University of Calgary

PRISM Repository **https://prism.ucalgary.ca**

The Vault Open Theses and Dissertations

2019-08-20

Strongly Linearizable Implementations of Fundamental Primitives

Ovens, Sean

Ovens, S. (2019). Strongly Linearizable Implementations of Fundamental Primitives (Master's thesis, University of Calgary, Calgary, Canada). Retrieved from https://prism.ucalgary.ca.

http://hdl.handle.net/1880/110755

Downloaded from PRISM Repository, University of Calgary

UNIVERSITY OF CALGARY

Strongly Linearizable Implementations of Fundamental Primitives

by

Sean Ovens

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

AUGUST, 2019

© Sean Ovens 2019

Abstract

Linearizability is the gold standard of correctness conditions for shared memory algorithms, and historically has been considered the practical equivalent of atomicity. However, it has been shown [1] that replacing atomic objects with linearizable implementations can affect the probability distribution of execution outcomes in randomized algorithms. Thus, linearizable objects are not always suitable replacements for atomic objects. A stricter correctness condition called strong linearizability has been developed and shown to be appropriate for randomized algorithms in a strong adaptive adversary model [1].

We devise several new lock-free strongly linearizable implementations from atomic registers. In particular, we give the first strongly linearizable lock-free snapshot implementation that uses bounded space. This improves on the unbounded space solution of Denysyuk and Woelfel [2]. As a building block, our algorithm uses a lock-free strongly linearizable ABA-detecting register. We obtain this object by modifying the wait-free linearizable ABA-detecting register of Aghazadeh and Woelfel [3], which, as we show, is not strongly linearizable.

Aspnes and Herlihy [4] identified a wide class types that have wait-free linearizable implementations from atomic registers. These types require that any pair of operations either commute, or one overwrites the other. Aspnes and Herlihy gave a general wait-free linearizable implementation of such types, employing a wait-free linearizable snapshot object. Replacing that snapshot object with our lock-free strongly linearizable one, we prove that all types in this class have a lock-free strongly linearizable implementation from atomic registers.

Acknowledgements

I thank my supervisor, Philipp Woelfel, for all of the helpful discussions, editing, and guidance he has provided for the past two years.

For all of the insightful conversations, group lunches, dinners, escape rooms, board games, hiking trips, etc. I thank the distributed computing group at the University of Calgary, consisting of: Lisa Higham, Mehrdad Jafari Giv, Zahra Aghazadeh, Maryam Elahi, Aryaz Eghbali, and all of those who joined us as guests.

I am especially grateful to my family and friends for their support throughout my education so far. Special thanks to Kendra Wannamaker for her assistance, reassurance, and for encouraging me to take a break every once in a while.

Table of Contents

Abstract ii

Acknowledgements iii

Table of Contents iv

List of Figures and Illustrations v

List of Tables vi

1 Introduction 1
- 1.1 Related Work 4
- 1.2 Results 7

2 Preliminaries 10

3 A Strongly Linearizable ABA-Detecting Register 15
- 3.1 A Linearizable ABA-Detecting Register 15
- 3.2 Making the Algorithm Strongly Linearizable 20
- 3.3 Lock-Freedom and Complexity Analysis 27
- 3.4 Remarks 29

4 A Strongly Linearizable Snapshot 31
- 4.1 Unbounded Implementation 31
- 4.2 Interpreted Value 35
- 4.3 Bounded Implementation 38
- 4.4 Lock-Freedom and Complexity Analysis 54
- 4.5 Remarks 62

5 General Construction 64
- 5.1 Storing a Precedence Graph 69
- 5.2 Proof of Strong Linearizability 77
- 5.3 Remarks 89

6 Dynamic Resiliency 91
- 6.1 Remarks 98

7 Conclusion 100

List of Figures and Illustrations

1.1 After observing the result of p's $O.Flip()$ operation, the strong adaptive adversary can decide whether r's $O.Read()$ operation returns 0 or 1. 3

6.1 The state machine representation of the type $\mathscr{T}_D$. Assume that all transitions that are not drawn are self-loops. Type $\mathscr{T}_D$ also has a *Read* invocation description (called *read*), which is not displayed. 94

6.2 A diagram of the transcript T_D. The beginning of each interval represents the atomic *root.scan* operation from line 115 (labelled by s), while the end of each interval represents the atomic *root.update* operation from line 125 (labelled by w). 97

List of Tables

6.1 The relationships between invocation descriptions in each state of the type depicted in Figure 6.1. Note that x, y, and z are *Update* invocation descriptions. Again, the invocation description *read* is not shown here; *read* is overwritten by every other invocation. 96

7.1 A summary of relevant results and further directions. The table should be interpreted as follows: the first column contains a set of base objects B, the second column contains a type T, and the third column contains a liveness property L. The fourth column contains an answer to the following question: does type T have a strongly linearizable implementation from B that satisfies L? In all cases above "atomic registers" refers to atomic *multi-writer* registers. S.L. stands for either strong linearizability or strongly linearizable. 103

1. Introduction

In general, correctness properties for concurrent objects are defined by the sequential behaviours they preserve. That is, overlapping operations on concurrent objects are expected to respond as they would in some sequential execution on the object. Linearizability, a particularly popular correctness condition, requires that concurrent executions correspond to sequential histories that preserve the real-time order of operations. Intuitively, operations on a linearizable implementation appear to take effect (i.e. linearize) at some atomic step between their invocation and response. Linearizable implementations adequately preserve the sequential behaviour of their atomic counterparts; that is, any execution of a linearizable implementation "appears" to be a sequential execution of atomic operations.

However, some subtle guarantees are lost with linearizable object implementations. For instance, it has been shown that the probability distribution over the execution outcomes of a randomized algorithm can change when atomic objects are replaced with linearizable implementations [1]. More generally, linearizability does not preserve any property that cannot be expressed as a set of allowable sequences of operations [5]. To address these shortcomings, Golab, Higham, and Woelfel [1] defined the notion of strong linearizability, and showed that replacing atomic objects with strongly linearizable implementations does not change the probability distribution of the algorithm's outcome under a *strong adversary*. A strong adversary has the power to schedule executions with complete knowledge of the system state, including all previous random operations (coin flips). Strong linearizability requires that, once an operation has linearized, its position in the linearization order does not change in the future. That is, operations cannot be retroactively inserted into the linearization order. Strongly linearizable objects can simplify randomized algorithm design, which provides motivation for developing strongly linearizable implementations or proving their non-existence.

We use a well-known implementation of a *counter* (provided for reference in Algorithm 1) to concretely demonstrate the difference between atomicity and linearizability in randomized algorithms. A counter is a type that stores a single integer (initially 0) and supports two operations: $Inc()$, which increases the stored value by 1, and $Read()$, which returns the stored value.

Algorithm 1: A linearizable counter implementation

```
   shared:
     register R[1...n] = [0,...,0]

1  Function Inc_p():
2  |  x ← R[p].Read()
3  |  R[p].Write(x + 1)

4  Function Read():
5  |  s ← 0
6  |  for i ∈ {1,...,n} do
7  |  |  s ← s + R[i].Read()
8  |  end
9  |  return s
```

In this implementation, every process p has an associated single-writer register $R[p]$. To perform an $Inc_p()$ operation, process p reads $R[p]$ to obtain the value x, then writes the value $x + 1$ to $R[p]$. A $Read()$ operation iteratively reads all of the registers in the array R and returns the sum of all of these values.

Consider the following programs for the processes p, q, r, where O is a counter object:

$$p = O.Inc_p(),\ c \leftarrow Flip()$$

$$q = O.Inc_q()$$

$$r = O.Read()$$

Note that the $Flip()$ operation picks a value uniformly and at random from the set $\{0, 1\}$. We proceed by describing a strong adversary that aims to make r's $O.Read()$ operation return

the randomly-selected value c.

First, suppose O is an atomic counter. Since c is either 0 or 1, the strong adaptive adversary should never schedule r's $O.Read()$ operation after both $O.Inc()$ operations (since the $O.Read()$ would return 2 in this circumstance). Hence, the adversary may schedule r's $O.Read()$ operation either before or after a single $O.Inc()$ operation (it does not matter whether this operation is performed by p or q). Therefore, r's $O.Read()$ operation returns c with probability $\frac{1}{2}$.

Now suppose O is implemented by Algorithm 1. By scheduling operations as in Figure 1.1, the adversary can force r's $O.Read()$ operation to return c with probability 1.

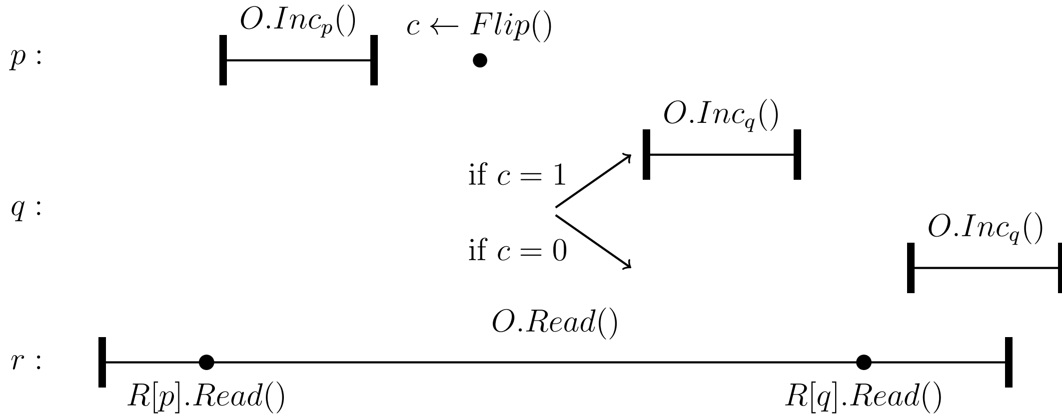

Figure 1.1: After observing the result of p's $O.Flip()$ operation, the strong adaptive adversary can decide whether r's $O.Read()$ operation returns 0 or 1.

In the execution in Figure 1.1, process r first reads 0 from $R[p]$ during its $O.Read()$ operation. Following this, p performs its entire $O.Inc_p()$ operation, writing 1 to $R[p]$ in the process. After this, p performs its $Flip()$ operation, which stores either 0 or 1 in the variable c. If $c = 0$, then the adversary allows r's $O.Read()$ operation to terminate and return the value 0. Otherwise, if $c = 1$, then the adversary allows q to perform its entire $O.Inc_q()$ operation, writing 1 to $R[q]$ in the process. Afterwards, the adversary allows r's $O.Read()$ operation to run until termination; the $O.Read()$ operation reads 1 from $R[q]$, and ultimately returns the value 1. Therefore, in both cases r's $O.Read()$ operation returns the value c. The execution

in Figure 1.1 demonstrates how a strong adversary may be given additional power when atomic base objects are replaced with linearizable implementations.

1.1 Related Work

The notion of strong linearizability was originally introduced by Golab, Higham, and Woelfel [1]. Strongly linearizable implementations are not only linearizable, but they also exhibit the "prefix-preservation" property. This extra condition ensures that the linearization order of operations does not change retroactively. Such implementations are resilient against strong adaptive adversaries, which have the power to schedule executions with complete knowledge of the system state, including all previous random operations (coin flips). In fact, it is known that strong linearizability is necessary to curtail the power of the strong adversary to influence the probability distribution of execution outcomes [1]. Importantly, strong linearizability, like traditional linearizability, is a local property. Roughly speaking, a correctness property is local if the system satisfies the property provided that each object in the system satisfies the property [6]. Hence, since strong linearizability is local, if every object in the set $\{O_1, \ldots, O_k\}$ is strongly linearizable, then any execution obtained by performing operations on any subset of $\{O_1, \ldots O_k\}$ is also strongly linearizable [1, 7]. Strong linearizability is also composable, meaning a strongly linearizable implementation O that uses an atomic base object B of type $\mathscr{T}$ remains strongly linearizable when B is replaced by B', where B' is a strongly linearizable implementation of $\mathscr{T}$ [1, 7]. Locality and composability can simplify distributed algorithm design, as we will see in Chapters 4 and 5 of this thesis.

More generally, it has recently been shown [7] that strong linearizability is a specific form of strong observational refinement. Traditional refinement [8] is a relationship between concrete objects and their specifications; that is, an object O_1 (i.e. a concrete object) refines an object O_2 (i.e. a specification) if the set of traces (i.e. sequences of possible actions) of O_1 is a subset of the set of traces of O_2. An object O_1 is said to *observationally* refine O_2 if every observation that can be made by a program using O_1 (i.e. effects of operation calls on O_1 that

are observable by the program) could also be made by the same program using O_2 instead of O_1. Finally, O_1 *strongly* observationally refines O_2 if, for every schedule of a program that uses O_1, there exists a schedule for the same program that uses O_2 instead of O_1, such that the program makes precisely the same observations in both scenarios. It has been shown [9] that observational refinement and linearizability are equivalent when the specification (i.e. O_2) is atomic. Similarly, strong observational refinement is equivalent to strong linearizability when the specification is atomic [7]. Strong observational refinements are designed to preserve hyperproperties, which are sets of sets of sequences of program observations. Hyperproperties are a generalization of the notion of "execution outcomes", mentioned previously. It is known that refinement (and observational refinement) preserves trace properties, which are sets of sequences of actions, but it does not preserve hyperproperties in general [7]. This provides further insight into the deficiencies of linearizability described by Golab, Higham, and Woelfel [1]. Since strong observational refinements preserve the hyperproperties satisfied by their specifications, this implies that strongly linearizable implementations preserve the hyperproperties satisfied by their atomic counterparts.

Unless otherwise noted we assume the standard asynchronous shared memory system, where n processes with unique IDs in $\{1, \ldots, n\}$ communicate through atomic read and write operations on shared (multi-reader multi-writer) registers. Almost all prior work on strong linearizability has focused on this model. However, it is known that standard wait-free universal constructions (e.g. [10]) using n-process consensus objects are also strongly linearizable [1]. Therefore, there exists a strongly linearizable implementation of any type using atomic compare-and-swap objects, for example. On the other hand, Attiya, Castañeda, and Hendler [11] have shown that any wait-free strongly linearizable implementation of a queue or a stack for n processes along with atomic registers can solve n-process consensus. Hence, any n-process wait-free strongly linearizable implementation of a queue or stack cannot be implemented from base objects with consensus number less than n.

Early results on strong linearizability have largely been negative (i.e. impossibility results).

Helmi, Higham, and Woelfel [12] have shown that essentially no non-trivial object has a deterministic wait-free strongly linearizable implementation from *single-writer* registers. Denysyuk and Woelfel [2] showed that for several fundamental types, including single-writer snapshots (defined below), counters, and unbounded max-registers, there exist no strongly linearizable wait-free implementations, even from multi-writer registers.

While many published results on strong linearizability (especially for lock/wait-free implementations) are discouraging, some fundamental types are known to have strongly linearizable implementations. For instance, Helmi, Higham, and Woelfel [12] describe a strongly linearizable wait-free implementation of a bounded max-register from multi-reader multi-writer registers [13]. A simple modification of this algorithm, which we describe in more detail in Section 4.1, results in a strongly linearizable lock-free implementation of an unbounded max-register from unbounded multi-reader multi-writer registers. Helmi, Higham, Woelfel [12] also demonstrate that there is a strongly linearizable obstruction-free implementation of a consensus object from multi-reader single-writer registers. This implies that any type has a strongly linearizable obstruction-free implementation from multi-reader single-writer registers. Denysyuk and Woelfel [2] have shown that there exists a universal lock-free strongly linearizable construction for versioned objects, which store version numbers that increase with each atomic update operation (a more detailed explanation of this construction, and of versioned objects, is provided in Section 4.1). This construction uses the unbounded modification of the max-register from [12], and therefore requires an unbounded number of registers. The algorithm inherently requires unbounded space, since the version number of the object must increase with each update.

The snapshot type [14] is a fundamental primitive in distributed algorithm design [4, 15, 16, 17, 18, 19]. In this thesis, we consider only single-writer snapshots, which contain an n-component vector of values. For any $p \in \{1, \ldots, n\}$, the p-th component of a single-writer snapshot is writable only by process p. An $update_p(x)$ invocation by process p changes the contents of the p-th component of the snapshot object to x. The snapshot type also supports

a *scan* invocation, which returns the entire stored vector. That is, the *scan* invocation allows processes to obtain a consistent view of multiple single-writer memory cells; if a *scan* invocation returns a vector V, then the snapshot object must have contained exactly V at some point in its execution interval. There are many wait-free linearizable implementations of the snapshot type from registers [14, 20, 21, 22, 23], but due to the results of Denysyuk and Woelfel [2], it is known that none of these implementations are strongly linearizable.

1.2 Results

An ABA-detecting register stores a single value from some domain D, and supports $DWrite$ and $DRead$ invocations. A $DWrite(x)$ invocation writes value $x \in D$, and a $DRead$ invocation returns the latest written value together with a Boolean flag. This flag indicates whether there has been a $DWrite$ operation since the previous $DRead$ by the same process. This type was originally defined by Aghazadeh and Woelfel [3], who also gave a wait-free linearizable implementation from $O(n)$ bounded registers. ABA-detecting registers are used to combat the ABA problem, which occurs when two $DRead$ operations with interleaving $DWrite$ operations return the same value; in this scenario the reading process is unable to distinguish between the actual execution and a different execution in which no $DWrite$ operations occur between the two $DRead$ operations. In Chapter 3 we show that Aghazadeh and Woelfel's wait-free linearizable ABA-detecting register is not strongly linearizable, and modify it to achieve strong linearizability. Our implementation sacrifices wait-freedom for lock-freedom.

Theorem 1. *There is a lock-free strongly linearizable implementation of an ABA-detecting register from $O(n)$ registers of size $O(\log n + \log |D|)$.*

The *amortized* step complexity of our implementation is $O(n)$ (an object has amortized step complexity k, if all processes combined execute at most $k \cdot \ell$ steps in any execution that comprises ℓ operation invocations). Moreover, each $DWrite$ needs only $O(1)$ steps, and each

DRead has constant step complexity in the absence of contention.

As mentioned previously, Denysyuk and Woelfel [12] sacrificed wait-freedom for lock-freedom to obtain a strongly linearizable snapshot implementation using an unbounded number of registers. In Chapter 4 we give the first such implementation that needs only bounded space.

Theorem 2. *There is a lock-free strongly linearizable implementation of a snapshot object from $O(n)$ registers of size $O(\log n + \log |D|)$.*

We provide an analysis of our implementation, showing that the amortized step complexity is $O(n^3)$. Our algorithm uses as base objects a linearizable snapshot object S, so the step complexity heavily depends on the implementation of S. But in the absence of contention, each *update* and *scan* operation of the strongly linearizable snapshot needs only a constant number of operations on S in addition to a constant number of register accesses.

Aspnes and Herlihy [4] defined a large class of types with wait-free linearizable implementations. Any two operations of this type must either commute (meaning the system configuration obtained after both operations have been executed consecutively is independent of the order of the two operations), or one operation overwrites the other (meaning that the system configuration obtained after the overwriting operation has been performed is not affected by whether or not the other operation is executed immediately before it). In this thesis we refer to such types as *simple* types. Aspnes and Herlihy [4] describe a general wait-free construction of any simple type. Their algorithm uses an atomic snapshot object, which may be replaced by a linearizable implementation to obtain a wait-free linearizable implementation of any simple type from registers. In Chapter 5 we prove that Aspnes and Herlihy's construction is also strongly linearizable. Combining this with Theorem 2, and using the composability of strong linearizability, we obtain the following:

Theorem 3. *Any simple type has a lock-free strongly linearizable implementation from registers.*

Aspnes and Herlihy introduce the notion of *linearization graphs*, which are directed acyclic graphs whose nodes are operations (we will define these structures more formally in Chapter 5). Aspnes and Herlihy define a linearization function based on topological orderings of these linearization graphs. However, since operations may be written to the "middle" of a linearization graph (i.e. an operation might have outgoing edges immediately as it is written to the graph), this linearization function is not prefix-preserving. Hence, even though we do not modify the algorithm beyond the snapshot object replacement, the proof of strong linearizability is involved.

Anderson and Moir [24] define a class of *readable* objects, which support a single read operation that returns the entire state of the object, along with a set of update operations that do not return anything and may modify the state of the object. Anderson and Moir claim that a "dynamic resiliency condition" is necessary and sufficient for the existence of a wait-free linearizable implementation of an object from registers. This condition is similar to Aspnes and Herlihy's definition of simple types, and Anderson and Moir claim that the sufficiency of the dynamic resiliency condition follows from the general construction in [4]. In Chapter 6 we show that Anderson and Moir's notion of dynamic resiliency is not equivalent to Aspnes and Herlihy's notion of simple types by defining a dynamically resilient type which is not simple. Therefore, Anderson and Moir's claim that a readable object has a wait-free implementation from registers if the object satisfies the resiliency condition remains unproven. Note that if Anderson and Moir's claim was true, combined with Theorem 3 this would imply that a readable object has a lock-free strongly linearizable implementation from atomic multi-reader multi-writer registers if and only if it satisfies the dynamic resiliency condition.

2. Preliminaries

We consider an asynchronous shared memory system with n processes, each of which has a unique identifier in $\{1, \ldots, n\}$. More precisely, processes communicate by performing operations on shared *objects*, which are each an instance of a *type*. A type may be defined as a state machine. That is, if $\mathscr{T}$ is a type, then $\mathscr{T} = (\mathcal{S}, s_0, \mathcal{O}, \mathcal{R}, \delta)$, where $\mathcal{S}$ is a set of *states*, $s_0 \in \mathcal{S}$ is an *initial state*, $\mathcal{O}$ is a set of *invocation descriptions*, $\mathcal{R}$ is a set of *responses*, and $\delta : \mathcal{S} \times \mathcal{O} \rightarrow \mathcal{S} \times \mathcal{R}$ is a *transition function.* An invocation description $invoke \in \mathcal{O}$ applied to an object of type $\mathscr{T}$ in state $s \in \mathcal{S}$ returns a value $resp \in \mathcal{R}$ and causes the object to enter state s', where $\delta(invoke, s) = (resp, s')$. Throughout this thesis, we only consider types for which $\delta(invoke, s)$ is defined for every invocation description *invoke* and for every state s. An invocation consists of a name, a set of arguments, and a process identifier (indicating the process that performs the invocation). The *sequential specification* of $\mathscr{T}$ is the set of allowable sequences of invocation/response pairs. We may define the sequential specification of $\mathscr{T}$ inductively; that is, a sequence $(invoke_1, resp_1), \ldots, (invoke_k, resp_k)$ is in the sequential specification of $\mathscr{T}$ if either (1) it is empty, or (2) $(invoke_1, resp_1), \ldots, (invoke_{k-1}, resp_{k-1})$ is in the sequential specification of $\mathscr{T}$, and $\delta(invoke_k, s_k) = (resp_k, s)$, for some state s, where s_k is the state reached after applying the invocations $invoke_1, \ldots, invoke_{k-1}$ in order. We define types using a descriptive approach; that is, instead of explicitly providing an automaton, we describe the values that an object stores, the invocation descriptions it supports, and how these invocation descriptions change the stored values in sequential executions. We present invocation descriptions using pseudocode; for instance, $op_p(x, y)$ denotes an invocation description named *op*, with x and y as arguments, and with p as the associated process identifier. Objects that are provided by the system are called *base objects.* An operation *op* consists of an *invocation event*, denoted $inv(op)$, and possibly a *response event*, denoted $rsp(op)$. Processes perform operations sequentially. An invocation event is a tuple (O, M, id),

where O is an object instance, M is a invocation description, and *id* is a unique integer that identifies the invocation event. A response event is a pair (r, id), where r is a return value, and *id* is an integer. An invocation event (O, M, id_i) *matches* a response event (r, id_j) (and vice versa) if and only if $id_i = id_j$. A *transcript* is a sequence of *steps*, each of which is either an invocation event or a response event of some operation. If T and U are transcripts, we use $T \circ U$ to denote the concatenation of T and U.

