Improved Time Bounds for Linearizable Implementations of Abstract Data Types

Jiaqi Wang    Edward Talmage    Hyunyoung Lee    Jennifer L. Welch

Department of Computer Science and Engineering
Texas A&M University

Abstract: Linearizability is a well-known consistency condition for shared objects in concurrent systems. We focus on the problem of implementing linearizable objects of arbitrary data types in message-passing systems with bounded, but uncertain, message delay and bounded, but non-zero, clock skew. We present an algorithm that exploits axiomatic properties of different operations to reduce the running time of each operation below that obtainable with previously known algorithms. We also prove lower bounds on the time complexity of various kinds of operations, specified by the axioms they satisfy, resulting in reduced gaps in some cases and tight bounds in others.

1 Introduction

With recent advances in hand-held devices such as smart phones and tablet computers, information sharing among distributed users plays a major role in many of today’s distributed applications, ranging from electronic commerce to social media. Shared data objects thus are a key component in such applications. When dispersed users of such applications perform various operations on shared objects, one of the major concerns is to maintain their consistency. Linearizability [12] (or atomicity [14]) is a well-known consistency condition for shared objects, which gives the illusion of sequential execution of operations. For broadly applicable results, we focus on abstract types of linearizable shared objects, which encompass many concrete data types including stacks, queues, sets, and read-modify-write objects, in addition to basic read/write registers.

The problem. Although shared objects are a convenient abstraction, they are not generally provided in large-scale distributed systems. We consider a system with a large number \( n \) of geographically dispersed processes communicating over a message passing system. We assume no process failures and a reliable point-to-point message passing system. We also assume that the system is partially synchronous in that it provides bounded but uncertain message delays in the range \( [d - u, d] \), and clocks that are approximately synchronized to within some constant \( \epsilon \).

A simple centralized algorithm to implement a linearizable object of arbitrary data type is to forward each operation invocation in a message to a distinguished process, which computes the result of the operation and sends the result back in a message to the invoker. The operations are linearized through the workings of the distinguished process and each operation takes up to \( 2d \) time. Another algorithm is to have each process use a total order broadcast primitive to notify all other processes when it invokes an operation; whenever a broadcast message arrives at a process, it updates a local copy of the object accordingly. However, this second method is not faster than the centralized scheme when taking into account the time overhead to implement the totally ordered broadcast on top of a point-to-point message system [3]. Increasing pressure to speed up today’s applications raises the question whether operations can be executed faster than the \( 2d \) achievable in these “folklore” algorithms. Our goal is to find optimally fast implementations for linearizable shared objects of abstract data types.

Background and our contributions. Lipton and Sandberg [15] initiated the study of performance limitations for consistency conditions by showing that in any implementation of a sequentially consistent register,
the time for read plus the time for write must be at least $d$; since sequential consistency is weaker than linearizability, the same lower bound is true for linearizability. Attiya and Welch [3] further studied the time complexity of both sequentially consistent and linearizable registers, stacks and queues; a combination of algorithms and lower bounds demonstrated an inherent gap in the complexity of the two consistency conditions. Mavronicolas and Roth [17] proved a lower bound of $d + \min\{\epsilon, u\}/2$ on the sum of the times for a write and a read operation on a shared object. Chaudhuri, Gawlick, and Lynch [6] gave an algorithm that uses a tradeoff parameter $c \in [0, d - u]$ to balance the speed of reads and writes, with the time for a read at most $u + c$ and the time for a write at most $d + u - c$, giving an upper bound of $d + 2u$ for the sum of a read and a write operation. Inspired by Weihl’s work [21] using commutativity properties of operations for transaction processing, Kosa [13] characterized operations by axioms on what operation sequences are legal and proved a variety of upper and lower bounds in different models.

We make the following contributions to this problem.

**Lower Bounds:** We first show several lower bounds on the elapsed time of various operations that can be achieved by any algorithm implementing linearizable shared objects in the model of interest. As in [13], the operations are characterized by general algebraic properties that they possess. Results for specific operations can be obtained via specialization, simply by showing that the operation in question satisfies the generic properties. In particular, in Theorem 2, we generalize a $u/4$ lower bound for read operations from [3] so that it holds for all operations that are “pure accessors” (informally, they don’t change the state of the shared object). We then, in Theorem 3 generalize and improve a $u/2$ lower bound—proved separately in [3] for writing a register, pushing onto a stack, and enqueuing into a FIFO queue—to $(1 - 1/n)u$: our new bound holds for any operation that, informally, changes the state of the object in a way that depends on which in a series of instances of the operation occurs last.

Using a new technique that we developed, we prove two more lower bounds. Our technique overcomes a limitation of the classic shifting technique [10], which we used in the first two lower bounds, and allows us to prove larger lower bounds. In our third lower bound (Theorem 4), we show that no operation can be implemented faster than $d + \min\{\epsilon, u, \frac{d}{4}\}$, if the operation both accesses and changes the state in a particular way. Finally, in Theorem 5, we show that the sum of the time for two types of operations must also be at least $d + \min\{\epsilon, u, \frac{d}{4}\}$, where, informally, the first operation changes the state in a certain way and the second operation is a pure accessor that can distinguish between the order in which two instances of the first operation occur.

Our results provide improved lower bounds for the operations Read-Modify-Write and Write on registers; Enqueue and Dequeue on FIFO queues; Push and Pop on stacks; and Insert and Delete on trees. We also show improved lower bounds for the sum of Enqueue and Peek on FIFO queues, and the sum of Insert or Delete with the Depth query on trees. Furthermore, we obtain the first lower bounds for Peek on stacks and queues, and for querying the depth of trees.

**Upper Bounds:** To the best of our knowledge, no algorithms for implementing objects of arbitrary data type have been previously proposed, other than the folklore algorithms mentioned above. We present the first such algorithm. The algorithm uses the axiomatic properties of different operations to reduce the running time of each operation to below $2d$. In particular, each operation that only observes the state takes $d - X$ time, each operation that only changes the state takes $X + \epsilon$ time, and each operation that both observes and changes the state takes $d + \epsilon$ time, where $X$ is a parameter in $[0, d - \epsilon]$ that provides a tradeoff between two classes of operations. As a result, we obtain faster implementations of many common data structures.

**Examples:** Combining our new upper and lower bounds, with appropriate choices of the parameter $X$ to the algorithm, we reduce the gap between the known bounds and in some cases have tight bounds. Here we present some example applications of our improved bounds to the most common operations on some of the fundamental data types. Table 1 shows our results applied to Read-Modify-Write registers; in addition to the standard Read and Write operations, they support the atomic mutator/accessor Read-Modify-Write, which returns the current value in the register before updating it. We have improved the lower bounds on both mutators for these objects. Tables 2 and 3 give the application of our bounds to queues and stacks, respectively. These data types support Enqueue/Push to add items, Dequeue/Pop to remove and return items, and Peek to return information without altering the state of the object. We have improved the lower bounds for all but one of the operations for these data types, and have tight or nearly tight bounds. Finally, in Table 4, we apply the bounds to inserting, deleting, and finding the depth of a node in a simple, rooted tree data type.
Tab. 1: Operation Bounds for Read/Write/Read-Modify-Write Registers

<table>
<thead>
<tr>
<th>Operation</th>
<th>Previous Lower Bound</th>
<th>New Lower Bound</th>
<th>New Upper Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-Modify-Write</td>
<td>(d_{[13]})</td>
<td>(d + \min{\epsilon, u, d/3}) (Thm. 4)</td>
<td>(d + \epsilon)</td>
</tr>
<tr>
<td>Write</td>
<td>(u/2_{[8]})</td>
<td>((1 - 1/n)u) (Thm. 3)</td>
<td>(\epsilon)</td>
</tr>
<tr>
<td>Read</td>
<td>(u/4_{[8]})</td>
<td>—</td>
<td>(\epsilon)</td>
</tr>
<tr>
<td>Write + Read</td>
<td>(d_{13})</td>
<td>—</td>
<td>(d + \epsilon)</td>
</tr>
</tbody>
</table>

Tab. 2: Operation Bounds for Queues

<table>
<thead>
<tr>
<th>Operation</th>
<th>Previous Lower Bound</th>
<th>New Lower Bound</th>
<th>New Upper Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enqueue</td>
<td>(u/2_{[3]})</td>
<td>((1 - 1/n)u) (Thm. 3)</td>
<td>(\epsilon)</td>
</tr>
<tr>
<td>Dequeue</td>
<td>(d_{[3]})</td>
<td>(d + \min{\epsilon, u, d/3}) (Thm. 4)</td>
<td>(d + \epsilon)</td>
</tr>
<tr>
<td>Peek</td>
<td>—</td>
<td>(u/4) (Thm. 2)</td>
<td>(\epsilon)</td>
</tr>
<tr>
<td>Enqueue + Peek</td>
<td>(d_{[13]})</td>
<td>(d + \min{\epsilon, u, d/3}) (Thm. 5)</td>
<td>(d + \epsilon)</td>
</tr>
</tbody>
</table>

Tab. 3: Operation Bounds for Stacks

<table>
<thead>
<tr>
<th>Operation</th>
<th>Previous Lower Bound</th>
<th>New Lower Bound</th>
<th>New Upper Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push</td>
<td>(u/2_{[3]})</td>
<td>((1 - 1/n)u) (Thm. 3)</td>
<td>(\epsilon)</td>
</tr>
<tr>
<td>Pop</td>
<td>(d_{[3]})</td>
<td>(d + \min{\epsilon, u, d/3}) (Thm. 4)</td>
<td>(d + \epsilon)</td>
</tr>
<tr>
<td>Peek</td>
<td>—</td>
<td>(u/4) (Thm. 2)</td>
<td>(\epsilon)</td>
</tr>
<tr>
<td>Push + Peek</td>
<td>(d_{[13]})</td>
<td>—</td>
<td>(d + \epsilon)</td>
</tr>
</tbody>
</table>

Tab. 4: Operation Bounds for Simple Rooted Trees

<table>
<thead>
<tr>
<th>Operation</th>
<th>Previous Lower Bound</th>
<th>New Lower Bound</th>
<th>New Upper Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>(u/2_{[13]})</td>
<td>((1 - 1/n)u) (Thm. 3)</td>
<td>(\epsilon)</td>
</tr>
<tr>
<td>Delete</td>
<td>(u/2_{[13]})</td>
<td>((1 - 1/n)u) (Thm. 3)</td>
<td>(\epsilon)</td>
</tr>
<tr>
<td>Depth</td>
<td>—</td>
<td>(u/4) (Thm. 2)</td>
<td>(\epsilon)</td>
</tr>
<tr>
<td>Insert + Depth</td>
<td>(d_{13})</td>
<td>(d + \min{\epsilon, u, d/3}) (Thm. 5)</td>
<td>(d + \epsilon)</td>
</tr>
<tr>
<td>Delete + Depth</td>
<td>(d_{13})</td>
<td>(d + \min{\epsilon, u, d/3}) (Thm. 5)</td>
<td>(d + \epsilon)</td>
</tr>
</tbody>
</table>

Please see Section 5 for a comprehensive discussion of the consequences of our bounds, in particular how the classification scheme used in the lower bounds interacts with that used in the algorithm.

A preliminary version of this work appeared as [20].

**Related work.** Several authors have used other variations on the shifting technique for proving lower bounds. Arjomandi, Fischer and Lynch [11], Attiya and Mavronicolas [2], and Rhee and Welch [18] used methods of reordering and retiming events based on causal dependence to move operations within a run without changing the result of the execution. Dolev, Halpern, and Strong [7] multiplicatively scaled the times of occurrence of all events at a single process to prove a lower bound on the achievable precision in clock synchronization. Fischer, Lynch, and Merritt [8] use an extension of this scaling technique, scaling each process’s local clock instead of just one’s, to generalize their result.

More recent work has focused on additional limitations of linearizability when considering asynchronous, and potentially unreliable, communication. Tradeoffs between safety and liveness of register implementations were pointed out by Brewer [5, 4] and formalized by Gilbert and Lynch [9, 10] (the so-called “CAP Theorem”). In some sense, these results can be viewed as the logical extreme of Lipton and Sandberg’s lower bound when \(d\) goes to infinity.

All of these negative results about the inherent performance costs of linearizability fuel interest in different ways to relax the original notion of linearizability. One direction is toward more relaxed consistency conditions (e.g., eventual consistency [19]), in which the definition of the (sequential) data type is unchanged but the set of allowed concurrent behaviors is expanded. Another direction, suggested by Henzinger et al. [11], is to keep the original notion of linearizability but apply it to a weaker, but still sequential, version.
of the data type.

2 Model and Definitions

2.1 Specifying Arbitrary Data Types

The data type $T$ of an object specifies a set of operations $OPS(T)$ which are meaningful for this type. Each operation $OP$ in $OPS(T)$ has an invocation $OP.inv$ and a response $OP.resp$. The invocation can include an argument, drawn from a set of possible arguments, and the response can have a return value, drawn from a set of possible return values.

In the sequential case, the invocation and response for an operation are indivisible. We thus represent a specific invocation $OP.inv(arg)$ and its immediately following specific response $OP.resp(ret)$ more concisely as $OP(arg, ret)$. This pair is called an instance of operation $OP$. (In the distributed case, invocation and response will be separated in time; see Section 2.2.)

