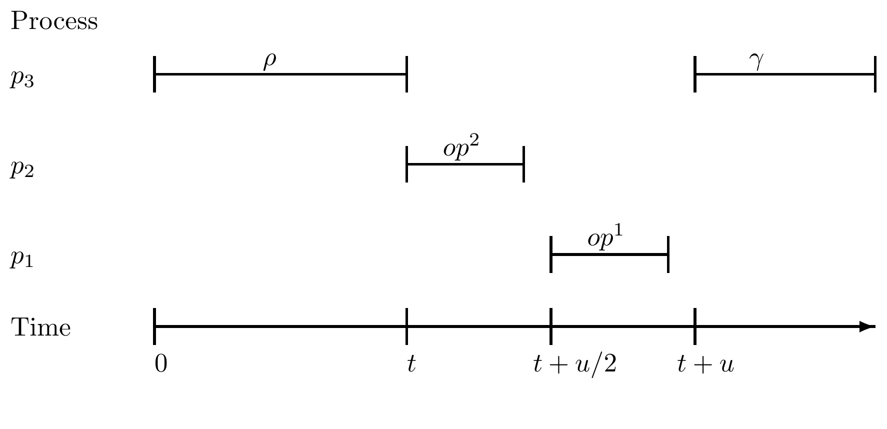
A variant of Theorem 21 can be obtained if the operation instances in the statement of the theorem are not in the same generic operation. See Figure 11. The conclusion is that at least one of the generic operations must take time at least $u/2$. We omit the proof because of its similarity to the proof of Theorem 21.

Theorem 22 *Let T be an abstract data type with operations OP_1 and OP_2 such that $\rho \circ op_1 \circ op_2$ does not look like $\rho \circ op_2 \circ op_1$ for some operation sequence ρ and some operation instances op_1 and op_2. For any linearizable implementation of an object of type T in a system with at least three different processes, $|OP_1| \geq u/2$ or $|OP_2| \geq u/2$. Figure 11 shows α and β.*

This next result is that an accessor operation must have worst-case time complexity of $\Omega(u)$ if the legality of an instance of the accessor depends on whether an instance of another operation is executed.



Execution α



Execution β

Figure 11: Counterexample for Theorem 21

Theorem 23 *Let T be an abstract data type with an operation MOP and an accessor AOP such that $\rho \circ aop^{beforemop}$ and $\rho \circ mop \circ aop^{aftermop}$ are legal but $\rho \circ aop^{aftermop}$ and $\rho \circ mop \circ aop^{beforemop}$ are not legal for some operation sequence ρ and some operation instances mop, $aop^{beforemop}$, and $aop^{aftermop}$. For any linearizable implementation of an object of type T in a system with at least three different processes, $|AOP| \geq u/2$.*

Proof The following proof generalizes the proof in [17] that $|READ| \geq u/2$ for read/write objects. Their proof improved a lower bound of $u/4$ in [5].

Let A be an object of type T. Let p_1 and p_2 be two processes that access A, and let p_3 be a process that performs operations on A.

Assume in contradiction that there exists a linearizable implementation of A for which $|AOP| < u/2$. Let $w = \lceil |MOP|/u \rceil$. By the specification of A and Lemma 31, there exists an admissible timed execution α that is as follows:

- $ops(\alpha)|p_3 = \rho[A, p_3] \circ mop[A, p_3]$, where ρ starts at time 0 and ends at time t, *mop*'s call occurs at time $t + u/2$, and its response occurs at or before time $t + u/2 + |MOP|$.
- $ops(\alpha)|p_1$ is a sequence of $w+1$ operations $aop^{2i}[A, p_1]$, where i ranges from 0 to w and the ith call occurs at time $t + iu$.
- $ops(\alpha)|p_2$ is a sequence of $w+1$ operations $aop^{2i+1}[A, p_2]$, where i ranges from 0 to w and the ith call occurs at time $t + iu + u/2$.

We assume that α has the following message delays: messages from p_1 to p_2 have delay d, messages from p_2 to p_1 have delay $d - u$, and all others have delay $d - u/2$.

By the definition of w, $t + u/2 + |MOP| < t + u/2 + wu$. Thus, *mop* completes before aop^{2w+1} begins. This fact, together with the linearizability of α and the accessor property of AOP, allows aop^{2w+1} to be an $aop^{aftermop}$ (i.e., it has the same return value). Also, ρ finishes before aop^0 begins, aop^0 completes before time $t+u/2$, and *mop* does not begin until time $t+u/2$, and therefore aop^0 is an $aop^{beforemop}$ by the linearizability of α and the accessor property of AOP. Thus, by the linearizability of α, there exists an index i, $0 \leq i \leq 2w$, such that aop^i is an $aop^{beforemop}$ and aop^{i+1} is an $aop^{aftermop}$.

This implies that the linearization of α is $\rho \circ aop^0 \circ \cdots \circ aop^i \circ mop \circ aop^{i+1} \circ \cdots \circ aop^{2w+1}$.