A *projection* of a transcript T onto an object O, denoted $T|O$, is the sequence of steps in T that are performed on O. Similarly, a projection of a transcript T onto a process p, denoted $T|p$, is the sequence of invocation and response events by process p. We say $e \in T$, for some invocation or response event e and some transcript T, if e is a member of the sequence defined by T. As a shorthand, we say $op \in T$, for an operation op and a transcript T, if $inv(op) \in T$. An operation op is *pending* in some transcript T if T contains its invocation but no matching response. If an operation $op \in T$ is not pending in T, then it is *complete*. A transcript T is *complete* if, for every operation $op \in T$, op is complete. Otherwise, T is *incomplete*. An operation op is *atomic* in a transcript T if $inv(op)$ is immediately followed by $rsp(op)$ in T. A transcript T is *well-formed* if T is empty, or for every $p \in \{1, \ldots, n\}$, $T|p = i \circ T_1 \circ r \circ T_2$, where i is some invocation event, T_1 and T_2 are well-formed transcripts, T_1 is complete, and r is either a response event that matches i, or r is empty. Throughout this paper, we assume all transcripts are well-formed.

An object O is *atomic* if every operation in any transcript on O is atomic. We say an object O of type $\mathscr{T}$ is *implemented* if each invocation description provided by $\mathscr{T}$ is associated with a *method*, which is a sequence of invocations. A process p that executes an operation invocation $inv(op)$ on an implemented object O, such that the invocation description of $inv(op)$ is associated with a method M, sequentially executes each invocation step described by M. During the execution of this method, other processes may also take steps, which interleave with the steps taken by p.

The order of operations in a transcript is a partial order, since some operations may

overlap. Operation op_1 *happens before* operation op_2 in transcript T, or $op_1 \xrightarrow{T} op_2$, if and only if $rsp(op_1)$ occurs before $inv(op_2)$ in T. Operations $op_1, op_2 \in T$, for some transcript T, are *concurrent* if op_1 does not happen before op_2, and op_2 does not happen before op_1.

A *history* is a transcript such that, for every process p, every operation in $H|p$ is atomic. A history may be considered a sequence of "high-level" invocation and response events. A *completion* of a history H is a complete history H' that is constructed from H as follows: for each pending operation op in H, either a response for op is appended to H', or op is removed from H'. A *sequential history* is a history that contains no concurrent operations. Suppose $S = inv(op_1) \circ rsp(op_1) \circ \ldots \circ inv(op_k) \circ rsp(op_k)$ is a sequential history on an object O of type $\mathscr{T}$, where $inv(op_i) = (O, invoke_i, id_i)$ and $rsp(op_i) = (O, resp_i, id_i)$ for every $i \in \{1, \ldots, k\}$. Then S is *valid* if and only if $(invoke_1, resp_1), \ldots, (invoke_k, resp_k)$ is in the sequential specification of $\mathscr{T}$. If T is a transcript, the *interpreted history* $\Gamma(T)$ consists of all "high-level" steps that exist in T. That is, $\Gamma(T)$ can be constructed by removing, for every process p, every step that appears after $inv(op)$ and not after $rsp(op)$, for any operation $op \in T$ with process identifier p. If $\mathcal{T}$ is a set of transcripts, then $\Gamma(\mathcal{T}) = \{\Gamma(T) : T \in \mathcal{T}\}$.

Linearizability was originally defined by Herlihy and Wing [25]. The following definition is taken from the textbook by Herlihy and Shavit [6]: a history H is *linearizable* if it has a completion H' such that there is a sequential history S with the following properties:

- All operations in H' are present in S, with identical invocations and responses;
- the sequential history S is valid; and
- the happens-before order of operations in S extends the happens-before order of operations in H'.

We call a sequential history S that satisfies the above properties a *linearization* of H. An implementation of an object is linearizable if every history in the set of possible histories on the object is linearizable. That is, if O is some object and $\mathcal{H}$ is the set of possible histories on O, then O is linearizable if and only if for all $H \in \mathcal{H}$, H is linearizable. If $\mathcal{H}$ is a set of

histories, and f is a function such that, for every $H \in \mathcal{H}$, $f(H)$ is a linearization of H, then f is called a *linearization function* for the set $\mathcal{H}$.

We often refer to the *time* at which particular steps are executed; this simply refers to the step's position in a transcript. That is, if e is a step in a transcript T, then $time_T(e) = t$ if the t-th element of T is e. Where T is clear from context, we simply write $time(e) = t$. If e is a step that is not present in a transcript T, then let $time_T(e) = \infty$. For the sake of brevity, if am is an atomic operation in a transcript T, then we say *am happens* at $time(rsp(am))$. If op is a complete operation whose implementation contains an atomic operation on line x, we use op^x to denote $rsp(am)$, where am is the operation invoked by the final call to line x performed by op.

Another characterization of linearizability relies on the notion of *linearization points*. Let O be a linearizable object. Then, for any transcript T, a *linearization point function pt* for O maps operations in $\Gamma(T|O)$ to points in time in T, such that

(i) for every operation $op \in \Gamma(T|O)$, $pt(op) \in \big[time_T(inv(op)), time_T(rsp(op))\big]$, and

(ii) there exists a linearization S of $\Gamma(T|O)$ such that for every operation $op \in \Gamma(T|O)$ such that $pt(op) \neq \infty$, $op \in S$, and for every pair of operations $op_1, op_2 \in S$, if $op_1 \xrightarrow{S} op_2$ then $pt(op_1) \leq pt(op_2)$.

Intuitively, a linearization point is a point in time between the invocation and response of an operation op at which op "appears" to take effect. In any transcript containing operations on a linearizable object O, each operation on O can be assigned a linearization point between its invocation and response, such that the sequential history that results from ordering each operation on O by these points is valid.

The *prefix closure* of a set of transcripts $\mathcal{T}$, denoted $close(\mathcal{T})$, is the set of all transcripts S such that there exists a transcript T such that $S \circ T \in \mathcal{T}$. A *strong linearization function* f for a set of transcripts $\mathcal{T}$ has the following properties [1]:

- The function f is a linearization function for $\Gamma(close(\mathcal{T}))$.

- For any two transcripts $S, T \in \mathcal{T}$ such that S is a prefix of T, $f(S)$ is a prefix of $f(T)$. That is, f is *prefix-preserving*.

An implementation O of a type $\mathscr{T}$ is called *strongly linearizable* if and only if the set of all transcripts on instances of O has a strong linearization function.

Let $\mathcal{T}$ be the set of transcripts of an implementation S of some type $\mathscr{T}$. A *continuation* of a transcript T is a transcript U such that $T \circ U$ is well-formed and $T \circ U \in \mathcal{T}$. Then S is *wait-free* if, for every transcript $T \in \mathcal{T}$ and every pending operation $op \in \Gamma(T)$ by process p, every continuation U of T that contains an infinite number of steps by p has a finite prefix U' such that $rsp(op) \in U'$. The implementation S is *lock-free* if, for every transcript $T \in \mathcal{T}$ such that $\Gamma(T)$ contains at least one pending operation, for every infinite continuation U of T, there exists an $op \in \Gamma(T)$ that is pending in T such that, for some finite prefix U' of U, $rsp(op) \in U'$. Intuitively, an implementation is wait-free if each pending operation by process p responds within a finite number of steps by p, and an implementation is lock-free if some pending operation responds provided that some process takes sufficiently many steps.

3. A Strongly Linearizable ABA-Detecting Register

An ABA-detecting register [3] is a type that stores a single value R from some domain D, and supports the invocation descriptions $DWrite_q(x)$ for $x \in D$, and $DRead_q()$ with the following sequential specification: Initially, $R = \perp \in D$, and a $DWrite_q(x)$ invocation changes the value of R to x. Invocation $DRead_q()$ returns a pair $(x, a) \in D \times \{true, false\}$, where x is the value of R, and a is $true$ if and only if q performed an earlier $DRead_q()$ operation, and a $DWrite_p$ was performed by some process p since q's last $DRead_q()$.

3.1 A Linearizable ABA-Detecting Register

Aghazadeh and Woelfel [3] presented a wait-free linearizable ABA-detecting register, which is included here as Algorithm 2 for reference. For a detailed description of the algorithm and a proof of its linearizability, see [26, 3]. This algorithm works by associating each write with a process identifier and a sequence number. Processes are also responsible for "announcing" the sequence numbers they read into a global array of single-writer registers. That is, if process p reads process identifier q and sequence number s, then it writes the pair (q, s) to the p-th entry of the announcement array A; a writer does not use a sequence number if it is paired with their process identifier in this announcement array. A writer also does not use any sequence number that is present in a local queue, called *usedQ* in Algorithm 2, which is a queue of $n + 1$ values. This queue stores the previous $n + 1$ sequence numbers chosen by the writing process, and it initially contains $n + 1$ elements valued $\perp$. Finally, if a write occurs during a read operation, the reader sets a local flag. This flag is used to delegate the task of acknowledging the modification to the reading process' next read operation.

Observation 4. *The ABA-detecting register in Algorithm 2 is not strongly linearizable.*

Algorithm 2 is not strongly linearizable because the point at which a *DRead* operation

Algorithm 2: A linearizable ABA-detecting register [3]

```
shared:
   register X = (⊥, ⊥, ⊥)
   register A[0 ... n − 1] = ((⊥, ⊥), ..., (⊥, ⊥))
local (to each process):
   Boolean b = False
   Queue usedQ[n + 1] = (⊥, ..., ⊥)
   Set na = {}
   Integer c = 0

Function DWrite_p(x):
10    s ← GetSeq()
11    X.Write(x, p, s)

Function GetSeq_p():
12    (r, s_r) ← A[c].Read()
13    if r = p then
14        na ← (na \ {(c, i) | i ∈ ℕ}) ∪ (c, s_r)
15    end
16    else
17        na ← na \ {(c, i) | i ∈ ℕ}
18    end
19    c ← (c + 1) mod n
20    choose arbitrary s ∈ ({0, ..., 2n + 1} \({i | (j, i) ∈ na} ∪ usedQ))
21    usedQ.enq(s)
22    usedQ.deq()
23    return s

Function DRead_q():
24    (x, p, s) ← X.Read()
25    (r, s_r) ← A[q].Read()
26    A[q].Write(p, s)
27    (x', p', s') ← X.Read()
28    if (p, s) = (r, s_r) then
29        ret ← (x, b)
30    end
31    else
32        ret ← (x, True)
33    end
34    if (x, p, s) = (x', p', s') then
35        b ← False
36    end
37    else
38        b ← True
39    end
40    return ret
```

takes effect depends on whether or not a $DWrite$ operation executes line 11 between lines 24 and 27 of the $DRead$; that is, if a process sets its b flag during a $DRead$ operation, then it must linearize on line 24, since the following read will detect any $DWrite$ operations that occur between lines 24 and 27. Conversely, if a process does not set its b flag during a $DRead$ operation, then this operation is responsible for detecting any $DWrite$ operations that occur between lines 24 and 27, and it must therefore linearize on line 27. This behaviour allows a scheduler to insert a $DRead$ operation in front of $DWrite$ operations that have already taken effect.

We prove Observation 4 by describing an execution between two processes, p and q. From this execution, we generate several possible transcripts. Assuming the implementation is strongly linearizable, this set of constructed transcripts must have a strong linearization function. We conclude the proof by showing that such a function cannot exist.

Proof of Observation 4. Consider an execution of Algorithm 2 where process p executes two $DRead_p$ operations dr_1 and dr_2, and process q executes an infinite sequence of $DWrite_q(x)$ operations $dw_1, dw_2, \ldots$ for some value x. Let dw_i, dw_{i+1} be two consecutive $DWrite_q(x)$ operations by q, and let s be the sequence number chosen by dw_i on line 20. Since q performs a $usedQ.enq(s)$ operation on line 21 of dw_i, and $usedQ$ contains $n+2$ elements after this operation, $usedQ$ contains s after the $usedQ.deq()$ operation performed by q on line 22 of dw_i. Following the $usedQ.deq()$ operation in dw_i, $usedQ$ is not modified again until line 21 is executed by q during dw_{i+1}. Hence, $usedQ$ contains s while q selects a sequence number on line 20 of dw_{i+1}. Thus, the sequence number chosen by dw_{i+1} is different from s. We have shown that

$$\text{no two consecutive } DWrite \text{ operations by } q \text{ choose the same sequence number.} \quad (3.1)$$

Additionally, since sequence numbers are chosen from a finite set of integers, in the infinite

sequence of $DWrite$ operations by q,

$$\text{there are distinct operations } dw_i \text{ and } dw_j \text{ that choose the same sequence number.} \tag{3.2}$$

Let dw_i, dw_j be two distinct $DWrite_q(x)$ operations in the infinite sequence performed by q (assume $i < j$), both of which choose the same sequence number s. Note that (3.1) guarantees that dw_{i+1} chooses a sequence number $s' \neq s$, and hence $dw_{i+1} \neq dw_j$. The following transcripts S, T_1, and T_2 are possible transcripts produced by the programs described for p and q:

$$\begin{aligned}
S &= dw_1 \circ \ldots \circ dw_i \circ (dr_1 \text{ to the end of line 25}) \circ dw_{i+1} \\
T_1 &= S \circ dw_{i+2} \circ \ldots \circ dw_j \circ (dr_1 \text{ from line 26 to completion}) \circ dr_2 \\
T_2 &= S \circ (dr_1 \text{ from line 26 to completion}) \circ dr_2
\end{aligned}$$

To derive a contradiction, assume that Algorithm 2 is strongly linearizable. Then there must be a strong linearization function for $\{S, T_1, T_2\}$; let f be such a strong linearization function. Since dw_i executes an $X.Write(x, q, s)$ operation on line 11, and no later $X.Write$ operations occur before dr_1 executes the $X.Read()$ operation on line 24 during S,

$$\text{the } X.Read() \text{ operation performed by } dr_1 \text{ on line 24 in } S \text{ returns } (x, q, s). \tag{3.3}$$

Since $dw_1, \ldots, dw_i$ are performed in sequence, and because each of these operations respond before any other operation is invoked in all of the above transcripts, each of $f(S)$, $f(T_1)$, and $f(T_2)$ must begin with $dw_1 \circ \ldots \circ dw_i$.

Suppose dr_1 linearizes prior to dw_{i+1} in S. That is, suppose

$$f(S) = dw_1 \circ \ldots \circ dw_i \circ dr_1 \circ dw_{i+1}. \tag{A-1}$$

The following table summarizes the steps that affect the return value of dr_2 in T_1:

Line #	**Operation**	**Code Statement**	**Response**
11	dw_j	$X.Write(x,q,s)$	—
26	dr_1	$A[p].Write(q,s)$	—
27	dr_1	$X.Read()$	(x,q,s)
35	dr_1	$b \leftarrow False$	—
24	dr_2	$X.Read()$	(x,q,s)
25	dr_2	$A[p].Read()$	(q,s)
27	dr_2	$X.Read()$	(x,q,s)
29	dr_2	$ret \leftarrow (x, False)$	—

Since dr_1 reads (x,q,s) on line 24 and line 27, dr_1 sets its b flag to $False$ on line 35 in T_1. Since dr_2 reads (x,q,s) on line 24 and line 27, by the condition on line 28 dr_2 executes $ret \leftarrow (x,b)$ on line 29 in T_1. Hence,

$$dr_2 \text{ returns } (x, False) \text{ in } T_1. \tag{A-2}$$

By (A-1), the fact that $dw_{i+2}, \ldots, dw_j$, and dr_2 are performed sequentially in T_1, and our supposition that f is prefix-preserving,

$$f(T_1) = dw_1 \circ \ldots \circ dw_i \circ dr_1 \circ dw_{i+1} \circ \ldots \circ dw_j \circ dr_2. \tag{A-3}$$

However, the history in (A-3) is not valid, since there is at least one $DWrite$ operation between dr_1 and dr_2, but dr_2 returns $(x, False)$ by (A-2).

Now suppose that dr_1 does not linearize prior to dw_{i+1} in S. That is, suppose

$$\text{either } f(S) = dw_1 \circ \ldots \circ dw_{i+1} \text{ or } f(S) = dw_1 \circ \ldots \circ dw_{i+1} \circ dr_1 \tag{B-1}$$

The following table summarizes the steps that affect the return value of dr_2 in T_2:

Line #	**Operation**	**Code Statement**	**Response**
11	dw_{i+1}	$X.Write(x, q, s')$	—
26	dr_1	$A[p].Write(q, s)$	—
24	dr_2	$X.Read()$	(x, q, s')
25	dr_2	$A[p].Read()$	(q, s)
32	dr_2	$ret \leftarrow (x, True)$	—

Due to (3.3), dr_1 executes an $A[p].Write(q, s)$ operation on line 26 in T_2. Hence, the $A[p].Read()$ operation performed by dr_2 on line 25 in T_2 must return (q, s). Since the $X.Read()$ operation performed by dr_2 on line 24 returns (x, q, s'), and $s \neq s'$, dr_2 executes the $ret \leftarrow (x, True)$ statement on line 32 by the condition on line 28 in T_2. Therefore,

$$dr_2 \text{ returns } (x, True) \text{ in } T_2. \tag{B-2}$$

By (B-1), the fact that dr_1 and dr_2 are performed sequentially, and our assumption that f is prefix-preserving,

$$f(T_2) = dw_1 \circ \ldots \circ dw_{i+1} \circ dr_1 \circ dr_2. \tag{B-3}$$

However, the history in (B-3) is not valid, since there are no $DWrite$ operations between dr_1 and dr_2, but dr_2 returns $(x, True)$ by (B-2).

Thus, no strong linearization function can be defined over the set $\{S, T_1, T_2\}$, which proves the observation. □

3.2 Making the Algorithm Strongly Linearizable

Algorithm 2 can be modified in order to make the implementation strongly linearizable. Our modification to the $DRead$ method of the linearizable ABA-detecting register is provided in Algorithm 3. The $GetSeq$ and $DWrite_p$ methods are the same as in [26]. Our strategy

is to "stretch" $DRead$ operations until a period of quiescence is observed by the reading process. Our new $DRead$ method performs the same sequence of reads as in Algorithm 2; however, each $DRead$ is now responsible for acknowledging all concurrent $DWrite$ operations, rather than delegating this task to the next $DRead$ by the same process. Processes no longer maintain a local b flag; instead, each $DRead$ operation begins by initializing a flag called *changed* to $False$ on line 41, before starting a repeat-until loop. During an iteration of the repeat-until loop by a $DRead$ operation, a process p that notices a difference in the values read from X on line 43 and line 46, or that $A[p]$ does not contain the same value as X, sets *changed* to $True$ on line 48 before repeating its sequence of reads. A process p performing a $DRead$ operation also announces the process identifier/sequence number pair read from X on line 43 to $A[p]$ on line 45. As before, the purpose of this announcement is to prevent $DWrite$ operations from choosing sequence numbers that have been observed recently. When p sees that its sequence of reads all return the same value, it can safely return this value along with the *changed* flag. In that case, p's return value is consistent with the state of the ABA-detecting register at the point of p's last shared memory operation, i.e., when it reads X for the last time. Hence, each $DRead$ method may now always be linearized at the time of its final read operation. Similarly, $DWrite$ operations linearize at their final shared memory operation, which is when they write to X. It is easy to see that if all operations can linearize with their final shared memory operation, the corresponding linearization function is prefix-preserving, and thus the object is strongly linearizable.

We now provide a formal argument that Algorithm 3 is strongly linearizable. For any operation op in a transcript T on an instance of the implementation in Algorithm 3, let $pt(op)$ be defined as follows:

Q-1 If op is a $DRead$ operation, then let $pt(op) = time(op^{46})$.

Q-2 If op is a $DWrite$ operation, then let $pt(op) = time(op^{11})$.

Let $\mathcal{T}$ represent the set of all possible transcripts of Algorithm 3. For every transcript $T \in \mathcal{T}$, define a sequential history $f(T)$ that orders operations according to pt, and excludes

Algorithm 3: *DRead* of a strongly linearizable ABA-Detecting register

```
   Function DRead_q:
41     changed = False
42     repeat
43         (x, p, s) ← X.Read()
44         (r, s_r) ← A[q].Read()
45         A[q].Write(p, s)
46         (x', p', s') ← X.Read()
47         if (p, s) ≠ (r, s_r) or (x, p, s) ≠ (x', p', s') then
48             changed ← True
49         end
50     until (p, s) = (r, s_r) and (x, p, s) = (x', p', s')
51     return (x', changed)
```

all operations op for which $pt(op) = \infty$. That is, for any two operations $op_1, op_2 \in \Gamma(T)$ such that $pt(op_1) \neq \infty$ and $pt(op_2) \neq \infty$, $op_1 \xrightarrow{f(T)} op_2$ if and only if $pt(op_1) < pt(op_2)$. Note that there is no pair of operations $op_i, op_j \in \Gamma(T)$ such that $pt(op_i) = pt(op_j) \neq \infty$, because, for every operation $op \in \Gamma(T)$, if $pt(op) \neq \infty$ then the step of T at $pt(op)$ is performed by op (see Q-1 and Q-2).

For the remainder of Section 3, let $T \in \mathcal{T}$ be some finite transcript of some ABA-detecting register implemented by Algorithm 3.

The following observation is immediate from the implementation of Algorithm 3 and Q-2:

Observation 5. *If an $X.Write(x, p, s)$ operation happens at time t, then there exists a $DWrite_p(x)$ operation dw by p such that $pt(dw) = t$.*

The following observations are immediately obtained from Claims 5.9 and 5.10 in [26]:

Observation 6. *(a) Consider two GetSeq calls g_1 and g_2 by some process p, where g_1 is invoked before g_2. If g_1 and g_2 return the same sequence number s, then p completes at least n GetSeq calls between g_1 and g_2.*

(b) Suppose $X = (x, p, s) \neq (\bot, \bot, \bot)$ at some point t, and $A[p] = s$ throughout $[t, t']$, where $t' \geq t$. Then process p does not write (x', p, s) to X during $(t, t']$ for any $x' \in D$.

Lemma 7. *Let dr be a complete $DRead_q()$ operation performed by process q, and suppose at least one $DWrite$ dw linearizes after $time(dr^{43})$ and before q invokes any other $DRead()$ operation following dr. Let p be the process executing dw, and s the sequence number associated with dw. Then*

(a) $A[q] \neq (p, s)$ *at* $time(dr^{44})$; *and*

(b) $pt(dw) \notin \big[time(dr^{43}), time(dr^{46})\big]$.

Proof. We first prove (a). For the purpose of a contradiction, assume $A[q] = (p, s)$ at $time(dr^{44})$. Since q executes dr^{44} in its final iteration of the repeat-until loop, it follows from the if-condition in line 50 that

$$\text{at } time(dr^{43}) \text{ process } q \text{ reads } (x, p, s) \text{ from } X \text{ for some value } x \in D. \tag{3.4}$$

Therefore, q writes (p, s) to $A[q]$ in dr^{45}. Since $A[q] = (p, s)$ prior to that write, and only process q can write to $A[q]$ (and only in line 45), it follows that $A[q] = (p, s)$ throughout the final iteration of the repeat-until loop of dr. Moreover, the value of $A[q]$ remains unchanged until q invokes another $DRead_q()$ operation. By the lemma assumption, $pt(dw)$ occurs before q's next $DRead_q()$ invocation, so

$$A[q] = (p, s) \text{ throughout } \big[time(dr^{43}), pt(dw)\big]. \tag{3.5}$$

By the assumption of the lemma and the fact that dw linearizes when q performs line 11 by Q-2,

$$\text{at } pt(dw) \text{ process } p \text{ writes } (y, p, s) \text{ to } X\text{, for some } y \in D. \tag{3.6}$$

Statements (3.4), (3.5), and (3.6) contradict Observation 6 (b). This completes the proof of part (a) of this lemma.

We now prove part (b). Suppose the statement is not true. Then let dw be the $DWrite$ with the latest linearization point $pt(dw) \in \big[time(dr^{43}), time(dr^{46})\big]$. Recall that process p

executes dw, and s is the sequence number used. That is, at $pt(dw)$ process p writes a triple (x, p, s) to X, for some $x \in D$.