A particular set of sequences of operation instances, denoted $L(T)$, represents all the legal sequences for the type $T$; these are the sequences that capture the semantics of the data type.

For example, the read-write register type over a set $V$ of values has operations read and write. Read has invocation $read(−)$, with no argument, and response $return(v), v ∈ V$. Write has invocation $write(v), v ∈ V$, and response $ack(−)$, with no return value. We denote operation instances more concisely as $read(−, v)$ and $write(v, −)$. Assuming the register has $v_0$ as its initial value, the set of legal sequences contains all (and only) sequences of operation instances such that each read returns the value written by the latest preceding write; if there is no preceding write then the read returns $v_0$.

We require the following constraints on the set of legal sequences $L(T)$:

- **Prefix Closure**: If $ρ$ is in $L(T)$, then every prefix of $ρ$ is also in $L(T)$. Also, the empty sequence is in $L(T)$.
- **Completeness**: If $ρ$ is in $L(T)$, then for every operation invocation $OP.inv(arg)$, there exists a response $OP.resp(ret)$ such that $ρ.OP(arg, ret)$ is in $L(T)$.
- **Determinism**: If $ρ.OP(arg, ret)$ is in $L(T)$, then there is no $ret′ ≠ ret$ such that $ρ(OP(arg, ret′)$ is in $L(T)$.

Two sequences $ρ_1$ and $ρ_2$ of operation instances are equivalent, denoted $ρ_1 ≡ ρ_2$, if for every sequence $ρ_3$ of operation instances, $ρ_1.ρ_3$ is legal if and only if $ρ_2.ρ_3$ is legal.

Our algorithm relies on a classification of operations into those that, informally speaking, change the state of the object, observe the state of the object, or do both. We now give more precise definitions of these kinds of operations with respect to some type $T$.

**Definition**: An operation $MOP$ is a mutator if there exists a sequence $ρ$ of operation instances (over all operations in $OPS(T)$) and an instance $mop$ of $MOP$ such that $ρ.mop$ is legal but $ρ ≠ ρ.mop$.

Informally, $mop$ changes the state of the object in such a way that some later sequence of operation instances can tell $mop$ occurred.

**Definition**: An operation $AOP$ is an accessor if there exists a legal sequence $ρ$ of operation instances, an operation instance $op$, and an operation instance $aop$ of $AOP$ such that $ρ.aop$ and $ρ.op$ are both legal but $ρ.op.aop$ is illegal.

Informally, $aop$ learns something about the state of the object, in particular, whether or not $op$ has occurred.

**Definition**: An operation is a pure mutator if it is a mutator but not an accessor, and it is a pure accessor if it is an accessor but not a mutator.

**Definition**: A mutator $MOP$ is an overwriter if, for every instance $mop$ of $MOP$ and every sequence $ρ.op$ of operation instances (over all operations in $OPS(T)$), if $ρ.mop$ and $ρ.op.mop$ are both legal, then $ρ.mop ≡ ρ.op.mop$. 
Informally, mop sets the entire state of the object.

Since an operation that is neither a mutator nor an accessor accomplishes nothing, we do not consider such operations. We also only consider data types that have at least one accessor and at least one mutator (which could be the same operation).

2.2 System Model

This subsection details the distributed system model that we use. We are interested in shared objects, which provide data storage to the user while disguising lower-level message passing. We view these objects as a layer between the user (or application) and the system’s message-passing interface. From above, a user can interact with our shared object by invoking operations, and accepting the object’s responses to those invocations. From below, the implementation of the shared object sends and receives messages between processes. We assume that a message is sent by a single process to a single receiving process, and contains a unique identifier specifying these two processes. The layer providing the illusion of the shared object is called a shared object implementation.

We consider a set \( \Pi = \{p_0, \ldots, p_{n-1}\} \) of \( n \) processes, each modeled as a state machine. Transitions of the state machines are triggered by the occurrence of an event. There are three kinds of events: the receipt of a message, a timer going off, and an invocation of an operation instance. In more detail, the state machine for each process consists of a set of states, a subset of which form the initial states, and a transition function. Each state contains some number of timers, which are variables that hold either a clock time or \( \perp \) (indicating that the timer is not set). In an initial state no timers can be set. The input to the transition function is the current state of the process, a triggering event, and a local clock value; the output of the transition function is a new state for the process, a set of messages to send, and optionally an operation instance response.

A step of a process is a 6-tuple \((s, T, C, M, R, s')\), where \( s \) and \( s' \) are states (the “old” state and the “new” state respectively), \( T \) is a trigger event, \( C \) is a clock value (real number), \( M \) is a set of messages, and \( R \) is either \( \emptyset \) or an operation instance response, such that \( M, R \) and \( s' \) are the result of the transition function operating on \( s, T, \) and \( C \).

A view of a process is a sequence of steps such that

- the old state of the first step is an initial state;
- the old state of each step is equal to the new state of the next step;
- in each step, the value of each timer in the old state is at most equal to the clock time of the step (this ensures that a timer must eventually go off unless the process changes the timer);
- if the trigger for a step is a timer going off, then there exists a timer in the old state of the step that is equal to the clock time of the step (this makes sure that timer events only occur if there really was a timer set for this time);
- the sequence of clock values in the steps is increasing, and if the sequence is infinite then the clock values increase without bound (the process can read the clock but not change it; some other component manages it);
- at most one operation instance is pending at any point in the sequence, i.e., the most recent invocation has no response yet (this is a constraint on the user).

A timed view is a view in which a real number, the “real time”, is associated with each step. Real times must be increasing, and if the sequence is infinite then they must increase without bound. Furthermore, the local clock must have no drift with respect to the real time, i.e., there exists \( c \) such that, for every step of the timed view, the difference between the local clock value and the real time, called the offset, is exactly \( c \).

A run is a set of timed views, one for each of the \( n \) processes, such that every message receipt has exactly one matching message send, and every message send has at most one message receipt. However, at this point we put no constraint on the delays; in fact, a message could have negative delay or not be received at all. Define last-time of a run to be \( \infty \) if at least one timed view is infinitely long, otherwise define it to be the maximum, over all the timed views in the run, of the real time associated with the last step of the view.

A run is complete if:
2.3 Correctness Condition

A run is admissible with respect to parameters $\epsilon$, $d$, and $u$ if:

- the local clocks are synchronized to within $\epsilon$, i.e., letting $c_i$ be the local clock offset for process $p_i$, $|c_i - c_j| \leq \epsilon$ for all processes $p_i$ and $p_j$, and
- all delays for messages that are received are in the range $[d - u, d]$; furthermore, if there is a send of a message at real time $t$, and the message has no matching receive, then the last real time for any step by the intended recipient is less than $t + d$.

2.3 Correctness Condition

There are two main parts to our definition of what it means for a shared object algorithm to be correct. First, we require that it respond to invocations:

- In every complete admissible run, every operation invocation has a matching response and every response has a matching invocation.

The second part requires that the values of the responses be “correct”. We rely on the sequential specification to characterize correctness via the property of linearizability [12]. Linearizability gives us the ability to treat each data type separately, since a run is linearizable if and only if the restriction of the run to each individual object is linearizable [12]. From here on, we will only consider a single arbitrary object.

Thanks to the first requirement, we know that every complete admissible run has matching invocations and responses. Thus we can bundle together each invocation with its matching response to get an operation instance; that is, we can extract a set of operation instances from the run. Our second requirement is that:

- In every complete admissible run $R$, there is a permutation $\pi$ of the set of operation instances in $R$ such that (i) $\pi$ is legal and (ii) if the time of the step of $R$ containing the response for operation instance $op_1$ is less than the time of the step of $R$ containing the invocation for operation instance $op_2$, then $op_1$ precedes $op_2$ in $\pi$.

The first criterion ensures that $\pi$ obeys the sequential specification of the data type. The second criterion ensures that the real-time order of non-overlapping operations is respected: two operations in $R$ which overlap in real time may occur in $\pi$ in either order, but non-overlapping operations must stay in their real-time order.

The time complexity of an operation $OP$ in an implementation of a data type is the maximum, over all complete admissible runs of the algorithm and all instances of $OP$ in those runs, of the time that elapses between the invocation of the instance and its response. This quantity is denoted $|OP|$.

In this paper, we will consider only algorithms satisfying the following two conditions:\footnote{In fact, History Oblivion is only needed for the lower bounds in Section 4, not those in Section 3.}

1. Eventual Quiescence: Every complete admissible run with a finite number of operations is finite (i.e., every view is finite).

2. History Oblivion: Let $R_1$ and $R_2$ be two complete admissible runs such that the same finite sequence of operation instances is executed by $p_0$ in both $R_1$ and $R_2$, and the other processes do not execute any operation instances in $R_1$ or $R_2$. Because of Eventual Quiescence, $R_1$ and $R_2$ are both finite. Then for each process $p_j$, the final state of $p_j$ must be the same in both $R_1$ and $R_2$. That is, nothing about the clock times at which events occur, or the order in which messages arrive, is permanently recorded in the state—all that matters is the sequence of operation instances executed.
2.4 Mechanisms for Lower Bounds

Many of our lower bounds consider specific runs in which all messages from any particular process to any particular other process have the same delay; we say such runs have pair-wise uniform delays.

The next definition is used to set up several of our lower bounds, by specifying a prefix of a run in which a sequence $\rho$ of operation instances is executed. Consider a fixed algorithm $A$. Given a finite legal sequence $\rho$ of operation instances, an $n$-vector $C$ of clock offsets, and an $n$-by-$n$ matrix $D$ of message delays, let $R_A(\rho, C, D)$ be the unique complete run of $A$ in which

- the clock offset of $p_i$ is $c_i$, $0 \leq i < n$,
- every message from $p_i$ to $p_j$ has delay $d_{ij}$, $0 \leq i, j < n$,
- $p_0$ invokes the operation instances in $\rho$ sequentially, starting at its clock time 0 and with no gaps between the return of one instance and the invocation of the next, and
- no other operations are invoked.

By linearizability of the algorithm and determinism of the data type, each instance in the run returns the same response as in $\rho$, and thus $p_0$ executes $\rho$. By Eventual Quiescence, $R_A(\rho, C, D)$ is finite.

We briefly review the classic shifting technique for proving lower bounds. Let $R$ be a run of an algorithm. Let $\vec{x} = (x_0, \ldots, x_{n-1})$ be a vector of $n$ real numbers. We define $\text{shift}(R, \vec{x})$ to be the result of adding $x_i$ to the real time associated with each step of $p_i$’s timed view in $R$, $0 \leq i < n$. It is straightforward to see that $\text{shift}(R, \vec{x})$ is also a run of the algorithm, and if $R$ is complete, then so is $\text{shift}(R, \vec{x})$. However, the property of admissibility is not necessarily preserved. The changes to the clock functions and message delays were quantified in [10] to be:

**Theorem 1.** Let $R$ be a run and $\vec{x}$ a vector of reals.

1. The clock offset for $p_i$ in $\text{shift}(R, \vec{x})$ is equal to $c_i - x_i$, where $c_i$ is the clock offset for $p_i$ in $R$.
2. If a message from $p_i$ to $p_j$ has delay $\delta$ in $R$, then the message has delay $\delta - x_i + x_j$ in $\text{shift}(R, \vec{x})$.

By construction, each process has the same view in $\text{shift}(R, \vec{x})$ as it does in $R$; only the real times at which steps take place have changed, but processes have no way of observing that.

3 Lower Bounds with Classic Shifting

In this section we prove two lower bounds, one on pure accessors and one on certain kinds of mutators. Both lower bounds use the classic shifting technique just reviewed.

3.1 Pure Accessor Lower Bound

We start by generalizing a standard shifting argument applied in [3] to read operations on read-write registers. We use this method to prove a lower bound of $\frac{3}{4}$ on all pure accessors.

**Theorem 2.** In any system with $n \geq 3$ processes, $c$-bounded clocks, and message delays in the interval $[d - u, d]$, any shared object implementation must have $|AOP| \geq \frac{n}{4}$ for every operation $AOP$ of the object’s type that is a pure accessor.

**Proof:** For the sake of contradiction, we assume that there is some linearizable implementation $A$ of the shared object and some pure accessor $AOP$ of the data type such that the algorithm $A$ achieves $|AOP| < \frac{n}{4}$. We will construct a complete admissible run of the algorithm that is not linearizable.

By the definition of accessor, there exists a legal sequence $\rho$ of operation instances, an operation instance $op$, and an operation instance $aop$ of $AOP$ such that $\rho.aop$ and $\rho.op$ are both legal but $\rho.op.aop$ is illegal. Since $AOP$ is a pure accessor, it is not a mutator, and thus $\rho \equiv \rho.aop$. Thus, $\rho.aop.op$ is legal, and in fact $\rho.aop...aop.op$ is legal for any number of instances of $aop$ between $\rho$ and $op$. Let $aop = AOP(arg, ret)$. By Completeness of the data type, there exists some $ret'$ such that $\rho.op.AOP(arg, ret')$ is legal, but since
ρ.op.aop is illegal, ret' must be different than ret. Denote AOP(arg, ret') by aop'. Thus ρ.op.aop'...aop' is legal, for any number of instances of aop' following op. And in fact ρ.aop...aop.op.aop'...aop' is legal.

Let \( R'_1 \) be \( R_A(\rho, C, D) \), where \( c_i = 0, 0 \leq i < n \), and \( d_{ij} = d - \frac{n}{\tau}, 0 \leq i, j < n \). \( R'_1 \) is complete by definition, admissible by choice of \( C \) and \( D \), and finite as noted above.

