Wait-Freedom is Harder than Lock-Freedom under Strong Linearizability

Oksana Denysyuk, Philipp Woelfel

Wait-Freedom is Harder than Lock-Freedom under Strong Linearizability

Oksana Denysyuk and Philipp Woelfel

Department of Computer Science, University of Calgary, Canada
{oksana.denysyuk,woelfel}@ucalgary.ca

Abstract. In randomized algorithms, replacing atomic shared objects with *linearizable* [1] implementations may affect probability distributions over outcomes [2]. To avoid this problem in the adaptive adversary model, it is necessary and sufficient that implemented objects satisfy *strong linearizability* [2]. In this paper we study the existence of strongly linearizable implementations from *multi-writer* registers. We prove the impossibility of *wait-free* strongly linearizable implementations for a number of standard objects, including snapshots, counters, and max-registers, all of which have wait-free linearizable implementations. To do so, we introduce a new notion of *group valency* that is useful to analyze (strongly linearizable) implementations from registers. Furthermore, we show that many objects, including snapshots, do have *lock-free* strongly linearizable implementations. These results separate lock-freedom from wait-freedom under strong linearizability.

1 Introduction

Linearizability [1] is the gold standard for correctness conditions of concurrent shared memory algorithms. The main reason for its attractiveness is that replacing atomic objects in a deterministic shared memory algorithm with linearizable ones preserves the worst-case behaviour of the algorithm. This simplifies programming concurrent code significantly, as it allows programmers to assume that the implemented linearizable operations get completed in a single atomic step. Unfortunately, linearizability has anomalies that can cause undesirable effects when used with randomized algorithms [2]: probability distributions over outcomes of an algorithm that uses atomic objects can differ significantly from those of the same algorithm using linearizable objects. As a result, algorithm designers cannot analyze running times or error probabilities of their algorithms under the assumption that linearizable operations complete in a single step.

To address this problem, *strong linearizability* [2] has been introduced. Roughly, strong linearizability requires operations to be linearized based on past and present behavior rather than the future. In a system where processes are scheduled by a strong adaptive adversary (i.e., the future schedule may depend on all past random decisions made by processes), this requirement preserves probability distributions over outcomes of algorithms, if atomic objects are replaced with strongly linearizable ones (see Section 2 for details). Moreover, strong linearizability is necessary to achieve this behaviour [2].

Unfortunately, little is known about whether and how strongly linearizable objects can be implemented. Clearly, using only registers, it is impossible to obtain wait-free or lock-free implementations of any type with consensus number two or greater [3] under linearizability. This implies *a fortiori* impossibilities under strong linearizability.

Many useful shared memory primitives, such as snapshots or counters, have wait-free linearizable implementations even from single-writer registers. Prior to our work it was unclear, however, whether those primitives have also wait-free strongly linearizable implementations. For systems providing only *single-writer* atomic registers, Helmi, Higham, and Woelfel [4] already showed that many objects have no wait-free strongly linearizable implementations, even though they have linearizable ones. In particular, under the stronger correctness condition multi-writer registers cannot be implemented from single-writer ones. But their proof technique does not apply to systems that readily provide atomic multi-writer registers. We present new proof techniques that yield the following result.

Theorem 1. *There are no deterministic strongly linearizable wait-free implementations of snapshots, counters, or max-registers for three or more processes, from multi-writer registers.*

We also show that, perhaps surprisingly, these types do have *lock-free* strongly linearizable implementations.

Theorem 2. *There exist deterministic strongly linearizable lock-free implementations of (general) counters, snapshots, and logical clock objects for any number of processes, from multi-writer registers.*

Theorems 1 and 2 provide a separation between these two progress conditions under strong linearizability. In fact, to our knowledge, this is the first result to show a separation of wait-free and lock-free implementations for natural types such as snapshots and counters. Prior work [5] claims a separation between wait-freedom and lock-freedom under linearizability for an *ad hoc* object called "iterated approximate agreement".

To prove Theorem 1, we show that a monotonic counter does not have a wait-free strongly linearizable implementation from registers (even though it has a wait-free linearizable implementation [6]). By reduction, this implies the other impossibilities stated in Theorem 1. To facilitate the proof, we introduce two new concepts, *group valency* and *supervalency*, which generalize the traditional notion of valency used in the FLP impossibility result for consensus [7,8]. (In a consensus algorithm, all participating processes have to agree on one of their input values.) In the consensus impossibility proof, a history H is multivalent if it has two different extensions, in which different values are output. Intuitively, that means that the decision has not been determined at the end of H. To show our result, we extend the notion of valency in two ways.

First, we consider the ability of a set G of processes to linearize the operation `op` of another process $p \notin G$. Roughly, v is in the G-valency of `op`, if processes in G executing alone can linearize `op`, causing `op` to return v. This is closely

tied to the notion of *helping* in wait-free implementations, where one or more processes help another process to complete `op`. In the impossibility proof for strongly linearizable monotonic counters, we apply the notion of group valency to a group G of processes that repeatedly increment the counter while another process $p \notin G$ wishes to execute `op` to read the counter. Using group valency, we show that if processes in G try to help p, they end up causing p to execute forever, thus violating wait-freedom.

Second, we introduce the notion of *supervalent histories*. In the traditional FLP proof, a multivalent (or bivalent) history is one in which the consensus output is undetermined (so both decisions of 0 or 1 are possible). We extend this notion to a history in which, not only the outcome of some operation `op` by $p \notin GF$ is undetermined, but processes in G can execute an unbounded number of steps alone without fixing the outcome of `op` (i.e., without linearizing `op`). This is called a G-supervalent history. Intuitively, in executing an unbounded number of steps, processes in G can influence the future return value of `op` to be any of an unbounded number of possibilities (e.g., as processes in G increment the counter, the future return value of the operation `op` that reads the counter can be arbitrarily high). We show that a supervalent history may remain supervalent forever, so that `op` can never return the correct value of the counter. The notion of supervalent histories is more powerful than the notion of multivalent histories for our impossibility result: we found algorithms for which it is impossible to show that a multivalent history can remain multivalent as `op` continues to execute.