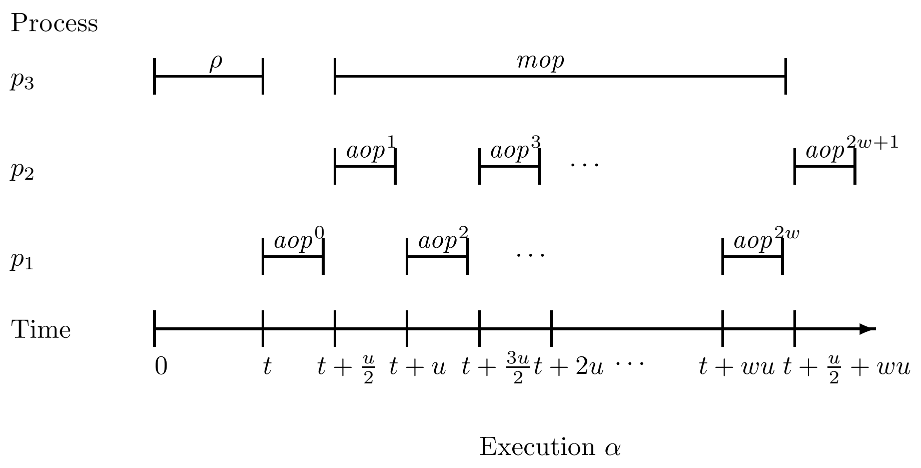
Since AOP is an accessor and the linearization is legal, $aop^1, \ldots, aop^{i-1}$ are of the form $aop^{beforemop}$, and $aop^{i+1}, \ldots, aop^{2w}$ are of the form $aop^{aftermop}$. We can assume that i is even so that aop^i is performed by p_1. Let $\beta = shift(shift(\alpha, p_1, -u/2), p_2, u/2)$. Now β is admissible, because by Lemma 20, the delay of a message from p_1 is $d - u$, the delay of a message from p_2 is d, and all other message delays are unchanged. In β, ρ precedes all AOP operations, and the order of the AOP operations performed on A by p_1 and p_2 is $aop^1, aop^0, aop^3, aop^2, \ldots, aop^{i+1}, aop^i, \ldots$. If mop is linearized before aop^i in β, then by the accessor property of AOP, $\rho \circ mop \circ aop^i$ is legal, a contradiction. If mop is linearized after aop^i in β, then by the accessor property of AOP, $\rho \circ aop^{i+1}$ is legal, a contradiction. ■

4.2 Sequentially consistent bounds

In a system with approximately synchronized clocks and uncertainty in message delay, we cannot automatically assume that processes receive and handle messages in the same order. Since sequential consistency needs a legal global ordering for operations in an execution that respects the individual local orderings of operations by the same process, sequential consistency may be easier to implement in a distributed system with a globally consistent message ordering.

The *atomic broadcast* communication primitive (see [6]) delivers all broadcast messages in the same order at all processes, ensuring that two messages broadcast by the same process are delivered in the order in which they are broadcast. [5] uses atomic broadcast in a modular fashion to yield a sequentially consistent implementation of queues, where enqueues are fast, and two implementations of read/write objects, one with fast reads (and slow writes) and the other with fast writes (and slow reads). [12] corrects potential deadlock problems in the atomic broadcast algorithm given in [5]. Accessor operations can be made fast.

Theorem 24 *Let T be an abstract data type with m generic accessor operations (AOP_i) and n generic mutator operations (MOP_j). Then there exists*



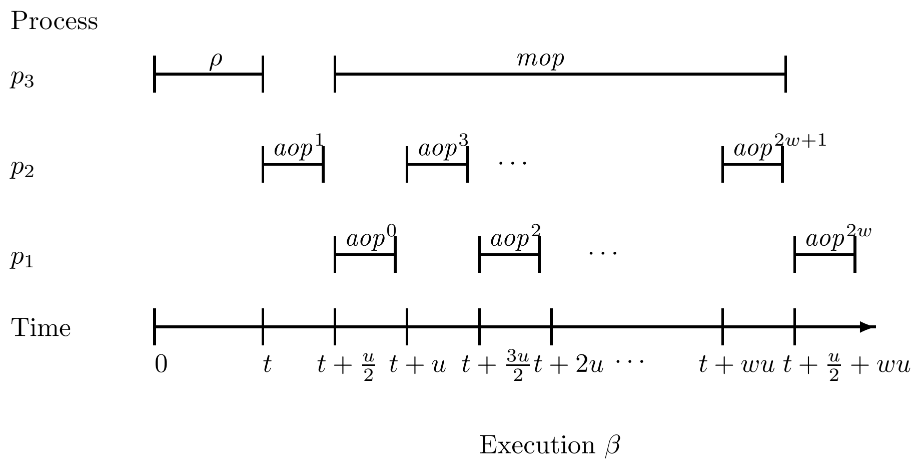


Figure 12: Counterexample for Theorem 23

a sequentially consistent implementation of T with $|AOP_i| = 0$ for each i in $\{1, \ldots, m\}$ and $|MOP_j| = h$ for each j in $\{1, \ldots, n\}$, where h is the maximum time for message delivery by the underlying atomic broadcast algorithm. Figure 12 shows α and β.