Since each write to X occurs at the linearization point of some $DWrite$ operation, and no other $DWrite$ linearizes in $\big(pt(dw), time(dr^{46})\big]$, we have that $X = (x, p, s)$ at point $time(dr^{46})$. Thus, process q reads (x, p, s) from X in line 46 during its final iteration of the repeat-until loop of dr. Then by the loop-guard in line 50, q reads (p, s) from $A[q]$ in line 44, i.e., when it executes dr^{44}. This contradicts part (a) of this lemma. □

Lemma 8. *Let dr_1, dr_2 be two complete DRead operations in T by process q, with $dr_1 \xrightarrow{T} dr_2$, where dr_1 is the latest DRead operation performed by q that precedes dr_2. Suppose dr_2 returns (val, a) for some value val and $a \in \{True, False\}$. Then $a = True$ if and only if some $DWrite$ operation linearizes in the interval $\big(pt(dr_1), pt(dr_2)\big)$.*

Proof. First, suppose $a = True$. Then q executes line 48 during dr_2. Since the if condition on line 47 is satisfied if and only if the loop-guard on line 50 is false, q must execute line 48 on the first iteration of the repeat-until loop on line 42 during dr_2. Let (x, p, s) and (x', p', s') be the tuples returned by the $X.Read()$ operations xr and xr' performed during the first iteration of the loop on line 43 and line 46, respectively. Also let (r, s_r) be the pair returned by the $A[q].Read()$ operation ar performed during the first iteration of the loop on line 44. Since dr_1 and dr_2 are performed in sequence, xr, xr', and ar all happen after $pt(dr_1)$. Additionally, since $pt(dr_2) = time(dr_2^{46})$ (i.e. dr_2 linearizes at its *final* execution of line 46), and dr_2 performs the repeat-until loop on line 42 more than once,

$$xr,\ xr',\ \text{and } ar \text{ all happen in } \big(pt(dr_1), pt(dr_2)\big). \tag{3.7}$$

By the condition on line 47, there are two cases:

(i) Suppose $(x, p, s) \neq (x', p', s')$. Since xr' returns (x', p', s'), there is an $X.Write(x', p', s')$ operation that happens in $\big(time(xr), time(xr')\big)$. Hence, by (3.7) and Observation 5 there exists a $DWrite_{p'}(x')$ operation that linearizes in $\big(pt(dr_1), pt(dr_2)\big)$.

(ii) Suppose $(p, s) \neq (r, s_r)$. Assume that dr_1 writes (p_1, s_1) to $A[q]$ in its final call to line 45. By the loop-guard on line 50,

$$dr_1^{46} \text{ returns } (x_1, p_1, s_1). \tag{3.8}$$

Since q performs dr_2 immediately after dr_1, and q is the only process that can write to $A[q]$, $(r, s_r) = (p_1, s_1)$. But since $(p, s) \neq (r, s_r)$ by assumption, $(x, p, s) \neq (x_1, p_1, s_1)$. Due to (3.8), there must be an $X.Write(x, p, s)$ operation that happens in $\big(time(dr_1^{46}), time(xr)\big)$. By Observation 5, there exists a $DWrite$ operation that linearizes in $\big(time(dr_1^{46}), time(xr)\big)$, and by (3.7), this $DWrite$ linearizes in $\big(pt(dr_1), pt(dr_2)\big)$.

Now suppose $a = False$. Assume, for the sake of a contradiction, that some $DWrite$ linearizes in $\big(pt(dr_1), pt(dr_2)\big)$. Let dw be the $DWrite$ operation that linearizes at the latest time in this interval. Suppose dw is a $DWrite_{p_1}(x_1)$ by process p_1 with associated sequence number s_1. Due to Lemma 7 (b), we know that no $DWrite$ can linearize in the interval $\big[time(dr_2^{43}), time(dr_2^{46})\big]$. Thus,

$$dw \text{ linearizes in } \big(time(dr_1^{46}), time(dr_2^{43})\big). \tag{3.9}$$

Since dw is the final $DWrite$ operation that linearizes prior to dr_2^{46},

$$X = (x_1, p_1, s_1) \text{ throughout } \big[pt(dw), pt(dr_2)\big]. \tag{3.10}$$

If dr_2 performs the repeat-until loop on line 42 more than once, then since the loop-guard on line 75 is true if and only if the condition line 47 is true, q sets *changed* to $True$ during dr_2, which is a contradiction. Hence, dr_2 must perform only one iteration of the repeat-until loop. By (3.9) and (3.10), the $X.Read()$ operation dr_2^{43} must return (x_1, p_1, s_1). Also, q reads (p_1, s_1) from $A[q]$ during dr_2^{44}, as otherwise the loop would repeat by the loop-guard on line 75.

Hence, the last write to $A[q]$ prior to dr_2^{44} must have been an $A[q].Write(p_1, s_1)$ operation. Since q only performs one iteration of the repeat-until loop during dr_2, and thus does not write anything to $A[q]$ prior to dr_2^{44} while executing dr_2, q must have written (p_1, s_1) to $A[q]$ in the final iteration of the repeat-until loop of dr_1 (i.e. during dr_1^{45}). Since the repeat-until loop of dr_1 also terminated after that write, q must have read (p_1, s_1) from $A[q]$ during dr_1^{44}. This, along with (3.9), contradicts Lemma 7 (a). □

Lemma 9. *Let dr be a complete $DRead_p()$ operation that returns (val, a) for some $val \in D$, $val \neq \bot$, and some $a \in \{True, False\}$. Then*

(1) there is some $DWrite$ operation that linearizes prior to $pt(dr)$, and

(2) if dw is a $DWrite_q(x)$ operation, such that no $DWrite$ operation linearizes in the interval $\big(pt(dw), pt(dr)\big]$, then $x = val$.

Proof. We first prove (1). Since dr returns (val, a), the first component of X contains val at $time(dr^{46}) = pt(dr)$. Hence, some $X.Write(val, p, s)$ operation occurs before $pt(dr)$, for some process identifier p and some sequence number s. Then by Observation 5, there is some $DWrite_p(val)$ operation that linearizes prior to $pt(dr)$.

We now prove (2). Let dw be defined as in the statement of (2). Since dw linearizes when it writes x to X, and no $DWrite$ linearizes in the interval $\big(pt(dw), pt(dr)\big]$, the first component of X contains x throughout the interval $\big(pt(dw), pt(dr)\big]$. □

Theorem 10. *The sequential history $f(T)$ is a linearization of the interpreted history $\Gamma(T)$.*

Proof. First note that for any operation $op \in \Gamma(T)$, $pt(op) \in \big[inv(op), rsp(op)\big]$, since $pt(op)$ is assigned directly to a line of code that is executed by op for both $DWrite$ and $DRead$ operations. Thus, $f(T)$ preserves the happens-before order of the interpreted history $\Gamma(T)$.

Lemma 8 and Lemma 9 ensure that the history $f(T)$ is valid with respect to the sequential specification of ABA-detecting registers. Thus, $f(T)$ is a linearization of $\Gamma(T)$. □

Lemma 11. *The linearization function f is prefix-preserving.*

Proof. Consider each step t of T, and some operation $op \in \Gamma(T)$. Then $pt(op) = t$ if

(i) operation op is some $DWrite$ operation, and t is the step at which op executes line 11 (this case follows from Q-2), or

(ii) operation op is some $DRead$ operation, and t is the final step at which op executes line 46. Note that whether t is the final execution of line 46 is entirely determined at step t, since all of the values compared on line 50 are stored in local memory at t (this case follows from Q-2).

Thus, at step t it is entirely determined which operations op satisfy $pt(op) = t$. That is, whether t satisfies $t = pt(op)$ depends entirely on steps that occur at or before t, and not on steps that occur after t. Hence, if T is a prefix of $T' \in \mathcal{T}$, then $f(T)$ is a prefix of $f(T')$. □

Theorem 12. *The implementation represented by Algorithm 3 is strongly linearizable.*

Proof. Theorem 10 shows that the sequential history $f(T)$ is a linearization of the interpreted history $\Gamma(T)$. Furthermore, Theorem 11 shows that f is prefix-preserving. Thus, f is a strong linearization function for $\mathcal{T}$. □

3.3 Lock-Freedom and Complexity Analysis

It is easy to see that each $DWrite_q(x)$ operation performs only two shared memory steps (a read of $A[q]$ and a write to X). Hence, the implementation of the $DWrite$ method is wait-free. However, a $DRead_q()$ operation by a process q may not terminate if it is "interrupted" by infinitely many $DWrite$ operations. But note that in each iteration of the repeat-until loop process q writes the same pair (p, s) to $A[q]$ in line 45 that it previously read in line 43 from the second and third component of X. Hence, if q executes sufficiently many steps while X does not change, then q will eventually read the same pair from X in line 43, from $A[q]$ in line 44, and from X again in line 46. After that, q's $DRead$ terminates. Thus, in any transcript in which q takes sufficiently many steps, either its $DRead$ terminates, or a

$DWrite$ terminates. We proceed by providing a precise analysis of the amortized complexity of Algorithm 3.

Lemma 13. *Let dr be some $DRead_q()$ operation by process q. Let xr_1, xr_2, and xr_3 be three consecutive $X.Read()$ operations on line 43 by dr. Then there exists a $DWrite$ operation that linearizes in the interval $\big(time(xr_1), time(xr_3)\big)$.*

Proof. If the values returned by xr_1 and xr_2 are not equal, then some $X.Write$ operation occurs in the interval $\big(time(xr_1), time(xr_2)\big)$. Then by Observation 5, a $DWrite$ operation linearizes in $\big(time(xr_1), time(xr_2)\big)$.

Now suppose xr_1 and xr_2 both return the same tuple (x, p, s). Then dr performs an $A[q].Write(p, s)$ operation on line 45 after xr_1, and $A[q]$ is not modified again before line 44 is performed by dr following xr_2. Hence, the $A[q].Read()$ operation on line 44 performed by dr following xr_2 returns (p, s). If the $X.Read()$ operation on line 46 performed by dr following xr_2 returns (x, p, s), then by the loop-guard on line 50, dr terminates after this iteration of the main loop. This is a contradiction, since dr must restart the repeat-until loop after xr_2 in order to perform xr_3. Therefore, the $X.Read()$ operation on line 46 performed by dr following xr_2 returns $(x', p', s') \neq (x, p, s)$. Then some $X.Write(x', p', s')$ must occur after $time(xr_2)$ and before the following execution of line 46 by dr, and hence in the interval $\big(time(xr_1), time(xr_3)\big)$. By Observation 5, there is a $DWrite_{p'}(x')$ operation that linearizes in this interval. □

Theorem 14. *(a) Each $DWrite()$ performs at most two shared memory operations; and*

(b) for any transcript that contains r $DRead$ and w $DWrite$ invocations, the total number of steps devoted to $DRead$ operations is $O(\min(r, n) \cdot w + r)$.

In particular, the implementation is lock-free and has amortized step complexity $O(n)$.

Proof. Part (a) follows immediately from the pseudocode (Algorithm 3).

By Lemma 13, each process reads X on line 43 at most $3w + 1$ times during a single $DRead$ operation. This immediately shows that the total number of steps devoted to $DRead$

operations is $O(r \cdot (w+1))$. This proves part (b) for the case $r \leq n$.

We now consider the case $r > n$. For any process let r_p denote the number of *DRead* invocations by process p. Further, for $i \in \{1, \ldots, r_p\}$ let $k_{p,i}$ denote the total number of times process p reads X in line 43 during its i-th *DRead* operation. From Lemma 13 we obtain $\sum_i k_{p,i} = O(w + r_p)$ for each process p. Using $r = \sum_p r_p$ we obtain that the total number of times all processes read X during all *DRead* operations is

$$\sum_p \sum_{i=1}^{r_p} k_{p,i} = \sum_p O(w + r_p) = O(n \cdot w + r)$$

This proves part (b) for the case $r > n$. □

Theorems 12 and 14 yield Theorem 1.

3.4 Remarks

In this chapter, we presented the first strongly linearizable implementation of an ABA-detecting register by modifying a previous linearizable implementation by Aghazadeh and Woelfel [3, 26]. An obvious extension of this work would examine the possibility of a wait-free strongly linearizable implementation of an ABA-detecting register. However, we strongly suspect that such an implementation is impossible. Denysyuk and Woelfel showed that there is no wait-free strongly linearizable implementation of a counter (defined in Chapter 1) from registers, and by reduction this implies that no such implementations exist for snapshots or max-registers, either [2]. The authors first assume there exists a wait-free implementation of a counter, then define a history in which a *Read* operation takes an infinite number of steps, but the return value of the *Read* is never determined; this contradicts the wait-freedom of the implementation. We believe a similar argument could be applied to the ABA-detecting register. For this thesis, our efforts were concentrated on the snapshot implementation; the strongly-linearizable ABA-detecting register was incidental. The analysis we performed on our snapshot implementation is not affected by the fact that the ABA-detecting register

implementation is only lock-free (as opposed to wait-free). Hence, we did not dedicate much time or effort to designing a wait-free strongly linearizable ABA-detecting register (or proving the impossibility of such an implementation). However, we would like to study this issue further. In fact, it would be interesting to know if *any* nontrivial type has a wait-free strongly linearizable implementation from registers.

4. A Strongly Linearizable Snapshot

A *(single-writer) snapshot* [14] is a type that provides the invocation descriptions $update_q(x)$ and $scan()$. A snapshot object has, for each process $p \in \{1, \ldots, n\}$, an *entry* that stores a value from some finite domain D. That is, the snapshot object contains a vector $X \in D^n$, which is initially $(\bot, \ldots, \bot)$. The $update_p(x)$ invocation, for any value $x \in D$ such that $x \neq \bot$, changes the p-th entry of X to x, and the $scan()$ invocation returns the vector X. We emphasize that the value $\bot$ strictly signifies the initial state of each entry; that is, once an entry of the snapshot contains a value $x \neq \bot$, no process may change the value of this entry back to $\bot$.

We use brackets to denote individual entries in vectors and snapshot objects. More precisely, if $X = (x_1, \ldots, x_k)$, then if $p \in \{1, \ldots, k\}$, $X[p] = x_p$. Additionally, if O is a snapshot object, then $O[p]$ denotes the p-th entry of the object (i.e. the entry that is writeable by process p).

4.1 Unbounded Implementation

Denysyuk and Woelfel [2] define a general lock-free construction for *versioned* objects, each storing a version number. A versioned object has an atomic update operation, which increases the object's version number every time it is invoked. A versioned object also supports a read operation which returns the state of the object along with its version number. The simple lock-free linearizable algorithm based on clean double collects from [14] can be transformed into a versioned snapshot object easily, by adding a sequence number field to each component that is incremented with each update of the component. The version number of the entire object may be obtained by calculating the sum of the sequence numbers of every component.

The strongly linearizable construction of a versioned object also uses a strongly linearizable bounded max-register described by Helmi, Higham, and Woelfel [12]. Denysyuk and

Woelfel augment the max-register implementation such that it stores a pair (x, y). The augmented max-register supports a $maxRead()$ invocation, which returns the pair (x, y), and a $maxWrite(x', y')$ invocation, which replaces the stored pair (x, y) with (x', y') provided that $x' > x$.

Denysyuk and Woelfel's construction uses a single instance of a versioned object S of type $\mathscr{T}$, along with a single instance of an augmented max-register R. A strongly linearizable object S' of type $\mathscr{T}$ is obtained as follows: to perform an $S'.update(x)$ operation, a process executes an $S.update(x)$ operation, reads S to obtain the pair (y, v), and finally performs an $R.maxWrite(v, y)$ operation (note that v represents the version number of the object). To execute an $S'.read()$ operation, a process performs an $R.maxRead()$ operation to obtain the pair (v, y), and returns the value y.

The fact that this algorithm is strongly linearizable follows from a simple argument. Let *up* be some $S'.update(x)$ operation. Suppose that S' has version number v immediately after *up* performs its $S.update(x)$ operation. Then *up* may be linearized as soon as some $R.maxWrite(v', y)$ operation, with $v' \geq v$, linearizes. If multiple $S'.update$ operations linearize at the same step, then these operations may be ordered by the times at which their atomic $S.update$ operations responded. An $S'.read()$ operation may be linearized as soon as its $R.maxRead()$ operation linearizes. Since each operation on S' can be linearized at the same step as an operation on R, then S' is strongly linearizable because R is strongly linearizable.

As mentioned previously, the construction uses an implementation of a *bounded* max-register, which means that the version number of the strongly linearizable object is bounded. Hence, without further modification, the algorithm only supports a finite number of update operations on the constructed object. The bounded max-register implementation from [12] is included here as Algorithm 4 for reference.

The bounded max-register can store a value up to some constant B. The implementation uses a shared array R of $B + 2$ registers of size $\log |D|$. A $maxWrite(x)$ operation, for some

Algorithm 4: A strongly linearizable bounded max-register [12]

```
   shared:
     register R[0 ... B + 1] = [0, ..., 0]

   local:
     Integer t = 0

   Function maxWrite(x):
52   for i = 1 ... x do
53     R[i] = x
54   end

   Function maxRead():
55   repeat
56     maxWrite(t)
57     r ← R[t + 1]
58     if r = 0 then
59       return t
60     end
61     else
62       t ← r
63     end
64   until true
```

$x \in D$, writes the value x to the registers $R[1], \ldots, R[x]$ in the loop on line 52. A process p performing a *maxRead*() repeatedly helps pending *maxWrite* operations by performing a $maxWrite(t)$ operation on line 56, where t is an integer that is local to each process. Following this, p checks the contents of register $R[t+1]$; if the register contains 0, then the *maxRead*() operation immediately terminates and returns t on line 59. Otherwise, p replaces the value t with the value observed in $R[t+1]$ on line 62, before restarting the *maxRead*() operation.

Notice that for each process p, t stores the largest value in R that has been observed by p. For any $i \in \{1, \ldots, B+1\}$, $R[i]$ may only contain a value $v > 0$ if some $maxWrite(v)$ operation writes to $R[i]$ on line 53. Hence, when a *maxRead*() operation helps pending calls to *maxWrite* on line 56, it is guaranteed that a $maxWrite(t)$ operation was invoked at some prior point in time. A $maxWrite(x)$ operation may be linearized at the first step at which

some $R[x].Write(x')$ operation occurs (for some value $x' \geq x$). If $R[x] \geq x$ at the invocation of some $maxWrite(x)$ operation, then it may be linearized immediately as it is invoked. A $maxRead()$ operation may be linearized as soon as it reads 0 from $R[t+1]$ on line 57. This selection of linearization points results in a prefix-preserving function because each of these points depends only on past events (that is, at any particular step of a transcript on Algorithm 4 it is entirely determined which operations linearize at that step).

The implementation of $maxWrite$ is wait-free because each $maxWrite(x)$ operation performs x iterations of the loop on line 52, and $x \leq B$ for some constant B. Similarly, the implementation of $maxRead$ is wait-free because each $maxRead()$ operation performs at most B iterations of the loop on line 55 (this follows from a simple argument based on the fact that t increases during every execution of line 62), and every operation invoked during the repeat-until loop is wait-free.

Algorithm 4 may be transformed into an unbounded implementation of a max-register with a simple modification: let R be an unbounded array of (unbounded) registers, and leave the implementations of $maxWrite$ and $maxRead$ unchanged. The $maxWrite$ implementation remains wait-free, since each $maxWrite(x)$ operation still only performs x iterations of the loop on line 52. However, a $maxRead()$ operation might never terminate if it is interrupted by infinitely many $maxWrite$ operations that write increasing values. Hence, the modification is only lock-free. However, the modified max-register may still be used in the lock-free construction of Denysyuk and Woelfel. Notice that the implementation of $S'.update$ remains wait-free, and each $S'.update$ operation invokes only a single $maxWrite$ operation. Hence, if a $maxRead()$ performed during an $S'.read()$ operation rd never terminates, the $maxRead()$ operation must continually read increasing values on line 57. This implies that infinitely many $maxWrite$ operations, which write increasing values, must terminate during rd. Since the contents of the max-register may only be increased by calls to $maxWrite$ performed during $S'.update$ operations, this implies that an infinite number of $S'.update$ operations terminate during rd. The outline of this argument is similar to the formal proofs of correctness provided

in Section 4.4.

4.2 Interpreted Value

Suppose O is an instance of a snapshot object, and let T be a transcript that contains operations on O. If O is atomic, then it is easy to determine the value of O at any step t of T, since *update* operations on O take effect instantaneously. However, if O is a linearizable implementation of a snapshot object (in particular, if *update* operations on O are non-atomic), then the value of O at any step t of T is not well-defined. To address this issue, we begin by introducing the notion of *interpreted value*, which allows us to reason about the contents of a linearizable object O at every step of T.

Let T be a fixed transcript, and let O be some linearizable snapshot object. Let pt be a linearization point function for O (recall that for any transcript T, pt maps operations in $\Gamma(T|O)$ to points in time in T). The *interpreted value* of $O[p]$ induced by pt at time t of T is x if and only if one of the following statements hold:

T-1 There exists an $O.update_p(x)$ operation $up \in \Gamma(T|O)$ by p such that $pt(up) < t$, and there does not exist any $O.update_p(x')$ operation $up' \in \Gamma(T|O)$ by p such that $pt(up) < pt(up') \leq t$, for any $x' \neq x$.

T-2 There is no such $O.update_p(x)$ operation by p in $\Gamma(T|O)$, and $x = \bot$.

When pt and T are clear from context, we say that the interpreted value of $O[p]$ at time t is x.

Intuitively, if the interpreted value of $O[p]$ at time t is x, then any *scan* operation $sc \in \Gamma(T)$ such that $pt(sc) = t$ must return a vector with x in its p-th entry (this simply follows from the sequential specification of the snapshot type, along with the assumption that S is a linearization of $\Gamma(T)$). Note that by the definition above, if up_1 and up_2 are two consecutive $O.update$ operations by process p such that up_1 and up_2 write distinct values and $pt(up_2) \neq \infty$, then the interpreted value of $O[p]$ at $pt(up_2)$ is $\bot$. However, if up_1 and up_2 both write the

same value v, then the interpreted value of $O[p]$ at $pt(up_2)$ is v.

Observation 15. *Let T be a transcript, let O be a linearizable snapshot object, and let pt be a linearization point function for O. Suppose the interpreted value of $O[p]$ induced by pt at time $t \neq \infty$ of T is $x \neq \bot$. Then for every scan operation sc such that $pt(sc) = t$, sc returns a vector $(x_1, \ldots, x_n)$ such that $x_p = x$.*

Proof. Since the interpreted value of $O[p]$ at time t is $x \neq \bot$, by T-1 there exists an $O.update_p(x)$ operation $up \in \Gamma(T|O)$ by p such that

$$pt(up) < t, \text{ and} \tag{4.1}$$

$$\begin{aligned}&\text{there is no } O.update_p(x') \text{ operation } up' \in \Gamma(T|O) \text{ by } p \text{ such that}\\ &pt(up) < pt(up') \leq t, \text{ for any } x' \neq x.\end{aligned} \tag{4.2}$$

By the definition of linearization point functions, there exists a linearization S of $\Gamma(T|O)$ such that,

$$\text{for any } op \in \Gamma(T|O) \text{ such that } pt(op) \neq \infty,\ op \in S, \text{ and} \tag{4.3}$$

$$\text{for any } op_1, op_2 \in S, \text{ if } op_1 \xrightarrow{S} op_2 \text{ then } pt(op_1) \leq pt(op_2). \tag{4.4}$$

From the lemma statement, $pt(sc) = t$ and $t \neq \infty$. By this, (4.1), and (4.3), $up, sc \in S$. Since $pt(up) < pt(sc)$, $up \xrightarrow{S} sc$ by contrapositive of (4.4).