Theorem 2 identifies several common primitives that have lock-free strongly linearizable implementations from registers. To prove this, we first define a class of *versioned types*, which are types that maintain a monotonic version number that increases for each update operation. Many objects of the standard types (snapshots, max-registers, counters, logical clocks) can be easily extended into objects of a versioned type by incorporating a counter as the version number. Moreover, many lock-free linearizable implementations of those types have the additional property that update operations consist of a single atomic step. We then transform such linearizable implementations into *strongly linearizable* lock-free ones. This transformation uses a simple generalization of a max-register, which admits a strongly linearizable lock-free implementation [4].

2 Preliminaries

We consider the standard shared memory model, where n asynchronous processes with distinct IDs in $\{0, \ldots, n-1\}$ communicate by accessing shared atomic multi-reader multi-writer registers. Each register R has initially value χ, and supports operations R.`read`(), which returns the value of R, and R.`write`(), where R.`write`(x) changes the value of R to x and returns nothing.

Atomicity and Linearizability. A type specifies operations, and the outcome of those operations in any sequential execution. An object is obtained by implementing the operations of type, by providing algorithms for them. A process p

executes an operation op by executing the steps of the algorithm beginning with an invocation step and ending with a response step. Other processes can be taking steps during the interval in which p is executing the method for o and these steps may interleave. This sequence of steps that results as processes execute their program is called a *history*. We restrict our attention to histories that *can* arise in an execution. Consider an object O and the histories that can arise as processes execute operations on O. A method of O is *atomic*, if it consists of a single shared memory step [1]. In this case, we may assume that the invocation and response step of the method occur at the same time as the shared memory step. An object is atomic, if all its methods are, and the histories that can arise by processes executing operations on such an object are *sequential*.

The behaviour of a type is given by its *sequential specification*, which is a set of sequential histories that are allowed to arise from atomic objects of that type. An *implemented history* on O arises when the operations on O may be non-atomic. The *interpretation* of an implemented history H, denoted $\Gamma(H)$, is formed by removing from H all the steps of every method call except the invocation and response steps. Let H be an implemented history arising from an execution of operations on O. Operation op completes in H if H contains the invocation and response of op. $Cmp(H)$ denotes the set of operations that complete in H. Operation op is pending in H if H contains the invocation but not the response of op. For implemented operations $\mathtt{op}_1$ and $\mathtt{op}_2$, $\mathtt{op}_1$ *happens-before* $\mathtt{op}_2$ in H, denoted $\mathtt{op}_1 \prec \mathtt{op}_2$, if the response of $\mathtt{op}_1$ precedes the invocation of $\mathtt{op}_2$ in H. Interpreted history H is *linearizable* if, for some subset S of pending operations in H, there is a sequential history H_{seq} that contains each operation in $Cmp(H) \cup S$ exactly once, is in the sequential specification of O, and preserves $\prec$. Such H_{seq} is *linearization* of H. An implementation of O is linearizable if every history that can arise from the implementation is linearizable. The property that makes linearizability attractive is the following: If $\mathcal{A}$ is a deterministic algorithm that uses objects of some type T, then for every history H that can arise from $\mathcal{A}$, the linearization of $\Gamma(H)$ can arise from the same algorithm using atomic objects of type T instead. But linearizability may not preserve probability distributions over outcomes, if $\mathcal{A}$ is a randomized algorithm. Thus, linearizable implementations are less suitable to accurately analyze the expected running times or error probabilities of randomized algorithms.

Strong Linearizability. In a randomized algorithm, processes can use local coin flips to decide which steps to execute in their program. The type and object of an operation may influence the speed with which an operation is executed, so the order in which processes take steps is indirectly influenced by their random decisions. Adversary models are used to capture that influence. One of the most common adversaries is the strong adaptive one. Informally, the strong adaptive adversary can look at the entire past execution, including the result of all coin flips made by processes, to decide which process will take the next step.

[1] Sometimes, however, in literature atomicity is defined to be the same as linearizability [9].

Let $close(\mathcal{H})$ denote the prefix closure of a set of histories in $\mathcal{H}$. That is, $G \in close(\mathcal{H})$ if and only if there is a sequence S of invocation and response steps such that $G \circ S \in \mathcal{H}$. (Operation $\circ$ denotes concatenation.) A function f that maps a set $\mathcal{H}$ of histories to a set $\mathcal{H}'$ of histories, is *prefix preserving*, if for any two histories $G, H \in \mathcal{H}$, where G is a prefix of H, $f(G)$ is a prefix of $f(H)$.

Definition 3. *[2]A set of histories $\mathcal{H}$ is* strongly linearizable *if there exists a function f mapping histories in $close(\mathcal{H})$ to sequential histories, such that for any $H \in close(\mathcal{H})$, $f(H)$ is a linearization of the interpreted history $\Gamma(H)$, and f is prefix preserving. A function satisfying these properties is called a* strong linearization function *for $\mathcal{H}$.*

Intuitively, strong linearizability requires that the linearization points of method calls are determined as the history is created. As soon as a step is taken, whether or not a particular method is linearized at that step is uniquely determined by the history up to this step; it cannot be influenced by future steps.

We say an object is strongly linearizable, if the set of histories that can be obtained by executions of operations on that object is strongly linearizable. Golab et al. [2] showed that strongly linearizable objects can serve the same purpose for randomized algorithms under a strong adaptive adversary model, as linearizable objects do for deterministic algorithms: Consider a randomized algorithms $\mathcal{A}$ and an adversary Z. For an infinite vector $\boldsymbol{c} = (c_1, c_2, \dots)$ over $\{0, 1\}$, let $H_{Z,\mathcal{A},\boldsymbol{c}}$ be the unique history obtained if algorithm $\mathcal{A}$ is scheduled by adversary Z, and the sequence of coin flips of $H_{Z,\mathcal{A},\boldsymbol{c}}$ is a prefix of $\boldsymbol{c}$ (or equals $\boldsymbol{c}$ if the history is infinite). Now suppose $\mathcal{A}$ is a randomized algorithm using atomic objects of some type, and $\mathcal{A}'$ is obtained by replacing those atomic objects with strongly linearizable ones of the same type. Golab et al. proved that for every strong adversary Z' there exists a strong adversary Z, such that for every coin flip vector $\boldsymbol{c}$, $\Gamma(H_{Z,\mathcal{A},\boldsymbol{c}})$ and $\Gamma(H_{Z',\mathcal{A}',\boldsymbol{c}})$ have a common linearization. Moreover, strong linearizability is necessary for this: If for some adversary Z' there exists an adversary Z such that $\Gamma(H_{Z,\mathcal{A},\boldsymbol{c}})$ and $\Gamma(H_{Z',\mathcal{A}',\boldsymbol{c}})$ have a common linearization for every coin flip vector $\boldsymbol{c}$, then the set of all histories $\Gamma(H_{Z',\mathcal{A}',\boldsymbol{c}})$ obtained from all possible coin flip vectors $\boldsymbol{c}$ is strongly linearizable. Hence, atomic objects can be replaced with implemented ones without changing the probability distribution over linearizations only, if the set of possible histories that can be obtained from any possible strong adversary is strongly linearizable.