We now explain the details of the algorithm. For each accessor operation, the invoking process applies just the operation to its local copy of the object and returns the result. For each mutator operation, the invoking process uses the atomic broadcast algorithm to send a message to all processes (including itself) containing the invoking process's identification, the name of the operation, the object on which the operation is invoked, and the argument list for the operation. The mutator operation does not complete until the invoking process receives the message and handles it.

Proof Our proof of sequential consistency generalizes the proof in [5] for read/write objects.

Claim 25 *For every admissible execution and every process p, p's local copies are updated to reflect all update operations, all update operations occur in the same order at every process, and this order preserves the order of update operations on a per-process basis.*

Proof of Claim 25 By the algorithm, an atomic broadcast send is performed exactly once for each update operation. By the guarantees of atomic broadcast, each process gets exactly one message for each update operation, these messages are received in the same order at all processes, and this order agrees with the sending order with respect to each individual process. ■

Claim 26 *For every admissible execution, every process p, and all objects X and Y, if accessor A of object Y follows mutator M of object X in $ops(\sigma)|p$, then A's access of p's local copy of Y follows M's modification of p's local copy of X.*

Proof of Claim 26 M does not complete until its update is performed at p, the process that begins it. ■

Rule 27 *Let ρ be an admissible execution. We systematically build the desired τ. We first order all mutator operations in the order that the atomic* *broadcast algorithm assigns to their messages. We determine where to place the accessor operations, starting from the beginning of ρ. We have that $aop_i^j[X,p]$ goes immediately after the latter of (1) the previous operation for process p or (2) the mutator that caused the latest update of p's copy of X before the generation of the response for $aop_i^j[X,p]$. We use process identifiers to break any ties.*

Now we prove that $ops(\rho)|p = \tau|p$ for all processes p. Choose some p. By the definition of τ, the relative ordering of two accessor operations in $ops(\rho)|p$ is the same as in $\tau|p$. Claim 25 guarantees that the relative ordering of two mutator operations in $ops(\rho)|p$ is the same as in $\tau|p$. If accessor A follows mutator M in $ops(\rho)|p$, then A follows M in τ by the definition of τ.

Suppose that accessor A precedes mutator M in $ops(\rho)|p$. Suppose in contradiction that M precedes A in τ. We follow the above rule for placing A in τ. Rule 27(1) does not apply, because M is not a previous operation for process p. Rule 27(2) applies. Thus, in ρ there is some accessor A' and some mutator M' such that the following hold:

1. A' is A or precedes A in ρ.

2. M' is M or follows M in the ordering of mutators generated by the atomic broadcast algorithm.

3. M' causes the latest update to p's copy of A's object that precedes A'.

A' finishes before M starts in ρ. Since mutator operations are performed in ρ in atomic broadcast order (Claim 25), A' does not see the update performed by M', a contradiction.

We must now prove that τ is legal. We must check all operations, mutators and accessors, for legality. By Claim 25, mutators are performed at every process in atomic broadcast order. Thus, each mutator returns correctly, based on all updates handled before it, ensuring that each mutator is legal.

Consider accessor $A = aop_i^j[X,p]$ in τ. Let M be the mutator (performed by process q) in ρ that causes the latest update to p's copy of X preceding A's access of p's copy of X. A follows M in τ by the definition of τ. We must show that no other mutator operation on X is placed between M and A in τ. Suppose in contradiction that a mutator M' performed by process

r does. By Claim 25, the update for M' follows the update for M at each process in ρ.

Suppose that $r = p$. M' precedes A in ρ by Rule 27. The update for M' follows the update for M in ρ. Thus A sees M''s update and not M's, contradicting the choice of M.

Suppose that $r \neq p$. By Rule 27 (A immediately follows the previous operation for process p), there is some operation in $ops(\rho)|p$ that precedes A and follows M' in τ; otherwise A would not follow M'. Let O be the first such operation. Thus, O immediately follows M' in τ.

Suppose O is a mutator operation on some object Y. By Claim 26, O's update to p's copy of Y precedes A's access of p's copy of X. Since updates are performed in atomic broadcast order, the update for M' occurs at p before the update for O, and also before A's access, contradicting the choice of M.

Suppose that O is an accessor operation. By Rule 27(2), O is an accessor of X, and M''s update to p's copy of X is the latest one preceding O's access; otherwise O would not immediately follow M'. Since updates are performed in atomic broadcast order, the value from M' supersedes the value from M, contradicting the choice of M. ■

Pure mutators can be made fast, too.

Theorem 28 *Let T be an abstract data type with m pure mutator operations (MOP_i) and n other operations (OP_j). Then there exists a sequentially consistent implementation of T with $|MOP_i| = 0$ for each i in $\{1, \ldots, m\}$ and $|OP_j| = h$ for each j in $\{1, \ldots, n\}$, where h is the maximum time for message delivery by the underlying atomic broadcast algorithm.*

Proof We now explain the details of the algorithm. For each operation, the invoking process uses the atomic broadcast algorithm to send a message to all processes (including itself) containing the invoking process's identification, the name of the operation, the name of the object on which the operation is invoked, and the argument list for the operation. *MOP* operations return immediately, while an OP_j operation does not complete (and return a result) until all pending updates by its invoking process have been completed. This is managed by maintaining a count of pending updates (initialized to 0) and waiting for its value to become 0.