Suppose there exists an $O.update_p(x')$ operation $up' \in S$ by p such that

$$x \neq x', \text{ and} \tag{4.5}$$

$$up \xrightarrow{S} up' \xrightarrow{S} sc. \tag{4.6}$$

By (4.4) and (4.6),

$$pt(up) \leq pt(up'), \text{ and} \tag{4.7}$$

$$pt(up') \leq pt(sc). \tag{4.8}$$

Note that by definition of linearization point functions,

$$pt(up) \in \big[time_T(inv(up)), time_T(rsp(up))\big], \text{ and} \tag{4.9}$$

$$pt(up') \in \big[time_T(inv(up')), time_T(rsp(up'))\big]. \tag{4.10}$$

By (4.9), (4.10), the fact that processes perform operations sequentially, and (4.7),

$$pt(up) < pt(up'). \tag{4.11}$$

Together, (4.5), (4.8), and (4.11) contradict (4.2). Hence, there is no $O.update_p(x')$ operation $up' \in S$ by p such that $up \xrightarrow{S} up' \xrightarrow{S} sc$, for any $x' \neq x$. By this, the fact that $up \xrightarrow{S} sc$, and the fact that S is a linearization of $\Gamma(T|O)$, the sequential specification of the snapshot type requires that sc returns a vector $(x_1, \ldots, x_n)$ with $x_p = x$. $\square$

Observation 16. *Let T be a transcript, let O be a linearizable snapshot object, and let pt be a linearization point function for O. Suppose that up is some $O.update_p(x)$ operation by p such that $pt(up) \neq \infty$. If there is no $O.update_p(x')$ operation up' by p with $x' \neq x$ such that $pt(up') \in \big(pt(up), t\big]$ for some time $t > pt(up)$, then the interpreted value of $O[p]$ is x throughout $\big(pt(up), t\big]$.*

Proof. This follows trivially from T-1. $\square$

4.3 Bounded Implementation

Golab, Higham, and Woelfel [1] have shown that the wait-free snapshot implementation designed by Afek, Attiya, Dolev, Gafni, Merritt, and Shavit [14] is not strongly linearizable. Previous strongly linearizable implementations of snapshot objects are either not lock-free [12], or use an unbounded number of registers [2]. We design a strongly linearizable implementation of a single-writer snapshot object using a linearizable instance of a single-writer snapshot object, along with an atomic ABA-detecting register.

The linearizable snapshot object used by our implementation can be any lock-free or wait-free linearizable implementation of a snapshot object. To achieve a strongly linearizable snapshot object that uses bounded space, we must ensure that the underlying linearizable snapshot implementation also uses only bounded space. For the sake of concreteness, we use an implementation by Attiya and Rachman [22], which is wait-free and linearizable. The bounded implementation presented in [22] uses $O(n^3)$ registers of size $O(\log n + \log |D|)$ to represent views of the snapshot object (where D is the set of values that may be stored by a component), plus $O(n^5)$ registers of size $O(\log n)$ to manage sequence numbers. This implementation therefore has space complexity $O(n^3(\log n+\log |D|)+n^5 \log n)$. The algorithm performs $O(n \log n)$ operations on MRSW registers during any *scan* or *update* operation. Let S be an instance of this bounded wait-free linearizable snapshot object implementation. Our algorithm also uses a shared atomic ABA-detecting register R. The snapshot object S is used to hold the contents of the strongly linearizable snapshot object, while the ABA-detecting register R contains a vector of size n that represents the state of S at some previous point in time. Since strong linearizability is a composable property [1, 7], we can replace the atomic ABA-detecting register R with our strongly linearizable one from Chapter 3.

In order to clearly distinguish between operations on the linearizable snapshot object S, and the implemented strongly linearizable snapshot, we call the operations on the latter one *SLupdate* and *SLscan*. Pseudocode for our implementation is presented in Algorithm 5.

Algorithm 5: A strongly linearizable snapshot object

```
shared:
   linearizable snapshot object S = (⊥, . . . , ⊥)
   atomic ABA-detecting register R = (⊥, . . . , ⊥)

Function SLupdate_p(x):
65 | S.update_p(x)
66 | s ← S.scan()
67 | R.DWrite_p(s)

Function SLscan_p():
68 | repeat
69 | | (s1, c1) ← R.DRead_p()
70 | | ℓ ← S.scan()
71 | | (s2, c2) ← R.DRead_p()
72 | | if !(s1 = ℓ = s2) then
73 | | | R.DWrite_p(ℓ)
74 | | end
75 | until (s1 = ℓ = s2) and !c2
76 | return s2
```

We employ a similar strategy as in the unbounded implementation, but the role of the max-register is now filled by the ABA-detecting register. The *SLupdate* operation is nearly identical to the update operation of the unbounded implementation: to perform an $SLupdate_p(x)$ operation, for some $x \in D$, a process p first performs an $S.update_p(x)$ operation on line 65, changing the contents of the p-th entry of S to x. Process p then performs an $S.scan()$ operation on line 66, and finally writes the result of this call to R on line 67. Since components of the snapshot object are single-writer, the vector returned by this $S.scan()$ operation must contain x in its p-th entry.

An $SLscan_p()$ operation, for some process p, is "stretched" until a period of time is observed during which the underlying objects S and R are not modified. This way, we force the operation to observe as many *SLupdate* operations as possible before allowing it to respond. The method works by repeatedly performing an $R.DRead_p()$ operation on line 69, then an $S.scan()$ operation on line 70, and finally another $R.DRead_p()$ operation on line 71. We will often refer to this sequence of operations as the *main loop* of the *SLscan* method.

Process p continues to perform this sequence of operations until the same vector is returned by all of these calls. Whenever p observes that the contents of R and S are inconsistent, p helps pending *SLupdate* operations by writing the previously-taken snapshot of S to R on line 73 before resuming its main loop. When p observes that the $S.scan()$ operation and the two $R.DRead_p()$ operations return the same vector, p will make sure that R was not changed between its most recent pair of $R.DRead_p()$ operations by checking the boolean flag returned by the second *DRead* on line 75; if this flag is false, then p can safely return the value that was returned by its final *DRead*. Thus, process p continues to perform its main loop until it observes that no process interferes during its most recently executed sequence of read operations.

Both *SLscan* and *SLupdate* operations work to stabilize the contents of S and R. The idea is that the underlying snapshot object S always contains the most recent state of the object, and operations write the state of S that they observed most recently to R (on both line 67 for *SLupdate* operations and line 73 for *SLscan* operations). A pending $SLupdate_p(x)$ operation by process p can be linearized as soon as some concurrent *SLscan* operation returns a vector that contains x in its p-th entry. The key is that we choose a linearization point for this *SLupdate* operation at which the interpreted value of $S[p]$ is x and the value of $R[p]$ is x. We choose to linearize *SLscan* operations on their last read of shared memory. That is, an *SLscan* operation linearizes at its final execution of the $R.DRead$ operation on line 71.

Our choice of linearization points results in a strong linearization function because *SLscan* operations always linearize at their final shared memory step, and when an *SLscan* operation linearizes, it is already determined which *SLupdate* operations are caused to linearize by this *SLscan* operation. Hence, no operations can be retroactively inserted anywhere in the established linearization order.

We now give an informal explanation of why the ABA-detecting register and the conditional statement on line 75 are helpful in Algorithm 5. The snapshot object implementation in Algorithm 6 is a modification of Algorithm 5, which replaces the ABA-detecting register

with an atomic multi-reader multi-writer register. The remaining details of the algorithm are nearly unchanged, besides method names (e.g. *DRead* is now simply *Read*). Notice that the Boolean flag test of c_2 from line 75 has been removed from line 87 (since the $R.Read()$ operation does not return a Boolean flag).

Algorithm 6: A snapshot object implemented without ABA-detecting registers

```
    shared:
      linearizable snapshot object S = (⊥, ..., ⊥)
      atomic multi-reader multi-writer register R = (⊥, ..., ⊥)

    Function SLupdate_p(x):
77  |  S.update_p(x)
78  |  s ← S.scan()
79  |  R.Write(s)

    Function SLscan_p():
80  |  repeat
81  |  |  s_1 ← R.Read()
82  |  |  ℓ ← S.scan()
83  |  |  s_2 ← R.Read()
84  |  |  if !(s_1 = ℓ = s_2) then
85  |  |  |  R.Write(ℓ)
86  |  |  end
87  |  until s_1 = ℓ = s_2
88  |  return s_2
```

Consider the following programs for the processes q, p, and r:

- Process q performs an $SLupdate_q(1)$ operation up_1 followed by an $SLupdate_q(2)$ operation up_2.
- Process p performs an $SLscan_p()$ operation sc_p.
- Process r performs an $SLscan_r()$ operation sc_r.

Define the following transcripts on Algorithm 6:

$$S = up_1 \circ (sc_r \text{ to the end of line 82}) \circ$$

$$(sc_p \text{ to the end of line 82}) \circ up_2 \circ (sc_r \text{ to the end of line 84})$$

$$T_1 = S \circ (sc_r \text{ to the end of line 85}) \circ (sc_p \text{ to completion})$$

$$T_2 = S \circ (sc_p \text{ to completion})$$

The $SLupdate_q(1)$ operation up_1 completes before any other operation is invoked in S. Therefore, when sc_p and sc_r run to the end of line 82 during S, they both see 1 in entry q of the vectors returned on lines 81 and 82 (we disregard the entries for p and r in this example, since neither process performs any *SLupdate* operations). The second time sc_r runs during S, to the end of line 84, it sees 2 in entry q of the vector returned on line 83 (since up_2 completed before the $R.Read()$ operation by sc_r on line 83 was invoked). Then sc_r must enter the conditional block on line 84. Hence, at the end of S, r is poised perform an $R.Write(X)$ operation on line 85, where $X[q] = 1$.

In transcript T_1, sc_r is allowed to complete its $R.Write(X)$ operation. Hence, when sc_p completes in transcript T_1, it reads a vector with 1 in entry q in line 83. Therefore, sc_p does not enter the conditional block on line 84, since the vectors it viewed on lines 81, 82, and 83 were all equal. Then sc_p returns a vector that contains 1 in entry q. Hence, sc_p must linearize after up_1, but before up_2 in any linearization of $\Gamma(T_1)$.

In transcript T_2, sc_p completes without sc_r performing its $R.Write(X)$ operation. In this case, since up_2 performs an $R.Write(Y)$ operation on line 79, with $Y[q] = 2$, sc_p reads a vector with 2 in entry q on line 83. Hence, sc_p restarts its loop, reads a vector with 2 in entry q in lines 81, 82, and 83, and then terminates and returns a vector with 2 in entry q. Therefore, sc_p must linearize after both up_1 and up_2 in any linearization of $\Gamma(T_2)$.

Consider the linearization $L_1 = up_1 \circ sc_p \circ up_2$ of S. Then L_1 is not a prefix of any linearization of $\Gamma(T_2)$, since sc_p must linearize after both up_1 and up_2 in any linearization of

$\Gamma(T_2)$. Now consider the linearization $L_2 = up_1 \circ up_2$ of S. Then L_2 is not a prefix of any linearization of $\Gamma(T_1)$, since sc_p must linearize after up_1 but before up_2 in any linearization of $\Gamma(T_1)$. This example provides intuition into why Algorithm 6 is not strongly linearizable. In the remainder of this section we show that Algorithm 5 is strongly linearizable, demonstrating that using ABA-detecting registers suffices to solve the problem we illustrated with this example.

For any operation op in a transcript T on some instance of the implementation provided in Algorithm 5, we define $pt(op)$ as follows:

R-1 Suppose op is some $SLscan$ operation. Then $pt(op)$ is the time at which the final shared memory step is performed by op. That is, $pt(op) = time(op^{71})$.

R-2 Suppose op is some $SLupdate_p(x)$ operation for some $x \in D$. If T contains an $SLscan$ operation sc such that $pt(sc) \neq \infty$, $time(inv(op)) < pt(sc)$, and sc returns a vector whose p-th entry contains x, then let sc_0 be such an $SLscan$ operation with minimum possible pt value. Let $t = pt(sc_0)$ if sc_0 exists, or $t = \infty$ otherwise. Then $pt(op) = \min\big(t, time(op^{67})\big)$. Recall that if $op^{67} \notin T$, then $time(op^{67}) = \infty$.

If $pt(op) \neq \infty$ for some operation op, then we say *op linearizes* at $pt(op)$. Let $\mathcal{T}$ represent the set of all possible transcripts on some instance of the object implementation from Algorithm 5. For every transcript $T \in \mathcal{T}$ define a sequential history $f(T)$, such that for every pair of distinct operations $op_1, op_2 \in \Gamma(T)$ by processes p_1 and p_2 respectively, with $pt(op_1) \neq \infty$ and $pt(op_2) \neq \infty$, $op_1 \xrightarrow{f(T)} op_2$ if and only if

U-1 operation op_1 linearizes before op_2 (i.e. $pt(op_1) < pt(op_2)$), or

U-2 operations op_1 and op_2 have the same linearization point (i.e. $pt(op_1) = pt(op_2)$), they both have the same invocation description (i.e. they are either both $SLupdate$ operations or both $SLscan$ operations), and $p_1 < p_2$, or

U-3 operations op_1 and op_2 have the same linearization point (i.e. $pt(op_1) = pt(op_2)$), op_1 is an $SLupdate$ operation, and op_2 is an $SLscan$ operation.

Note that $f(T)$ does not contain any operation $op \in \Gamma(T)$ for which $pt(op) = \infty$.

For the remainder of Chapter 4, let $T \in \mathcal{T}$ be some finite transcript on a snapshot object O implemented by Algorithm 5. Additionally, fix a linearization point function pt_S for S, where S is the linearizable snapshot object used by Algorithm 5. By definition of linearization point functions, there exists a linearization L of $\Gamma(T|S)$ such that

$$\text{for every } op \in \Gamma(T|S) \text{ such that } pt_S(op) \neq \infty,\ op \in L, \text{ and} \tag{4.12}$$

$$\text{for every } op_1, op_2 \in L, \text{ if } op_1 \xrightarrow{L} op_2 \text{ then } pt_S(op_1) \leq pt_S(op_2). \tag{4.13}$$

Lemma 17. *For any operation $op \in \Gamma(T)$, $pt(op) \in \big[time(inv(op)), time(rsp(op))\big]$.*

Proof. Let $op \in \Gamma(T)$ be some $SLscan_p()$ operation by p. Then $pt(op) = time(op^{71})$ by R-1. Hence, $pt(op) \in \big[time(inv(op)), time(rsp(op))\big]$.

Let $op \in \Gamma(T)$ be some $SLupdate_p(x_p)$ operation by p for some value $x_p \in D$. By R-2, $pt(op)$ is explicitly defined as a time after $time(inv(op))$ and not after $time(rsp(op))$, and therefore $pt(op) \in \big[time(inv(op)), time(rsp(op))\big]$. □

Lemma 18. *Suppose $up \in \Gamma(T)$ is some $SLupdate_p(x)$ operation by p, for some value $x \in D$. If there exists an $SLscan()$ operation $sc \in \Gamma(T)$ such that $time(inv(up)) < pt(sc)$ and sc returns some vector with x in its p-th entry, then $pt(up) \leq pt(sc)$.*

Proof. This is trivially true if $pt(sc) = \infty$. Otherwise, the lemma follows from R-2. □

Lemma 19. *Let $op \in \Gamma(T)$ be some complete $SLscan_p()$ operation by some process p. Then no $R.DWrite$ operation happens in the interval $\big[time(op^{69}), time(op^{71})\big]$.*

Proof. This follows directly from the sequential specification of ABA-detecting registers and the if-statement on line 75. □

Lemma 20. *Let $up \in \Gamma(T)$ be some $SLupdate_p(x)$ operation with $pt(up) \neq \infty$, for some process p and some $x \in D$. Then the interpreted value of $S[p]$ induced by pt_S is x and $R[p] = x$ at $pt(up)$.*

Proof. There are two cases:

(i) Operation up linearizes at $time(up^{67})$ (that is, $pt(up) = time(up^{67})$). Let up_u be the $S.update_p(x)$ operation performed by up on line 65. Since entries of S are single-writer, and the $SLupdate$ method only contains a single $S.update$ call,

$$\text{the interpreted value of } S[p] \text{ is } x \text{ throughout the interval } \big(pt_S(up_u), time(rsp(up))\big]. \tag{4.14}$$

By definition of linearization point functions,

$$pt_S(up_u) \in \big[time(inv(up_u)), time(rsp(up_u))\big]. \tag{4.15}$$

By (4.15) and the fact that processes perform operations sequentially, $pt_S(up_u) < time(up^{67})$. By this and (4.14),

$$\text{the interpreted value of } S[p] \text{ is } x \text{ at } time(up^{67}). \tag{4.16}$$

Let sc_u be the $S.scan()$ operation performed by up on line 66. Again, since processes perform operations sequentially and $pt_S(up_u)$ and $pt_S(sc_u)$ both occur between the invocations and responses of up_u and sc_u, respectively, $pt_S(up_u) < pt_S(sc_u)$. By this, (4.14), and Observation 15, sc_u returns a vector whose p-th entry contains x. Then the vector written to R by up on line 67 contains x in its p-th entry. Hence,

$$R[p] = x \text{ at } time(up^{67}). \tag{4.17}$$

By (4.16), (4.17), and the assumption that $pt(up) = time(up^{67})$, we obtain the claim in the lemma statement.

(ii) Operation up linearizes before $time(up^{67})$ (that is, $pt(up) < time(up^{67})$). Then by R-2 there is some $SLscan$ operation sc that returns a vector whose p-th entry contains x

while up is pending, and $pt(sc) = pt(up)$. By this and the fact that $pt(sc) = time(sc^{71})$ by R-1,

$$time(inv(up)) < time(sc^{71}). \tag{4.18}$$

Since sc linearizes at its final execution of line 71, the vectors returned by the $R.DRead()$ operations on line 69 and line 71, along with the vector returned by the $S.scan()$ operation on line 70, must satisfy the condition on line 75. In particular, if sc_s is the final $S.scan()$ operation performed by sc, then

$$sc_s \text{ returns a vector with } x \text{ in its } p\text{-th entry.} \tag{4.19}$$

Since L is a linearization of $\Gamma(T|S)$, by (4.19) and the sequential specification of the snapshot type, there exists an $S.update_p(x)$ operation up_{u_x} by p such that

$$up_{u_x} \xrightarrow{L} sc_s, \text{ and} \tag{4.20}$$

$$\begin{array}{l}\text{there is no } S.update_p(x') \text{ operation } up'_u \text{ by } p \text{ such that } x \neq x' \\ \text{and } up_{u_x} \xrightarrow{L} up'_u \xrightarrow{L} sc_s.\end{array} \tag{4.21}$$

Applying (4.13) to (4.20),

$$pt_S(up_{u_x}) \leq pt_S(sc_s). \tag{4.22}$$

Suppose that up_{u_x} is performed by some $SLupdate_p(x)$ operation other than up. There cannot exist an $S.update_p(x')$ operation up'_u by p such that $x' \neq x$ and $pt_S(up_{u_x}) < pt_S(up'_u) < pt_S(sc_s)$, since, using the contrapositive of (4.13), this would contradict (4.21). Also, there cannot exist such an up'_u by p such that $pt_S(up_{u_x}) = pt_S(up'_u)$ by Lemma 17, since up_{u_x} is also by process p and processes perform operations sequentially. Now suppose there exists an $S.update_p(x')$ operation up'_u by p such that $x' \neq x$ and

$$pt_S(sc_s) \leq pt_S(up'_u) < time(inv(up)). \tag{4.23}$$

Let up' be the $SLupdate_p(x')$ operation that performs up'_u on line 65. Clearly, $up' \neq up$, since $x \neq x'$. Since processes perform operations sequentially, $time(rsp(up')) < time(inv(up))$. But by this, (4.18), and (4.23), up' must perform its $R.DWrite$ operation from line 67 in the interval $\big[pt_S(sc_s), time(sc^{71})\big)$, and hence in the interval $\big[time(sc^{69}), time(sc^{71})\big]$. This contradicts Lemma 19, and therefore

$$\text{no } S.update_p(x') \text{ operation } up'_u \text{ by } p \text{ such that } x' \neq x \text{ satisfies} \\ pt_S(up'_u) \in \big[pt_S(up_{u_x}), time(inv(up))\big). \tag{4.24}$$

Since up only performs a single $S.update_p(x)$ operation, and up does not perform any $S.update_p(x')$ operations for any $x' \neq x$, by (4.24) and Observation 16,

$$\text{the interpreted value of } S[p] \text{ is } x \text{ throughout } \big(pt_S(up_{u_x}), time(rsp(up))\big]. \tag{4.25}$$

If up_{u_x} is performed by p during up, then (4.25) is implied by the fact that up only performs a single $S.update$ operation, along with Observation 16.
Since processes perform operations sequentially, $pt_S(sc_s) < time(sc^{71})$. Thus, $pt_S(sc_s) < pt(up)$. By this, (4.22), and (4.25)

$$\text{the interpreted value of } S[p] \text{ is } x \text{ at } pt(up). \tag{4.26}$$

Since sc linearizes at the time of its final $R.DRead()$ operation on line 71, and sc returns a vector with x in its p-th entry,

$$R[p] = x \text{ at } pt(sc) = pt(up). \tag{4.27}$$

By (4.26) and (4.27) we obtain the claim in the lemma statement.

$\square$

Lemma 21. *Let $sc \in \Gamma(T)$ be some $SLscan_q()$ operation by process q such that $pt(sc) \neq \infty$, which returns $(x_1, x_2, \ldots, x_n)$. Then $x_p \neq \bot$ if and only if there exists some $SLupdate_p(x_p)$ operation $up \in \Gamma(T)$, for some $x_p \in D$, such that $pt(up) \leq pt(sc)$.*

Proof. Let sc_s be the final $S.scan()$ operation performed by sc. Since sc returns $(x_1, \ldots, x_n)$,

$$sc_s \text{ returns } (x_1, \ldots, x_n), \text{ and} \tag{4.28}$$

$$R[p] = x_p \text{ at } time(sc^{71}) = pt(sc). \tag{4.29}$$

Additionally, since $pt(sc) \neq \infty$ by the lemma assumption, and $pt(sc) = time(sc^{71})$ by R-1, sc_s is complete in T. By this and the fact that $pt_S(sc_s) \in \big[time(inv(sc_s)), time(rsp(sc_s))\big]$ by definition of linearization point functions,

$$pt_S(sc_s) \neq \infty. \tag{4.30}$$

By (4.30) and (4.12),

$$sc_s \in L. \tag{4.31}$$

Suppose $x_p \neq \bot$ for some process p. Then by (4.28), (4.31), and the sequential specification of snapshot objects (along with the fact that L is a linearization of $\Gamma(T|S)$), there must exist an $S.update_p(x_p)$ operation up_u by p such that $up_u \xrightarrow{L} sc_s$. By this and (4.13),

$$pt(up_u) \leq pt(sc_s). \tag{4.32}$$

Let up_u be invoked by the $SLupdate_p(x_p)$ operation up. Since up is invoked before $pt(sc)$, $pt(up) \leq pt(sc)$ by Lemma 18.