Thus, strong linearizability is the correctness condition of choice for randomized algorithm against the strong adaptive adversary.

Configurations, Schedules, and Progress Conditions. A *configuration* C of a system with n processes and m registers is a tuple $(s_1, \dots, s_n, v_1, \dots, v_m)$, which denotes that process p_i, $1 \leq 1 \leq n$, is in state s_i, and register r_j, $1 \leq j \leq m$, has value v_j. The initial configuration is denoted by C_0. We usually assume without mentioning it explicitly that histories are obtained by processes taking steps starting in C_0.

A *schedule* σ is a (possibly infinite) sequence of process indices. Let C be a configuration resulting from execution of a finite history H. $H \triangleright \sigma$ denotes a his-

tory resulting from executing a sequence of steps in σ beginning in configuration C and moving through successive configurations one at a time. At each step, next process p indicated in σ takes the next step in its deterministic program. If σ is a sequence of length one, we say $\sigma{=}p$. If σ and π are finite schedules then $\sigma\pi$ denotes the concatenation of σ and π. Let P be a set of processes, and σ a schedule. We say σ is *P-only* if only indices of processes in P appear in σ.

Configurations $C_1{=}(s_1, \ldots, s_n, r_1, \ldots, r_m)$ and $C_2{=}(s'_1, \ldots, s'_n, r'_1, \ldots, r'_m)$ are indistinguishable to process p_i, denoted $C_1 \overset{p_i}{\sim} C2$, if $s_i{=}s'_i$ and $r_j{=}r'_j$ for $1{\leq}j{\leq}m$. If S is a set of processes, and $C_1 \overset{p}{\sim} C_2$ for every process $p \in S$, then we write $C_1 \overset{S}{\sim} C_2$; if $S{=}\{1, \ldots, n\}$ is the set of all processes, we simply write $C_1 \sim C_2$. If $C_1 \overset{p}{\sim} C_2$, then for any S-only schedule σ, configurations resulting from execution of σ from C_1 and C_2 are indistinguishable to every process in S. Two histories H_1 and H_2 are indistinguishable, denoted $H_1 \sim H_2$, if H_1 and H_2 generate indistinguishable configurations.

An implementation is *lock-free* if, for any history H and every infinite schedule σ, there exists a process p with a pending operation `op` in H, and p takes infinitely many steps in σ, then `op` completes in a finite number of steps in history $H{\triangleright}\sigma$. An implementation is *wait-free* [7] if, in any history, any process with a pending operation completes in a finite number of steps, regardless of the steps taken by other processes.

Some Common Types. We refer to the types monotonic counters, (general) counters, max-registers, and snapshots, as defined below: A *monotonic counter* has two operations, `increment`() and `read`(), where `increment`() increases the counter value by one, and `read`() returns the counter value. A *(general) counter* is defined similarly, but the `increment`() operations takes an argument, x, and increases the value of the counter by x. A *max-register* has two operations, `maxWrite`(v) and `read`(), such that `read`() returns the largest value written by any preceding `maxWrite` operation. A snapshot object stores n segments, one for each process. It supports two operations, `update`(v) and `scan`(). Operation `update`(v), when executed by process i, changes the value of the i-th segment to v, and `scan`() returns a vector of n elements containing the n segments.

3 Impossibilities

We show that there is no strongly linearizable wait-free implementation of a monotonic counter from registers. Assume by contradiction that there exists such an implementation. We will consider an execution with three processes: r, w_0, and w_1. We call r the reader and w_i the writers. Initially, r starts executing `read`(), while w_0 and w_1 start executing `increment`(). If process w_0 or w_1 finishes executing `increment`(), it invokes the operation again and again in an infinite loop. We will construct an infinite fair schedule, i.e., a schedule in which every process takes infinitely many steps, such that r never finishes its `read`(). This contradicts the assumption that the implementation is wait-free.

3.1 Group Valency and Super Valency

We now define the notions *total valency* and *group valency*. In the definitions, `op` denotes an operation, S a set of processes, and H a finite history.

Definition 4 (Total Valency). *The* total valency *of H (w.r.t. `op`) is the set of values ν such that, for some finite schedule σ, `op` returns ν in history $H{\triangleright}\sigma$.*

In the proofs, `op` is a fixed operation so we often omit references to it. To prove the impossibility, it will be critical to consider the possible values the reader may return if it gets linearized by the writers. To facilitate this we define the notion of *group-valency*.

Definition 5 (Group valency). *The* S-valency *of H (with respect to `op`) is the set of values ν for which there exists an S-only schedule σ, such that in $f(H{\triangleright}\sigma)$ `op` returns ν.*

Some histories have the property that all sufficiently long S-only schedules will linerize `op`, even if `op` is not an operation by a process in S. These are called S-closed. Histories that are not S-closed are called S-supervalent.