We now explain why the algorithm guarantees sequential consistency. Let ρ be an admissible execution. We construct τ by ordering all operations in the

Table 3: Corollaries for Section 4

Operations	Type/table references	Theorems	Citations
ENQ	Table 5	21, 28	[5]
PUSH	Table 6	21, 28	[5]
WRITE	Read/write objects	21, 28	[5]
UP	Table 9	21	
PEEK	Tables 5, 6	23, 24	
READ	Read/write objects	23, 24	[5, 17]
SEARCH	Table 7	23, 24	
BALANCE	Table 8	23, 24	
FIND	Table 9	23, 24	
DEPOSIT	Table 8	28	
$INC, HALF$	Table 10	28	

order that the atomic broadcast algorithm assigned to their messages. Since atomic broadcast preserves the per-process message orders, $\tau|p = ops(\rho)|p$ for each process p. We now show why τ is legal. All pure mutator operations are legal in τ, because the return values for pure mutator operations do not depend on the states of the objects on which they are invoked. Now we must show that each op_j^i is legal. In ρ, $op_j^i[X, p]$ returns based on the state of p's copy of object X when its message is handled. The state of p's copy of object X reflects all changes made at all processes before op_i^j's message is handled. Thus $op_j^i[X, p]$ returns a legal value list in τ. ■

Table 3 displays types applicable to the theorems in this section. For each row of the table, the operation from the first column, which is from the type or table reference listed in the second column, meets the hypotheses for the theorem(s) listed in the third column. Any references to a specific lower or upper bound corresponding to the given operation are presented in the fourth column.

We now discuss the implications of the results in this section with respect to the relative costs of sequential consistency and linearizability in systems with only approximately synchronized clocks and uncertainty in the network message delay. Now we have lower bounds on the costs in linearizable implementations of single operations satisfying certain algebraic properties. The necessary property for Theorem 21 is very general. In order for an abstract data type to be at all useful, all operation sequences cannot look like each other. The necessary property for Theorem 23 is reasonably general, too. Accessor instances should not be freely interchangeable after sequences including mutator instances.

We also have two sequentially consistent implementations of abstract data types, with accessors taking time 0 to complete in one (Theorem 24) and pure mutators taking time 0 to complete in the other (Theorem 28). In many abstract data types, pure mutators satisfy the hypothesis of Theorem 21, and accessors satisfy the hypothesis of Theorem 23, requiring at least $u/2$ time in linearizable implementations of their abstract data types. Thus, our results imply that sequential consistency is less expensive than linearizability for a reasonably large class of abstract data types.

5 Hybrid consistency

Attiya and Friedman [4] show that if weak operations are used mostly, hybrid consistency is cheaper than sequential consistency or linearizability for read/write objects. We show that hybrid consistency is not necessarily cheaper than stronger guarantees.

The lower bounds in Table 4 are analogous to the lower bounds for sequential consistency. We omit proofs because they are very similar to previous proofs. We assume all processes have perfectly synchronized clocks. WOP indicates OP's weak version, SOP indicates OP's strong version, and $*OP$ indicates an arbitrary version of OP. *Wop* indicates a weak instance of OP, and *Sop* indicates a strong instance of OP.

Since hybrid consistency is weaker than sequential consistency, intuition tells us that there may be some abstract data types in which hybrid consistent implementations of weak operations are faster than their sequentially consistent (or linearizable) counterparts. Indeed, [4] describes a hybrid consistent implementation of read/write objects in which weak reads and writes are fast, and strong reads and writes take time $O(d)$. Is it always possible to develop hybrid consistent implementations with similar time bounds (ideally, weak operations that are faster than strong operations)? The answer is negative.

Why could weak operations for read/write objects be optimized? One

Table 4: Hybrid consistency analogues

Theorem for sequential consistency	Result for hybrid consistency
Theorem 1	$\lvert *OP\rvert \geq d$
Theorem 2	$\lvert SOP_1\rvert + \lvert WOP_2\rvert \geq d$
Theorem 3	$\lvert *OP_1\rvert + \lvert *OP_2\rvert \geq 2d$
Theorem 4	$\lvert SOP_i\rvert \geq d$ for some i in $\{1,\ldots,n\}$
	$\lvert WOP_i\rvert \geq d$ for some i in $\{1,\ldots,n\}$
Theorem 5	$\lvert *OP_1\rvert + \lvert *OP_2\rvert + \lvert *OP_3\rvert \geq 2d$
Theorem 6	$\lvert *OP_1\rvert + \lvert *OP_2\rvert \geq d$, or
	$\lvert *OP_3\rvert \geq d$
Theorem 7	$\sum_{i=1}^{n}(\lvert SOP_i\rvert + \lvert WOP_i\rvert) \geq 2d$
	$(\lvert NC(T)\rvert + \lvert Maxdom(RCG(T))\rvert)$
Theorem 8	$\sum_{i=1}^{n}(\lvert SOP_i\rvert + \lvert WOP_i\rvert) \geq$
Omit the -1 if $n - \lvert NC(T)\rvert$ is even.	$2\lvert NC(T)\rvert d + (n - \lvert NC(T)\rvert - 1)d$

plausible reason is that read/write objects have simple semantics. It may be possible to optimize weak operations for other abstract data types with simple semantics. Let us consider the PMR objects defined in Subsection 3.3.