Now suppose $x_p = \bot$. To derive a contradiction, suppose that some $SLupdate_p(x)$

operation up linearizes at or before $pt(sc)$, for some $x \in D$ such that $x \neq \bot$. That is,

$$pt(up) \leq pt(sc). \tag{4.33}$$

By Lemma 20,

$$\text{the interpreted value of } S[p] \text{ at } pt(up) \text{ is } x\text{, and} \tag{4.34}$$

$$R[p] = x \text{ at } pt(up). \tag{4.35}$$

By (4.29) and our assumption that $x_p = \bot$, $R[p] = \bot$ at $pt(sc)$. This along with (4.33) and (4.35) implies that there must exist an $R.DWrite_q(X)$ operation dw by some process q with $X[p] = \bot$, such that

$$pt(up) < time(dw) \leq pt(sc). \tag{4.36}$$

By (4.36) and Lemma 19,

$$pt(up) < time(dw) < time(sc^{69}). \tag{4.37}$$

By (4.34) and the definition of interpreted value, there exists an $S.update_p(x)$ operation up_u by p such that

$$pt_S(up_u) < pt(up). \tag{4.38}$$

By (4.37), (4.38), and the fact that $pt_S(sc_s) \in \big[time(inv(sc_s)), time(rsp(sc_s))\big]$ by definition of linearization point functions,

$$pt_S(up_u) < pt_S(sc_s). \tag{4.39}$$

By (4.39) and the contrapositive of (4.13), $up_u \xrightarrow{L} sc_s$. Since there are no $S.update_q(\perp)$ operations for any process q by assumption, $up_u \xrightarrow{L} sc_s$ implies that sc_s returns some vector with $x' \neq \perp$ in its p-th entry. This along with our assumption that $x_p = \perp$ contradicts (4.28). $\square$

Lemma 22. *Let sc be some complete $SLscan_q()$ operation in T by process q that returns $(x_1, x_2, \ldots, x_n)$, and suppose $x_p \neq \perp$ for some process p. Then there is some $SLupdate_p(x_p)$ operation up_{x_p} such that $pt(up_{x_p}) \leq pt(sc)$, and there is no $SLupdate_p(x)$ operation up_x such that $pt(up_{x_p}) < pt(up_x) \leq pt(sc)$, for any $x \in D$.*

Proof. Let up_{x_p} be the last $SLupdate_p(x_p)$ operation by p that linearizes at or before $pt(sc)$ (Lemma 21 guarantees that such an operation exists). To derive a contradiction, suppose that some $SLupdate_p(x)$ operation by p with $x \neq x_p$ exists, which linearizes in the interval $\big(pt(up_{x_p}), pt(sc)\big]$. Let up_x be the latest such operation. So $pt(up_{x_p}) < pt(up_x) \leq pt(sc)$. Additionally, since sc returns a vector with x_p in its p-th entry, and up_x writes $x \neq x_p$, up_x does not linearize at $pt(sc)$ by R-1 and R-2 (note that at most a single $SLscan$ operation may linearize at any step, since each $SLscan$ linearizes at one of its own steps by R-1). Hence,

$$pt(up_{x_p}) < pt(up_x) < pt(sc). \tag{4.40}$$

By Lemma 20,

$$\text{the interpreted value of } S[p] \text{ is } x \text{ at } pt(up_x)\text{, and} \tag{4.41}$$

$$R[p] = x \text{ at } pt(up_x). \tag{4.42}$$

Let sc_ℓ be the final $S.scan()$ operation performed by sc. Since processes perform operations sequentially, and $pt_S(sc_\ell) \in \big[time(inv(sc_\ell)), time(rsp(sc_\ell))\big]$ by definition of linearization point functions,

$$time(sc^{69}) < pt_S(sc_\ell) < time(sc^{71}). \tag{4.43}$$

There are two cases:

(i) Operation up_x linearizes after sc_ℓ. That is,

$$pt(up_x) > pt_S(sc_\ell). \tag{4.44}$$

By R-1,

$$pt(sc) = time(sc^{71}). \tag{4.45}$$

By (4.43) and (4.44), $time(sc^{69}) < pt(up_x)$. By (4.40) and (4.45), $pt(up_x) < time(sc^{71})$. Hence,

$$time(sc^{69}) < pt(up_x) < time(sc^{71}). \tag{4.46}$$

By (4.42) and the fact that sc^{71} returns a vector with x_p in its p-th entry, there is an $R.DWrite(X)$ operation dw, with $X[p] = x_p$, such that

$$pt(up_x) < time(dw) < time(sc^{71}). \tag{4.47}$$

But by (4.46) and (4.47), $time(dw) \in \big(time(sc^{69}), time(sc^{71})\big)$, which contradicts Lemma 19.

(ii) Operation up_x linearizes not after sc_ℓ. That is,

$$pt(up_x) \leq pt_S(sc_\ell). \tag{4.48}$$

By (4.41) and the definition of interpreted values, there must exist an $S.update_p(x)$ operation up_{u_x} by p such that

$$pt_S(up_{u_x}) < pt(up_x), \text{ and} \tag{4.49}$$

$$\text{there does not exist an } S.update_p(x') \text{ operation } up'_{u_x} \text{ by } p \text{ such that } x' \neq x \text{ and } pt_S(up_{u_x}) < pt_S(up'_{u_x}) \leq pt(up_x). \tag{4.50}$$

By (4.48) and (4.49), $pt_S(up_{u_x}) < pt_S(sc_\ell)$. Then by contrapositive of (4.13),

$$up_{u_x} \xrightarrow{L} sc_\ell. \tag{4.51}$$

Since sc returns a vector with $x_p \neq x$ in its p-th entry, sc_ℓ must also return a vector with $x_p \neq x$ in its p-th entry. By this, (4.51), the fact that L is a linearization of $\Gamma(T|S)$, and the sequential specification of the snapshot type, there must exist an $S.update_p(x_p)$ operation up_{u_p} by p such that

$$up_{u_x} \xrightarrow{L} up_{u_p} \xrightarrow{L} sc_\ell. \tag{4.52}$$

By (4.13) and (4.52),

$$pt_S(up_{u_x}) \leq pt_S(up_{u_p}), \text{ and} \tag{4.53}$$

$$pt_S(up_{u_p}) \leq pt_S(sc_\ell). \tag{4.54}$$

Note that by definition of linearization point functions,

$$pt_S(up_{u_x}) \in \big[time(inv(up_{u_x})), time(rsp(up_{u_x}))\big], \text{ and} \tag{4.55}$$

$$pt_S(up_{u_p}) \in \big[time(inv(up_{u_p})), time(rsp(up_{u_p}))\big]. \tag{4.56}$$

By (4.55), (4.56), and the fact that both up_{u_x} and up_{u_p} are performed by p, $pt_S(up_{u_x}) < pt_S(up_{u_p})$. By this and (4.50),

$$pt(up_x) < pt_S(up_{u_p}). \tag{4.57}$$

Due to (4.40) and the fact that up_{x_p} and up_x are performed by p, the $S.update_p(x_p)$ operation by up_{x_p} linearizes before the invocation of up_x. This along with (4.57) implies that up_{u_p} is performed during some $SLupdate_p(x_p)$ operation up'_{x_p} by p such that

$up'_{x_p} \neq up_{x_p}$. Then by Lemma 18,

$$pt(up'_{x_p}) \leq pt(sc). \tag{4.58}$$

Additionally, since up_{x_p} and up'_{x_p} are both performed by p, they are performed in sequence. This along with (4.58) contradicts the fact that up_{x_p} is the final $SLupdate_p(x_p)$ operation by p that linearizes not after $pt(sc)$.

□

Lemma 23. *Algorithm 5 is linearizable.*

Proof. Lemma 17 ensures that f preserves the happens-before order of the interpreted history $\Gamma(T)$. Lemma 22 ensures that Algorithm 5 satisfies the sequential specification of a snapshot object, and therefore $f(T)$ is a valid sequential history. Thus, $f(T)$ is a linearization of the interpreted history $\Gamma(T)$. □

Lemma 24. *The linearization function f is prefix-preserving.*

Proof. Consider each time t of T, and some operation $op \in \Gamma(T)$ by p. Then $pt(op) = t$ if

(i) operation op is some $SLscan_p()$ operation and op^{71} happens at t (this case follows from R-1), or

(ii) operation op is some $SLupdate_p(x_p)$ operation for some value $x_p \in D$, and t is the first point in time not before $time(inv(op))$ that satisfies $t = time(sc^{71})$ for some $SLscan()$ operation sc, and sc^{71} returns a vector whose p-th entry contains x_p (this case follows from both R-1 and R-2), or

(iii) operation op is some $SLupdate_p(x_p)$ operation for some value $x_p \in D$ that did not linearize at any step prior to t, and op^{67} happens at t (this case follows from R-2).

Thus, at step t it is entirely determined which operations linearize at t. That is, for any operation $op \in \Gamma(T)$, whether $pt(op) = t$ or not can be decided soley based on previous steps

(i.e. steps earlier than t) in T. Additionally, once it is determined that $pt(op) = t$, $pt(op)$ does not change with any future step (i.e. steps later than t) of T. Along with the fact that processes perform operations sequentially, Observation 17 implies that for any time t of T, the set of operations Op that linearize at t contains at most one operation by each process. Therefore, U-2 imposes a total order on the operations in Op. Hence, if T is a prefix of $T' \in \mathcal{T}$, then $f(T)$ is a prefix of $f(T')$. $\square$

Theorem 25. *The snapshot object implemented by Algorithm 5 is strongly linearizable.*

Proof. Lemma 23 shows that the sequential history $f(T)$ is a linearization of $\Gamma(T)$. Furthermore, Lemma 24 proves that f is prefix-preserving. Thus, f is a strong linearization function for $\mathcal{T}$. $\square$

4.4 Lock-Freedom and Complexity Analysis

To simplify our analysis of the amortized complexity of our strongly linearizable snapshot object implementation, we first present a modification to Algorithm 5. The pseudocode of this implementation is provided in Algorithm 7. Our modification adds a sequence number field to each entry of the snapshot object; that is, each entry of the snapshot object stores a pair (x, s), where $x \in D$ is a value, and s is an unbounded sequence number. Initially, the snapshot contains the vector $\big((\bot, 0), \ldots, (\bot, 0)\big)$. Each process also stores a local sequence number (called *seq* in Algorithm 7), which is incremented every time the process updates the snapshot object on line 90. Following this, process p performs an $S.update_p(x, seq)$ operation on line 90, which stores the pair (x, seq) in the p-th entry of the snapshot object. The remainder of the modified implementation is essentially identical to Algorithm 5. Note that $S.scan()$ and $R.DRead()$ operations now return vectors of pairs. That is, if X is returned by some $S.scan()$ or $R.DRead()$ operation, then $X = \big((x_1, s_1), \ldots, (x_n, s_n)\big)$, such that each $x_i \in D$ is a value, and each s_i is a sequence number. We use $vals(X)$ to denote the vector of values stored by X (i.e. $vals(X) = (x_1, \ldots, x_n)$).

Algorithm 7: Modified Strongly Linearizable Snapshot Object

```
shared:
   atomic snapshot object S = ((⊥, 0), . . . , (⊥, 0))
   atomic ABA-detecting register R = ((⊥, 0), . . . , (⊥, 0))

local (to each process):
   Integer seq = 0

Function SLupdate_p(x):
89     seq ← seq + 1
90     S.update_p(x, seq)
91     s ← S.scan()
92     R.DWrite_p(s)

Function SLscan_p():
93     repeat
94         (s_1, c_1) ← R.DRead_p()
95         ℓ ← S.scan()
96         (s_2, c_2) ← R.DRead_p()
97         if !(vals(s_1) = vals(ℓ) = vals(s_2)) then
98             R.DWrite_p(ℓ)
99         end
100    until (vals(s_1) = vals(ℓ) = vals(s_2)) and !c_2
101    return s_2
```

Note that Algorithm 7 also uses an atomic snapshot object for S rather than a linearizable one. This is acceptable for our analysis because we aim to express the amortized complexity of Algorithm 7 in terms of the number of operation invocations on S and R, without regard for how S or R are implemented. This simplification allows us to avoid using the notion of interpreted value from Section 4.2. Instead, for any transcript $T \in \mathcal{T}$, we may simply reference the *value* stored by S at any particular time t of T. Since *update* operations on S are atomic, the value of S at any particular step of T is well-defined.

Since Algorithm 7 uses unbounded sequence numbers, the implementation uses unbounded space. However, since Algorithm 7 performs exactly the same shared memory operations as in Algorithm 5, it is easy to see that these implementations have the same amortized complexity. Hence, while our analysis is performed directly on Algorithm 7, all of the results

in this section may also be applied to Algorithm 5.

We first introduce some notation that will simplify our argument throughout this section. Let X be a vector that is returned by some $S.scan()$ or $R.DRead()$ operation. For a process p and an entry $X[p]$ of X, define $seq(X[p])$ as the second entry of the pair stored in $X[p]$ (i.e. the sequence number stored by $X[p]$). Define $seq(X) = \sum_{i=1}^{n} seq(X[i])$. We first observe that as the underlying snapshot object S is modified, *seq* values of subsequent $S.scan()$ operations never decrease. For the sake of clarity, when we say $X = Y$, for some vectors X and Y, we mean $vals(X) = vals(Y)$ and $seq(X) = seq(Y)$. For the remainder of this section, fix a transcript T on Algorithm 7.

Observation 26. *Suppose that the value of S is X at time t in T. If the value of S is X' at time $u \geq t$ of T, then*

(a) $seq(X'[p]) \geq seq(X[p])$, for every process p, and

(b) $seq(X') \geq seq(X)$.

Proof. Part (a) follows from the fact that each process that performs an *SLupdate* operation increments its local *seq* variable on line 89, prior to invoking the *S.update* operation on line 90. Part (b) follows trivially from part (a). □

Observation 27. *Let X_1 and X_2 be vectors returned by two $S.scan()$ operations in T. If $seq(X_1) = seq(X_2)$, then $X_1 = X_2$.*

Proof. Let op_1 and op_2 be two complete $S.scan()$ operations that return X_1 and X_2 respectively, such that $seq(X_1) = seq(X_2)$. Without loss of generality, suppose $time(op_1) \leq time(op_2)$. To derive a contradiction, suppose that $X_1 \neq X_2$. Thus, some *S.update* operation by p must happen in the interval $\big(time(op_1), time(op_2)\big)$, for some process p. Since the sequence numbers written by subsequent *S.update* operations by p always increase (by the increment on line 89),

$$X_1[p] < X_2[p]. \tag{4.59}$$

By Observation 26 (a), $X_1[q] \leq X_2[q]$ for every process q. This, combined with (4.59), implies that $seq(X_1) < seq(X_2)$, which is a contradiction. □

Observation 28. *Suppose that at time t, the value of S is X, while R contains a vector X'. Then $seq(X) \geq seq(X')$.*

Proof. Notice that the only $R.DWrite(X')$ statements present in Algorithm 5 acquire X' from some previous $S.scan()$ operation. Hence, $seq(X') \geq seq(X)$ by Observation 26 (b). □

Lemma 29. *Let sc be an $S.scan()$ operation that returns X, such that no $S.scan()$ operation sc' with $time(sc') < time(sc)$ returns X. Suppose there are k operations*

$$R.DWrite(X_1), R.DWrite(X_2), \dots, R.DWrite(X_k)$$

that happen after $time(sc)$ in T, such that $seq(X_i) < seq(X)$ for all $i \in \{1, \dots, k\}$. Then $k \leq n-1$.

Proof. By Observation 26, any $S.scan()$ operation in T that is invoked after $time(sc)$ returns a vector Y with $seq(Y) \geq seq(X)$. Thus,

any $S.scan()$ operation that returns a vector X' with $seq(X') < seq(X)$ must be invoked before $time(sc)$. (4.60)

Note that each $R.DWrite$ statement in Algorithm 7 is preceded by an $S.scan$ statement in the same method (i.e. the $R.DWrite$ on line 92 in the $SLupdate$ method is preceded by the $S.scan$ on line 91, and the $R.DWrite$ on line 98 in the $SLscan$ method is preceded by the $S.scan$ on line 95). Furthermore,

for any vector Y and any $R.DWrite_p(Y)$ operation dw by p, the latest $S.scan()$ operation by p that is invoked before dw returns Y. (4.61)

Together, (4.60) and (4.61) imply that, for any $R.DWrite(X')$ operation dw by p with

$seq(X') < seq(X)$, the latest $S.scan()$ operation invoked by p before dw must have been invoked before $time(sc)$. This immediately implies that

$$\text{each process performs at most one } R.DWrite(X') \text{ operation that happens after } time(sc), \text{ with } seq(X') < seq(X). \tag{4.62}$$

Suppose sc is performed by process q. Since each process performs operations sequentially, (4.60) and (4.61) together imply that no $R.DWrite_q(X')$ operation by q happens after $time(sc)$. This combined with (4.62) implies the statement in the lemma. □

Lemma 30. *Suppose that the value of S is X at some step t in T. Let $sc_Y \in T$ be an $S.scan()$ operation that returns Y with $seq(Y) > seq(X)$, such that there does not exist an $S.scan()$ operation $sc_{Y'}$ that returns a vector Y' with $seq(Y') > seq(X)$ and $time(sc_{Y'}) < time(sc_Y)$. Then,*

(a) for any pair of $R.DWrite_p(X)$ operations dw_1, dw_2 by p on line 98 (i.e. during SLscan operations) such that $time(dw_1) < time(dw_2) < time(sc_Y)$, there exists an $R.DWrite(X')$ operation that happens in $\big(time(dw_1), time(dw_2)\big)$, such that $seq(X') < seq(X)$, and

(b) if $R.DWrite_p(X_1), \ldots, R.DWrite_p(X_k)$ is a sequence of operations $dw_1, \ldots, dw_k$ performed by p such that $seq(X_1) = \ldots = seq(X_k) = seq(X)$, then $k \leq 2n$.

Proof of Lemma 30 (a). Since dw_2 is an $R.DWrite_p(X)$ operation, by the pseudocode in Algorithm 7 the latest $S.scan()$ operation on line 95 that was invoked by p prior to $time(dw_2)$ must have returned X. Let V_1 and V_2 be the vectors returned by the last executions of $R.DRead_p()$ by process p prior to $time(dw_2)$ on lines 94 and 96, respectively. By the condition on line 97, either $vals(V_1) \neq vals(X)$ or $vals(V_2) \neq vals(X)$. Then either $V_1 \neq X$ or $V_2 \neq X$. In either case, since dw_1 writes X to R, there must exist an $R.DWrite(X')$ operation $dw_{X'}$ (with $X' \neq X$, and either $X' = V_1$ or $X' = V_2$) that happens after $time(dw_1)$, but before the

$R.DRead_p()$ operation that returns a vector distinct from X. Hence,

$$time(dw_{X'}) \in \big(time(dw_1), time(dw_2)\big). \tag{4.63}$$

Since all $R.DWrite$ operations in Algorithm 7 write the result of some previous $S.scan()$ operation to R, there must be an $S.scan()$ operation $sc_{X'}$ that returns X' and happens prior to $time(dw_{X'})$. Then by (4.63), $time(sc_{X'}) < time(dw_2)$, and hence $time(sc_{X'}) < time(sc_Y)$. By the assumption of the lemma, $seq(X') \leq seq(X)$. Since $X' \neq X$, $seq(X') \neq seq(X)$ by the contrapositive of Observation 27, and therefore $seq(X') < seq(X)$. □

Proof of Lemma 30 (b). Suppose dw_i is performed by an *SLupdate* operation *up* by p, for some $i \in \{1, \ldots, k\}$. First assume that $i > 1$. Then

$$R \text{ contains } X_{i-1} \text{ at } time(dw_{i-1}). \tag{4.64}$$

Since *SLupdate* operations perform at most a single $R.DWrite$ operation (line 92),

$$time(dw_{i-1}) < time(inv(up)). \tag{4.65}$$

Suppose that the value of S at $time(dw_{i-1})$ is V_1. Then by (4.64) and Observation 28, $seq(V_1) \geq seq(X_{i-1}) = seq(X)$. If the value of S at $time(inv(up))$ is V_2, then by (4.65) and Observation 26 (b) $seq(V_2) \geq seq(V_1) \geq seq(X)$. By the increment on line 89, p increases the sequence number of the vector stored by S when it performs the *S.update* operation on line 90 during *up*. Then the vector X' returned by the $S.scan()$ operation performed by *up* on line 66 satisfies $seq(X') > seq(V_2) \geq seq(X)$. By the pseudocode of Algorithm 7, dw_i is an $R.DWrite_p(X')$ operation (i.e. $X_i = X'$), which contradicts the assumption that $seq(X_i) = seq(X)$. Therefore, $i = 1$, and

$$\text{only a single member of the sequence may belong to an } SLupdate \text{ operation.} \tag{4.66}$$

By (4.66), $dw_2, \ldots, dw_k$ are all performed during $SLscan$ operations (on line 98). Let sc_X be the earliest $S.scan()$ operation in T that returns X (i.e. there does not exist an $S.scan()$ operation sc'_X which returns X such that $time(sc'_X) < time(sc_X)$). Since $seq(X_i) = seq(X)$, Observation 27 implies that $X_i = X$, for all $i \in \{1, \ldots, k\}$. Hence, since every $R.DWrite$ operation writes a vector returned by some earlier $S.scan()$ operation,

$$time(sc_X) < time(dw_i) \text{ for all } i \in \{1, \ldots, k\}. \tag{4.67}$$

By Lemma 30 (a), for every pair of consecutive operations dw_i, dw_{i+1} that happen before $time(sc_Y)$, there exists an $R.DWrite(X')$ operation that happens in $\big(time(dw_i), time(dw_{i+1})\big)$ with $seq(X') < seq(X)$. By Lemma 29 there are at most $n-1$ such $R.DWrite(X')$ operations that happen after $time(sc_X)$. Combining this with (4.67), we obtain the following:

$$\text{There are at most } n \; R.DWrite_p(X_i) \text{ operations performed during } SLscan \text{ operations that happen before } time(sc_Y). \tag{4.68}$$

By Lemma 29,

$$\text{there are at most } n-1 \; R.DWrite_p(X_i) \text{ operations performed during } SLscan \text{ operations that happen after } time(sc_Y). \tag{4.69}$$

Combining (4.66), (4.68), and (4.69) yields $k \leq 2n$. □

Lemma 31. *Let op* $\in \Gamma(T)$ *be an operation by process p, and suppose op performs a sequence of* $S.scan()$ *operations* $sc_1, sc_2, \ldots, sc_k$*, such that* sc_i *returns a vector* X_i *for all* $i \in \{1, \ldots, k\}$*, and* $seq(X_1) = \ldots = seq(X_k)$*. Then* $k \leq 2n^2 + 1$*.*

Proof. Clearly, every $SLupdate$ operation performs at most one $S.scan()$ operation.

Suppose *op* is an $SLscan$ operation. Since $seq(X_1) = \ldots = seq(X_k)$, by Observation 27 $X_1 = \ldots = X_k$. Let $X = X_1 = \ldots = X_k$. Consider some sc_i operation, with $i \in \{1, \ldots, k-1\}$.

Operation op must restart its main loop after sc_i, and so the condition on line 100 must hold at the end of the iteration of the main loop during which sc_i is performed by p. There are two cases:

(i) The vectors of values compared on line 100 are unequal. Then by the condition on line 100, process p performs an $R.DWrite_p(X)$ on line 73 following sc_i.

(ii) The Boolean flag returned by the $R.DRead_p()$ operation performed by p on line 96 following sc_i is $true$ (but the vectors of values compared on line 75 are equal). Then, by the sequential specification of ABA-detecting registers, some $R.DWrite(X)$ operation occurs after the $R.DRead()$ operation performed by p on line 94 prior to sc_i, and before the $R.DRead()$ operation performed by p on line 96 following sc_i.

In both cases, some $R.DWrite(X)$ operation happens during the iteration of the main loop in which sc_i is performed. Lemma 30 (b) ensures that each process p performs at most $2n$ $R.DWrite_p(X)$ operations. Since there are n processes, we obtain $k - 1 \leq 2n^2$, and hence $k \leq 2n^2 + 1$. □

Theorem 32. *(a) Each SLupdate performs at most one S.update, one S.scan, and one R.DWrite operation.*

(b) For any transcript that contains u SLupdate and s SLscan invocations, the total number of operation invocations on S and R during SLscan operations is $O(s + n^3 u)$.

In particular, the implementation is lock-free provided that S and R are.

Proof. Part (a) follows immediately from the pseudocode of Algorithm 5.

We now prove part (b). For any process let s_p denote the number of $SLscan$ invocations by process p. Further, for $i \in \{1, \ldots, s_p\}$ let $k_{p,i}$ denote the total number of times process p calls $S.scan()$ in line 95 during its i-th $SLscan()$ operation. From Lemma 31 we obtain $\sum_{i=1}^{s_p} k_{p,i} = O(un^2 + s_p)$ for each process p. Using $s = \sum_p s_p$ we obtain that the total number

of $S.scan()$ calls during all $SLscan()$ operations is

$$\sum_p \sum_{i=1}^{s_p} k_{p,i} = O\left(\sum_p un^2 + s_p\right) = O(s + un^3).$$

□

As mentioned previously, we can use any lock-free or wait-free linearizable snapshot implementation for S. Instead of an atomic ABA-detecting register R, we can use the lock-free strongly linearizable one from Chapter 3. Thus, Theorems 1, 25, and 32 yield Theorem 2.