Definition 6 (Supervalency). *We say that H is S-closed (w.r.t. `op`) if there exists an integer $K \geq 0$ such that, for every S-only schedule σ of length at least K, `op` appears in $f(H{\triangleright}\sigma)$. We say that H is S-supervalent (w.r.t. `op`) if H is not S-closed, that is, for every K, there is an S-only schedule σ of length at least K such that `op` does not appear in $f(H{\triangleright}\sigma)$.*

The above definitions immediately imply the following:

Observation 7. *(1) If H is S-closed then the S-valency of H is not empty. (2) All S-only extensions of an S-closed history are S-closed. (3) From an S-supervalent history H, there exists an S-only non-empty schedule σ such that $H{\triangleright}\sigma$ is also S-supervalent. (4) For any finite schedule σ, the S-valency of $H{\triangleright}\sigma$ is contained in the S-valency of H.*

3.2 Impossibility Proof

In the proof we analyze the possible outputs of the `read`() operation using the concepts of group valency and supervalency when the group is the set of writers. Specifically, we fix `op` to be the `read`() operation of r, and we fix S to be the set of both writers $\{w_0, w_1\}$. For a finite history H, we denote by $V(H)$ the total valency of H (w.r.t. `op`), and we denote by $\mathcal{W}(H)$ the S-valency of H (w.r.t. `op`). Because S is the set of writers, we often use the terms writers-valency, writers-supervalent, and writers-closed to refer to the concepts of S-valency, S-supervalent, and S-closed defined above, respectively.

By the standard argument we obtain the following.

Lemma 8. *Consider some valid histories H and H':*

(a) If H is writers-supervalent then `read`() *is not in* $f(H)$.
(b) If $|\mathcal{W}(H)| \geq 2$ *then* `read`() *is not in* $f(H)$.
(c) If $H \sim H'$ *then* $V(H) = V(H')$ *and* $\mathcal{W}(H) = \mathcal{W}(H')$.
(d) If H *is writers-closed and* $H \overset{w_i,r}{\sim} H'$, *for some* $i \in \{0,1\}$, *then* $\mathcal{W}(H) \cap \mathcal{W}(H') \neq \emptyset$.

In the following lemma we show that if history H is writers-closed and $|\mathcal{W}(H)| \geq 2$, then there exists a step by a writer w_i such that $|\mathcal{W}(H \triangleright w_i)| \geq 2$.

Lemma 9. *If H is writers-closed and $|\mathcal{W}(H)| \geq 2$, then for some $i \in \{0,1\}$, $H \triangleright w_i$ is writers-closed and $|\mathcal{W}(H \triangleright w_i)| \geq 2$.*

Proof. By Lemma 8(b), since $|\mathcal{W}(H)| \geq 2$, `read` is not in $f(H)$. Suppose H is writers-closed. From Observation 7, for all writers-only schedules σ, $H \triangleright \sigma$ is also writers-closed. By contradiction, suppose that for all $i \in \{0,1\}$, $|\mathcal{W}(H \triangleright w_i)| \leq 1$. By Observation 7, $\mathcal{W}(H \triangleright w_i)$ is not empty. Thus, $\mathcal{W}(H \triangleright w_0) = \{x_0\}$ and $\mathcal{W}(H \triangleright w_1) = \{x_1\}$ for two distinct $x_0, x_1 \in \mathcal{W}(H)$.

Case 1 *For some $i \in \{0,1\}$, w_i is poised to read in H.* Then $H \triangleright w_i \overset{r,w_{1-i}}{\sim} H$ and so $H \triangleright w_i w_{1-i} \overset{r,w_{1-i}}{\sim} H \triangleright w_{1-i}$. However, $\mathcal{W}(H \triangleright w_i w_{1-i}) \subseteq \mathcal{W}(H \triangleright w_i)$ and $\mathcal{W}(H \triangleright w_{1-i}) \cap \mathcal{W}(H \triangleright w_i) = \emptyset$, and so $\mathcal{W}(H \triangleright w_{1-i}) \cap \mathcal{W}(H \triangleright w_i w_{1-i}) = \emptyset$. This contradicts Lemma 8(d).
Case 2 *Both writers are poised to write to different registers.* Then $H \triangleright w_0 w_1 \sim H \triangleright w_1 w_0$. Since $\mathcal{W}(H \triangleright w_0) \cap \mathcal{W}(H \triangleright w_1) = \emptyset$, $\mathcal{W}(H \triangleright w_0 w_1) \cap \mathcal{W}(H \triangleright w_1 w_0) = \emptyset$. This contradicts Lemma 8(d).
Case 3 *Both writers are poised to write to the same register.* Then $H \triangleright w_0 w_1 \overset{r,w_1}{\sim} H \triangleright w_1$. Since $\mathcal{W}(H \triangleright w_0) \cap \mathcal{W}(H \triangleright w_1) = \emptyset$, $\mathcal{W}(H \triangleright w_0 w_1) \cap \mathcal{W}(H \triangleright w_1) = \emptyset$. This contradicts Lemma 8(d). □

The above lemma implies that the writers-valency of writers-closed histories contains only one value.

Lemma 10. *If history H is writers-closed, then $|\mathcal{W}(H)| = 1$.*

In the following we show that, from a writers-supervalent history H, no writers-only schedules can linearize the `read` operation, i.e. $\mathcal{W}(H) = \emptyset$. To proof is by contradiction, assuming that $\mathcal{W}(H)$ contains an element x. In the following lemma, we first show that if such a writers-supervalent history H exists, then we can extend that history to a history H', so that the writers-valencies obtained by a single step of w_0 respectively w_1 are distinct.

Lemma 11. *If there exists a writers-supervalent history H such that $x \in \mathcal{W}(H)$, then there is a finite writers-only schedule σ and an index $j \in \{0,1\}$, such that*

(a) $H \triangleright \sigma w_j$ is writers-supervalent and $x \notin \mathcal{W}(H \triangleright \sigma w_j)$; and
(b) $H \triangleright \sigma w_{1-j}$ is writers-closed and $\mathcal{W}(H \triangleright \sigma w_{1-j}) = \{x\}$.

In particular, $\mathcal{W}(H \triangleright \sigma w_j) \cap \mathcal{W}(H \triangleright \sigma w_{1-j}) = \emptyset$.

Proof. Let σ be a longest possible writers-only schedule such that for each prefix σ'' of σ, history $H{\triangleright}\sigma''$ is writers-supervalent and $x \in \mathcal{W}(H{\triangleright}\sigma'')$.