Let \( t = \text{last-time}(R'_1) + \frac{u}{\tau} \) and let \( k = \left\lceil \frac{|\text{OP}|}{u/\tau} \right\rceil \). We now extend \( R'_1 \) into a complete run \( R_1 \) as follows:

- Invoke \( \text{OP}(\text{arg}) \) at \( p_0 \) at real time \( t + \frac{u}{\tau} \), where \( \text{op} = \text{OP}(\text{arg}, \text{oret}) \).
- Invoke \( \text{AOP}(\text{arg}) \) at \( p_{i\%2} \) at real time \( t + i \cdot \frac{u}{\tau}, 0 \leq i \leq k + 1 \).
- Message delays are all \( d - \frac{n}{\tau} \) (as in \( R'_1 \)).

Thus \( p_0 \) and \( p_1 \) execute \( k + 2 \) alternating, non-overlapping instances of \( \text{AOP} \) with the first one occurring before the \( \text{OP} \) instance is invoked and the last one occurring after the \( \text{OP} \) instance finishes; denote the \( \text{AOP} \) instances by \( \text{aop}_0, ..., \text{aop}_{k+1} \) in order of occurrence and the \( \text{OP} \) instance by \( \text{op}' = \text{OP}(\text{aop}, \text{oret}) \).

By the assumed correctness of the algorithm, there must exist some \( j \), with \( 0 \leq j \leq k \), such that \( \rho.\text{aop}_0...\text{aop}_j.\text{op}'.\text{aop}_{j+1}...\text{aop}_{k+1} \) is illegal. Since \( \text{AOP} \) is a pure accessor, \( \rho.\text{aop}_0...\text{aop}_j.\text{op}'.\text{aop}_{j+1}...\text{aop}_{k+1} \) is illegal. Thus, since the arguments above and determinism of the data type, \( \text{op}' = \text{op} \). Thus, from the arguments above and determinism of the data type, \( \text{aop}_0 \) through \( \text{aop}_j \) are all equal to \( \text{aop} \), and \( \text{aop}_{j+1} \) through \( \text{aop}_{k+1} \) are all equal to \( \text{aop}' \).

We now have two cases, depending on the parity of \( j \), which, informally speaking, indicates whether \( p_0 \) or \( p_1 \) was the last process to see the old state of the object.

Case 1: \( j \equiv 0 \pmod{2} \). In this case, \( \text{aop}_j \) is invoked at \( p_0 \). Let \( R_2 = \text{shift}(R_1, (\frac{u}{\tau}, -\frac{n}{\tau}, 0, \ldots, 0)) \). By Theorem 1 this new run has message delays \( d'_{ij} \), where

- \( d'_{01} = (d - \frac{u}{\tau}) - \frac{u}{\tau} + (-\frac{u}{\tau}) = d - u \) and \( d'_{10} = d \)
- \( d'_{02} = d - \frac{3u}{\tau} \) and \( d'_{20} = d - \frac{u}{\tau} \)
- \( d'_{12} = d - \frac{u}{\tau} \) and \( d'_{21} = d - \frac{3u}{\tau} \)

Message delays between all other pairs of processes remain \( d - \frac{n}{\tau} \). The maximum clock skew in \( R_2 \) occurs between \( p_0 \) and \( p_1 \), which are shifted in opposite directions, each by \( \frac{u}{\tau} \), giving a total skew of \( \frac{2u}{\tau} \), which is less than \( \epsilon \), because the smallest possible value of \( \epsilon \) is \( (1 - \frac{1}{n})u \geq \frac{2u}{3} \) since we have at least 3 processes. Thus \( R_2 \) is admissible.

In \( R_2 \), the sequence \( \rho \) still ends before any of the \( \text{aop}_j \)'s begin, as we made sure that in \( R_1 \) there was a gap of size \( \frac{u}{\tau} \) between the end of \( \rho \) and the start of \( \text{aop}_0 \). However, in \( R_2 \) we have \( \text{aop}_{j+1} \) responding before \( \text{aop}_j \) is invoked. There is no way to linearize these operations, as we now argue. If \( \text{op} \) is linearized before \( \text{aop}_j = \text{aop} \), then the linearization is illegal, since instances of \( \text{AOP} \) after \( \text{op} \) must be \( \text{aop}' \), not \( \text{aop} \). On the other hand, if \( \text{op} \) is linearized after \( \text{aop}_j \), then it must also be after \( \text{aop}_{j+1} = \text{aop}' \), and the linearization is again illegal, since instances of \( \text{AOP} \) after \( \text{op} \) must be \( \text{aop} \), not \( \text{aop}' \). Since \( R_2 \) is complete and admissible, this is a contradiction to our assumption that \( |\text{AOP}| < \frac{u}{\tau} \).

Case 2: \( j \equiv 1 \pmod{2} \). In this case, \( \text{aop}_j \) is invoked at \( p_1 \). The proof is symmetric to that for Case 1, defining \( R_2 = \text{shift}(R_1, (-\frac{u}{\tau}, \frac{u}{\tau}, 0, \ldots, 0)) \).

### 3.2 Last-Sensitive Mutator Lower Bound

By studying the behavior of two concurrent instances of the same operation, it has been shown that a class of operations which are sensitive to order, such as write, push, and enqueue, have a time lower bound of \( \frac{u}{\tau} \). We extend our consideration to more than two concurrent operation instances. In the following theorem, we prove, using the standard shifting technique, a lower bound of \( (1 - \frac{1}{k})u \), where \( k \) is any positive integer at most \( n \), for a group of operations including write, push, and enqueue. Specifically, for common operations in this class, such as writing a register, pushing on a stack, and enqueuing on a queue, if there are \( n \) processes in a run, we can set \( k \) equal to \( n \), so the lower bound is \( (1 - \frac{1}{n})u \).

**Definition:** An operation \( \text{OP} \) is transposable if for any two distinct instances \( \text{op}_1, \text{op}_2 \in \text{OP} \) and any sequence \( \rho \) of operation instances, if \( \rho.\text{op}_1 \) and \( \rho.\text{op}_2 \) are both legal, then \( \rho.\text{op}_1.\text{op}_2 \) and \( \rho.\text{op}_2.\text{op}_1 \) are also both legal.
We next show that transposability can be extended to more than two instances.

**Lemma 1.** If $OP$ is a transposable operation with $k$ distinct instances $op_i$, $0 \leq i < k$, and $\rho$ is a sequence of operation instances such that $\rho.op_i$ is legal, $0 \leq i < k$, then for any permutation $\pi$ of any subset of $\{op_0, \ldots, op_{k-1}\}$, $\rho.\pi$ is legal.

**Proof:** We prove the lemma by induction on $|\pi|$, the length of $\pi$. First we prove the two base cases. Suppose $|\pi| = 0$. Since $\rho.op_0$ is legal, the Prefix Closure property of the data type implies that $\rho$ is legal. The base case $|\pi| = 1$ follows from the hypothesis that $\rho.op_i$ is legal for each $op_i$.

For the inductive case, consider permutation $\pi = \pi'.op_i$, $\rho.\pi'$ of length at least 2. By the inductive hypothesis, $\rho.\pi'.op_i$ and $\rho.\pi'.op_j$ are both legal. By the definition of a transposable operation, then, $(\rho.\pi').op_i.op_j$ and $(\rho.\pi').op_j.op_i$ are both legal. Specifically, $\rho.\pi = \rho.\pi'.op_i.op_j$ is legal.

We now consider a particular class of transposable operations, those in which the last operation instance makes a difference:

**Definition:** A transposable operation $OP$ is said to be last-sensitive if there exist $k$ distinct instances of $OP$, $op_i$, $0 \leq i < k$, and a sequence $\rho$ of operation instances with $\rho.op_i$ legal, $0 \leq i < k$, such that for any two permutations $\pi$ and $\pi'$ of $\{op_0, \ldots, op_{k-1}\}$ where last($\pi$) $\neq$ last($\pi'$) $\rho.\pi$ and $\rho.\pi'$ are not equivalent.

**Theorem 3.** For any last-sensitive operation $OP$, in any implementation in a system with $n \geq k$ processes, $\epsilon$-bounded clocks, and message delays in the interval $[d – u, d]$, $|OP| \geq (1 – 1/k)u$.

**Proof:** We prove the theorem by contradiction. Suppose there is an implementation $A$ of a data type with a last-sensitive operation $OP$ in which $|OP| < (1 – 1/k)u$. Let $\rho$ and $op_0, \ldots, op_{k-1}$ be as in the definition of last-sensitive.

Our proof will have three steps. First, we construct a run using $\rho$ and the $op_i$’s. Then, for any linearization of the operation instances of that run, we use shifting to construct a second run. Finally, we argue—relying on the assumed short duration of $OP$ instances—that there is no linearization of the operation instances in our second run, giving a contradiction to the assumed correctness of the algorithm.

**Step 1: The First Run**

Let $C$ be the $n$-vector consisting of all 0’s. Let $D$ be the $n$-by-$n$ matrix with

$$d_{ij} = \begin{cases} d - \frac{(i-j)\%k}{k} \cdot u & \text{if } 0 \leq i, j < k \\ d - \frac{u}{2} \cdot u & \text{otherwise} \end{cases}$$

Let $R_1'$ be $R_A(\rho, C, D)$. $R_1'$ is complete by definition, admissible by choice of $C$ and $D$, and finite since $\rho$ is finite.

Let $t = $ last-time($R_1'$) + $\frac{u}{2}$. We now extend $R_1'$ into a complete run $R_1$ as follows (see top part of Figure 1):

- Invoke $OP(arg_i)$ at $p_i$ at real time $t$, where $op_i = OP(arg_i, ret_i)$, $0 \leq i < k$.

- Message delays are pair-wise uniform according to $D$, as in $R_1'$.

$R_1$ is admissible.

By assumption on the correctness and complexity of $A$, each invocation of $OP(arg_i)$ returns before real time $t + u$ in $R_1$; let $ret'_i$ be its return value and let $op'_i$ denote the operation instance $OP(arg_i, ret'_i)$. We now argue that $ret'_i$ equals $ret_i$, and thus that $op'_i$ equals $op_i$. By the assumed correctness of $A$, there exists at least one permutation $\pi$ of $\{op'_0, \ldots, op'_{k-1}\}$ such that $\rho.\pi$ is legal. Fix any such permutation $\pi$ for the rest of the proof. (Think of $\pi$ as the one chosen by the algorithm to linearize the concurrent $OP$ instances.) Let $\pi(i)$ denote the subscript of the $i$-th operation instance in $\pi$, $0 \leq i < k$.

**Claim 1.** $op'_i = op_i$, $0 \leq i < k$.  

$^{2}$ last($\pi$) denotes the last element of $\pi$
3.2 Last-Sensitive Mutator Lower Bound

Proof: We prove this by induction, but instead of doing induction on \( i \), which just indicates the id of the invoking process, we do induction on the order in which the instances occur in \( \pi \). More precisely, we show by strong induction that \( \text{op}'_{\pi(i)} = \text{op}_{\pi(i)} \), \( 0 \leq i < k \).

Assume that for all \( j, 0 \leq j < i \), \( \text{op}'_{\pi(j)} = \text{op}_{\pi(j)} \). Thus \( \rho.\text{op}'_{\pi(0)} \ldots \text{op}'_{\pi(i-1)} = \rho.\text{op}_{\pi(0)} \ldots \text{op}_{\pi(i-1)} \) which, by Lemma \( 1 \) is legal.

Since \( \rho.\pi \) is legal, the Prefix Closure property of the data type implies that \( \rho.\text{op}_{\pi(0)} \ldots \text{op}_{\pi(i-1)} \text{op}'_{\pi(i)} \) is legal. By Lemma \( 1 \) \( \rho.\text{op}_{\pi(0)} \ldots \text{op}_{\pi(i-1)} \text{op}_{\pi(i)} \) is also legal. By determinism of the data type it must be that
\[
\text{op}'_{\pi(i)} = \text{op}_{\pi(i)},
\]
as desired. \( \square \)

Fig. 1: Runs used in the proof of Theorem 3
Step 2: The Second Run

We now construct a second run by shifting $R_1$ (see bottom part of Figure 1). Letting $op_z$ be last($\pi$), define $R_2 = shift(R_1, \bar{x})$, where

$$x_i = \begin{cases} \frac{-(1-k)}{2k} + \frac{(z-i)k}{k} \cdot u & \text{if } 0 \leq i < k \\ 0 & \text{if } k \leq i < n \end{cases}$$

Claim 2. $|x_i| \leq \frac{n}{2}$ for all $i$, $0 \leq i < n$.