4.5 Remarks

In this chapter we presented the first lock-free strongly linearizable implementation of a snapshot object that requires only bounded space. The time complexity of our implementation is unfortunate; even with a small number of processes, the use of our implementation seems impractical due to its $O(n^3)$ runtime. However, when contention is low (i.e. there are few overlapping calls to *SLupdate* and *SLscan*), our snapshot implementation performs reasonably well. Notice that in a low-contention scenario, *SLscan* operations are seldom forced to repeat their main loop, and therefore their runtime is dominated by the *S.scan* call on line 70. Hence, choosing a reasonably efficient linearizable snapshot implementation for S results in an efficient strongly linearizable snapshot. However, when contention is high and the number of processes is sufficiently large (and *SLupdate* is called sufficiently often), it becomes increasingly likely that *SLscan* operations *never* terminate, as they are interrupted infinitely often by *S.update* or $R.DWrite$ calls performed during other concurrent *SLupdate* or *SLscan* operations. A natural extension of this work would aim to develop more efficient implementations.

Previous strongly linearizable implementations of counters and max-registers (such as the lock-free modification of the max-register in [12]) required an unbounded number of

unbounded registers. Our bounded snapshot implementation can be used to implement a lock-free strongly linearizable counter or max-register using only a bounded number of registers. Note that these implementations still inherently require registers to store unbounded values, since the state space of both counters and (unbounded) max-registers is infinite.

5. General Construction

Aspnes and Herlihy [4] defined the large class of *simple* types, and demonstrated that any type in this class has a wait-free linearizable implementation from atomic multi-reader multi-writer registers. Simple types require that any pair of operations either commute, or one overwrites the other (see below for a formal definition). Algorithm 8 depicts Aspnes and Herlihy's general wait-free linearizable implementation of an arbitrary simple type $\mathscr{T}$. Processes communicate only through an atomic snapshot object, *root* (which can be replaced with a wait-free linearizable implementation from registers). Suppose $\mathscr{T}$ supports the set of invocation descriptions $\mathcal{O}$. Then for every $invoke \in \mathcal{O}$, *invoke* is implemented by the *execute*(*invoke*) method. Aspnes and Herlihy proved that Algorithm 8 is linearizable with respect to the sequential specification of the simulated type $\mathscr{T}$. We prove that it is in fact strongly linearizable, and thus it remains strongly linearizable if *root* is a strongly linearizable snapshot object. Thus, using our lock-free snapshot implementation from Chapter 4 for *root* yields Theorem 3.

Throughout this chapter, we only consider types $\mathscr{T} = (\mathcal{S}, s_0, \mathcal{O}, \mathcal{R}, \delta)$ such that, for every $s \in \mathcal{S}$ and for every $invoke \in \mathcal{O}$, $\delta(s, invoke)$ is defined. Two sequential histories H and H' are *equivalent* if, for any sequential history S, $H \circ S$ is valid if and only if $H' \circ S$ is valid. The invocation events $inv(op_1)$ and $inv(op_2)$ *commute* if, for all sequential histories H such that $H \circ op_1$ and $H \circ op_2$ are valid, $H \circ op_1 \circ op_2$ and $H \circ op_2 \circ op_1$ are valid and equivalent. The invocation event $inv(op_2)$ *overwrites* $inv(op_1)$ if, for all sequential histories H such that $H \circ op_1$ and $H \circ op_2$ are valid, $H \circ op_1 \circ op_2$ is valid and equivalent to $H \circ op_2$. As a shorthand, we say op_1 commutes with (resp. overwrites) an operation op_2 if $inv(op_1)$ commutes with (resp. overwrites) $inv(op_2)$. An invocation description $invoke_1$ commutes with (resp. overwrites) the invocation description $invoke_2$ if, for all invocation events $inv(op_1) = (O, invoke_1, id_1)$ and $inv(op_2) = (O, invoke_2, id_2)$, $inv(op_1)$ commutes with (resp. overwrites) $inv(op_2)$. These

properties allow us to describe the class of *simple* types.

Definition 33. *Let $\mathscr{T}$ be a type that supports a set of invocation descriptions $\mathcal{O}$. Then $\mathscr{T}$ is* simple *if, for every pair of invocation descriptions $invoke_1, invoke_2 \in \mathcal{O}$, either $invoke_1$ and $invoke_2$ commute, or one overwrites the other.*

For the remainder of this chapter, let $\mathscr{T} = (\mathcal{S}, s_0, \mathcal{O}, \mathcal{R}, \delta)$ be some simple type, and let O be an object of type $\mathscr{T}$ implemented by Algorithm 8; that is, each invocation description $invoke \in \mathcal{O}$ is implemented by the $execute(invoke)$ method. Let $\mathcal{T}$ be the set of all transcripts on O. For ease of notation, for every operation ex on O let $invoc(ex)$ denote the invocation description of ex; that is, if $inv(ex) = (O, invoke, id)$, then $invoc(ex) = invoke$.

Algorithm 8 maintains a representation of a graph in a shared snapshot object called *root*. Each entry of the *root* variable contains a reference to an instance of type *node*, which has three fields: *invocation*, *response*, and *preceding*. The *invocation* and *response* fields contain an invocation description and response, respectively. The *preceding* field is an array containing n references to nodes. For a node x, $x.preceding[i]$ contains either $\perp$ or a pointer to a node y.

A *precedence graph* $G = (V, E)$ of a history H is a directed graph whose vertices are operations, such that for any $op_1, op_2 \in V$ there is a directed path of length at least 1 from op_1 to op_2 in G if and only if $op_1 \xrightarrow{H} op_2$. Notice that the happens-before relation is the transitive closure of G.

The following notion of *dominance* is used to break ties between mutually overwriting operations.

Definition 34. *An invocation event $inv(op_2)$ of process p* dominates *$inv(op_1)$ of process q if either*

(1) $inv(op_2)$ overwrites $inv(op_1)$ but not vice-versa, or

(2) $inv(op_1)$ and $inv(op_2)$ overwrite each other and $p > q$.

An invocation description $invoke_2$ of process p dominates $invoke_1$ of process q if, for all invocation events $inv(op_1) = (O, invoke_1, id_1)$ and $inv(op_2) = (O, invoke_2, id_2)$, $inv(op_2)$ dominates $inv(op_1)$. A *linearization graph*, $lingraph(G)$, is constructed by adding directed edges to the precedence graph G as follows: First, an arbitrary topological order $op_1, \ldots, op_k$ of G is fixed. Then all pairs (i, j), $1 \leq i < j \leq k$, are considered in lexicographical order, and if one of the two invocation descriptions in $\{op_i, op_j\}$ dominates the other, an edge is added from the dominated operation to the dominating one, provided that edge does not close a cycle. A precise description of the construction of a linearization graph is provided in the *lingraph* method of Algorithm 8.

Suppose $T \in \mathcal{T}$ is a transcript on O. Let $ex \in \Gamma(T)$ be an operation on O by process p. Process p begins ex by performing a $root.scan()$ operation on line 115. This $root.scan()$ operation returns a vector *view* of references to nodes. In line 116 process p computes a precedence graph G of operations on O using a straightforward graph search, starting with the nodes stored in *view* (we will explain the *precgraph* method in more detail later). It then calculates a sequential history H on line 117 by topologically sorting a linearization graph $lingraph(G)$. Now p constructs a new node x that stores the invocation description $invoc(ex)$, and creates the invocation event $inv(op) = (O, invoc(ex), id)$, for some integer id. Process p then constructs a response event $rsp(op) = (resp, id)$ such that $H \circ inv(op) \circ rsp(op)$ is valid with respect to the sequential specification of $\mathscr{T}$ (lines 118-121). The existence of the response *resp* is guaranteed by our assumption that for every $s \in \mathcal{S}$ and every $invoke \in \mathcal{O}$, $\delta(s, invoke)$ is defined. Note that the node referenced by $view[q]$ is the most recent node written to *root* by process q prior to the $root.scan()$ operation performed by ex. For every process $q \in \{1, \ldots, n\}$, p stores $view[q]$ in $x.preceding[q]$ on line 123. Finally, p writes the address of the constructed node to the snapshot object on line 125. For any operation $ex \in \Gamma(T)$ such that $ex^{125} \in T$, let $node(ex)$ be the node constructed by ex (i.e. $node(ex)$ is the node whose address is written to *root* during ex^{125}). Since each operation instantiates a new node instance, and each node reference remains in the shared precedence graph representation

forever, the algorithm uses unbounded space.

We proceed by first outlining an argument for the linearizability of Algorithm 8. On line 121, an operation ensures that it calculates a valid response with respect to the sequential history H constructed on line 117. Then assuming that H is valid, the algorithm is correct as long as writing out a node constructed by a pending operation does not invalidate the response of some concurrent operation. Suppose ex_1 and ex_2 are concurrent operations. For this example, assume no other operations are concurrent with either ex_1 or ex_2. Hence, both ex_1 and ex_2 construct the same precedence graph (call it G) on line 116. For simplicity, assume that ex_1 and ex_2 both compute the same topological ordering (call it H) of $lingraph(G)$ on line 117 (one of the key results of Aspnes and Herlihy [4] is that every pair of topological orderings of any linearization graph are equivalent — we state this result more formally in Section 5.2). Hence, on lines 120 and 121 ex_1 and ex_2 construct operations op_1 and op_2, respectively, such that $H \circ op_1$ and $H \circ op_2$ are valid. Since the simulated type is simple, either $invoc(ex_1)$ and $invoc(ex_2)$ commute, or one overwrites the other. If $invoc(ex_1)$ and $invoc(ex_2)$ commute, then it does not matter which node is written to *root* first, since $H \circ op_1 \circ op_2$ and $H \circ op_2 \circ op_1$ are both valid and equivalent, by definition of commutativity. If $invoc(ex_2)$ overwrites $invoc(ex_1)$, then again it does not matter which operation writes its node to *root* first, since any operation that views the precedence graph after both ex_1 and ex_2 have written their nodes to *root* adds a dominance edge from op_1 to op_2 in the linearization graph. Hence, in any topological ordering of this linearization graph op_1 occurs immediately before op_2 (recall our assumption that no other operations are concurrent with either ex_1 or ex_2). Since $H \circ op_1 \circ op_2$ is valid by the definition of overwriting, the responses of both op_1 and op_2 are valid. A symmetric argument applies if $invoc(ex_1)$ overwrites $invoc(ex_2)$.

We now outline the intuition behind our strong linearization function for Algorithm 8; a full proof of the strong linearizability of Algorithm 8 is provided in Section 5.2. Suppose $T \in \mathcal{T}$ is a transcript, and let $ex \in \Gamma(T)$ be an operation by process p. After p performs the $root.scan()$ operation on line 115 during ex, the response of ex is entirely determined,

Algorithm 8: Implementation of a simple type [4]

```
    struct node :
        invocation description, invocation ∈ 𝒪
        response, response ∈ ℛ
        pointers to nodes, preceding[1 ... n]
    shared
        atomic snapshot object root = (null, ..., null)

    Function lingraph(G):
102     let op_1, ..., op_k be a topological sort of G
103     L ← G
104     for i ∈ {1, ..., k − 1} do
105         for j ∈ {i + 1, ..., k} do
106             if op_i dominates op_j and adding (op_j, op_i) to L does not complete a cycle
                 then
107                 add (op_j, op_i) to L
108             end
109             if op_j dominates op_i and adding (op_i, op_j) to L does not complete a cycle
                 then
110                 add (op_i, op_j) to L
111             end
112         end
113     end
114     return L

    Function execute_p(invoke):
115     view ← root.scan()
116     G ← precgraph(view)
117     H ← topological sort of lingraph(G)
118     initialize a new node e = {⊥, ⊥, ⊥}
119     e.invocation ← invoke
120     inv(op) ← (O, invoke, id)
121     rsp(op) ← (resp, id) such that H ∘ inv(op) ∘ rsp(op) is valid; e.response ← resp
122     for i ∈ {1, ..., n} do
123         e.preceding[i] ← view[i]
124     end
125     root.update_p(address of e)
126     return e.response
```

since it is chosen based on the contents of the precedence graph constructed from *view* on line 116. If during ex, an operation ex' writes its constructed node to *root*, and $invoc(ex')$ dominates $invoc(ex)$, then ex may be linearized immediately before ex', since ex no longer has any effect on the responses calculated by subsequent operations. Hence, operations may only linearize when some node is written to *root* on line 125, as it is entirely determined which operations linearize when a particular node is written to the graph (i.e. all concurrent operations that are dominated by the writing operation).

5.1 Storing a Precedence Graph

In this section we prove that a precedence graph may be extracted from *root*. More specifically, we show that for any vector *view* returned by a $root.scan()$ operation, $precgraph(view)$ returns a precedence graph of some history (the implementation of *precgraph* is provided in Algorithm 9).

The $precgraph(view)$ method begins by calling $nodegraph(view)$ on line 127, which computes a graph whose vertices are nodes. The $nodegraph(view)$ method begins with a straightforward graph search starting from the nodes present in *view*. First, a process p performing a $nodegraph(view)$ operation initializes an empty graph $G = (V, E)$ and an empty queue *queue* (lines 140 and 141). Next, p adds all nodes referenced in *view* to both *queue* and the vertex set V during the loop on line 142. The main loop of the *nodegraph* method begins on line 149, and continues until all nodes have been removed from *queue*. During the main loop, p first removes a node $node(ex)$ from *queue* on line 150. For each node $node(ex')$ referenced in $node(ex).preceding$, p adds the edge $\big(node(ex'), node(ex)\big)$ to E on line 152; if $node(ex')$ is not present in V, then p adds $node(ex')$ to *queue* and V on lines 154 and 155. After the main while-loop has terminated, the $nodegraph(view)$ operation returns the computed graph G. In the $precgraph(view)$ method, after $nodegraph(view)$ has been computed, process p computes a topological ordering $node(ex_1), \ldots, node(ex_k)$ of the vertices in G. Next, process p initializes an array $id[1 \ldots n] = [1, \ldots, 1]$ on line 129.

Process p then begins a for-loop on line 130; during this loop, p computes an operation on O for each node present in G. That is, for each $node(ex_i)$ present in the topological ordering of G, p constructs an operation op_i, with $inv(op_i) = (O, node(ex_i).invocation), opid)$, and $rsp(op_i) = (node(ex_i), opid)$, where $opid$ is an integer. The operation identifier $opid$ is computed on line 132, where it is assigned the value $(id[p] \cdot n) + (p - 1)$, where p is the process that performed ex_i. The final statement of the loop increments $id[p]$ by 1. In general, the operation constructed for the j-th node in the topological ordering of G which was written by process p is assigned the identifier $(j \cdot n) + (p - 1)$. This way, each operation constructed during the loop is assigned a unique identifier (note that if ex_j is performed by process p, and $inv(ex_j)$ has the identifier $opid$, then $opid \equiv (p - 1) \pmod{n}$). After the for-loop, on line 137 p replaces each $node(ex_i) \in V$ with op_i (each edge is replaced similarly on line 138).

Let $T \in \mathcal{T}$ be a transcript, and let $\mathcal{G}(T) = (V_T, E_T)$ be a graph such that

$$\begin{aligned} V_T &= \{node(ex) \; : \; ex^{125} \in T\}, \text{ and} \\ E_T &= \Big\{ \big(node(ex_1), node(ex_2)\big) \; : \; node(ex_1), node(ex_2) \in V_T \wedge \\ &\qquad \exists p \in \{1, \ldots, n\} \text{ s.t. } node(ex_2).preceding[p] = \text{address of } node(ex_1) \Big\}. \end{aligned}$$

We will show that if $root$ contains the vector $view$ after all steps in T are performed in order, then $nodegraph(view) = \mathcal{G}(T)$.

Observation 35. *If* $\big(node(ex_1), node(ex_2)\big) \in E_T$, *then* $time(ex_1^{125}) < time(ex_2^{115})$.

Proof. Since $\big(node(ex_1), node(ex_2)\big) \in E_T$, $node(ex_2).preceding[p]$ contains the address of $node(ex_1)$, for some process p. On line 118 of Algorithm 8, each operation initializes its own node, and only modifies the fields of that node. Hence, only ex_2 modifies the $preceding$ field of $node(ex_2)$. Thus, ex_2 must place the address of $node(ex_1)$ into $node(ex_2).preceding[p]$ when it performs line 123 (during the p-th iteration of the for-loop on line 122). Therefore, the address of $node(ex_1)$ must be in $view[p]$, where $view$ is the vector returned by ex_2^{115}. Then by the sequential specification of snapshot objects, $time(ex_1^{125}) < time(ex_2^{115})$. □

Algorithm 9: Extraction of a precedence graph from a vector of node references.

```
    Function precgraph(view):
127     G = (V, E) ← nodegraph(view)
128     let node(ex_1), ..., node(ex_k) be a topological sort of G
129     let id[1 ... n] = [1, ..., 1]
130     for i ∈ {1, ..., k} do
131         suppose ex_i is by process p
132         let opid = (id[p] · n) + (p − 1)
133         let inv(op_i) = (O, node(ex_i).invocation, opid)
134         let rsp(op_i) = (node(ex_i).response, opid)
135         id[p] ← id[p] + 1
136     end
137     replace node(ex_i) ∈ V with op_i, for all i ∈ {1, ..., k}
138     replace each (node(ex_i), node(ex_j)) ∈ E with (op_i, op_j)
139     return G

    Function nodegraph(view):
140     let G = (V, E) be an empty graph
141     let queue be an empty queue
142     for v ∈ view do
143         if v ≠ null then
144             let node(ex) be the node addressed by v
145             enqueue node(ex) to queue
146             V ← V ∪ {node(ex)}
147         end
148     end
149     while queue is not empty do
150         dequeue node(ex) from queue
151         for each node(ex') referenced in node(ex).preceding do
152             E ← E ∪ {(node(ex'), node(ex))}
153             if node(ex') ∉ V_N then
154                 enqueue node(ex') to queue
155                 V ← V ∪ {node(ex')}
156             end
157         end
158     end
159     return G
```

Observation 36. *If there is a path of length at least 1 from $node(ex_1)$ to $node(ex_2)$ in $\mathcal{G}(T)$, then $time(ex_1^{125}) < time(ex_2^{115})$.*

Proof. If the length of the path from $node(ex_1)$ to $node(ex_2)$ is 1, then the observation immediately follows from Observation 35. Now

$$\text{assume that the observation holds for any path of length } k \geq 1. \tag{5.1}$$

Suppose that there is a path of length $k+1$ from $node(ex_1)$ to $node(ex_2)$. Let $node(ex_\ell)$ be the second-last node on this path. So

$$\text{there is a path from } node(ex_1) \text{ to } node(ex_\ell) \text{ of length } k \text{, and} \tag{5.2}$$

$$\big(node(ex_\ell), node(ex_2)\big) \in E_T. \tag{5.3}$$

By (5.1) and (5.2),

$$time(ex_1^{125}) < time(ex_\ell^{115}). \tag{5.4}$$

By (5.3) and Observation 35,

$$time(ex_\ell^{125}) < time(ex_2^{115}). \tag{5.5}$$

By the pseudocode of the *execute* method in Algorithm 8, $time(ex_\ell^{115}) < time(ex_\ell^{125})$. This, along with (5.4) and (5.5), imply that $time(ex_1^{125}) < time(ex_2^{115})$. Hence, the observation follows by induction. □

Observation 37. *For some process p, let $ex_1 \circ \ldots \circ ex_k$ be the longest prefix of $\Gamma(T)|p$ with $ex_k^{125} \in T$. Then for any $i, j \in \{1, \ldots, k\}$ such that $i < j$, there is a path of length $j - i$ from $node(ex_i)$ to $node(ex_j)$.*

Proof. We show that, for any $i \in \{1, \ldots, k-1\}$, $\big(node(ex_i), node(ex_{i+1})\big) \in E_T$. Since ex_i and ex_{i+1} are both performed by p, they are performed in sequence. Hence, $time(ex_i^{125}) <$

$time(ex_{i+1}^{115})$, and since each operation performs only a single *root.update* operation (line 125), there is no *root.update* operation by p that happens in $\big(time(ex_i^{125}), time(ex_{i+1}^{115})\big)$. Then if *view* is the vector returned by ex_{i+1}^{115}, $view[p]$ contains the address of $node(ex_i)$. Therefore, ex_{i+1} places the address of $node(ex_i)$ into $node(ex_{i+1}).preceding[p]$ on line 123 (during the p-th iteration of the for-loop on line 122). Then by the definition of E_T, $\big(node(ex_i), node(ex_{i+1})\big) \in E_T$. □

Observation 38. *Let $ex_q, ex_p \in \Gamma(T)$ be two operations by processes q and p, respectively, such that $ex_q^{125}, ex_p^{125} \in T$. If $time(ex_q^{125}) < time(ex_p^{115})$, then there is a path from $node(ex_q)$ to $node(ex_p)$ in $\mathcal{G}(T)$.*

Proof. If $q = p$, then the observation statement follows from Observation 37.

Now assume $q \neq p$. Suppose $ex_{q,1}, \ldots, ex_{q,k}$ is the sequence of *execute* operations by q in $\Gamma(T)$. Let $ex_{q,i}$ be the final operation in this sequence such that $time(ex_{q,i}^{125}) < time(ex_p^{115})$ (the existence of $ex_{q,i}$ is guaranteed by the observation assumption). That is,

$$\text{there is no } j \in \{i+1, \ldots, k\} \text{ such that } time(ex_{q,j}^{125}) < time(ex_p^{115}). \tag{5.6}$$

Let *view* be the vector returned by ex_p^{115}. By (5.6) and the fact that $time(ex_{q,i}^{125}) < time(ex_p^{115})$,

$$view[q] \text{ contains the address of } node(ex_{q,i}). \tag{5.7}$$

By (5.7), ex_p places the address of $node(ex_{q,i})$ into $node(ex_p).preceding[q]$ on line 123. Then by definition of E_T,

$$\big(node(ex_{q,i}), node(ex_p)\big) \in E_T. \tag{5.8}$$

If $ex_q = ex_{q,i}$, then the observation statement is implied by (5.8). Otherwise, if $ex_q = ex_{q,j} \neq ex_{q,i}$, then $j < i$ by (5.6) along with the observation assumption that $time(ex_q^{125}) < time(ex_p^{115})$. Hence, by Observation 37, there is a path from $node(ex_q)$ to $node(ex_{q,i})$ in $\mathcal{G}(T)$. By this and (5.8), there is a path from $node(ex_q)$ to $node(ex_p)$ in $\mathcal{G}(T)$. □

Observation 39. *Suppose $T \in \mathcal{T}$ is a transcript. Suppose that, after all steps in T are performed in order, root contains the vector view. Let $G = (V, E) = nodegraph(view)$. Then for every operation $ex \in \Gamma(T)$ such that $ex^{125} \in T$,*

(a) $node(ex) \in V$, and

(b) for every $node(ex')$ referenced in $node(ex).preceding$, $\big(node(ex'), node(ex)\big) \in E$.

That is, $nodegraph(view) = \mathcal{G}(T)$.

Proof. To prove part (a), we show that for every process p, each node whose address is written to *root* during T is added to V in the calculation of $nodegraph(view)$. Let $ex_1, \dots, ex_k \in \Gamma(T)$ be the (nonempty) sequence of operations by p such that $ex_i^{125} \in T$, for every $i \in \{1, \dots, k\}$. (If there are no such operations by process p, then V trivially contains all nodes written to *root* by p in T.) Note that $view[p]$ contains a reference to $node(ex_k)$. Hence, in the computation of $nodegraph(view)$, $node(ex_k)$ is added to V on line 146. Now

$$\text{assume that for some } j \in \{2, \dots, k\},\ node(ex_j) \in V. \tag{5.9}$$

We proceed by showing that $node(ex_{j-1}) \in V$. By Observation 37, there exists a path of length 1 (i.e. an edge) from $node(ex_{j-1})$ to $node(ex_j)$ in $\mathcal{G}(T)$. Then

$$node(ex_j).preceding[p] \text{ contains a reference to } node(ex_{j-1}). \tag{5.10}$$

Due to (5.9), in the computation of $nodegraph(view)$, $node(ex_j)$ must have been added to V on either line 146 or line 155. In either case, $node(ex_j)$ is added to *queue* on the previous line. Since the while-loop on line 149 does not terminate until *queue* is empty, $node(ex_j)$ is eventually dequeued from *queue* on line 150. By (5.10), the for-loop on line 151 eventually reaches $node(ex_{j-1})$. Hence, at the if-statement on line 153, either $node(ex_{j-1}) \in V$ already, or $node(ex_{j-1})$ is added to V on line 155. Therefore, $node(ex_{j-1}) \in V$, and by induction $node(ex_1), \dots, node(ex_k) \in V$. This completes the proof of part (a).