Let us consider read/add/multiply objects. $read()(v)[X,p]$ is legal if the value stored in X is v. The effect of $add(v)()[X,p]$ is to add v to the value stored in X. The effect of $mul(v)()[X,p]$ is to multiply v to the value stored in X.

Let X be one such object, initialized to 1. Suppose there is a hybrid consistent implementation of X for which $\lvert WADD\rvert = \lvert WMUL\rvert = \lvert WREAD\rvert = 0$. Consider the admissible execution ρ:

- $ops(\rho)|p_1$ is $Wadd(5)()[X,p_1] \circ Wread()(6)[X,p_1]$, where the add starts at time 0 and the read completes before time d.
- $ops(\rho)|p_2$ is $Wmul(3)()[X,p_2] \circ Wread()(3)[X,p_2]$, where the multiply starts at time 0 and the read completes before time d.

This execution is not hybrid. p_1's read must return 6, because the execution is indistinguishable to p_1 from an execution in which p_1 is running alone. Similarly, p_2's read must return 3.

There are two ways to order the operations that are not reads:

- $Wmul(3)()[X,p_2] \circ Wadd(5)()[X,p_1]$. $Wread()(6)[X,p_1]$ must be placed after $Wadd(5)()[X,p_1]$ for hybrid consistency, making it illegal.

- $Wadd(5)()[X, p_1] \circ Wmul(3)()[X, p_2]$. $Wread()(3)[X, p_2]$ must be placed after $Wmul(3)()[X, p_2]$ for hybrid consistency, making it illegal.

Both reads cannot be legally placed. We use this example to derive a new lower bound on the worst-case time complexity for weak operations in hybrid consistent implementations of abstract data types. A generic operation OP is *doubly noninterleavable with respect to* OP_1 *and* OP_2 if there exist operation sequence ρ and operation instances op^1, op^2, op_1, op_2 such that $\rho \circ op_1 \circ op^1$ and $\rho \circ op_2 \circ op^2$ are legal, but there is no way to place both op^1 and op^2 after ρ in $\rho \circ op_1 \circ op_2$ and $\rho \circ op_2 \circ op_1$ to form a legal sequence.

Theorem 29 *Let T be an abstract data type, and let AOP, OP_1, and OP_2 be operations on objects of type T in a system with at least two different processes, where AOP is a generic accessor operation. Suppose AOP is doubly noninterleavable with respect to OP_1 and OP_2. Then, in any hybrid consistent implementation of objects of type T, $|WOP_1| + |WAOP| \geq d$ or $|WOP_2| + |WAOP| \geq d$.*

Proof Let A be an object of type T in a system with at least two different processes. Let processes 1 and 2 use A. Since AOP is doubly noninterleavable with respect to OP_1 and OP_2, there exists an operation sequence α and operation instances aop^1, aop^2, op_1, op_2 such that $\alpha \circ op_1 \circ aop^1$ and $\alpha \circ op_2 \circ aop^2$ are legal, but there is no way to place both aop^1 and aop^2 after ρ in $\rho \circ op_1 \circ op_2$ and $\rho \circ op_2 \circ op_1$ to form a legal sequence. Assume in contradiction that there exists some hybrid consistent implementation of A_α for which $|WOP_1| + |WAOP| < d$ and $|WOP_2| + |WAOP| < d$.

By the sequential specification for A_α, there exists an admissible execution ρ_1 with $ops(\rho_1)$ equal to $Wop_1[A_\alpha, 1] \circ Waop^1[A_\alpha, 1]$. $Wop_1[A_\alpha, 1]$ starts at real time 0, and $Waop^1[A_\alpha, 1]$ starts immediately after $Wop_1[A_\alpha, 1]$ finishes. Because we have assumed that the real time after the end of ρ_1 is less than d, no process receives a message during ρ_1.

By the sequential specification for A_α, there exists an admissible execution ρ_2 with $ops(\rho_2)$ equal to $Wop_2[A_\alpha, 2] \circ Waop^2[A_\alpha, 2]$. Then $Wop_2[A_\alpha, 2]$ starts at real time 0, and $Waop^2[A_\alpha, 2]$ starts immediately after $Wop_2[A_\alpha, 2]$ finishes. Because we have assumed that the real time after the end of ρ_2 is less than d, no process receives a message during ρ_2.

Figure 13 shows admissible executions ρ_1 and ρ_2. Since no messages are received in ρ_1 and ρ_2, we can produce an admissible hybrid execution ρ by replacing process 2's history in ρ_1 with its history in ρ_2.