Proof: If $i$ is between $k$ and $n - 1$ inclusive, then $x_i$ is 0. Suppose $i$ is between 0 and $k - 1$ inclusive. We show that

$$|\frac{-(1-k)}{2k} + \frac{(z-i)k}{k}| \leq \frac{1}{2}$$

Recall that $z$ must be between 0 and $k - 1$ inclusive, since $p_z$ is one of the processes that executes an $OP$ instance. If $z \geq i$, then $(z-i)k = z-i$, and if $z < i$, then $(i-z)k = i-z$. In both cases, the second term in the sum ranges from 0 to $\frac{k-1}{k}$. Thus the smallest that the sum can be is $\frac{-(k-1)}{2k}$, which is between $-\frac{1}{2}$ and 0, while the largest that the sum can be is $\frac{k-1}{2k}$, which is between 0 and $\frac{1}{2}$. \□

Clearly, $\rho$ is still executed in sequence by $p_0$ in $R_2$. By construction of $R_1$, there is a gap of duration $\frac{n}{2}$ between the end of $R_1$ (the prefix in which $\rho$ is executed) and the concurrent invocations of the $OP$ instances. Since, by Claim 3.2, no process is shifted by more than $\frac{n}{2}$ in either direction, it remains true in $R_2$ that the earliest invocation of an $OP$ instance does not overlap the execution of $\rho$.

Claim 3. $R_2$ is an admissible run.

Proof: We use the formulas in Theorem 1 to calculate the clock skews and message delays in $R_2$.

Recall that all the clock offsets in $R_1$ are 0, leading to a maximum clock skew of 0. Thus the maximum clock skew in $R_2$ is

$$\max_{0 \leq i, j < k} \left[ \left( \frac{-(1-k)}{2k} + \frac{(z-i)k}{k} \right) u - \left( \frac{-(1-k)}{2k} + \frac{(z-j)k}{k} \right) u \right] = (1 - \frac{1}{k})u.$$ 

Since the smallest that $\epsilon$ can be is $(1 - \frac{1}{n})u$, and $k \leq n$, it follows that the maximum clock skew in $R_2$ is at most $\epsilon$.

It remains to check that all message delays are in the range $[d - u, d]$ in $R_2$. The new message delay from $p_i$ to $p_j$, $0 \leq i, j < k$, is:

$$d'_{i,j} = d - \frac{(i-j)k}{k}u - \left( \frac{-(1-k)}{2k} + \frac{(z-i)k}{k} \right) u + \left( \frac{-(1-k)}{2k} + \frac{(z-j)k}{k} \right) u$$

$$= d - \frac{(i-j)k}{k}u - \frac{(z-i)k}{k}u + \frac{(z-j)k}{k}u.$$ 

For each choice of $i$ and $j$ in $\{0, \ldots, k-1\}$, there are six cases for the possible relative ordering of $i$, $j$, and $z$, where $p_z$ is the process with operation instance is last in $\pi$. We give the calculations for the first case, then omit them for the other cases, because they are entirely similar.

- $i < j \leq z$:

$$d'_{i,j} = d + \left( \frac{-(i-j+k)k}{k} - \frac{(z-i)k}{k} + \frac{(z-j)k}{k} \right) u$$

$$= d + \left( \frac{(j-i-k)}{k} - \frac{z-i}{k} + \frac{z-j}{k} \right) u$$

$$= d - u \quad \text{\footnote{3 We use the algebraic modulo operator, where for } a > 0, (-a)\%k = k - (a\%k).}$$
3.2 Last-Sensitive Mutator Lower Bound

Let $\pi$ be a sequence of operation instances such that $p.\pi$ is legal. Furthermore, $\pi'$ must respect the ordering of non-overlapping operation instances in $R_2$. Fix a particular such $\pi'$ for the rest of the proof.

**Step 3: The Contradiction**

To reach a contradiction, we show that, because of the way we have constructed $R_2$, $op_z$ cannot be the last instance in $\pi'$. $p_z$ is shifted by $-(k-1)u$ and $p_{(z+1)\%k}$ is shifted by $(-\frac{(k-1)}{2k} + \frac{k-1}{k})u$. We have assumed that every instance of operation $OP$ completes in time less than $(1 - 1/k)u = \frac{k-1}{k}u$. Thus, since

$$\frac{-(k-1)}{2k}u + \frac{k-1}{k}u = \left(-\frac{(k-1)}{2k} + \frac{k-1}{k}\right)u$$

we have that, in $R_2$, $op_z$ returns strictly before $op_{(z+1)\%k}$ is invoked. This means that $op_z$ must come before $op_{(z+1)\%k}$ in $\pi'$, by the rules of linearizability.

Since $op_z = last(\pi)$, we now have $last(\pi) \neq last(\pi')$. By the assumption that $OP$ is last-sensitive, there is some sequence $\rho'$ of operation instances such that exactly one of $\rho, \pi, \rho'$ and $\rho, \pi', \rho'$ is legal. We assume first that $\rho, \pi, \rho'$ is legal and $\rho, \pi', \rho'$ is illegal.

By Eventual Quiescence, $R_1$ is finite. Let $t_1 = last-time(R_1) + \frac{u}{2}$. Extend $R_1$ to a complete run $R''_1$ by having $p_0$ invoke the operation instances in $\rho'$ sequentially, starting at real time $t_1$ and with no gaps between the return of one instance and the invocation of the next. No other operations are invoked in the extension. Message delays are pair-wise uniform according to $D$, as in $R_1$, ensuring that $R''_1$ is admissible. By linearity of the algorithm and determinism of the data type, each instance returns the same response as in $\rho'$, and thus $p_0$ executes $\rho'$ in the extension.

Define $R''_2$ to be $shift(R''_1, \bar{x})$, where $\bar{x}$ is the same vector of shift amounts that was used to create $R_2$ from $R_1$. $R''_2$ is admissible for the same reason that $R_2$ is admissible. Since there is a gap in $R''_1$ of duration $\frac{u}{2}$ between the end of $R_2$ and the first invocation of $\rho'$, and no process is shifted by more than $\frac{u}{2}$ (by Claim 3.2), there is no overlap in $R''_2$ of the $OP$ instances and the instances making up $\rho$. Thus the linearization of $R''_2$ must be $\rho, \pi', \rho'$, which is illegal, contradicting the assumed correctness of the algorithm.

If $\rho, \pi', \rho'$ is legal and $\rho, \pi', \rho$ is illegal, we do the “reverse” of the procedure in the previous two paragraphs: We append $\rho'$ to $R_2$, and shift the resulting run by $-\bar{x}$ to get $R_1$ with $\rho'$ appended to it, which is illegal. □

**Corollary 1.** $|Write|, |Push|$, and $|Enqueue|$ are all at least $(1 - 1/n)u$.

**Proof:** We show that all the operations satisfy the hypotheses of Theorem 3. Since $Write$, $Push$, and $Enqueue$ are pure mutators, any instance of one of them is legal after any legal sequence $\rho$ of operation instances. Also, since these operations can take any argument, there are at least $k = n$ distinct instances of each. Finally, by executing a Read, a Pop, or a sufficiently long string of Dequeue’s, respectively for registers, stacks, and queues, we can determine which instance of $Write$, $Push$, or $Enqueue$ was executed last, which means that permutations with different last operation instances are not equivalent. Thus, the theorem holds for these operations. □
4 Lower Bounds with Modified Shift

4.1 Shifting and Chopping

The classic shifting technique is subject to the limitation that the shifted run must still be admissible. We now describe a technique that uses shifting in a more general way, which allows us to prove larger lower bounds. We start with an admissible run containing pair-wise uniform message delays, and then shift in a way that results in a single invalid message delay. This problem is fixed by chopping the shifted run before a wrong message delay occurs, and then extending the result in a valid way.

Another complication is that we sometimes need to append a shifted run to another run. To deal with this, we define shifting for run fragments (which do not need to start in initial process states) and define what it means to append one run fragment to another. This procedure relies on the History Oblivion assumption.

Define a run fragment to be a relaxation of the definition of run in which the first step in each view need not be an initial state of the process. Define the shift function to work on run fragments as well as runs.

Let $R$ be a run fragment with pair-wise uniform message delays given by $n$-by-$n$ matrix $D$ such that exactly one delay, say $d_{sr}$, is invalid. Given parameter $\delta \in [d - u, d]$, define $\text{chop}(R, \delta)$ to be the set of timed views defined as follows:

- Let $t_m$ be the real time when the first message from $p_s$ to $p_r$ is sent in $R$.
- Let $p_r$’s timed view be the prefix of $p_r$’s timed view in $R$ ending just before real-time $t^* = t_m + \min\{d_{sr}, \delta\}$.
- For each $p_i$ other than $p_r$, let $p_i$’s timed view be the prefix of $p_i$’s timed view in $R$ ending just before real-time $t^* + \delta_{ri}$, where $\delta_{ri}$ is the length of the shortest path from $p_r$ to $p_i$ with respect to the message delays in $D$.

**Lemma 2.** If $R$ is a run fragment with pair-wise uniform message delays, exactly one of which is invalid, then $R' = \text{chop}(R, \delta)$ is a run fragment with pair-wise uniform message delays all of which are valid, for all $\delta \in [d - u, d]$.

**Proof:** Let $d_{sr}$ be the invalid message delay. First, $R'$ is a run fragment since every message received in $R'$ is sent in $R'$. We can see this by supposing the contrary – that there is a message $m'$ sent from $p_i$ to $p_j$ in $R$ after $p_i$’s chopping point $t^* + \delta_{ri}$ and received in $R$ before $p_j$’s chopping point $t^* + \delta_{rj}$. Then we have that $\delta_{rj} - \delta_{ri} > d_{ij}$, so $\delta_{ri} + d_{ij} < \delta_{rj}$, so $\delta_{rj}$ is not actually the shortest path from $p_r$ to $p_j$, which is a contradiction.

Second, the message delays in $R'$ are pair-wise uniform by construction. We next argue that they are all valid:

1. All messages received in $R'$ have delays in the range $[d - u, d]$, because only one ordered pair of processes has an invalid delay, and we chop before the first message with that delay is received.
2. For each message $m$ sent at real time $t_m$ in $R'$ but not received in $R'$, the view of the recipient is chopped before $t + d$. For $m$ from $p_i$ to $p_j$, $t^* \leq t_m + \delta \leq t_m + d$, so $p_j$ is chopped before $t_m + d$. All other messages have delay at most $d$, so if they are not received in $R'$, the receiving process must have been chopped less than $d$ time after the message was sent.

Run fragment $R_2$ is appendable to run fragment $R_1$ if (1) $R_1$ is complete, (2) $R_1$ and $R_2$ have the same set of clock functions, (3) first-time($R_2$) > last-time($R_1$). (4) For each process, its last state in $R_1$ equals its first state in $R_2$. The result of appending $R_2$ to $R_1$ is the run fragment obtained by appending the timed view for $p_i$ from $R_2$ to the timed view for $p_i$ in $R_1$, for each process $p_i$.

4.2 Pair-Free Operation Lower Bound

We now use the modified shift technique to prove a lower bound on a class of operations which are both accessors and mutators. That is, operations in this class both return information about the state of the
shared object and change that state. That means that they are very sensitive to concurrent operation, since
the value which the operations should return depends on where in the linearization order they are placed.

**Definition:** An operation $OP$ is pair-free if there are two instances $op_1, op_2 \in OP$, and a sequence $\rho$ of
operation instances, such that $\rho. op_1$ and $\rho. op_2$ are both legal, but $\rho. op_1. op_2$ and $\rho. op_2. op_1$ are both illegal.

**Lemma 3.** Every pair-free operation is both an accessor and a mutator.

**Proof:** Let $OP$ be a pair-free operation and let $\rho, op_1, op_2$ be an operation sequence and two instances of
$OP$, respectively, as in the definition of pair-free operations. Since $op_2$ is legal after $\rho$ but not after $\rho. op_1$,
$\rho \neq \rho. op_1$. Then, by the definition of a mutator, $OP$ is a mutator. $OP$ also fits the definition of accessor,
since $\rho. op_1$ and $\rho. op_2$ are legal but $\rho. op_1. op_2$ is illegal. □

To prove this bound, we consider two processes in the system. If the operations respond too quickly, we
can invoke concurrent operations on the two processes, one slightly before the other, then the first of the
two operations will have the same return value as it would if it were invoked alone. With the modified shift,
though, we can shift the operations in opposite directions, so that the second operation is invoked first in
real time. Then it will have the same return value as if it were invoked alone, and the first operation will
have a different return value, contradicting the first conclusion. Thus, the operations must take longer to
force consistent linearization.

**Theorem 4.** If $OP$ is a pair-free operation, then $|OP| \geq d + m$, where $m = \min\{\epsilon, u, \frac{d}{3}\}$ on any linearizable
shared object implemented on $n \geq 2$ processes with message delays in the range $[d - u, d]$ and $\epsilon$-synchronized
clocks.

**Proof:** Assume in contradiction that there is an algorithm $A$ for which $|OP| < d + m$. Let $\rho, op_0 =
OP(arg_0, ret_0)$, and $op_1 = OP(arg_1, ret_1)$ be such that $\rho. op_0$ and $\rho. op_1$ are both legal, but $\rho. op_0. op_1$ and
$\rho. op_1. op_0$ are both illegal, as guaranteed by the definition of pair-free.

Here is an overview of the proof. (See Figure 3)

- We first construct a run $R_1$ that consists of a prefix in which one process executes $\rho$, and a suffix in
  which $p_0$ executes $op_0$.

- Next we define a run $R_2$ that is obtained from $R_1$ by having $p_1$ execute $op_1'$ (which has the same
  invocation as $op_1$ but a different return value) starting slightly after $op_0$ begins.