Due to (a), and the fact that each node is added to *queue* before it is added to V during the calculation of $nodegraph(view)$,

$$\text{for every } ex \in \Gamma(T) \text{ such that } ex^{125} \in T,\ node(ex) \text{ is dequeued from } queue \text{ on line 150 at some point during the computation of } nodegraph(view). \tag{5.11}$$

When $node(ex)$ is dequeued from *queue* on line 150, for every node $node(ex')$ that is referenced in $node(ex).preceding$, the edge $\big(node(ex'), node(ex)\big)$ is added to E (if it is not already present in E) on line 152; part (b) follows from this along with (5.11). □

Let $T \in \mathcal{T}$ be a transcript such that *root* contains the vector *view* after all steps of T are performed in order. Let $oper(ex)$ denote the operation on O with $inv\big(oper(ex)\big) = \big(O, node(ex).invocation, (j \cdot n) + (p-1)\big)$ and $rsp\big(oper(ex)\big) = \big(node(ex).response, (j \cdot n) + (p-1)\big)$, where ex is the j-th operation by p in $\Gamma(T)$.

Observation 40. *Consider the operation op_i constructed from $node(ex_i)$ on lines 133 and 134 during the computation of $precgraph(view)$, for any $i \in \{1, \ldots, k\}$. Then $op_i = oper(ex_i)$.*

Proof. Suppose ex_i is the j-th operation performed by p in $\Gamma(T)$. Let G be the graph returned by the $nodegraph(view)$ operation on line 127. Consider the topological ordering $node(ex_1), \cdots, node(ex_k)$ of G constructed on line 128; let $node(ex_{p,1}), \ldots, node(ex_{p,\ell})$ be the subsequence of $node(ex_1), \cdots, node(ex_k)$ consisting of all and only those nodes constructed by p. Observation 37 implies that $ex_i = ex_{p,j}$, as otherwise there would be a backwards edge among the sequence $node(ex_{p,1}), \ldots, node(ex_{p,\ell})$ in G. Hence, $node(ex_i)$ is the j-th node by process p that is encountered during the for-loop on line 130 (that is, each of $node(ex_{p,1}), \ldots, node(ex_{p,j-1})$ is encountered during the for-loop prior to $node(ex_{p,j}) = node(ex_i)$). On line 133, the invocation description of op_i is set to $node(ex_i).invocation$, and on line 134, the response of op_i is set to $node(ex_i).response$. Now consider *opid*, the operation identifier calculated for op_i on line 132. Since ex_i is the j-th node by process p that is encountered during the for-loop on line 130, $id[p]$ has been incremented $j-1$

times by the time op_i is constructed. Hence, $opid = (id[p] \cdot n) + (p-1) = (j \cdot n) + (p-1)$. Therefore, $inv(op_i) = \big(O, node(ex_i).invocation, (j \cdot n) + (p-1)\big) = inv\big(oper(ex_i)\big)$, and $rsp(op_i) = \big(node(ex_i).response, (j \cdot n) + (p-1)\big) = rsp\big(oper(ex_i)\big)$. □

By Observation 40 and the replacements performed on line 137, if V is the vertex set of the graph returned by $precgraph(view)$, then $V = \{oper(ex) : node(ex) \text{ is in } \mathcal{G}(T)\}$. We will use this fact without referencing Observation 40 for the remainder of the chapter. Let H be a history on an object O of type $\mathscr{T}$ obtained from $T|root$ by doing the following for each step t of $T|root$:

(i) If $t = time(ex^{115})$ for some operation ex by process p, then replace this step by the invocation event $inv\big(oper(ex)\big)$ in H.

(ii) If $t = time(ex^{125})$ for some operation ex by process p, then replace this step by the response event $rsp\big(oper(ex)\big)$ in H.

(iii) If $t = time\big(inv(op)\big)$, for any $root.scan$ or $root.update$ operation op, remove this step from H.

Lemma 41. *The graph $precgraph(view)$ is a precedence graph of the history H.*

Proof. Suppose $oper(ex_1) \xrightarrow{H} oper(ex_2)$. Since $oper(ex_1) \xrightarrow{H} oper(ex_2)$, $rsp\big(oper(ex_1)\big)$ occurs before $inv\big(oper(ex_2)\big)$ in H. Then by the construction rules for H, $time_{T|root}(ex_1^{125}) < time_{T|root}(ex_2^{115})$. This immediately implies that $time_T(ex_1^{125}) < time_T(ex_2^{115})$. Hence, there is a path from $node(ex_1)$ to $node(ex_2)$ in $\mathcal{G}(T)$ by Observation 38. Then by the replacements performed on lines 137 and 138, there is a path from $oper(ex_1)$ to $oper(ex_2)$ in $precgraph(view)$.

Now suppose there is a path from $oper(ex_1)$ to $oper(ex_2)$ in $precgraph(view)$. Then there is a path from $node(ex_1)$ to $node(ex_2)$ in $\mathcal{G}(T)$. By Observation 36, $time_T(ex_1^{125}) < time_T(ex_2^{115})$. This immediately implies that

$$time_{T|root}(ex_1^{125}) < time_{T|root}(ex_2^{115}). \tag{5.12}$$

By the construction rules for H, $rsp(ex_1^{125})$ is replaced with $rsp\big(op_1\big)$ in H, and $rsp(ex_2^{115})$ is replaced with $inv\big(oper(ex_2)\big)$ in H. By this and (5.12), $rsp\big(oper(ex_1)\big)$ occurs before $inv\big(oper(ex_2)\big)$ in H, so $oper(ex_1) \xrightarrow{H} oper(ex_2)$. $\square$

Throughout the remainder of this section, for any transcript $T \in \mathcal{T}$ such that *root* contains the vector *view* after all steps in T are performed in order, we refer to $precgraph(view)$ as the precedence graph *induced by* $\mathcal{G}(T)$.

For operations $oper(ex_1), oper(ex_2)$ in a precedence graph G, we say $oper(ex_1)$ *precedes* $oper(ex_2)$ in G if and only if there is a path from $oper(ex_1)$ to $oper(ex_2)$ in G. If there is no path between operations $oper(ex_1)$ and $oper(ex_2)$ in G, then we say $oper(ex_1)$ and $oper(ex_2)$ are *concurrent* in G. If $T \in \mathcal{T}$ is a transcript such that G is the precedence graph induced by $\mathcal{G}(T)$, then following two observations are immediate from Observation 36 and Observation 38, respectively:

Observation 42. *Operation $oper(ex_1)$ precedes $oper(ex_2)$ in G if and only if $time(ex_1^{125}) < time(ex_2^{115})$.*

Observation 43. *Operations $oper(ex_1)$ and $oper(ex_2)$ are concurrent in G if and only if $time(ex_2^{115}) < time(ex_1^{125})$ and $time(ex_1^{115}) < time(ex_2^{125})$*

5.2 Proof of Strong Linearizability

Notice that we cannot use Aspnes and Herlihy's linearization function (i.e. topological orderings of linearization graphs) to prove strong linearizability, since this function is not prefix preserving. This is because operations may be written to the "middle" of the linearization graph; we clarify this argument with an example: Consider a transcript T containing only the operations ex and ex'. Suppose $invoc(ex')$ dominates $invoc(ex)$. While ex is pending, suppose ex' writes its constructed node to *root*. Let T' be the prefix of T that ends with the response of the *root.update* operation by ex' on line 125. Suppose that ex completes its operation in T. Let G' be the precedence graph induced by $\mathcal{G}(T')$, and let G be the precedence graph

induced by $\mathcal{G}(T)$. Then $lingraph(G')$ only contains $oper(ex')$, while $lingraph(G)$ contains both operations along with a dominance edge from $oper(ex)$ to $oper(ex')$. Hence, $oper(ex')$ is the only topological ordering of $lingraph(G')$, while $oper(ex) \circ oper(ex')$ is the only topological ordering of $lingraph(G)$. This demonstrates how operations can unavoidably be written to the middle of the linearization order. For this reason, we must define our own linearization function for Algorithm 8.

Let $T \in \mathcal{T}$ be some transcript. Let $ex_1, \ldots, ex_k$ be a sequence consisting of all operations in $\Gamma(T)$, such that for any $i, j \in \{1, \ldots, k\}$ with $i < j$, $invoc(ex_j)$ does not dominate $invoc(ex_i)$ (such an ordering is guaranteed to exist, since dominance is a strict partial order). For every $i \in \{1, \ldots, k\}$, let $pt_T(ex_i)$ be defined inductively as follows:

I-1 If $i = 1$, then define

$$pt_T(ex_i) = time(ex_i^{125}).$$

I-2 If $i > 1$, then let Dom_i be the set of all operations ex_h such that $h < i$ and $invoc(ex_h)$ dominates $invoc(ex_i)$. Define

$$pt_T(ex_i) = \min\Big(\big\{pt(ex_h) \ : \ ex_h \in Dom_i \wedge time(ex_i^{115}) < pt(ex_h)\big\} \cup \big\{time(ex_i^{125})\big\}\Big)$$

In I-2, we emphasize that $Dom_i \subseteq \{ex_1, \ldots ex_{i-1}\}$; this ensures that I-1 and I-2 together form a proper inductive definition. However, note that if ex_h dominates ex_i, then it is guaranteed that $h < i$ since $ex_1, \ldots, ex_k$ is ordered by dominance. That is, every operation whose invocation dominates $invoc(ex_i)$ must be earlier in the sequence $ex_1, \ldots, ex_k$ than ex_i. Therefore, the following property is guaranteed for any $ex \in \Gamma(T)$:

$$\begin{aligned} pt_T(ex) = \min\Big(&\big\{time(ex^{125})\big\} \cup \\ &\big\{pt(ex') \ : \ invoc(ex') \text{ dominates } invoc(ex) \wedge time(ex^{115}) < pt(ex')\big\}\Big). \end{aligned} \tag{5.13}$$

To simplify our proofs, we decompose property (5.13) into the following two rules:

J-1 If there exists an operation $ex' \in \Gamma(T)$ such that $invoc(ex')$ dominates $invoc(ex)$ and $time(ex^{115}) < pt_T(ex') < pt_T(ex^{125})$, then let ex_0 be such an operation with minimal pt_T value. Then $pt_T(ex) = pt_T(ex_0)$.

J-2 If no such operation ex' exists, then $pt_T(ex) = time(ex^{125})$. Recall that if $ex^{125} \notin T$, then $time(ex^{125}) = \infty$.

If $pt_T(ex) \neq \infty$ for some operation ex, then we say ex *linearizes* at $pt_T(ex)$. Also, if T is clear from context, then we shorten $pt_T(ex)$ to $pt(ex)$.

Let $f(T)$ be a sequential history consisting of all and only those operations $ex_1, ex_2 \in \Gamma(T)$ by processes q and p, respectively, with $pt(ex_1) \neq \infty$ and $pt(ex_2) \neq \infty$, such that $ex_1 \xrightarrow{f(T)} ex_2$ if and only if

K-1 operation ex_1 linearizes before ex_2 (i.e. $pt(ex_1) < pt(ex_2)$), or

K-2 operations ex_1 and ex_2 linearize at the same time (i.e. $pt(ex_1) = pt(ex_2)$) and $invoc(ex_2)$ dominates $invoc(ex_1)$, or

K-3 operations ex_1 and ex_2 linearize at the same time (i.e. $pt(ex_1) = pt(ex_2)$), $invoc(ex_2)$ does not dominate $invoc(ex_1)$, $invoc(ex_1)$ does not dominate $invoc(ex_2)$, and $q < p$.

Since dominance is a strict partial order [4], K-2 and K-3 impose a total order on the set of operations that linearize at any step of T.

Observation 44. *For every operation $ex \in \Gamma(T)$, $pt(ex) \in \big(time(ex^{115}), time(ex^{125})\big]$.*

Proof. If $pt(ex)$ satisfies J-1, then $pt(ex) \in \big(time(ex^{115}), time(ex^{125})\big)$ explicitly. Otherwise, if $pt(ex)$ satisfies J-2, then $pt(ex) = time(ex^{125})$. □

Observation 45. *Let $ex_1 \in \Gamma(T)$ be an operation. If there exists an operation $ex_2 \in \Gamma(T)$ such that $time(ex_1^{115}) < pt(ex_2)$ and $invoc(ex_2)$ dominates $invoc(ex_1)$, then $pt(ex_1) \leq pt(ex_2)$.*

Proof. This is trivial if $pt(ex_2) = \infty$. If $time(ex_1^{125}) \leq pt(ex_2)$, then the observation follows from Observation 44.

Suppose $pt(ex_2) < time(ex_1^{125})$. Let $ex_3 \in \Gamma(T)$ be an operation such that $time(ex_1^{115}) < pt(ex_3) < time(ex_1^{125})$, $invoc(ex_3)$ dominates $invoc(ex_1)$, and $invoc(ex_3)$ has the lowest possible pt value of any such operation. Hence, $pt(ex_3) \leq pt(ex_2)$, and $pt(ex_1) = pt(ex_3)$ by J-1. □

Observation 46. *Suppose ex_1 is an operation such that $pt(ex_1) < time(ex_1^{125})$. Then there exists an operation ex_2 such that $invoc(ex_2)$ dominates $invoc(ex_1)$ and $pt(ex_1) = pt(ex_2) = time(ex_2^{125})$.*

Proof. Since dominance is a strict partial order, there is a maximal element in the set of operations that linearize at $pt(ex_1)$. That is, there exists an operation ex_2 with $pt(ex_2) = pt(ex_1)$, such that there is no operation ex_3 with $pt(ex_3) = pt(ex_1)$ and $invoc(ex_3)$ dominates $invoc(ex_2)$. Therefore, $pt(ex_2)$ does not satisfy J-1. Hence, $pt(ex_2)$ must satisfy J-2, meaning $pt(ex_2) = time(ex_2^{125})$. □

Lemma 47. *Let $T \in \mathcal{T}$ be a transcript, where G is the precedence graph induced by $\mathcal{G}(T)$. Suppose $ex_1, ex_2 \in \Gamma(T)$ are operations such that*

$$oper(ex_1) \text{ and } oper(ex_2) \text{ are concurrent in } G, \tag{5.14}$$

$$invoc(ex_1) \text{ dominates } invoc(ex_2), \text{ and} \tag{5.15}$$

$$ex_1 \xrightarrow{f(T)} ex_2. \tag{5.16}$$

Then there exists an operation $ex_3 \in \Gamma(T)$ such that $invoc(ex_3)$ dominates $invoc(ex_1)$ and $oper(ex_3)$ precedes $oper(ex_2)$ in G.

Proof. If $pt(ex_1) = pt(ex_2)$, then $ex_2 \xrightarrow{f(T)} ex_1$ by K-2. This contradicts (5.16), so $pt(ex_1) \neq pt(ex_2)$. If $pt(ex_1) > pt(ex_2)$, then $ex_2 \xrightarrow{f(T)} ex_1$ by K-1. Again this contradicts (5.16), so

$$pt(ex_1) < pt(ex_2). \tag{5.17}$$

By (5.14) and Observation 43

$$time(ex_1^{115}) < time(ex_2^{125}), \text{ and} \tag{5.18}$$

$$time(ex_2^{115}) < time(ex_1^{125}). \tag{5.19}$$

Suppose that $time(ex_2^{115}) < pt(ex_1)$. Using this and (5.15), $pt(ex_2) \leq pt(ex_1)$ by Observation 45. This contradicts (5.17). Hence,

$$pt(ex_1) < time(ex_2^{115}). \tag{5.20}$$

By (5.19), (5.20), and Observation 46, there exists an operation ex_3 such that

$$invoc(ex_3) \text{ dominates } invoc(ex_1), \text{ and} \tag{5.21}$$

$$pt(ex_3) = pt(ex_1) = time(ex_3^{125}). \tag{5.22}$$

By (5.20) and (5.22), $time(ex_3^{125}) < time(ex_2^{115})$, which implies that $oper(ex_3)$ precedes $oper(ex_2)$ in G by Observation 42. This, combined with (5.21) and (5.22), imply the statement of the lemma. □

The key result of Aspnes and Herlihy [4] is stated below (note that we have combined two results from [4] — specifically, Lemma 11 and Theorem 17):

Lemma 48. *For any transcript $T \in \mathcal{T}$, where G is the precedence graph induced by $\mathcal{G}(T)$, any topological ordering of $lingraph(G)$ is a linearization of $\Gamma(T)$.*

We proceed to show that f is a linearization function by demonstrating that if $f(T) = ex_1 \circ \ldots \circ ex_k$, then $oper(ex_1) \circ \ldots \circ oper(ex_k)$ is a topological ordering of $lingraph(G)$, where G is the precedence graph induced by $\mathcal{G}(T)$, for any transcript $T \in \mathcal{T}$. There are two facts that make our task nontrivial. First, $f(T)$ may contain operations whose constructed nodes are not present in $\mathcal{G}(T)$. We claim that these operations are overwritten before they are

added to the shared precedence graph, and the responses that are eventually calculated for each of these operations are valid for their position in the linearization order. Second, it is not immediately apparent that the order of $oper(ex_1) \circ \ldots \circ oper(ex_k)$ satisfies the dominance order that is present in a topological ordering of $lingraph(G)$.

The following lemma is taken directly from Aspnes and Herlihy [4].

Lemma 49. *Let $T \in \mathcal{T}$ be a transcript, where G is the precedence graph induced by $\mathcal{G}(T)$, and let $L_1 \circ oper(ex_1) \circ L_2$ be a topological sort of $lingraph(G)$. If there exists $oper(ex_2) \in L_2$ that is concurrent with $oper(ex_1)$ in G, and $oper(ex_1)$ dominates $oper(ex_2)$, then there is $oper(ex_3) \in L_2$ such that $invoc(ex_3)$ dominates $invoc(ex_1)$ and $oper(ex_3)$ precedes $oper(ex_2)$ in G.*

Lemma 50. *Let $T \in \mathcal{T}$ be a transcript, where G is the precedence graph induced by $\mathcal{G}(T)$, and let L be some topological ordering of $lingraph(G)$. There is no pair of operations $ex_1, ex_2 \in f(T)$ such that*

$$ex_1 \xrightarrow{f(T)} ex_2, \tag{5.23}$$

$$oper(ex_2) \xrightarrow{L} oper(ex_1), \text{ and} \tag{5.24}$$

$$invoc(ex_1) \text{ dominates } invoc(ex_2). \tag{5.25}$$

Proof. Suppose there exists a pair of operations $ex_1, ex_2 \in f(T)$ satisfying (5.23)-(5.25), and let ex_1 be the first operation in $f(T)$ for which these properties are satisfied. That is,

$$\text{there is no pair of operations } ex'_1, ex'_2 \in f(T) \text{ that satisfy (5.23)-(5.25), such that } ex'_1 \xrightarrow{f(T)} ex_1. \tag{5.26}$$

If $oper(ex_2)$ precedes $oper(ex_1)$ in G, then $time(ex_2^{125}) < time(ex_1^{115})$ by Observation 42. But by Observation 44 this would imply that $pt(ex_2) < pt(ex_1)$, which would mean $ex_2 \xrightarrow{f(T)} ex_1$

by K-1. This contradicts (5.23). Hence,

$$oper(ex_2) \text{ does not precede } oper(ex_1) \text{ in } G. \tag{5.27}$$

If $oper(ex_1)$ precedes $oper(ex_2)$ in G, then this contradicts (5.24), since L is a topological sort of $lingraph(G)$. Therefore,

$$oper(ex_1) \text{ does not precede } oper(ex_2) \text{ in } G. \tag{5.28}$$

By (5.27) and (5.28), $oper(ex_1)$ and $oper(ex_2)$ are concurrent in G. By this, (5.23), (5.25), and Lemma 47 there exists an operation ex_j such that

$$invoc(ex_j) \text{ dominates } invoc(ex_1), \tag{5.29}$$

$$pt(ex_j) = pt(ex_1) = time(ex_j^{125}), \text{ and} \tag{5.30}$$

$$oper(ex_j) \text{ precedes } oper(ex_2) \text{ in } G. \tag{5.31}$$

By (5.29), (5.30), and K-2,

$$ex_1 \xrightarrow{f(T)} ex_j \tag{5.32}$$

By (5.31) and the fact that L is a topological sort of $lingraph(G)$,

$$oper(ex_j) \xrightarrow{L} oper(ex_2). \tag{5.33}$$

By Observation 44, $time(ex_j^{115}) < pt(ex_j) \leq time(ex_j^{125})$ and $time(ex_1^{115}) < pt(ex_1) \leq time(ex_1^{125})$. This along with (5.30) implies that $time(ex_j^{115}) < time(ex_1^{125})$ and $time(ex_1^{115}) < time(ex_j^{125})$. By this and Observation 43,

$$oper(ex_j) \text{ and } oper(ex_1) \text{ are concurrent in } G. \tag{5.34}$$

By (5.24), (5.33), and the transitivity of happens-before order,

$$oper(ex_j) \xrightarrow{L} oper(ex_1). \tag{5.35}$$

By (5.29), (5.34), (5.35), and Lemma 49, there exists an operation $oper(ex_i)$ such that

$$oper(ex_j) \xrightarrow{L} oper(ex_i) \xrightarrow{L} oper(ex_1), \tag{5.36}$$

$$invoc(ex_i) \text{ dominates } invoc(ex_j), \text{ and} \tag{5.37}$$

$$oper(ex_i) \text{ precedes } oper(ex_1) \text{ in } G. \tag{5.38}$$

By (5.38) and Observation 42, $time(ex_i^{125}) < time(ex_1^{115})$. This along with Observation 44 implies that $pt(ex_i) < pt(ex_1)$. Hence, by K-1,

$$ex_i \xrightarrow{f(T)} ex_1. \tag{5.39}$$

By (5.32), (5.39), and the transitivity of happens-before order,

$$ex_i \xrightarrow{f(T)} ex_j. \tag{5.40}$$

Together, (5.36), (5.37), and (5.40) imply that ex_i and ex_j satisfy (5.23)-(5.25). By (5.39), this contradicts (5.26). Hence, there is no pair of operations that satisfies (5.23)-(5.25). □

Lemma 51. *The function f is prefix-preserving.*

Proof. Let $T \in \mathcal{T}$ be a transcript. Consider each step t of T, and some operation $ex \in \Gamma(T)$ by p. By Observation 44 are two cases:

(i) Suppose $pt(ex) < time(ex^{125})$. Then by Observation 46, there exists an operation ex_ℓ such that $pt(ex) = pt(ex_\ell) = time(ex_\ell^{125})$. Hence, $pt(ex) = t$ if and only if $time(ex_\ell^{125}) = t$.

(ii) Suppose $pt(ex) = time(ex^{125})$. Then $pt(ex) = t$ if and only if $time(ex^{125}) = t$.

Thus, at step t it is entirely determined which operations linearize at t. That is, whether t satisfies $t = pt(op)$ can be deduced solely by examining the step at time t of T. By this and the fact that K-2 and K-3 impose a total order on the set of operations that linearize at each point in time in T, f is prefix-preserving. □

We now address the fact that $f(T)$ may contain operations that are not in the precedence graph G induced by $\mathcal{G}(T)$. We aim to show that if $f(T) = ex_1 \circ \ldots \circ ex_k$, then $oper(ex_1) \circ \ldots \circ oper(ex_k)$ is a topological ordering of G; this does not hold if $oper(ex_1) \circ \ldots \circ oper(ex_k)$ contains operations that are not present in G. To resolve this issue, we define a notion of a "completion" of a transcript (recall that completion is only defined for histories), which we call $fill$. For any transcript $T \in \mathcal{T}$, $fill(T)$ is constructed by allowing every incomplete operation present in $f(T)$ to finish (see below for a precise definition). If $ex \in f(T)$ is an operation by p, and ex is incomplete in T, then let the p-solo *completion* of ex in T be the transcript T_p such that $(T \circ T_p) \in \mathcal{T}$, T_p contains only steps by p, and the final step of T_p is $rsp(ex)$. Note that the existence of T_p is guaranteed by the fact that Algorithm 8 is wait-free. Additionally, since $ex \in f(T)$, $pt(ex) \neq \infty$, and therefore $ex^{115} \in T$ by Observation 44. Since the response of ex is entirely determined by the vector returned by ex^{115} (by examination of the *execute* method of Algorithm 8), the response of ex is entirely determined in T (that is, no future steps by any process can change the response of ex). Therefore, if $(T \circ T') \in \mathcal{T}$ and T' contains no steps by p, then $(T \circ T' \circ T_p) \in \mathcal{T}$.