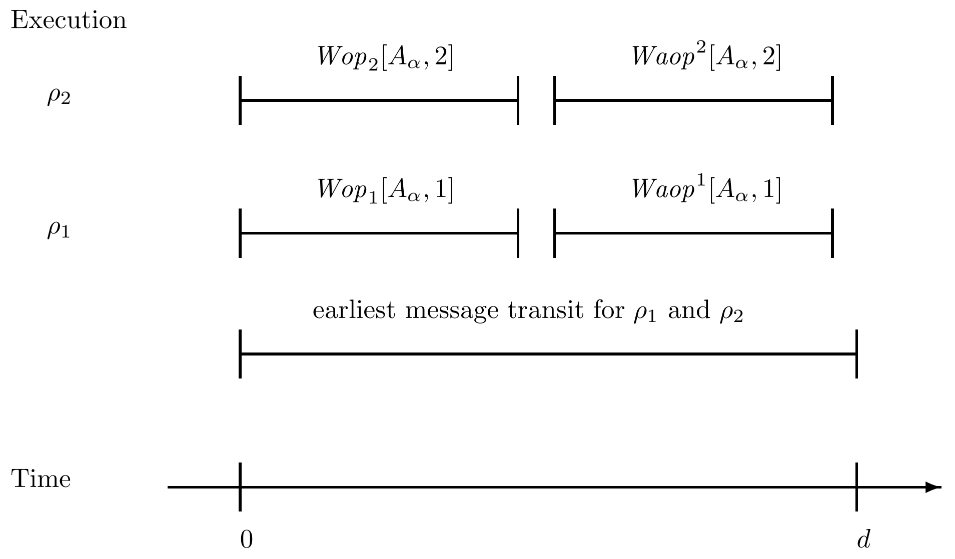


Figure 13: Counterexample for double noninterleavability lower bound (Theorem 29)

By assumption, ρ is hybrid. Consider τ_1. In τ_1, $Wop_1[A_\alpha, 1]$ precedes $Waop^1[A_\alpha, 1]$. If $Wop_1[A_\alpha, 1]$ precedes $Wop_2[A_\alpha, 2]$, then by the double noninterleavability property, all possible placements of both $Waop^1[A_\alpha, 1]$ and $Waop^2[A_\alpha, 2]$ in $Wop_1[A_\alpha, 1] \circ Wop_2[A_\alpha, 2]$ are illegal for A_α. If $Wop_2[A_\alpha, 2]$ precedes $Wop_1[A_\alpha, 1]$, then by the double noninterleavability property, all possible placements of both $Waop^1[A_\alpha, 1]$ and $Waop^2[A_\alpha, 2]$ in $Wop_2[A_\alpha, 2] \circ Wop_1[A_\alpha, 1]$ are illegal for A_α. Thus, we cannot produce a τ_1 that is legal for A_α, because we need to place all operations. ■

Corollary 30 *In any implementation of* read/add/multiply *objects that is hybrid consistent,* $|WADD| + |WREAD| \geq d$ *or* $|WMUL| + |WREAD| \geq d$.

For the above classes of PMR objects, the lower bound on total time complexity of the operations matches the upper bound given in the linearizable implementations. Thus, for these classes of objects, sequential consistency and linearizability cost no more than hybrid consistency.

6 Summary

We have studied the impact of algebraic properties of operations, the degree of process synchronization, and the type of consistency guarantee on the time complexities of distributed implementations of abstract data types. We have shown that sequential consistency and linearizability are equally costly in systems with perfectly synchronized clocks under certain reasonable assumptions; that sequential consistency is cheaper than linearizability in systems with only approximately synchronized clocks under certain reasonable assumptions; and that hybrid consistency is not necessarily cheaper than the stronger consistency guarantees.

We attempted to find an algebraic property of operations such that an operation could be optimized in a linearizable implementation assuming perfect process synchrony if it satisfied the property and it could not be optimized otherwise. We determined a reasonably general algebraic property, self-obliviousness, such that if an operation is self-oblivious, then it could be optimized in a linearizable implementation assuming perfect process synchrony. However, we still do not know if we can optimize an operation that is not self-oblivious. The results in [14] support the hypothesis that self-obliviousness is necessary for optimizing a single operation.

Optimizing a given operation or group of operations is beneficial when it is frequently used. Although we do not have a complete characterization of when a given operation or group of operations cannot be optimized, we have made reasonable progress on this problem, and we hope that our work can be used to determine this elusive complete characterization.

A Examples of abstract data types

The *augmented queue* abstract data type (Table 5) has enqueue, dequeue, and peek operations. A peek returns the front item of a queue without changing the queue. Queues can be of arbitrary length. A $\perp$ is returned by a dequeue or peek invoked on an empty queue. An e (respectively, d) in entry (i, j) of the table means that operation i and operation j eventually (respectively, immediately) do not commute. No relationship is assumed between the argument lists for operation i and operation j. No item in an argument list has $\perp$ for its value. Any operation returning $\perp$ is *abnormal.*

Table 5: Commutativity for the augmented queue abstract data type

Queue operation	$enq(x)\cdot$ $ok()$	$deq()\cdot$ $ret(x)$	$deq()\cdot$ $ret(\perp)$	$peek()\cdot$ $ret(x)$	$peek()\cdot$ $ret(\perp)$
$enq(y) \circ ok()$	e		d		d
$deq() \circ ret(y)$		d	d	d	d
$deq() \circ ret(\perp)$	d	d		d	
$peek() \circ ret(y)$		d	d		d
$peek() \circ ret(\perp)$	d	d		d	

Table 6: Commutativity for the augmented stack abstract data type

Stack operation	$push(x)\cdot$ $ok()$	$pop()\cdot$ $ret(x)$	$pop()\cdot$ $ret(\perp)$	$peek()\cdot$ $ret(x)$	$peek()\cdot$ $ret(\perp)$
$push(y) \circ ok()$	e	d	d		d
$pop() \circ ret(y)$		d	d	d	d
$pop() \circ ret(\perp)$	d	d		d	
$peek() \circ ret(y)$		d	d		d
$peek() \circ ret(\perp)$	d	d		d	

All other operations are *normal.* These definitions hold for all tables.