- We extract the suffix $S_2$ of $R_2$ containing $op_0$ and $op_1'$. We shift $S_2$ so that $p_1$ invokes $op_1'$ at the same
time that $p_0$ invokes $op_0$. This results in one illegal message delay, so we chop the shifted version of $S_2$
  and argue that the result contains $op_1'$. We then append the result to a run containing $\rho$ (with clock
  functions that match those in the shift), and extend to completion. Call the resulting run $R_3$. We show
  that in $R_3$, $p_0$ must still execute $op_0$.

- We extract the suffix $S_3$ of $R_3$ containing $op_0$ and $op_1'$. We shift $S_3$ so that $p_0$ invokes $op_0$ slightly after
  $p_1$ invokes $op_1'$. This results in one illegal message delay, so we chop the shifted version of $S_3$ and show
  that the result contains $op_0$. We then append the result to a run containing $\rho$ (with clock functions
  that match those in the shift), and extend to completion. Call the resulting run $R_4$. We prove that in
  $R_4$, $p_1$ must still execute $op_1'$.

- Finally, we let $R_5$ be the run that is the same as $R_4$ except that the execution of $op_0$ by $p_0$ is deleted.
  We argue that $p_1$ cannot tell the difference while it is executing $op_1'$ between $R_4$ and $R_5$, and thus
  executes $op_1'$ in $R_5$. But this is illegal, a contradiction.

**Step 1: Solo Execution by $p_0$**

Let $C_0$ be the $n$-vector $(-m, 0, \ldots, 0)$, $C_1$ be the $n$-vector $(0, -m, 0, \ldots, 0)$, and $C_2$ be the $n$-vector $(0, \ldots, 0)$. Since $m \leq \epsilon$, all of these result in valid clock skew when used as clock offsets. Let $D_v$ be any $n$-by-$n$ matrix
of valid message delays. Let $t = \max\{last-time(R_A(\rho, C_i, D_v)) : i = 0, 1, 2\} + m$. 

14
Let $D^1$ be the $n$-by-$n$ matrix of message delays defined as follows (also see Figure 2):

$$d_{ij}^1 = \begin{cases} 
  d - m & \text{if } i \neq 1 \text{ and } j = 0 \\
  d - m & \text{if } i = 1 \text{ and } j \neq 0 \\
  d & \text{otherwise}
\end{cases}$$

All these delays are valid since $m \leq u$.

We extend $R_A(\rho, C_1, D_v)$ to a complete run $R_1$ as follows:

- $p_0$ invokes $OP(\arg_0)$ at real time $t$; no other operations are invoked;
- message delays in the extension are pair-wise uniform according to $D^1$.

$R_1$ is admissible and finite. By the assumed correctness and complexity of the algorithm $A$ and the determinism of the data type, there is a response at $p_0$ before real time $t + d + m$ and that response is $ret_0$; i.e., $p_0$ executes $op_0$.

**Step 2: Adding an Execution by $p_1$**

We now modify $R_1$ to make another admissible run $R_2$ by inserting an invocation of $OP(\arg_1)$ by $p_1$ at real time $t + m$ and continuing to completion.

**Claim 4.** The view of $p_0$ in $R_2$ during real-time interval $[t, t + d + m]$ is the same as the view of $p_0$ in $R_1$ during real-time interval $[t, t + d + m]$.

**Proof:** There is no difference between $R_1$ and $R_2$ until real time $t + m$, when $p_1$ invokes $OP(\arg_1)$ in $R_2$ but not in $R_1$. The earliest that $p_0$ could receive information about this new invocation directly from $p_1$ is real time $t + m + d$, since $d_{01} = d$. The earliest that $p_0$ could receive such information indirectly, via one or more intermediate processes, is $t + m + 2(d - m)$, which is at least $t + m + d$, since $m \leq \frac{d}{3}$. In all cases, then $p_0$ cannot learn about the existence of $p_1$’s invocation until $t + d + m$ or later. \hfill \square

Thanks to this claim, we see that in $R_2$, the operation invoked by $p_0$ returns $ret_0$, and thus $p_0$ executes $op_0$ just as it does in $R_1$. Since $R_2$ is admissible and complete, $p_1$’s invocation of $OP(\arg_1)$ must have a response, and by the definition of pair-free, the response cannot be that of $op_1$. Denote this operation instance by $op_1' = OP(\arg_1, ret_1')$, with $ret_1' \neq ret_1$.

**Step 3: Shifting $p_1$ Earlier and Chopping**

Let $S_2$ be the run fragment that is the suffix of $R_2$ following $R_A(\rho, C_1, D_v)$. Let $\vec{x}$ be the $n$-vector $(0, m, 0, \ldots, 0)$ and $S'_2 = \text{shift}(S_2, \vec{x})$. Recalling that $C_1 = (0, -m, 0, \ldots, 0)$, we see that the clock offset vector in $S'_2$ is $C_2 = (0, \ldots, 0)$. The shift causes the delays of all messages sent by $p_1$ to increase by $m$ and the delays of all messages received by $p_1$ to decrease by $m$, while the remaining delays are unchanged. Thus
4.2 Pair-Free Operation Lower Bound

Fig. 3: Runs used in the proof of Theorem 4; prefixes containing \( \rho \) are omitted. Dotted arrows indicate illegal messages that we prevent by chopping. Crossed out delays correspond to those illegal messages. Dashed lines indicate where we chop a run.
the messages from \( p_1 \) to \( p_0 \) now have delay \( d + m \), the messages from \( p_1 \) to the other processes now have delay \( d \), and the messages received by \( p_1 \) now all have delay \( d - m \). The only invalid delay is that from \( p_1 \) to \( p_0 \). (See Figure 4.)

Thus \( S_2' \) is a run fragment that satisfies the requirements for the \( \text{chop} \) procedure. Let \( S_2'' = \text{chop}(S_2', d - m) \). We now show that \( p_1 \)'s timed view is chopped after \( \text{op}_1' \) returns, and thus \( p_1 \) executes \( \text{op}_1' \) in its entirety in \( S_2'' \).

In \( S_2 \), \( \text{op}_1' \) is invoked at real time \( t + m \) and returns before real time \( t + m + d + m \), by assumption that \( |\text{OP}| < d + m \). Thus in \( S_2'' \), after shifting \( p_1 \) by \(-m\), \( \text{op}_1' \) is invoked at real time \( t \) and returns before real time \( t + d + m \). The first message sent from \( p_1 \) to \( p_0 \) is sent at time \( t \) or later. By the definition of \( \text{chop} \), \( p_0 \)'s timed view is chopped at real time \( t + \min\{d + m, d - m\} = t + d - m \) or later.

The shortest path from \( p_0 \) to \( p_1 \) with respect to delays is the edge from \( p_0 \) to \( p_1 \), which has delay \( d - m \), since all delays are at least \( d - m \). By the definition of \( \text{chop} \), \( p_1 \)'s timed view is chopped at the real time that is \( d - m \) later than the time when \( p_0 \)'s timed view is chopped; this time is \((t + d - m) + (d - m)\) or later. Since \( m \geq \frac{d}{2} \), \( p_1 \)'s timed view is chopped at time \( t + d + m \) or later.

**Step 4: Repairing and Extending**

We now show that \( S_2'' \) is appendable to \( R = R_A(\rho, C_2, D_v) \). (1) By definition, \( R \) is complete. (2) Both \( S_2'' \) and \( R \) have \( C_2 \) as their clock offset vector. (3) Recall that the first step in \( S_2 \) occurs at real-time \( t + m \), where \( t \geq \text{last-time}(R) \). Thus, \( \text{first-time}(S_2'') \) occurs no earlier than \( t \) since the maximum shift amount is \( m \), which is at least \( \text{last-time}(R) \). (4) Since \( S_2 \) is the suffix of run \( R_2 \) that follows \( R_A(\rho, C_1, D_v) \), for each process, its first state in \( S_2'' \) equals its last state in \( R_A(\rho, C_1, D_v) \), which by History Oblivion, equals its last state in \( R \).

We now extend \( R.S_2'' \) (the result of appending \( S_2'' \) to \( R \)) to a complete run, call it \( R_3 \), so that in the suffix after \( R \), the message delays are the same as in \( S_2'' \) except that messages from \( p_1 \) to \( p_0 \) have delay \( d - m \) (see Figure 5), and no further operations are invoked. This choice of delays is consistent with any messages received in \( S_2'' \), since in \( S_2'' \), no message is received by \( p_0 \) and less than \( d - m \) time elapses in \( p_0 \)'s timed view after any message is sent to it by \( p_1 \). Since \( R_3 \) is admissible and complete, the operation invoked by \( p_0 \) in \( S_2'' \), but not completed in \( S_2'' \), must be completed subsequently; let \( \text{ret}_0' \) be its return value and \( \text{op}_0' = \text{OP}(\text{arg}_0, \text{ret}_0') \).

The operation instances in \( R_3 \) consist of the sequence \( \rho \) followed by \( \text{op}_0' \) and \( \text{op}_1 \), which are concurrent. There are only two possibilities for linearizing the operation instances in \( R_3 \). If they are linearized as \( \rho, \text{op}_1, \text{op}_0' \), we get a contradiction, since by Determinism of the data type, the fact that \( \rho \) \( \text{op}_1 \) is legal means that \( \rho, \text{op}_0' \) is illegal. Thus they must be linearized as \( \rho, \text{op}_0', \text{op}_1' \). By Determinism of the data type and the fact that \( \rho, \text{op}_0 \) is legal, it follows that \( \text{op}_0' \) must equal \( \text{op}_0 \). Thus \( p_0 \) continues to execute \( \text{op}_0 \) in \( R_3 \), even though the latter part of its execution might be quite different from that in \( R_2 \).
4.2 Pair-Free Operation Lower Bound

Step 5: Shifting $p_0$ Later and Chopping

Let $S_3$ be the run fragment that is the suffix of $R_3$ following $R_A(\rho, C_2, D_v)$. Let $\vec{y}$ be the $n$-vector $\langle -m, 0, 0, \ldots, 0 \rangle$ and $S'_3 = \text{shift}(S_3, \vec{y})$. Recalling that $C_2 = (0, \ldots, 0)$, we see that the clock offset vector in $S'_3$ is $C_0 = \langle -m, 0, \ldots, 0 \rangle$. The shift causes the delays of all messages sent by $p_0$ to decrease by $m$ and the delays of all messages received by $p_0$ to increase by $m$, while the remaining delays are unchanged. Thus the messages from $p_0$ to $p_1$ now have delay $d - 2m$, the messages from $p_0$ to the other processes now have delay $d - m$, and the messages received by $p_0$ now all have delay $d$. The only invalid delay is that from $p_0$ to $p_1$. (See Figure 6.)

Thus $S'_3$ is a run fragment that satisfies the requirements for the chop procedure. Let $S''_3 = \text{chop}(S'_3, d - m)$. We now show that $p_0$’s timed view is chopped after $op_0$ returns, and thus $p_0$ executes $op_0$ in its entirety in $S''_3$.

In $S'_3$, $op_0$ is invoked at real time $t$ and returns before real time $t + d + m$, by assumption that $|OP| < d + m$. Thus in $S'_3$, after shifting $p_0$ by $m$, $op_0$ is invoked at real time $t + m$ and returns before real time $t + d + 2m$. The first message sent from $p_0$ to $p_1$ is sent at time $t + m$ or later. By the definition of chop, $p_1$’s timed view is chopped at real time $t + m + \min\{d - 2m, d - m\} = t + d - m$ or later.

The shortest path from $p_1$ to $p_0$ with respect to delays is the edge from $p_1$ to $p_0$, which has delay $d$, since all edges outgoing from $p_1$ have delay $d$. By the definition of chop, $p_0$’s timed view is chopped at the real time that is $d$ later than the time when $p_1$’s timed view is chopped; this time is $(t + d - m) + d$ or later. Since $m \geq \frac{d}{3}$, $p_0$’s timed view is chopped at time $t + d + m$ or later.

Step 6: Repairing and Extending Again

We now show that $S''_3$ is appendable to $R' = R_A(\rho, C_0, D_v)$. (1) By definition, $R'$ is complete. (2) Both $S''_3$ and $R'$ have $C_0$ as their clock offset vector. (3) Recall that the first step in $S_3$ occurs at real-time $t$, where
Let $R$.

Step 7: Solo Execution by $p_1$.

Let $R_5$ be the complete run obtained by extending $R_A(\rho, C_0, D_v)$ as follows:

- $p_1$ invokes $OP(arg_1)$ at real time $t$; no other operations are invoked;
- message delays in the extension are pair-wise uniform according to Figure 7.

$R_5$ is admissible and finite. By the assumed correctness and complexity of the algorithm $A$ and the Determinism of the data type, there is a response at $p_1$ before real time $t + d + m$ and that response is $ret_1$; i.e., $p_1$ executes $op_1$.

Claim 5. The view of $p_1$ in $R_5$ during real-time interval $[t, t + d + m]$ is the same as the view of $p_1$ in $R_4$ during real-time interval $[t, t + d + m]$.

Proof: There is no difference between $R_5$ and $R_4$ until real time $t + m$, when $p_0$ invokes $OP(arg_0)$ in $R_4$ but not in $R_5$. The earliest that $p_1$ could receive information about this new invocation, either directly from $p_0$ or indirectly via one or more intermediate processes, is $t + m + d$, since every edge incoming to $p_1$ has delay $d$.

The previous claim shows the contradiction, since $p_1$ returns $ret_1$ in $R_5$ but returns $ret_1' \neq ret_1$ in $R_4$.