First we prove that σ is finite. Suppose it is not. Then in $H{\triangleright}\sigma$ at least one of the writers takes infinitely many steps. By wait-freedom that writer completes infinitely many `increment` operations in $H{\triangleright}\sigma$. Let σ' be some finite prefix of σ such that $H{\triangleright}\sigma'$ contains at least $x{+}1$ complete `increment` operations. By the construction of σ, $H{\triangleright}\sigma'$ is writers-supervalent and $x \in \mathcal{W}(H{\triangleright}\sigma')$. From writers-supervalency it follows that the `read` cannot appear in $f(H{\triangleright}\sigma')$, while on the other hand this linearization contains at least $x{+}1$ `increment` operations. Since f is prefix preserving, if the `read` appears in $f(H{\triangleright}\sigma'\lambda)$ for any schedule λ, then it must be preceded by at least $x{+}1$ `increment` operations, and thus return a value of at least $x{+}1$. Hence, $x \notin \mathcal{W}(H{\triangleright}\sigma')$, contradicting the construction of σ.

We conclude that σ is finite. In particular, for every writer w_i, $i{\in}\{0,1\}$, either $H{\triangleright}\sigma w_i$ is writers-closed or $x{\notin}\mathcal{W}(H{\triangleright}\sigma)$. According to Observation 7, the extensions $H{\triangleright}\sigma w_0$ and $H{\triangleright}\sigma w_1$ cannot be both writers-closed. Hence, there is an index $j{\in}\{0,1\}$ such that $H{\triangleright}\sigma w_j$ is writers-supervalent and $H{\triangleright}\sigma w_{1-j}$ is writers-closed. Since $H{\triangleright}\sigma w_j$ is writers-supervalent, we know from the definition of σ that $x{\notin}\mathcal{W}(H{\triangleright}\sigma w_j)$. But since $x{\in}\mathcal{W}(H{\triangleright}\sigma){=}\mathcal{W}(H\sigma w_0){\cup}\mathcal{W}(H{\triangleright}\sigma w_1)$, it must be in $\mathcal{W}(H{\triangleright}\sigma w_{1-j})$. Because $H{\triangleright}\sigma w_{1-j}$ is writers-closed, we obtain from Lemma 10 that $\mathcal{W}(H{\triangleright}\sigma w_{1-j}){=}\{x\}$. Hence, (a)-(b) are satisfied, and thus $\mathcal{W}(H{\triangleright}\sigma w_j){\cap}\mathcal{W}(H{\triangleright}\sigma w_{1-j}){=}\emptyset$. □

Lemma 12. *If history H is writers-supervalent, then $\mathcal{W}(H) = \emptyset$.*

Proof. Suppose that H is writers-supervalent and assume by contradiction that there exists some value $\nu \in \mathcal{W}(H)$. By Lemma 8(a), `read` is not in $f(H)$. By Lemma 11, there is an extension H' of H and an index $i{\in}\{0,1\}$ such that

$$H'{\triangleright}w_{1-i} \text{ is writers-closed and } \mathcal{W}(H'{\triangleright}w_{1-i}) \cap \mathcal{W}(H'{\triangleright}w_i) = \emptyset. \tag{1}$$

Let R_{1-i} and R_i be the registers that w_{1-i} and w_i are poised to access in H'.

Case 1 *There is an index $j{\in}\{0,1\}$ such that in H', w_j is poised to read R_j.* Then $H'{\triangleright}w_j \overset{r,w_{1-j}}{\sim} H'$ and so $H'{\triangleright}w_jw_{1-j} \overset{r,w_{1-j}}{\sim} H'{\triangleright}w_{1-j}$. Now, either $H'{\triangleright}w_jw_{1-j}$ or $H'{\triangleright}w_{1-j}$ is writers-closed (depending on whether $j{=}i$ or $j{=}1{-}i$). Thus, by Lemma 8(d), $\mathcal{W}(H'{\triangleright}w_jw_{1-j}){\cap}\mathcal{W}(H'{\triangleright}w_{1-j}){\neq}\emptyset$. This contradicts Eq. (1).
Case 2 *w_0 is poised to write to R_0 in H', w_1 is poised to write to R_1 and $R_0{\neq}R_1$.* Then $H'{\triangleright}w_0w_1{\sim}H'{\triangleright}w_1w_0$. Also, either $H'{\triangleright}w_0w_1$ or $H'{\triangleright}w_1w_0$ is writers-closed. By Lemma 8(d), $\mathcal{W}(H'{\triangleright}w_0w_1) \cap \mathcal{W}(H'{\triangleright}w_1w_0) \neq \emptyset$, contradicting Eq. (1).
Case 3 *w_0 is poised to write to R_0 in H', w_1 is poised to write to R_1 and $R_0{=}R_1$.* Then $H'{\triangleright}w_0w_1 \overset{r,w_1}{\sim} H'{\triangleright}w_1$. Now, either $H'{\triangleright}w_0w_1$ or $H'{\triangleright}w_1$ is writers-closed (depending on whether $i{=}1$ or $i{=}0$). By Lemma 8(d), $\mathcal{W}(H'{\triangleright}w_0w_1){\cap}\mathcal{W}(H'{\triangleright}w_1){\neq}\emptyset$. This contradicts Eq. (1).

In all cases the assumption that $\nu{\in}\mathcal{W}(H)$ is contradicted. Hence, $\mathcal{W}(H){=}\emptyset$. □

Lemma 13. *Let H be a writers-supervalent history and $S \subseteq \{w_1, w_2\}$. Then:*

(a) For any integer y and any infinite S-only schedule γ, there is a prefix γ' of γ such that for every schedule λ, either $\mathcal{W}(H{\triangleright}\gamma'\lambda){=}\emptyset$ or $\min(\mathcal{W}(H{\triangleright}\gamma'\lambda)){>}y$.
(b) If $x \in \mathcal{W}(H{\triangleright}r)$, then there exists a finite S-only schedule σ, such that $x \in \mathcal{W}(H{\triangleright}\sigma r)$ and $x \notin \mathcal{W}(H{\triangleright}\sigma w_i r)$ for any $w_i \in S$.