The *augmented stack* abstract data type (Table 6) has push, pop, and peek operations. A peek returns the top item of a stack without changing the stack. Stacks can be of arbitrary height. $\perp$ is returned by a pop or peek operation invoked on an empty stack.

The *dictionary set* abstract data type (Table 7) has insert (an element and its key), delete (an element and its key), and search (for an element and return its key) operations. A set can have an arbitrary number of (element,key) pairs. $\perp$ is returned by a delete or search operation that is performed when the input argument is not an element in the set.

The bank account abstract data type [18] (Table 8) has deposit, withdraw, and balance operations. $\perp$ is returned by a withdraw operation performed when the account contains less money than the amount specified in the input argument.

Table 7: Commutativity for the dictionary set abstract data type

Set operation	$ins(x,v)\cdot ok()$	$del(x)\cdot ack(ok)$	$del(x)\cdot ack(\perp)$	$search(x)\cdot ret(v)$	$search(x)\cdot ret(\perp)$
$ins(y,w)\cdot ok()$		d	d		d
$del(y)\cdot ok(k)$	d	d	d	d	
$del(y)\cdot ack(\perp)$	d	d			
$search(y)\cdot ret(w)$		d			
$search(y)\cdot ret(\perp)$	d				

Table 8: Commutativity for the bank account abstract data type

Bank account operation	$deposit(j)\cdot ok()$	$withdraw(j)\cdot ack(ok)$	$withdraw(j)\cdot ack(\perp)$	$balance()\cdot ret(j)$
$deposit(i)\cdot ok()$			d	d
$withdraw(i)\cdot ack(ok)$		d		d
$withdraw(i)\cdot ack(\perp)$	d			
$balance()\cdot ret(i)$	d	d		

The *reference-count set* abstract data type (Table 9) has insert, update, and find (search) operations. A reference-count set can have an arbitrary number of elements. When an item is inserted into this set, its data field is initialized to 1. All inserts are normal. When an item is updated, its data field is incremented by 1 (if the item is present). When an item is searched, its data field is returned (if the item is present). $\perp$ is returned by an update or search operation performed when the input argument is not in the set.

In the *increment-half* abstract data type (Table 10), the values are real numbers, and the operations are read, increment, and half. The half operation causes the value of the object on which it is invoked to become half of its previous value.

Table 9: Commutativity for the reference-count set

Reference set operation	$ins(x) \cdot ok()$	$up(x) \cdot ack(ok)$	$up(x) \cdot ack(\perp)$	$find(x) \cdot ret(v)$	$find(x) \cdot ret(\perp)$
$ins(y) \circ ok()$			d		d
$up(y) \circ ack(ok)$				d	
$up(y) \circ ack(\perp)$	d				
$find(y) \circ ret(w)$		d			
$find(y) \circ ret(\perp)$	d				

Table 10: Commutativity for the increment-half abstract data type

INC-HALF	$read() \circ ret(v)$	$inc(x) \circ ok()$	$half() \circ ok()$
$read() \circ ret(v)$		d	d
$inc() \circ ok()$	d		e
$half() \circ ok()$	d	e	

B Axioms about executions

Lemma 31 *If α is a legal sequence of operations for a set of objects $\mathcal{O}$, and α is of the form $op_1[\mathcal{O}_1, p_1] \cdots op_n[\mathcal{O}_n, p_n]$, then there exists an admissible execution σ such that $ops(\sigma)$ is α.*[8]

Proof Choose nonnegative numbers $t_{1s}, t_{1f}, \ldots, t_{ns}, t_{nf}$, where the numbers form a nondecreasing sequence. op_i starts at time t_{is} and finishes at time t_{if}. Let all messages have delay in the range $[d-u, d]$. By construction, $ops(\sigma) = \alpha$, and at most one call per process is pending at a time. We have satisfied the definition of admissibility. ∎

Lemma 32 *If α is a legal sequence of operations for a set of objects $\mathcal{O}_1$ that are performed by process p_1, and β is a legal sequence of operations for a set*

[8] op_i's are not necessarily instantiations of different generic operations, $\mathcal{O}_i$'s are not necessarily distinct objects, and the p_i's are not necessarily distinct processes.

of objects $\mathcal{O}_2$ that are performed by process p_2, where $\mathcal{O}_1$ and $\mathcal{O}_2$ are disjoint, then there exists an admissible execution σ such that $ops(\sigma) = \alpha \circ \beta$.[9]