Corollary 2. Since Read – Modify – Write in Read-Modify-Write registers, Dequeue in queues, and Pop in stacks all fall in this category, we have a lower bound of $d + \min\{c, u, \frac{d}{2}\}$ for each.

Fig. 7: Message delays in $S_4$, after repairing illegal delay from $p_0$ to $p_1$ (proof of Theorem 4).

$t \geq \text{last-time}(R')$. Thus, first-time($S''_4$) occurs no earlier than $t$ since the maximum shift amount is $m$, which is at least last-time($R'$).

(4) Since $S_3$ is the suffix of run $R_3$ that follows $R_A(\rho, C_2, D_v)$, for each process, its first state in $S''_4$ equals its last state in $R_A(\rho, C_2, D_v)$, which by History Oblivion, equals its last state in $R'$.

We now extend $R'.S''_4$ (the result of appending $S''_4$ to $R'$) to a complete run, call it $R_4$, so that in the suffix after $R'$, the message delays are the same as in $S''_4$ except that messages from $p_0$ to $p_1$ have delay $d$ (see Figure 7), and no further operations are invoked. This choice of delays is consistent with any messages received in $S''_4$, since in $S''_4$, no message is received by $p_1$ and less than $d - m$ time elapses in $p_1$’s timed view after any message is sent to it by $p_0$. Since $R_4$ is admissible and complete, the operation invoked by $p_1$ in $S''_4$, but not completed in $S''_4$, must be completed subsequently: let $ret_0''$ be its return value and $op''_1 = \text{OP}(arg_1, ret''_1)$.

The operation instances in $R_4$ consist of the sequence $\rho$ followed by $op_0$ and $op''_1$, which are concurrent. There are only two possibilities for linearizing the operation instances in $R$. Suppose they are linearized as $\rho.op''_1.op_0$. Since $\rho.op_1$ is legal, Determinism of the data type implies that $op''_1$ must equal $op_1$. But then $\rho.op_1.op_0$ is illegal, by the definition of pair-free, which contradicts the assumed correctness of the algorithm. So the operations must be linearized as $\rho.op_0.op''_1$. By the Determinism of the data type, $op''_1$ must equal $op_1'$. Thus $p_1$ continues to execute $op''_1$ in $R_4$, even though the latter part of its execution might be quite different from that in $R_3$.  

4.2 Pair-Free Operation Lower Bound
4.3 Sum of Transposable Operation and Pure Accessor Lower Bound

We next show, in Theorem 5, a lower bound on the sum of the times for the execution of a certain kind of transposable operation, $OP$, and a pure accessor, $MOP$, where the two operations bear a certain relationship to each other. The hypotheses of the theorem imply that $OP$ is not an overwriter; in fact, the assumption is slightly stronger, as for a pair of operation instances $op_1, op_2 \in OP$, neither overwrites the other. Second, although $OP$ is transposable it is not commutative as invoking $op_1$ and $op_2$ in different orders results in non-equivalent states. We require a property stronger than just assuming non-commutativity, though, because we require that we can detect the difference between the non-equivalent states with a single operation, instead of an arbitrary-length operation sequence. Furthermore, $OP$ and $AOP$ must not be mutually-transposable.

We state these conditions precisely using the notion of a “discriminator”, a pair of instances of $AOP$ that can distinguish between different sequences of instances that end with $op_1$ and/or $op_2$.

An example of an operation pair which satisfies these conditions is $enqueue$ and $peek$ on a queue. The first operation is a pure mutator which “adds” to the state, instead of overwriting it, and the second is a pure accessor which is not solely dependent on the last change made to the object’s state. It is easy to see that all three assumptions hold for $enqueue$ and $peek$ if $op_1 = enqueue(1, -), op_2 = enqueue(2, -), aop_1 = aop_3 = peek(-, 1), aop_2 = peek(-, 2)$, and the queue’s initial state after $\rho$ is empty. While many results are the same for stacks and queues, this does not hold for stacks, because in a run implementing a stack that has only pushes and peeks, a $peek$ is solely dependent on the last $push$, as if it were an overwriter.

A discriminator in $AOP$ for two legal sequences of operation instances, $\rho_1$ and $\rho_2$, is a pair of operation instances in $AOP$ with the same argument but different return values that can tell the difference between the two operation instance sequences. That is, $aop_1 = AOP(arg, ret_1)$ and $aop_2 = AOP(arg, ret_2)$ such that $\rho_1.aop_1$ and $\rho_2.aop_2$ are legal but $\rho_1.aop_2$ and $\rho_2.aop_1$ are illegal.

**Theorem 5.** Let $OP$ be a transposable operation and $AOP$ a pure accessor. Suppose there exist $op_0, op_1 \in OP$ and a sequence $\rho$ of operation instances such that $\rho.op_0$ and $\rho.op_1$ are both legal and there exist discriminators in $AOP$ for

- $\rho.op_0$ and $\rho.op_1.op_0$,
- $\rho.op_1$ and $\rho.op_0.op_1$, and
- $\rho.op_2$ and $\rho.op_1.op_3$.

Then for any implementation in a message-passing system with at least three processes, message delays in the range $[d - u, d]$, and clock skew at most $\epsilon$, $|OP| + |AOP| \geq d + m$, where $m = \min\{\epsilon, u, \frac{d}{2}\}$.

**Proof:** Assume in contradiction there exists an algorithm $A$ for which $|OP| + |AOP| < d + m$.

Let $op_0 = OP(a_0, r_0)$ and $op_1 = OP(a_1, r_1)$.

Let $(aop_0^{\rho_1}, aop_0^{\rho_2})$ be the discriminator for $\rho.op_0$ and $\rho.op_0.op_0$, with common argument $arg_0$ and return values $ret_0^{\rho_1}$ and $ret_0^{\rho_2}$ respectively.

Let $(aop_1^{\rho_1}, aop_1^{\rho_2})$ be the discriminator for $\rho.op_1$ and $\rho.op_0.op_1$, with common argument $arg_1$ and return values $ret_1^{\rho_1}$ and $ret_1^{\rho_2}$ respectively.

Let $(aop_2^{\rho_1}, aop_2^{\rho_2})$ be the discriminator for $\rho.op_0.op_1$ and $\rho.op_1$, with common argument $arg_2$ and return values $ret_2^{\rho_1}$ and $ret_2^{\rho_2}$ respectively.

Let $C_1$ be the $n$-vector $(0, \ldots, 0)$ and $C_2$ be the $n$-vector $(0, m, 0, \ldots, 0)$. Since $m \leq \epsilon$, both vectors result in valid clock skews when used as clock offsets. Let $D_v$ be any $n$-by-$n$ matrix of valid message delays. Let $t = \max\{\text{last-time}(R_A(\rho, C_i, D_v)) : i = 1, 2\} + m$.

Let $D$ be the $n$-by-$n$ matrix of valid message delays defined as

$$d_{ij} = \begin{cases} d - m & \text{if } j = 1 \text{ or } j = 0 \\ d & \text{otherwise} \end{cases}$$

We extend $R_A(\rho, C_1, D_v)$ to a complete run $R_1$ as follows (see Figure 8):

---

4 Commutativity of an operation is a strengthening of transposability, in which both operation orders are equivalent, as well as legal.

5 We say that two operations $OP_1$ and $OP_2$ are mutually transposable if, given $op_1 \in OP_1$, $op_2 \in OP_2$, and an operation sequence $\rho$, $\rho.op_1$ legal and $\rho.op_2$ legal implies that both $\rho.op_1.op_2$ and $\rho.op_2.op_1$ are legal.
4.3 Sum of Transposable Operation and Pure Accessor Lower Bound

Wednesday 30th October, 2013, 11:41

Fig. 8: Message delays in the suffix $S_1$ of $R_1$ (proof of Theorem 5).

- All message delays in the extension are pair-wise uniform according to $D$.
- $p_0$ invokes $OP(a_0)$ at real time $t$ and gets response $r'_0$. Let $op'_0 = OP(a_0, r'_0)$.
- $p_1$ invokes $OP(a_1)$ at real time $t$ and gets response $r'_1$. Let $op'_1 = OP(a_1, r'_1)$.
- $p_0$ invokes $AOP(arg_0)$ at real time $t_{max}$ (the later of the finishing times of $op'_0$ and $op'_1$) and gets response $ret_0$ before time $t + d + m$ by assumed time bounds of the algorithm. Let $aop_0 = AOP(arg_0, ret_0)$.
- $p_1$ invokes $AOP(arg_1)$ at real time $t_{max}$ and gets response $ret_1$ before time $t + d + m$ by assumed time bounds of the algorithm. Let $aop_1 = AOP(arg_1, ret_1)$.
- $p_2$ invokes $AOP(arg_2)$ at real time $t_{max} + m$ and gets response $ret_2$ before time $t + d + 2m$. Let $aop_2 = AOP(arg_2, ret_2)$.

Note: The reason $aop_2$ returns before time $t + d + 2m$ is that

$$t_{max} + m + |aop_2| = t + \max(|op_0|, |op_1|) + m + |aop_2|$$

$$\leq t + |OP| + m + |AOP|$$

$$< t + d + 2m,$$

where the last line follows by the assumed upper bound on $|OP| + |AOP|$.

Claim 6. $op'_0 = op_0$ and $op'_1 = op_1$.

Proof: The operations in $R_1$ consist of the sequence $\rho$, followed by the concurrent operations $op'_0$ and $op'_1$, followed by the concurrent operations $aop_0$, $aop_1$, and $aop_2$. Suppose the algorithm linearizes $op'_0$ before $op'_1$. By Determinism, since $\rho.op_0$ is legal, and $op_0$ and $op'_0$ have the same argument, $op'_0 = op_0$. Next, $\rho.op_0.op'_1$ must be legal, since we assume the algorithm is correct. By transposability, $\rho.op_0.op_1$ is legal, so by Determinism, $op'_1 = op_1$ as they have the same argument. An analogous argument holds if the algorithm
4.3 Sum of Transposable Operation and Pure Accessor Lower Bound  

Wednesday 30th October, 2013, 11:41

4.3.1 Techniques

**Claim 7.** \( aop_0 = aop_0^2 \) and \( aop_2 = aop_2^2 \).

**Proof:** The algorithm must linearize \( aop_0, aop_1 \), and \( aop_2 \) in some order after \( \rho.op_1.op_0 \), say \( aop_i, aop_j, aop_k \). Since \( AOP \) is a pure accessor, \( \rho.op_1, \rho.op_0 = \rho.op_1, \rho.op_0, \rho.op_1, \rho.op_0, \rho.op_2, \rho.op_2, \rho.op_2, \rho.op_1, \rho.op_2, \rho.op_2, \rho.op_2 \) are all legal. This means that \( \rho.op_1, \rho.op_0, aop_0, \rho.op_1, \rho.op_0, aop_1, aop_2, \rho.op_1, \rho.op_0, aop_2 \) are all legal.

Let’s focus on the fact that \( \rho.op_1, \rho.op_0, aop_0 \) is legal. Recall that \( aop_0 = AOP(\text{arg}_0, \text{ret}_0) \), and that \( AOP(\text{arg}_0, \text{ret}_0^1) \) and \( AOP(\text{arg}_0, \text{ret}_0^2) \) form a discriminator for \( \rho.op_0 \) and \( \rho.op_1, \rho.op_0 \). Thus \( aop_0 \) must be \( AOP(\text{arg}_0, \text{ret}_0^2) = aop_0^2 \).

Similarly, \( aop_2 = AOP(\text{arg}_2, \text{ret}_2) = aop_2^2 \). □

Let \( S_1 \) be the run fragment that is the suffix of \( R_1 \) following \( R_A(\rho, C_1, D_v) \). Let \( \vec{x} \) be the \( n \)-vector \( \langle 0, m, 0, \ldots, 0 \rangle \) and \( S_1' = \text{shift}(S_1, \vec{x}) \). Recalling that \( C_1 = \langle 0, \ldots, 0 \rangle \), we see that the clock offset vector in \( S_1 \) is \( C_2 = \langle 0, -m, 0, \ldots, 0 \rangle \). The shift causes the delays of all messages sent by \( p_1 \) to increase by \( m \) and the delays of all messages received by \( p_1 \) to decrease by \( m \), while the remaining delays are unchanged. See Figure 10. The only invalid delay is that from \( p_1 \) to \( p_0 \), which is \( d - 2m \).

Thus \( S_1' \) is a run fragment that satisfies the requirements for the \( \text{chop} \) procedure. Let \( S_1'' = \text{chop}(S_2', d - m) \).

### Claim 8. \( S_1'' \) contains \( op_1 \) and \( aop_1 \) (executed by \( p_1 \)) and \( aop_2 \) (executed by \( p_2 \)).
Fig. 10: Message delays after shifting $S_1$ (proof of Theorem 5).

Proof: We chop $p_0$’s view at real time $t^*$, which is the earliest time when an illegal message could be sent plus the smaller of the illegal delay $(d - 2m)$ and the parameter $\delta = d - m$. Since the earliest that $p_1$ could send a message to $p_0$ is $t + m$ (when $op_1$ is invoked), $t^* = t + d - m$.