Proof. We first prove Part (a). In $H{\triangleright}\gamma$ at least one of the writers executes infinitely many steps, and thus by wait-freedom infinitely many `increment` operations. Hence, there is a finite prefix γ' of γ such that in $H{\triangleright}\gamma'$ at least $y{+}1$ `increment` operations complete. By Lemma 12, $\mathcal{W}(H) = \emptyset$, and since γ' is writers-only, the `read` does not linearize in $H{\triangleright}\gamma'$. I.e., $f(H{\triangleright}\gamma')$ does not contain a `read`, while it contains at least $y{+}1$ `increment` operations. Since f is prefix-preserving it follows that if the `read` appears in $f(H{\triangleright}\gamma'\lambda)$ for any schedule λ, then it is preceded by at least $y{+}1$ `increment` operations and thus returns a value of at least $y{+}1$. Hence, either $\mathcal{W}(H{\triangleright}\gamma'\lambda) = \emptyset$ or $\min \mathcal{W}(H{\triangleright}\gamma'\lambda) \geq y{+}1$.

For Part (b), we let σ be a longest possible S-only schedule with $x{\in}\mathcal{W}(H{\triangleright}\sigma r)$. From Part (a) (with $\lambda{=}r$) we obtain that σ is finite. Hence, by construction $x{\in}\mathcal{W}(H{\triangleright}\sigma r)$ and $x{\notin}\mathcal{W}(H{\triangleright}\sigma w_i r)$ for any $w_i{\in}S$. This completes the proof. □

Below we state and prove our main lemma. It says that from any writers-supervalent history H we can construct a finite schedule σ, which includes at least one step by r, such that $H{\triangleright}\sigma$ is writers-supervalent.

Lemma 14. *If a history H is writers-supervalent, then there exists a finite writers-only schedule σ, such that $H{\triangleright}\sigma r$ is also writers-supervalent.*

Proof. Let H be a writers-supervalent history. For the purpose of a contradiction, we suppose that for every finite writers-only schedule σ, $\mathcal{W}(H{\triangleright}\sigma r)$ is writers-closed. By Lemma 10, for every such σ, $|\mathcal{W}(H{\triangleright}\sigma r)|{=}1$.

By Lemma 13 (b) there exists a writers-only schedule yielding an extension H' of H such that $\mathcal{W}(H'{\triangleright}r) = \{x\}$, and $x \notin \mathcal{W}(H'{\triangleright}w_0 r) \cup \mathcal{W}(H'{\triangleright}w_1 r)$. By our assumption $H'{\triangleright}w_i r$ is writers-closed for any $i{\in}\{0,1\}$, and by Lemma 10, $|\mathcal{W}(H'{\triangleright}w_i r)| = 1$. Thus, there exist values y_0, y_1 such that

$$\forall i{\in}\{0,1\}:\ \mathcal{W}(H'{\triangleright}w_i r) = \{y_i\} \neq \{x\} = \mathcal{W}(H'{\triangleright}r). \tag{2}$$

In particular, for any schedule λ, by $\mathcal{W}(H'{\triangleright}r\lambda) \subseteq \mathcal{W}(H'{\triangleright}r)$, we have

$$\forall i{\in}\{0,1\}:\ \mathcal{W}(H'{\triangleright}w_i r) \cap \mathcal{W}(H'{\triangleright}r\lambda) = \emptyset. \tag{3}$$

We look at the steps that the processes are poised to take in H'. Let R_0, R_1, and R_2 be registers accessed by w_0, w_1, and r respectively. Recall that by assumption all extensions $H' \circ H''$ of H' are writer-closed provided that r takes a step in H''.

Case 1 *There is an index $i{\in}\{0,1\}$ such that in H' process w_i is poised to read R_i.* Then, $H'{\triangleright}r \overset{r,w_{1-i}}{\sim} H'{\triangleright}w_i r$, and thus by Lemma 8(d), $\mathcal{W}(H'{\triangleright}r){\cap}\mathcal{W}(H'{\triangleright}w_i r) \neq \emptyset$. This contradicts (3).
Case 2 *Both w_0 and w_1 are poised to write in H'*:

Case 2.1 *There is an index* $i\in\{0,1\}$ *such that* $R_i \neq R_2$. This means that $H'\triangleright w_i r \sim H'\triangleright r w_i$ and thus by Lemma 8(d), $\mathcal{W}(H'\triangleright w_i r)\cap\mathcal{W}(H'\triangleright r w_i)\neq\emptyset$, which contradicts (3).

Case 2.2 *All three processes access the same register.* I.e., there exists register R such that for any $i\in\{0,1,2\}$, $R=R_i$.

Case 2.2.1 r *is poised to write in* H'. Then, $H'\triangleright w_0 r w_1 \sim H'\triangleright r w_0 w_1$ and thus by Lemma 8(d), $\mathcal{W}(H'\triangleright w_0 r w_1)\cap\mathcal{W}(H'\triangleright r w_0 w_1)\neq\emptyset$. This contradicts (3).

Case 2.2.2 r *is poised to read in* H'. We will construct two indistinguishable histories, in which r outputs different values. This establishes a contradiction.

Recall that by our assumption, for any writers-only schedule σ, $H'\sigma r$ is writers-closed and thus by Lemma 10, $|\mathcal{W}(H'\sigma r)|=1$. Then according to Lemma 13 (a), there is a w_1-solo schedule $w_1^{k_1}$ of length k_1 such that for any schedule λ, the unique value in $\mathcal{W}(H'\triangleright w_1^{k_1}\lambda r)$ is larger than y_0. Let z be the value in $\mathcal{W}(H'\triangleright w_1^{k_1} r)$. Applying Lemma 13 (b), we obtain a w_1-solo schedule $w_1^{k_2}$ of length k_2 such that for $k=k_1+k_2$ we have $z\in\mathcal{W}(H'\triangleright w_1^k r)$ and $z\notin\mathcal{W}(H'\triangleright w_1^{k+1} r)$. In particular,

$$\mathcal{W}(H'\triangleright w_1^k r)\cap\mathcal{W}(H'\triangleright w_1^{k+1} r)=\emptyset. \tag{4}$$

We now consider the histories

$$H_1 = H'\triangleright w_1^k w_0 r w_1 \quad\text{and}\quad H_2 = H'\triangleright w_0 r w_1^{k+1}.$$

Recall that by the construction above, for any schedule λ, the unique value in $\mathcal{W}(H'\triangleright w_1^{k_1}\lambda r)$ is larger than y_0. In particular, this is true for $\lambda=w_1^{k_2}w_0$, and thus $\mathcal{W}(H'\triangleright w_1^k w_0 r\}=\mathcal{W}(H_1)=\{z'\}$, for some integer $z'>y_0$. On the other hand, by (2), $\mathcal{W}(H'\cdot w_0 r)=\{y_0\}$, and thus $\mathcal{W}(H_2)=\{y_0\}$. Therefore, $\mathcal{W}(H_1)\neq\mathcal{W}(H_2)$. We now show that $H_1\sim H_2$. This contradicts Lemma 8(d) according to which $\mathcal{W}(H_1)=\mathcal{W}(H_2)$.