Proof Suppose α has n operations in it. By Lemma 31, there is an admissible execution σ_1 such that $ops(\sigma_1) = \alpha$. Let the last operation finish at time t_{nf}. By Lemma 31, there is an admissible execution σ_2 such that $ops(\sigma_1) = \beta$. Add $t_{nf} + \epsilon$ for some $\epsilon \geq 0$ to the start and finish times for all operations in σ_2. Consider $\sigma_1 \circ \sigma_2$. It is easy to see that $\sigma_1 \circ \sigma_2$ satisfies the definition of admissibility. ∎

Lemma 33 *Any prefix of an admissible execution is admissible.*

Proof Let $\rho \circ \sigma$ be an admissible execution. ρ is a prefix of $\rho \circ \sigma$. The message delays for the operations in ρ are the same as what they were in $\rho \circ \sigma$, and no process has multiple simultaneous pending operations in ρ if it does not have them in $\rho \circ \sigma$. ∎

Lemma 34 *If β and γ are admissible executions with a common prefix α, and $(\beta - \alpha)$,*[10] *and $(\gamma - \alpha)$ are such that*

1. *any messages sent in $(\beta - \alpha)$ would not be received until after the completion time of γ, and*

2. *any messages sent in $(\gamma - \alpha)$ would not be received until after the completion time of β,*

then, for each process i,

1. *replacing process i's history after α in $\alpha(\beta - \alpha)$ with its history in $(\gamma - \alpha)$ results in an admissible execution $\beta_{i\gamma}$, and*

2. *replacing process i's history after α in $\alpha(\gamma - \alpha)$ with its history in $(\beta - \alpha)$ results in admissible execution $\gamma_{i\beta}$.*

Proof The message delays remain unchanged in $\beta_{i\gamma}$ and $\gamma_{i\beta}$ from their respective delays in β and γ. Process i is the only process for which we need to check for multiple simultaneous pending operations, because all other process

[9] p_1 and p_2 do not have to be distinct processes.

[10] If $z = x \circ y$, then $y = z - x$.

histories are unchanged. Process i's history in $\beta_{i\gamma}$ is the same as its history in γ, and its history in $\gamma_{i\beta}$ is the same as its history in β. Since both β and γ are admissible, process i has no multiple simultaneous pending operations in $\beta_{i\gamma}$ and $\gamma_{i\beta}$. Thus, $\beta_{i\gamma}$ and $\gamma_{i\beta}$ are admissible. ■

Acknowledgments

We thank Soma Chaudhuri, Roy Friedman, Donald Stanat, Jennifer Welch, and the referees for their helpful comments.

Acknowledgment of support

Part of this work was performed at the University of North Carolina at Chapel Hill, supported in part by National Science Foundation grant CCR-9010730 and an IBM Faculty Development Award. This work was partially presented in [13].

References

[1] S. Adve and M. Hill. Weak ordering—A new definition. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 2–14, Seattle, Washington, 1990. IEEE Computer Society Press.

[2] Y. Afek, G. Brown, and M. Merritt. A lazy cache algorithm. *ACM Transactions on Programming Languages and Systems*, 13(1):182–205, 1993.

[3] J. H. Anderson and B. Grošelj. Beyond atomic registers: Bounded wait-free implementations of nontrivial objects. In *Proceedings of the Fifth International Workshop on Distributed Algorithms*, pages 52–70, Delphi, Greece, 1991. Springer-Verlag.

[4] H. Attiya and R. Friedman. A hybrid condition for shared memory Consistency. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, pages 679–690, Victoria, British Columbia, Canada, 1992. ACM Press.

[5] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994.

[6] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[7] M. Dubois, C. Scheurich, and F. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–22, 1988.

[8] R. Friedman. Implementing hybrid consistency with high-level synchronization operations. *Distributed Computing*, 9(3):119–129, 1995.

[9] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessey. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 15–26, Seattle, Washington, 1990. IEEE Computer Society Press.

[10] P. Gibbons, M. Merritt, and K. Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Proceedings of the Third ACM Symposium on Parallel Algorithms and Architectures*, pages 292–303, Hilton Head, South Carolina, 1991. ACM Press.

[11] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 10(3):463–492, 1990.

[12] L. Higham and J. Warpechowska-Gruca. Notes on atomic broadcast. Technical Report TR 95/562/14, University of Calgary Department of Computer Science, 1995.

[13] M. J. Kosa. Making operations of concurrent data types fast. In *Proceedings of the Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 32–41, Los Angeles, 1994. ACM Press.

[14] M. J. Kosa. What critical algebraic property allows operations of concurrent abstract data types to be "fast"? In *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*, page 244, Philadelphia, Pennsylvania, 1996. ACM Press.

[15] R. A. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, 1988.

[16] J. Lundelius and N. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2/3):190–204, 1984.

[17] M. Mavronicolas and D. Roth. Efficient, strongly consistent implementations of shared memory. In *Proceedings of the Sixth International Workshop on Distributed Algorithms*, pages 346–361, Haifa, Israel, 1992. Springer-Verlag.

[18] W. Weihl. The impact of recovery on concurrency control. *Journal of Computer and System Sciences*, 47(1):157–184, 1993.