We chop $p_1$’s view at real time $t^* + \delta_{01}$ (recall that $\delta_{01}$ is the shortest path distance from $p_0$ to $p_1$ w.r.t. delays). Since all edges incoming to $p_1$ have delay $d$, $\delta_{01} = d$, and thus we chop at $t + 2d - m$. Since $m \leq \frac{d}{3}$, $t + 2d - m \geq t + d + 2m$, and thus $p_1$ finishes $aop_1$ before the chopping point.

A similar argument (noting that all edges outgoing from $p_0$ have delay $d$) shows that $p_2$’s view is chopped at real time $t + d + 2m$ also, which is after $aop_2$ finishes.

We now show that $S_1''$ is appendable to $R = R_A(\rho, C_2, D_v)$. (1) By definition, $R$ is complete. (2) Both $S_1''$ and $R$ have $C_2$ as their clock offset vector. (3) Recall that the first step in $S_1$ occurs at real time $t + m$, where $t \geq \text{last-time}(R)$. Thus first-time($S_1''$) occurs no earlier than $t$ since the maximum shift amount is $m$, which is at least last-time($R$). (4) Since $S_1$ is the suffix of run $R_1$ that follows $R_A(\rho, C_1, D_v)$, for each process, its first state in $S_1''$ equals its last state in $R_A(\rho, C_1, D_v)$, which by History Oblivion, equals its last state in $R$.

Let $R_2$ be the complete admissible run obtained by extending $R.S_1''$ (the result of appending $S_1''$ to $R$) as follows:

- in the suffix after $R$, the message delays are the same as in $S_1'$ (see Figure 8) except that messages from $p_1$ to $p_0$ have delay $d$; and
- if $p_0$ is chopped before the invocation of $aop_0$, add an invocation $AOP(arg_0)$ immediately after the invocation of $OP(a_0)$ finishes.

Let $r_0''$ be the return value of the $OP(a_0)$ invocation in $R_2$ and let $op''_0 = OP(a_0, r_0'')$. Let $ret'_0$ be the return value of the $AOP(arg_0)$ invocation in $R_2$ and let $aop'_0 = AOP(arg_0, ret'_0)$.

Claim 9. $op''_0 = op_0$ and $aop'_0 = aop_0'$.
Proof: Because of the assumed correctness of the algorithm, the operation instances in $R_2$ must be linearized as $\rho$ followed by some permutation of $\op'_0, \op_1, \aop'_0, \aop_1, \aop_2$. Since AOP is a pure accessor, the occurrences of $\aop'_0$, $\aop_1$, and $\aop_2$ do not affect how $\op'_0$ and $\op_1$ are linearized. By the same arguments made in the proof of Claim 6, we see that $\op'_0 = \op_0$.

Suppose $\op_0$ is linearized before $\op_1$ in $R_2$. In $R_1$, $\op_1$ ends before $t_{\text{max}}$, so in $R_2$, $\op_1$ ends before $t_{\text{max}} + m$, which is when $\aop_2$ begins. Thus $\aop_2$ must be linearized after $\op_1$. By Claim 7, $\aop_2 = \aop_2^2$, which, by definition of discriminator, is illegal after $\op_1, \aop_1$. Contradiction.

Now suppose $\op_0$ is linearized after $\op_1$ in $R_2$. Since $\op_0$ executes $\aop'_0$ after it executes $\op_0$, $\aop'_0$ must be linearized after $\op_0$. So we have the linearization $\rho, \op_1, \op_0, \aop'_0$. By Determinism, and since they have the same argument, $\aop'_0$ must equal $\aop^2_0$.

We now show that $R_2$ is indistinguishable to $p_0$ from some other admissible run $R_3$ during a specific time interval. $R_3$ is the same as $R_2$ except that $p_1$ does not execute $\op_1$. The time interval is $[t, t + d + 2m]$, which encompasses all of $\aop^2_0$. Thus the operation instances in $R_3$ must be linearized as $\rho, \op_0, \aop_0^2$. But by definition of discriminator, $\aop_0^2$ is illegal after $\op_0$. Contradiction.

We have now finished the proof, assuming that $\op_1$ is linearized before $\op_0$ in $R_1$. If $\op_0$ is linearized before $\op_1$ in $R_1$, essentially the same proof holds, reversing the roles of $p_0/\aop_0$ and $p_1/\aop_1$. □

5 Upper Bounds for Linearizable Objects

Now that we have developed lower bounds on a variety of operations on shared objects with arbitrary data types, we give an algorithm to prove upper bounds for all types of operations. In some interesting cases, these upper bounds are tight with our lower bounds and in others they are nearly tight.

The following implementation assumes that we have a message passing system with clocks that do not drift relative to real time. From [16], we know that the optimal clock synchronization error $\epsilon$ is $(1 - \frac{1}{2}) u < u$. Algorithms for achieving this optimal error $\epsilon$ already exist, so we proceed under the assumption that some such algorithm has already synchronized the clocks in our system to within $\epsilon$.

5.1 The Algorithm

We describe the algorithm for one shared object. Each process stores a local copy of the state of the object. The heart of the algorithm is how the copies are synchronized. The algorithm is interrupt-driven, with three types of events which can trigger actions: the invocation of an operation instance at a process, the receipt of a message at a process from another process, and the expiration of a previously-set timer at a process. The timers are used to make sure that events happen in a coordinated way at every process, despite the delays in communication. Each timer has an associated operation instance and action. The algorithm uses these to determine the return value of the operation instance.

In our algorithm, we use timestamps to determine the order in which to execute operation instances to satisfy linearizability. Each operation instance is given a timestamp on invocation, which is an ordered pair containing the local clock time of invocation and the id of the process where the instance was invoked. At each process, we maintain a priority queue of operation instances which are waiting for the correct time to execute. The priority function is lexicographic ordering of the timestamps of the instances, with the lowest first.

To allow local execution of operations, each process maintains the sequence of operations it has executed locally. Together with the invocation and argument, this uniquely determines the return value for each operation. In theory, this variable will increase without bound in an infinite execution. This does not prevent practical implementation of the algorithm, since it can be optimized to contain only the currently-relevant information for specific data types. Also, many data types, such as queues and stacks, can increase without bound, so maintaining operation history does not asymptotically increase the worst-case space requirement.

We partition operations into three categories—pure accessors, pure mutators, and other (or mixed) operations—and denote elements of these categories as AOP, MOP, and OOP, respectively.
5.1 The Algorithm

5.1.1 OOP
Operations in OOP are both accessors and mutators, so we must ensure that they are executed in the same order at each process and we must also ensure that they do not return until they have learned about all mutators that have completed before their invocation.

When a user invokes an op ∈ OOP at process pi, we immediately send a message specifying the operation and its timestamp to all other processes informing them of the invocation. Each other process pj, i ≠ j, adds op to its own execute queue immediately upon receiving this message. Because these messages are delayed an unknown length of time in the range [d – u, d], pi should not add op to its execute queue immediately on invocation. Instead, pi sets a timer for d – u, the minimum message delay to inform another process of the invocation, pretending that it must wait for a message about the invocation just like any other process. Only when this timer expires does pi add op to its execute queue.

Now, any process pi that has added an op ∈ OOP to its queue needs to wait long enough to ensure that no other op’ with smaller timestamp than op’s is added after op is executed, because we use timestamp order to ensure that every process executes the operations in the same order. To do this, when pi adds op to its queue, it starts a timer for u + ϵ. Since every process has already waited at least d – u for the message delay (or simulated delay at the invoking process), no message about an op’ with equal timestamp could be received after this timer expires, since the maximum message delay is d, and the clock skew (the size of the interval of real time after an invocation during which a later operation at another process could receive a lower timestamp) is ϵ. When this timer expires, then, pi executes every operation in its execute queue with timestamp less than or equal to that of op, in timestamp order. At this point, op’s invoking process returns control to the user, allowing further operation invocations.

5.1.2 MOP
Since pure mutators modify the object, it is possible that executing them in different orders on different processes’ local copies leaves those copies in different states. Thus, we must ensure, in the same way as for OOP’s, that all mutators are executed in the same order at every process. Again, we use timestamps to define the canonical order in which we want to execute the operations. To do this, we use exactly the same mechanism of message passing and timers as we used for OOP’s.

The difference between MOP’s and OOP’s is that pure mutators don’t return any information about the object, so they can return control to the user (“respond”) independent of the actual execution time at any process. There must be a delay of at least ϵ, to ensure that timestamps reflect operation order between processes despite the clock skew.

Following the approach of [17], we use a parameter X to balance the tradeoff in achievable response speed between pure mutators and accessors. X must be in the range [0, d – ϵ] and pure mutators respond X + ϵ time after invocation.

5.1.3 AOP
Since a pure accessor aop ∈ AOP does not modify the object, the invoking process does not need to tell any other process about aop. This means we don’t have to wait for message delays in broadcasting aop. Instead, we just have to wait long enough that any mutators which respond before aop’s invocation are executed locally before aop returns. We also have to ensure that AOPs invoked at different processes have timestamps corresponding to their order, as with pure mutators, since we use timestamp order to sort adjacent AOPs in the linearization.

It can take d time for aop’s invoking process to learn about a mutator invoked at another process, so we must wait that long to determine whether another process invoked a mutator which could have responded before aop was invoked. However, since every mutator takes at least X time to respond (pure mutators take X + ϵ, mixed operations take at least d + ϵ ≥ X), we are only interested in those mutators which were invoked at least X before aop, as mutators invoked less than X before aop are certain not to have returned before aop is invoked. This means that we only need to wait d after t – X, where t is the time of aop’s invocation. Since we locally execute all known mutators with lower timestamps before executing aop, every aop ∈ AOP can respond d – X time after invocation.
Algorithm 1 Code for each process $p_i$

**Initialization:** No timers are set, $To\_Execute$ is empty.

- $aop$ denotes an element of $AOP$, $xop$ denotes an element of $OOP \cup MOP$, $op$ denotes an element of $AOP \cup MOP \cup OOP$
- $history$ is the sequence of operations executed at $p_i$, initially empty

1. **HandleEvent** $\text{InvokeAOP}(aop, arg)$  \hspace{1cm} \text{▷ Invoke a Pure Accessor}
2. \hspace{0.5cm} $\text{set\_timer}(d - X, \langle aop, arg, \langle\text{local\_time} - X, i\rangle\rangle, \text{respond})$
3. **HandleEvent** $\text{ExpireTimer}(\langle aop, arg, ts \rangle, \text{respond})$  \hspace{1cm} \text{▷ Execute a Pure Accessor}
4. \hspace{0.5cm} \textbf{while} $ts \geq To\_Execute\_min()$ \hspace{0.7cm} \textbf{do}
5. \hspace{1cm} $\langle op', arg', ts' \rangle = To\_Execute\_extract\_min()$
6. \hspace{1cm} $\text{execute\_Locally}(op', arg')$
7. \hspace{1cm} $\text{cancel\_timer}(\langle op', arg', ts' \rangle, \text{execute})$
8. \hspace{0.5cm} \textbf{end while}
9. \hspace{0.5cm} \textbf{return} $\text{execute\_Locally}(aop, arg)$

10. **HandleEvent** $\text{InvokeOP}(xop, arg)$  \hspace{1cm} \text{▷ Invoke a mutator}
11. \hspace{0.5cm} \textbf{if} $xop \in MOP$ \hspace{0.7cm} \textbf{then}
12. \hspace{1cm} $\text{set\_timer}(X + \epsilon, \langle xop, arg, \langle\text{local\_time}, i\rangle\rangle, \text{respond})$
13. \hspace{1cm} \textbf{end if}
14. \hspace{0.5cm} $\text{set\_timer}(d - u, \langle xop, arg, \langle\text{local\_time}, i\rangle\rangle, \text{add})$
15. \hspace{0.5cm} $\text{send}(\langle xop, arg, ts \rangle, i)$ to all other processes.

16. **HandleEvent** $\text{ExpireTimer}(\langle xop, arg, ts \rangle, \text{respond})$  \hspace{1cm} \text{▷ Pure Mutators respond}
17. \hspace{0.5cm} \textbf{return} ACK
18. **HandleEvent** $\text{ExpireTimer}(\langle xop, arg, ts \rangle, \text{add})$  \hspace{1cm} \text{▷ $p_i$ adds a mutator to its $To\_Execute$ queue or $Receive((xop, arg, ts), j)$}
19. \hspace{0.5cm} $\text{set\_timer}(u + \epsilon, \langle xop, arg, ts \rangle, \text{execute})$
20. \hspace{0.5cm} $To\_Execute.add(\langle xop, arg, ts \rangle)$

21. **HandleEvent** $\text{ExpireTimer}(\langle xop, arg, ts \rangle, \text{execute})$  \hspace{1cm} \text{▷ Execute a mutator}
22. \hspace{0.5cm} \textbf{while} $ts \geq To\_Execute\_min()$ \hspace{0.7cm} \textbf{do}
23. \hspace{1cm} $\langle op', arg', ts' \rangle = To\_Execute\_extract\_min()$
24. \hspace{1cm} $\text{ret} = \text{execute\_Locally}(op', arg')$
25. \hspace{1cm} $\text{cancel\_timer}(\langle op', arg', ts' \rangle, \text{execute})$
26. \hspace{1cm} \textbf{if} $ts' == (\ast, i)$ \hspace{0.7cm} \textbf{then}
27. \hspace{1cm} \hspace{0.7cm} \textbf{return} $\text{ret}$
28. \hspace{1cm} \hspace{0.7cm} \textbf{end if}
29. \hspace{1cm} \textbf{end while}

30. **Function** $\text{execute\_Locally}(op, arg)$  \hspace{1cm} \text{▷ Perform local execution of $op$}
31. \hspace{0.5cm} Let $\text{ret}$ be the unique return value such that $\text{history.op}(arg, ret)$ is legal
32. \hspace{0.5cm} $\text{history} = \text{history.op}(arg, ret)$
33. \hspace{0.5cm} \textbf{return} $\text{ret}$
5.2 Proof of Linearizability

We prove that the algorithm correctly linearizes operations by describing a legal ordering of the operations in every complete admissible run which respects the order of non-overlapping operations. First, to have well-defined sequences of operation instances, we need to prove that every operation instance terminates. All of the proofs in section are based on analyzing the timing imposed by the algorithm in various possible executions.