First, observe that all processes take equally many steps after H'. By the assumption of Case 2, the first step by each process w_0 and w_1 following H' is a write to R, while the first step by r is a read of R. Hence, in both histories in their single steps following H', process w_0 writes some value ν to R and process r reads that value ν.

Observe that w_1 is poised to write to R in $H'\triangleright w_1^k$. Otherwise, the steps w_1 and r would be commutative and thus $H'\triangleright w_1^{k+1} r \overset{r,w_1}{\sim} H'\triangleright w_1^k r w_1$. Then Lemma 8(d) would imply $\mathcal{W}(H'\triangleright w_1^{k+1} r)\cap\mathcal{W}(H'\triangleright w_1^k r w_1)\neq\emptyset$, which contradicts (4). Since the first step by w_1 is also a write to R, in both histories following H', in each single step process w_1 either writes to R, it reads from R what itself has written to R, or it accesses a register other than R. In any of those cases, w_1 cannot distinguish between H_1 and H_2. Thus, we conclude that $H_1\sim H_2$.

Hence, the assumption that from a writers-supervalent history, all finite schedules $\{w_0,w_1\}^* r$ lead to writers-closed histories, leads to contradictions in all cases. This completes the proof of the lemma. □

Lemma 15. *Any history* H*, in which* r *has taken no steps, is writers-supervalent.*

Proof. For the purpose of a contradiction assume that H is writers-closed. By Lemma10 there is an integer $x\geq 0$ such that $\mathcal{W}(H)=\{x\}$. Then there is a

writers-only schedule γ such that $f(H \triangleright \gamma)$ contains a `read` operation that returns x. By wait-freedom, there is a w_0-only schedule σ such that in $H \triangleright \gamma\sigma$ process w_0 completes at least $x{+}1$ `increment` operations. Again by wait-freedom, for a long enough r-only schedule λ, the `read` operation returns in history $H \triangleright \gamma\sigma\lambda$. In that history, the `read` is invoked after at least $x{+}1$ `increment` operations completed, so in $f(H \triangleright \gamma\sigma\lambda)$ the `read` also appears only after at least $x{+}1$ `increment` operations. But then $f(H \triangleright \gamma\sigma)$, where the `read` appears after only x `increment` operations, cannot be a prefix of $f(H \triangleright \gamma\sigma\lambda)$, contradicting the prefix-preserving property of f. □

Theorem 16. *There is no (deterministic) strongly linearizable wait-free implementation of a monotonic counter for three processes, from registers.*

Proof. Suppose by contradiction that there exists such a wait-free strongly linearizable implementation of a counter. Consider an algorithm, where processes w_0 and w_1 execute repeated `increment` operations in an infinite loop and process r executes a single `read` operation.

We prove by induction that for any integer $k \geq 0$ there is a writers-supervalent history $H_0 \circ H_1 \ldots \circ H_k$ in which r takes at least k steps. We let H_0 be the empty history. By Lemma 15, H_0 is writers-supervalent. Now suppose we constructed a writers-supervalent history $H_0 \circ \ldots \circ H_k$ in which r takes at least k steps. By Lemma 14, there is a schedule σ such that $r \in \sigma$ and history $H_0 \circ \ldots \circ H_k \circ H_{k+1} := H_0 \circ \ldots \circ H_k \triangleright \sigma$ is also writers-supervalent. In that history r takes at least $k{+}1$ steps, and the inductive hypothesis follows. Since $H_0 \circ \ldots \circ H_k$ is writers-supervalent, the `read` is pending in this history, as it does not appear in $f(H_0 \circ \ldots \circ H_k)$. Hence, there exists a history in which r takes infinitely many steps but never finishes its `read` operation. This contradicts wait-freedom. □

Strong linearizability is a composable property [2]. Hence, if there is a strongly linearizable implementation of a type T from atomic base objects of types in a set B, then T also has a strongly linearizable implementation from strongly linearizable objects of types in B. Strongly linearizable monotonic counters can be implemented from atomic (and thus from strongly linearizable) snapshot objects and general counters. Thus, Theorem 1 for snapshots and general counters follows from Theorem 16.

Now suppose there is a wait-free strongly linearizable max-register R. In Section 4, we give an algorithm that uses a linearizable object V of a type T from a certain class of types together with R, and yields a strongly linearizable object V_{strong} of type T. The algorithm itself is wait-free, so if V and R are wait-free, then so is V_{strong}. We can apply this algorithm, using for V a standard wait-free implementation of a monotonic counter with atomic `increment` operations. Using that we obtain a wait-free strongly linearizable monotonic counter V_{strong}, contradicting Theorem 16. As a consequence, the assumption that there is a wait-free strongly linearizable max-register R is wrong. This completes the proof of Theorem 1.

4 Lock-Free Implementations

We now explain how to obtain several lock-free strongly linearizable objects from atomic multi-writer registers. These objects include monotonic counters, snapshot objects, general counters, and logical clocks. We first define the notion of a *versioned* object, which is an object that increases a version number whenever it changes the state of the object. We give several examples of *linearizable* lock-free versioned objects including counters and snapshot objects. All those implementations have in common that update operations are atomic, and only the read operations are non-atomic. Then, we show how to transform any lock-free linearizable versioned object with atomic update operations into a lock-free strongly linearizable object of the same type. This transformation yields many lock-free strongly linearizable implementations from multi-writer registers.

Versioned Objects Many objects are easy to augment with version numbers that increase with every successful update operation. In the following we define such versioned variants of those types formally.