Lemma 4. All operations terminate. Specifically,

- Every operation in AOP takes \(d - X\) time,
- Every operation in MOP takes \(X + \varepsilon\) time, and
- Every operation in OOP takes \(d + \varepsilon\) time.

Proof: From lines 2 and 9, we see that a pure accessor attempts to execute locally and return \(d - X\) time after it is invoked. From lines 14, 19, and 24, we see that a mutator attempts to execute at each process in the interval \([d + \varepsilon, d + u + \varepsilon]\) after invocation. The only thing that could delay these executions is locally executing operations in the To_Execute queue with smaller timestamps. But, since every operation is only in the queue for a finite amount of time (until its execute timer goes off), and operations can only be added at a finite rate (since each process can only invoke one operation instance at a time, and all instances take a positive amount of time to respond), there can only be finitely many operations on the To_Execute queue. Local execution is assumed to take no time, so the operation whose timer has expired executes without delay, a finite time after invocation.

AOPs and OOPs return when they execute at their invoking process, \(d - X\) and \(d + \varepsilon\) time after invocation, respectively. In lines 11-13 and 16-17, we see that MOPs return \(X + \varepsilon\) time after invocation. Thus all operation instances execute and return to the user in finite time, so every operation terminates. □

Because every operation terminates, we can now construct an ordering of the operations in the run.

Construction 1. We construct a permutation \(\pi\) of the operation instances executed in a run.

- Add all mutators in increasing timestamp order.
- Insert each pure accessor \(aop\) invoked at process \(p_i\) into the sequence immediately after the last mutator which \(p_i\) executed on its local copy before \(aop\) returned. (If there is no such mutator, \(aop\) is placed at the beginning of the sequence.)
- Wherever there are adjacent pure accessors, sort them in increasing timestamp order.

For \(\pi\) to be a correct linearization, we need to show that every process executes all mutators in increasing timestamp order and that at each process, pure accessors are executed in increasing timestamp order. Then we argue that this ordering means that every operation which completes before another operation \(op\) is invoked is linearized before \(op\).

Lemma 5. Each process locally executes all mutators in increasing timestamp order.

Proof: If we have two mutators, \(op_1\) and \(op_2\), with \(ts(op_1) \leq ts(op_2)\), let \(t\) and \(t'\) be the real times at which \(op_1\) and \(op_2\) are invoked, respectively. The delays at lines 14 and 19 and the message delays guarantee that \(op_2\) is executed at every process in the real time interval \([t' + d + \varepsilon, t' + d + u + \varepsilon]\). By the message delays and line 14, every process knows about \(op_1\) (has added it to the local To_Execute) by real time \(t + d \leq t' + d + \varepsilon\), because \(t'\) could be up to \(\varepsilon\) before \(t\) due to clock skew. But this means that every process knows about \(op_1\) before it executes \(op_2\), and, in lines 22-28, executes \(op_1\) before \(op_2\). This shows that the algorithm executes all mutators in increasing timestamp order. □

Each process executes locally-invoked pure accessors in timestamp order, because it must execute and return from one operation before another can be invoked at that process and since an AOP operation takes time \(|AOP| = d - X > 0\), subsequent locally-invoked AOPs have increasing timestamps.
Lemma 6. The construction of $\pi$ respects the order of non-overlapping operations in the run.

Proof: Consider two non-overlapping operations, $op_1$ and $op_2$, where $op_1$ responds before $op_2$ is invoked. Say that they are invoked at $p_i$ and $p_j$, respectively.

- $op_1$ and $op_2$ are both mutators.
  Since every mutator takes at least $\epsilon$ time to respond (lines 12, 19), then $ts(op_1) < ts(op_2)$. That is, non-overlapping mutators are assigned timestamps in their invocation order. Construction then tells us that non-overlapping mutators are inserted into $\pi$ in the correct order.

- $op_1$ is a mutator and $op_2$ is a pure accessor.
  As in the previous case, we can conclude that $ts(op_1) < ts(op_2)$. Since $op_1$ returns before $op_2$ is invoked, we know that $op_1$ is invoked at least $X + \epsilon$ before $op_2$, since every mutator takes at least $X + \epsilon$ real time, by Lemma[5]. This means that $p_i$ (op$_2$’s invoking process) learns about $op_1$ no later than $d - X - \epsilon$ time after $op_2$ is invoked, since the message delay from $p_i$ to $p_j$ is no more than $d$. Since $p_j$ waits $d - X + \epsilon$ after invoking $op_2$ (line 2), it knows about and executes $op_1$ locally before $op_2$ returns. Thus $op_2$ is inserted in the permutation after $op_1$, respecting the non-overlapping order of the operations.

- $op_1$ and $op_2$ are both pure accessors.
  Accessors are inserted in the linearization in timestamp order after the last mutator which executed before them at their local process. What we must show now is that pure accessors at different processes that do not overlap are linearized correctly. If the last mutator locally executed before each of $op_1$ and $op_2$ is different, then this is true, because we know from Lemma[5] that mutators are executed in the same order at every process, so $op_1$ is added to $\pi$ after the mutator which $op_2$ didn’t see, respecting the order of non-overlapping operations.

If non-overlapping pure accessors are linearized adjacentally, we show that they are ordered correctly. Suppose $op_1$ and $op_2$ are invoked with timestamps $ts(op_1), ts(op_2)$ such that $op_1$ responds before $op_2$ is invoked, in real time. Since a pure accessor takes $d - X \geq \epsilon$ time to respond, $ts(op_2) > ts(op_1)$. Thus, ordering adjacent pure accessors in $\pi$ by their timestamps respects the order of non-overlapping operations.

- $op_1$ is a pure accessor and $op_2$ is a mutator.
  Assume in contradiction that $op_1$ is linearized after $op_2$. Then, by the construction of $\pi$, $op_2$ must execute at $p_i$ (op$_1$’s invoking process) before $op_1$ does. There are two places $op_2$ may execute, lines 6 and 24. Since a pure accessor takes $d - X \geq \epsilon$ time to respond (Lemma[5]), then $ts(op_1) < ts(op_2)$. This means that $op_2$ is not executed at line 6, even if it is in the To Execute queue at $p_i$ before $op_1$ executes.

At line 24, $op_2$ is executed only if the execute timer for a mutator with timestamp at least as large as $op_2$’s expires. This can happen no sooner than $d + \epsilon$ after $op_2$ is invoked, from lines 14, 15 and 19. But $op_1$ executes $d - X < d + \epsilon$ after it is invoked, which is strictly before $op_2$’s invocation, because $ts(op_1) < ts(op_2)$. Thus, $op_2$ is not executed at line 24 before $op_1$ has executed.

Thus, $op_2$ can never execute before $op_1$, so the assumption that $op_2$ is linearized before $op_1$ is false, and $op_1$ is linearized before $op_2$, as desired.

Thus, linearizing the operations according to the above construction is valid for any execution performed by our algorithm. \qed

Lemma 7. The construction gives a legal sequence of operation instances.

Proof: Let $\pi$ be the sequence the construction generates. We use induction on the length of $\pi$. If $\pi$ is empty, then it is vacuously legal.

If $|\pi| > 0$, we assume that $\pi = \pi'. op$, where $\pi'$ is a legal sequence of operation instances and $op = OP(arg, ret)$ is an operation instance. Let $p_i$ be the process where $op$ is invoked and let $h_i$ be the sequence of operations in $p_i$’s history variable when $op$ is executed.
We need to show that the value which \( op \) returns when executed by the algorithm after \( h_i \) is the same as the value it would return when executed after \( \pi' \). It is sufficient to show that \( \pi'|_m = h_i|_m \), where \( |_m \) indicates the sequence formed by removing all pure accessors from the given sequence of operation instances. By determinism, if \( \pi'|_m = h_i|_m \), then \( OP(arg) \) has the same return value when executed after \( \pi' \) or \( h_i \), because pure accessors do not affect the execution of other operations. Thus, \( \pi = \pi'.op \) would be legal.

If \( op \) is a pure accessor, since mutators are both executed at \( p_1 \) and inserted into \( \pi \) in increasing timestamp order (Lemma 4 and Construction 1), and \( op \) is placed in \( \pi \) after the same mutator it is executed after in \( h_i \), \( \pi'|_m = h_i|_m \), so \( \pi \) is legal.

Similarly, if \( op \) is both an accessor and mutator, then it is executed at \( p_1 \) and inserted into \( \pi \) in timestamp order with all other mutators. That is, \( \pi'|_m = h_i|_m \) is the sequence of mutators with timestamps smaller than \( op \), ordered by increasing timestamps, so \( \pi \) is legal.

If \( op \) is a pure mutator, it is always legal regardless of previous operations, so \( \pi \) is legal.

Thus, by induction, the sequence of operation instances given by Construction 1 is legal. \( \square \)

We thus have

**Theorem 6.** Algorithm 1 is a correct implementation of a linearizable object of an arbitrary data type.

## 6 Conclusion

### 6.1 Summary

In this paper we gave lower bounds on several widely-used classes of operations. We defined these classes in terms of algebraic properties, which allows us to prove general results applicable to a variety of specific operations on specific data types. To achieve these bounds, we used shifting arguments. We started with the classic style of shifting proof, then introduced an extension to the shifting technique for proving lower bounds in message-passing executions. This new technique allows us to prove larger lower bounds in certain cases by carefully breaking and repairing the message delay constraints. We then gave an algorithm which, by using a classification orthogonal to that used in the lower bounds, based on whether an operation accesses or changes the shared object, gives good upper bounds on all operations of arbitrary data types.

In Figure 11, we show the relationships between the classes of operations for which we have lower bounds, relative to the broad classification of operations as accessors, mutators, or both, for which we have upper bounds. The class of last-sensitive mutators (cf. Theorem 3) is a subset of the class of mutators that includes both pure mutators as well as mixed operations (both mutators and accessors). The class of pair-free operations (cf. Theorem 4) is in the intersection of mutators and accessors.

In Table 5, we summarize our upper and lower bounds. There are several facts which do not stand out in the table, but are worth noting. First, the lower bound for the sum of a pure accessor and a pure mutator is greater than the sum of the individual upper bounds for the operations because of the tradeoff between optimizing them. In different implementations, it is possible to make either pure accessors or pure mutators very fast, but whenever we make one fast, we must make the other slow to maintain linearizability.

The lower bound on \( \lceil Read \rceil + \lceil Write \rceil \) in [15] can be generalized easily to hold for any two operations \( OP_1 \) and \( OP_2 \) that “interfere” with each other, in the sense that there exists a sequence \( \rho \) of operation sequences, an instance \( op_1 \in OP_1 \), and an instance \( op_2 \in OP_2 \) such that \( \rho.op_1.op_2 \) is legal, but \( \rho.op_2 \) is illegal. In this case, \( OP_1 \) is a mutator and \( OP_2 \) is an accessor. The intuition is that the accessor must learn about the mutator in order to return the correct value, so there must be time for a message to get from the invoker of \( op_1 \) to the invoker of \( op_2 \) after \( op_1 \) begins and before \( op_2 \) ends.

Since we assume that clocks do not drift, they can be synchronized so that the clock skew \( \epsilon \) is \( (1 - \frac{1}{n})u < u \) [16]. Also, \( k \) in the bound \( (1 - \frac{1}{k})u \) from Theorem 3 is \( n \) if there are at least \( n \) different possible arguments for the operation. In this case, our bounds on last-sensitive pure mutators are tight for operations with large domains, such as the integers or reals. Furthermore, if \( \epsilon < \frac{1}{3} \), then our bounds on pair-free operations are tight.


6.2 Future Work

Most directly, future work includes tightening the bounds we give here. Some of the categories of operations we consider still have gaps between the upper and lower bounds.

Next, there is still much work to be done to complete the algebraic taxonomy of operations. While it is straightforward to provide upper bounds for all operations based on functionality (mutator, accessor), the techniques we have available for proving lower bounds are not general enough to easily cover all operations. We have used a variety of specific algebraic properties to define classes of operations for which we were able to prove lower bounds, covering many fundamental operations, but these classes are not exhaustive. A complete categorization of operations based on algebraic properties would be an important step to proving tight bounds on all possible operations.

Finally, the model we use can be relaxed. For example, we could remove the constraint that the data type be wholly deterministic. E.g. a Set data type could support the extraction of an arbitrary element. It seems possible that the freedom to remove any element, instead of a specific element, such as the head in a stack or queue, may allow faster operation. However, the proofs in this paper will no longer apply, as they rely heavily on the determinism of the shared object.

Acknowledgement

This work was supported in part by NSF grant 0964696.
References