We consider a class $\mathcal{T}$ of types that support two operations, `read`() and `update`(v). The sequential specification of each type in the class is uniquely defined by the state space Q of the sequential object of that type, its initial state, q_0, and two functions, f and g. For the following discussion, the initial state, q_0, is not relevant, so we ignore it, and denote such a type as $T_{Q,f,g}$. A `read`() operation on the sequential object does not change the state of the object, but returns $f(q)$, where q is its current state. The operation `update`(v) changes the state of the object from its current state, s, to $g(s)$, and does not return anything. It is easy to see that snapshots, counters, and max-registers are all types in $\mathcal{T}$. For example, the monotonic counter is the type $T_{Q,f,g}$ with $Q = \mathbb{N} \cup \{0\}$, $f(x) = x$, and $g(x) = x + 1$.

Let $T_{Q,f,g}$ be some type in $\mathcal{T}$. A type $T_{Q',f',g'}$ is called a *versioned* variant of type T, if $Q' = Q \times \mathbb{N}$, $f'(x, v) = (f(x), v)$, and $g'(x, v) = (g(x), v')$, where $v' > v$. I.e., the versioned variant of type T stores exactly the same information as T in addition to a *version number*, v. That version number gets returned by `read` operations, and increased with every `update` operation. For example, a versioned variant of the monotonic counter is the type $T_{Q',f',g'}$, where $Q'{=}\mathbb{N}{\times}\mathbb{N}$, $f(x, x){=}(x, x)$, and $g(x){=}(x{+}1, x{+}1)$.

It is easy to obtain linearizable versioned variants of some popular types, including snapshots, by embedding in each object an internal counter that gets incremented atomcially with each update operation. The lock-free linearizable snapshot implementation by [6] has the property that `update` operations are atomic. Hence, for the versioned variants of all types mentioned above, in particular snapshots, (general) counters and logical clocks, we obtain lock-free linearizable implementations from registers, with atomic `update` operations.

Making Linearizable Versioned Objects Strongly Linearizable We show that any lock-free linearizable implementation of a versioned object can be transformed into a lock-free strongly linearizable one, provided that `update` operations of the

versioned object are atomic. For that we use the lock-free strongly linearizable max-register implementation of Helmi et al. [4]. We augment the integer value stored in a max-register with some additional information.

An *augmented max-register* stores a pair (x, y), where $x \in \mathbb{N} \cup \{0\}$, and y is from some arbitrary domain D. It supports the operations `maxRead`() and `maxWrite`(x, y). If the state of the object is (x, y), then a `maxRead`() returns (x, y), and `maxWrite`(x', y') changes the object's state to (x', y') provided that $x' > x$. Otherwise, the object's state remains unchanged.

Existing linearizable max-register implementations from registers (e.g., [10]) can be easily transformed into linearizable augmented max-register objects. This is also true for the lock-free strongly linearizable max-register implementation of Helmi et al. [4].

We now give an implementation of an object V_{strong} of type $T \in \mathcal{T}$, from an implementation V of a versioned variant of T and an augmented max-register R. Object V_{strong} is strongly linearizable, provided that R is strongly linearizable, V is linearizable, and the `update` operations of V are atomic.

The idea is simple: to execute V_{strong}.`update`(x), a process first updates V using V.`update`(x), and then reads V to obtain the pair (y, vno), where vno is the current version number of the object. Finally, it max-writes the pair (vno, y) into the augmented max-register R. To read object V_{strong}, a process simply returns the augmented value read from the max-register R.

Lemma 17. *If R is strongly linearizable, V is linearizable, and operations V.`update` are atomic, then V_{strong} is strongly linearizable.*

The implementation of V_{strong} uses only wait-free code in addition to the operations on V and R. Hence, if V and R are lock-free, then so is V_{strong}. As mentioned, there exists a lock-free implementation of augmented max-registers. Thus, we obtain the following theorem, which immediately implies Theorem 2.

Theorem 18. *Let T be a type in $\mathcal{T}$, and T' a versioned variant of T. If T' has a lock-free* linearizable *implemenation with atomic `update` operations, then T' also has a lock-free* strongly linearizable *implementation.*

5 Discussion

In this paper, we proved that several important types, such as snapshots, counters, and max-registers, have lock-free, but not wait-free, strongly linearizable implementations from registers. The negative results show that in a system with atomic registers, strong linearizability is significantly harder to obtain than linearizability.

On the other hand, recall that strong linearizability is necessary to preserve probability distributions when replacing atomic objects with implemented ones in randomized algorithms scheduled by a strong adaptive adversary [2]. Therefore, it remains an important task to find ways of implementing synchronization primitives that are robust for randomized algorithms. This can be achieved, for

example, by using stronger base objects, such as compare-and-swap. However, care needs to be taken to ensure that the system that provides those base objects (e.g., the hardware) ensures that they are at least strongly linearizable. Another way could be to use *randomized* wait-free implementations of objects. Note that strong linearizability has been defined only for deterministic objects (whereas the algorithms that use those objects can be randomized). Additional work is needed to formalize an equivalent notion for randomized objects.

Acknowledgments This research was undertaken, in part, thanks to funding from the Canada Research Chairs program and from the Discovery Grants program of the Natural Sciences and Engineering Research Council of Canada (NSERC).

We thank Hagit Attiya for the useful discussion on wait-freedom versus lock-freedom.

References

1. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12** (1990) 463–492
2. Golab, W., Higham, L., Woelfel, P.: Linearizable implementations do not suffice for randomized distributed computation. In: Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing. STOC '11, New York, NY, USA, ACM (2011) 373–382
3. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13** (1991) 124–149
4. Helmi, M., Higham, L., Woelfel, P.: Strongly linearizable implementations: Possibilities and impossibilities. In: Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing. PODC '12, New York, NY, USA, ACM (2012) 385–394
5. Herlihy, M.: Impossibility results for asynchronous pram (extended abstract). In: Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures. SPAA '91, New York, NY, USA, ACM (1991) 327–336
6. Afek, Y., Dolev, D., Attiya, H., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. In: Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing. PODC '90, New York, NY, USA, ACM (1990) 1–13
7. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32** (1985) 374–382
8. Loui, M.C., Abu-Amara, H.H.: Memory requirements for agreement among unreliable asynchronous processes. Advances in Computing Research (1987) 163–183
9. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996)
10. Aspnes, J., Attiya, H., Censor, K.: Max registers, counters, and monotone circuits. In: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing. PODC '09, New York, NY, USA, ACM (2009) 36–45