For any transcript $T \in \mathcal{T}$, let $fill(T)$ be a transcript constructed from T as follows:

F-1 Initially, let $T' = T$.

F-2 For every operation $ex \in f(T)$ by process p that is incomplete in T, append the p-solo completion of ex in T to T'. That is, if T_p is the p-solo completion of ex in T, then let $T' = T' \circ T_p$.

F-3 Let $fill(T) = T'$.

Observation 52. *For any $T \in \mathcal{T}$,*

(a) $f(T) = f\big(fill(T)\big)$*, and*

(b) if $f(T) = ex_1 \circ \ldots \circ ex_k$*, then the precedence graph* G' *induced by* $\mathcal{G}\big(fill(T)\big)$ *contains all and only those operations in* $\big\{oper(ex_1), \ldots, oper(ex_k)\big\}$.

Proof. Let $T \in \mathcal{T}$, and for ease of notation let $T' = fill(T)$. Since F-2 only appends steps in the construction of T', T is a prefix of T'. By this and Lemma 51,

$$f(T) \text{ is a prefix of } f(T'). \tag{5.41}$$

Suppose $f(T) \neq f(T')$. By this and (5.41), $f(T') = f(T) \circ ex_{\ell_1} \circ \ldots \circ ex_{\ell_k}$, for some nonempty sequence of operations $ex_{\ell_1}, \ldots, ex_{\ell_k}$ with

$$ex_{\ell_i} \notin f(T) \text{ for all } i \in \{1, \ldots, k\}. \tag{5.42}$$

Due to (5.42) and the fact that F-2 only adds steps of operations in $f(T)$,

$$\text{no steps of } ex_{\ell_i} \text{ are added in the construction of } T' \text{, for any } i \in \{1, \ldots, k\}. \tag{5.43}$$

Suppose $pt_{T'}(ex_{\ell_1}) = time_{T'}(ex_{\ell_1}^{125})$. Since $ex_{\ell_1} \in f(T')$, $pt_{T'}(ex_{\ell_1}) \neq \infty$, and therefore $time_{T'}(ex_{\ell_1}^{125}) \neq \infty$. By this and (5.43), $ex_{\ell_1}^{125} \in T$. Then $ex_{\ell_1} \in f(T)$, which is a contradiction. Therefore, $pt_{T'}(ex_{\ell_1}) < time_{T'}(ex_{\ell_1}^{125})$. Then by Observation 46 there exists an operation $ex_\alpha \in \Gamma(T')$ such that

$$pt_{T'}(ex_{\ell_1}) = pt_{T'}(ex_\alpha) = time_{T'}(ex_\alpha^{125}) \text{ in } T' \text{, and} \tag{5.44}$$

$$invoc(ex_\alpha) \text{ dominates } invoc(ex_{\ell_1}). \tag{5.45}$$

By (5.44), (5.45), and K-2,

$$ex_{\ell_1} \xrightarrow{f(T')} ex_\alpha. \tag{5.46}$$

Since $pt_{T'}(ex_{\ell_1}) \neq \infty$, $time_{T'}(ex_\alpha^{125}) \neq \infty$ by (5.44). This, along with (5.43), implies that

$ex_\alpha^{125} \in T$. Hence, $ex_\alpha \in f(T)$. This, along with (5.41) and (5.46) imply that $ex_{\ell_1} \in f(T)$, which is a contradiction. Therefore, we have arrived at a contradiction in all cases, and our initial supposition is false. That is, $f(T) = f(T')$, which concludes the proof of part (a).

Let G' be the precedence graph induced by $\mathcal{G}(T')$. During the construction of T', F-2 ensures that, for every operation $ex \in f(T)$, ex is complete. In particular,

$$\text{for every operation } ex \in f(T),\ ex^{125} \in T'. \tag{5.47}$$

By definition, $\mathcal{G}(T')$ contains every $node(ex)$ such that $ex^{125} \in T'$, and therefore G' contains every $oper(ex)$ such that $ex^{125} \in T'$. This, together with (5.47), implies part (b). □

Lemma 53. *For any transcript $T \in \mathcal{T}$, $f(T)$ is a linearization of $\Gamma(T)$.*

Proof. Let $T' = fill(T)$, and let G' be the precedence graph induced by $\mathcal{G}(T')$. Suppose $f(T') = ex_1 \circ \ldots \circ ex_k$. Notice that the only difference between the sequences $ex_1, \ldots, ex_k$ and $oper(ex_1), \ldots, oper(ex_k)$ are operation identifiers, so $f(T')$ is equivalent to the history $oper(ex_1) \circ \ldots \circ oper(ex_k)$. Hence, if we prove that $oper(ex_1) \circ \ldots \circ oper(ex_k)$ is a linearization of $\Gamma(T')$, this implies that $f(T')$ is also a linearization of $\Gamma(T')$. To accomplish this, it suffices to demonstrate that $oper(ex_1) \circ \ldots \circ oper(ex_k)$ is a topological ordering of $lingraph(G')$ by Lemma 48. More specifically, we show that the sequential history $oper(ex_1) \circ \ldots \circ oper(ex_k)$ has no "back-edges"; that is, there are no edges $\big(oper(ex_j), oper(ex_i)\big)$ in $lingraph(G')$ such that $j > i$.

First note that by Observation 52 (b), G' contains all and only those operations $oper(ex_i)$ such that $ex_i \in f(T)$. Since $f(T) = f(T')$ by Observation 52 (a), this implies that

$$ex \in f(T') \text{ if and only if } oper(ex) \text{ is in } G'. \tag{5.48}$$

Let L be a topological ordering of $lingraph(G')$. Let $i, j \in \{1, \ldots, k\}$ with $i < j$. Hence,

$$ex_i \xrightarrow{f(T')} ex_j. \tag{5.49}$$

By (5.48), $oper(ex_i), oper(ex_j)$ are in G', and hence $oper(ex_i), oper(ex_j)$ are in $lingraph(G')$. Suppose that

$$\text{there exists an edge from } oper(ex_j) \text{ to } oper(ex_i) \text{ in } lingraph(G'). \tag{5.50}$$

Suppose $oper(ex_j)$ precedes $oper(ex_i)$ in G'. Then $time(ex_j^{125}) < time(ex_i^{115})$ by Observation 42. This implies that $pt(ex_j) < pt(ex_i)$ by Observation 44. By K-1, $ex_j \xrightarrow{f(T')} ex_i$, which contradicts (5.49). Therefore,

$$oper(ex_j) \text{ does not precede } oper(ex_i) \text{ in } G'. \tag{5.51}$$

By (5.50) and the fact that L is a topological ordering of $lingraph(G')$,

$$oper(ex_j) \xrightarrow{L} oper(ex_i). \tag{5.52}$$

Due to (5.50) and (5.51), there must be a dominance edge from $oper(ex_j)$ to $oper(ex_i)$ in $lingraph(G')$. That is,

$$invoc(ex_i) \text{ dominates } invoc(ex_j). \tag{5.53}$$

But (5.49), (5.52), and (5.53) contradict Lemma 50. Hence, no "back-edges" exist in $lingraph(G')$ among the sequential history $oper(ex_1) \circ \ldots \circ oper(ex_k)$. Along with (5.48), this implies that $oper(ex_1) \circ \ldots \circ oper(ex_k)$ is a topological ordering of $lingraph(G')$. By Lemma 48, $oper(ex_1) \circ \ldots \circ oper(ex_k)$ is a linearization of $\Gamma(T')$. Since $oper(ex_1) \circ \ldots \circ oper(ex_k)$ and $f(T')$ are equivalent, $f(T')$ is also a linearization of $\Gamma(T')$. By Observation 52 (a),

$$f(T) \text{ is a linearization of } \Gamma(T'). \tag{5.54}$$

By construction of T', every operation $ex \in f(T)$ is present and complete in T' (and in

$\Gamma(T'))$, and every complete operation $ex \in \Gamma(T')$ is in $f(T)$ by Observation 44. That is,

$$ex \in \Gamma(T') \text{ is incomplete if and only if } ex \notin f(T). \tag{5.55}$$

Let H_C be the completion of $\Gamma(T')$ obtained by removing all incomplete operations from $\Gamma(T')$. By (5.55),

$$H_C \text{ and } f(T) \text{ contain precisely the same operations.} \tag{5.56}$$

Since H_C is a completion of $\Gamma(T')$, $ex_1 \xrightarrow{H_C} ex_2$ implies that $ex_1 \xrightarrow{\Gamma(T')} ex_2$. By this, (5.54), and (5.56), $f(T)$ is a linearization of H_C. Since the construction of T' adds no new "high-level" invocations (i.e. invocations that are not present in $\Gamma(T)$) to $\Gamma(T')$, H_C is also a completion of $\Gamma(T)$. Hence, since $f(T)$ is a linearization of H_C, $f(T)$ is a linearization of $\Gamma(T)$. □

Theorem 54. *Algorithm 8 is strongly linearizable.*

Proof. Lemma 53 shows that f is a linearization function for $\Gamma(\mathcal{T})$, and Lemma 51 shows that f is prefix preserving. Hence, f is a strong linearization function for $\mathcal{T}$. □

Theorems 2 and 54, along with the fact that strong linearizability is composable, yield Theorem 3.

5.3 Remarks

In this chapter, we proved that Aspnes and Herlihy's general construction for simple types [4] is strongly linearizable. Typically, proving strong linearizability is only marginally more difficult than proving linearizability; as long as one chooses linearization points for each operation carefully, it is easy to show that a linearization function satisfies the prefix-preservation property. With this claim in mind, the length of this chapter might be perplexing, since Aspnes and Herlihy have already shown that Algorithm 8 is linearizable [4]. However, their

linearization function is inherently not prefix-preserving, since operations may be written to the "middle" of the linearization graph (as discussed at the beginning of Section 5.2). Hence, in Section 5.2 we were forced to start from scratch, defining our own linearization function that satisfies the prefix-preservation property. A considerable amount of effort was also spent on proving that the shared snapshot object *root* always contains a representation of a particular precedence graph. This is an invariant that was taken for granted in [4]; for the sake of completeness, we decided to prove it formally in Section 5.1.

We also note that Aspnes and Herlihy's construction requires unbounded memory, since operations are never removed from the shared precedence graph. This is unfortunate, since the general construction does not benefit from the fact that our snapshot implementation requires only bounded space. Aspnes and Herlihy claim that "for any particular data type, it should be possible to apply type-specific optimizations to discard most of the precedence graph" [4]. It would be interesting to study types for which such a simplification could be used to bound the space of Aspnes and Herlihy's construction. Moreover, it may be possible to adjust Aspnes and Herlihy's algorithm so that it only requires bounded space for any simple type. For instance, this might be achieved by storing states, rather than operations, inside the nodes of a precedence graph.

6. Dynamic Resiliency

Anderson and Moir [24] define a class of *readable* types; such types support a single invocation description that returns the current state (i.e. the *Read* invocation), along with a set of *Update* invocations that do not return anything and may modify the state of the object. That is, if $\mathscr{T} = (\mathcal{S}, s_0, \mathcal{O}, \mathcal{R}, \delta)$ is a readable type, then $\mathcal{R} = \mathcal{S} \cup \{\bot\}$, and for every state $s \in \mathcal{S}$,

(a) $\delta(read, s) = (s, s)$, where $read \in \mathcal{O}$ is the *Read* invocation supported by $\mathscr{T}$, and

(b) $\delta(up, s) = (\bot, s')$ for every invocation $up \in \mathcal{O} \setminus \{read\}$, where $s' \in \mathcal{S}$ is some state, and $\bot \in \mathcal{R}$ is a special value denoting the empty response.

Note that since *Update* operations do not return anything, for any object O of a readable type, if H is a valid sequential history on O, then for any *Update* operation up on O, $H \circ up$ is also valid.

Anderson and Moir define a *dynamic resiliency condition*, provided below, and proceed to claim that this condition is both necessary and sufficient for the existence of a wait-free implementation of a readable type from atomic registers. The dynamic resiliency condition requires different definitions for *commute* and *overwrite* than those used by Aspnes and Herlihy. Let $\mathscr{T} = (\mathcal{S}, s_0, \mathcal{O}, \mathcal{R}, \delta)$ be a type. If σ and τ are sequences of invocation descriptions, then let $\sigma \circ \tau$ denote the concatenation of σ and τ. For any sequence of invocation descriptions $\tau \in \mathcal{O}^*$, the state $\langle \tau \rangle \in \mathcal{S}$ may be defined inductively:

$$\langle \tau \rangle = \begin{cases} s_0 & \text{if } \tau \text{ is empty} \\ s & \text{if } \tau = \tau' \circ invoke \text{ and } \delta(invoke, \langle \tau' \rangle) = (s, resp) \end{cases}$$

The invocation description $invoke_1$ *commutes* with the invocation description $invoke_2$ after τ if $\langle \tau \circ invoke_1 \circ invoke_2 \rangle = \langle \tau \circ invoke_2 \circ invoke_1 \rangle$. The invocation description $invoke_2$ *overwrites*

the invocation description $invoke_1$ after τ if $\langle \tau \circ invoke_1 \circ invoke_2 \rangle = \langle \tau \circ invoke_2 \rangle$. As before, we use the notion of dominance to break ties between mutually overwriting invocations; that is, the invocation description $invoke_2$ by operation p *dominates* the invocation description $invoke_1$ by operation q if either $invoke_2$ overwrites $invoke_1$ and not vice versa, or both invocations overwrite each other and $p > q$.

Definition 55. *Type* $\mathscr{T} = (\mathcal{S}, s_0, \mathcal{O}, \mathcal{R}, \delta)$ *is* dynamically resilient *if, for every sequence of invocation descriptions* $\tau \in \mathcal{O}^*$ *and for every pair of invocations* $invoke_1$ *and* $invoke_2$ *by different processes, either* $invoke_1$ *and* $invoke_2$ *commute after* τ*, or one overwrites the other after* τ*.*

The relationships between pairs of invocations on a dynamically resilient type can change; that is, after a particular sequence τ two invocations may commute, while after another sequence τ' these same two invocations may no longer commute, while instead one overwrites the other. The main result claimed by Anderson and Moir [24] is that a readable type has a wait-free implementation from registers if and only if it is dynamically resilient. They claim that "the sufficiency of this condition follows from results presented by Aspnes and Herlihy in [4], where it is shown that any object satisfying the condition has an unbounded, wait-free implementation".

We claim that the dynamic resiliency condition defined by Anderson and Moir is different from the definition of *simple* types, originally described by Aspnes and Herlihy [4], and described in Chapter 5 of this thesis. Recall that, according to Aspnes and Herlihy, a pair of invocation events $inv(op)$ and $inv(op')$ commute if, for all sequential histories H, $H \circ op \circ op'$ and $H \circ op' \circ op$ are valid and equivalent, provided $H \circ op$ and $H \circ op'$ are valid. That is, two invocations commute if the state obtained after both invocations have been applied consecutively is independent of the order of the two invocations. In Aspnes and Herlihy's definition, this relationship between invocations remains the same regardless of the starting state. By contrast, Anderson and Moir define commutativity on a state-by-state basis. There is a similar distinction between the definitions of overwriting operations.

Aspnes and Herlihy's general construction is not well-defined for dynamically resilient types that are not simple. During the construction of a linearization graph, a process adds dominance edges between pairs of operations; that is, for every pair of operations $oper(ex_1)$ and $oper(ex_2)$ in the precedence graph, a dominance edge is added from $oper(ex_1)$ to $oper(ex_2)$ if $oper(ex_2)$ dominates $oper(ex_1)$ and the addition of the edge does not create a cycle in the resulting graph. It is not clear how a linearization graph might be constructed to account for the changing relationships between operations that can occur in a dynamically resilient type.

Algorithm 10: Modified linearization graph construction

```
    Function getlinearization(G):
160     H is an empty sequential history
161     while G is not empty do
162         L ← lingraph(G)
163         op ← node of L with no incoming edges
164         H ← H ∘ op
165         G ← G − op
166     end
167     return H
```

One might consider a modification of Aspnes and Herlihy's algorithm that reconstructs the linearization graph every time an operation is added to the topological ordering. That is, instead of simply topologically sorting $lingraph(G)$ to calculate a linearization, line 117 of Algorithm 8 could be replaced with $H \leftarrow getlinearization(G)$, where *getlinearization* is implemented as in Algorithm 10. Using this strategy, the structure of the linearization graph changes as operations are added to H.

As in Algorithm 8, let G be the precedence graph returned by the $precgraph(view)$ operation on line 116. A process p performing a $getlinearization(G)$ operation begins by initializing an empty history H on line 160. Following this, p begins a while-loop on line 161; at the beginning of the loop, p calculates a linearization graph L using $lingraph(G)$ on line 162. Next, on line 163 p identifies an operation op in the linearization graph L such that op has no incoming edges. This operation op is appended to the linearization H on line 164. Finally, p removes op from the precedence graph G on line 165 (note that this also removes

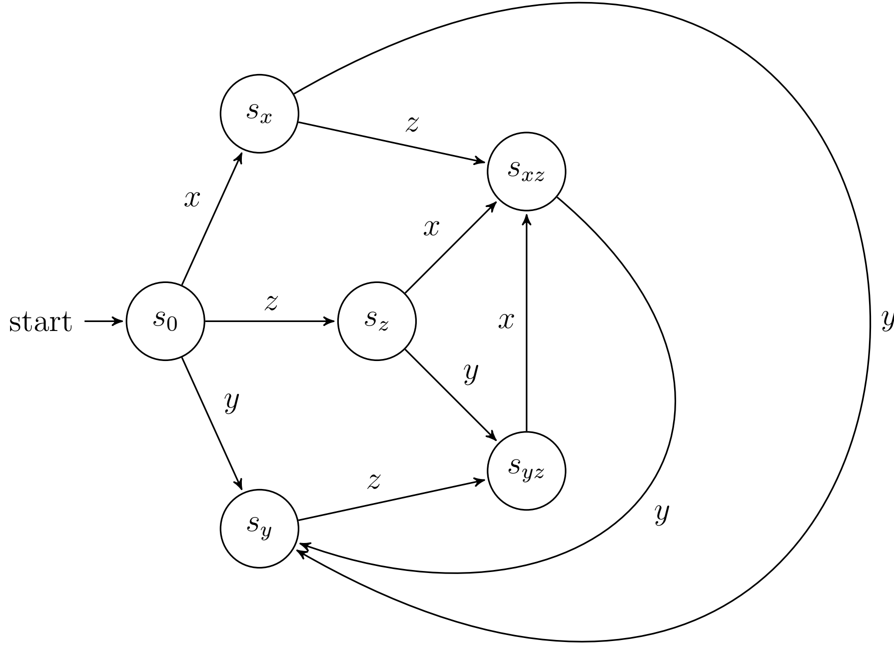

Figure 6.1: The state machine representation of the type $\mathscr{T}_D$. Assume that all transitions that are not drawn are self-loops. Type $\mathscr{T}_D$ also has a *Read* invocation description (called *read*), which is not displayed.

all outgoing edges (op, op') from G, for any operation op'). The while-loop continues until all operations have been removed from G.

Suppose that, immediately following some execution of line 162, $H = oper(ex_1) \circ \ldots \circ oper(ex_k)$. Then at this point, the linearization graph L contains dominance edges that are consistent with the state $\langle invoc(ex_1) \circ \ldots \circ invoc(ex_k)\rangle$; that is, there is a dominance edge from $oper(ex_i)$ to $oper(ex_j)$ in L only if $oper(ex_j)$ dominates $oper(ex_i)$ in state $\langle invoc(ex_1) \circ \ldots \circ invoc(ex_k)\rangle$.

However, this modified algorithm does not preserve the dominance order of operations as nodes are removed from the graph. Suppose that there are two concurrent operations $oper(ex_1)$ and $oper(ex_2)$ in a precedence graph, and there is a dominance edge from $oper(ex_1)$ to $oper(ex_2)$ in the initial state. Let $oper(ex_3)$ be an operation that is concurrent with both

$oper(ex_1)$ and $oper(ex_2)$ such that, after applying $invoc(ex_3)$, $oper(ex_1)$ overwrites $oper(ex_2)$, but $oper(ex_2)$ does not overwrite $oper(ex_1)$ (so $oper(ex_3)$ effectively flips the dominance order between $oper(ex_1)$ and $oper(ex_2)$). Hence, if a process obtains a precedence graph G on line 116 that contains $oper(ex_1)$, $oper(ex_2)$, and $oper(ex_3)$, then the linearization constructed using the $getlinearization(G)$ operation depends on the relative order of $oper(ex_1)$ and $oper(ex_3)$. Notice that, since $oper(ex_1)$ and $oper(ex_3)$ commute in the initial state, there is no order imposed upon them by the *getlinearization* method. Suppose a process p performs a $getlinearization(G)$ operation; if $oper(ex_1)$ is the first operation selected by p on line 163, then p's $getlinearization(G)$ operation will either return $oper(ex_1) \circ oper(ex_2) \circ oper(ex_3)$ or $oper(ex_1) \circ oper(ex_3) \circ oper(ex_2)$. However, if $oper(ex_3)$ is the first operation selected by p on line 163, then p's $getlinearization(G)$ operation will return $op(ex_3) \circ oper(ex_2) \circ oper(ex_1)$.

The type $\mathscr{T}_D$ depicted in Figure 6.1 is used to illustrate this problematic scenario more concretely. Type $\mathscr{T}_D$ supports four invocation descriptions: x, y, and z are *Update* invocations, while *read* is the *Read* invocation.

Theorem 56. *Type $\mathscr{T}_D$ is dynamically resilient, but is not simple.*

Proof. We prove that the type is dynamically resilient by showing that in each state, every pair of invocations either commute, or one overwrites the other. In Table 6.1, we use $u > v$ to denote that invocation u overwrites invocation v, and we use $u \sim v$ to denote that invocations u and v commute.

State	x and y	x and z	y and z
s_0	$y > x$ $x \sim y$	$x \sim z$	$y \sim z$
s_x	$y > x$ $x \sim y$	$z > x$ $x \sim z$	$y > z$
s_z	$x > y$	$x > z$ $x \sim z$	$y > z$ $y \sim z$
s_y	$x > y$ $y > x$ $x \sim y$	$z > x$	$z > y$
s_{xz}	$y > x$ $x \sim y$	$x > z$ $z > x$ $x \sim z$	$y > z$
s_{yz}	$x > y$	$x > z$ $x \sim z$	$y > z$ $z > y$ $y \sim z$

Table 6.1: The relationships between invocation descriptions in each state of the type depicted in Figure 6.1. Note that x, y, and z are *Update* invocation descriptions. Again, the invocation description *read* is not shown here; *read* is overwritten by every other invocation.

In state s_0, y overwrites x, x does not overwrite y, and y and x commute. In state s_z, y does not overwrite x, x overwrites y, and y and x do not commute. Hence, $\mathscr{T}_D$ is not simple. □

We now outline a problem that arises when type $\mathscr{T}_D$ is implemented by Algorithm 8 with our modification in Algorithm 10. We will describe a transcript T_D on this implementation and show that T_D has no linearization.

Let O be an instance of $\mathscr{T}_D$, implemented by Algorithm 8, along with our modification in Algorithm 10. Consider the transcript T_D on O depicted in Figure 6.2. The transcript consists of five operations: ex_x is an instance of invocation x, ex_y is an instance of invocation y, ex_z is an instance of invocation z, and ex_{r1} and ex_{r2} are instances of the *read* invocation. The

history contains operations by three processes: process p_1 performs operation ex_x, process p_2 performs operations ex_y and ex_{r1}, and process p_3 performs operations ex_z and ex_{r2}.

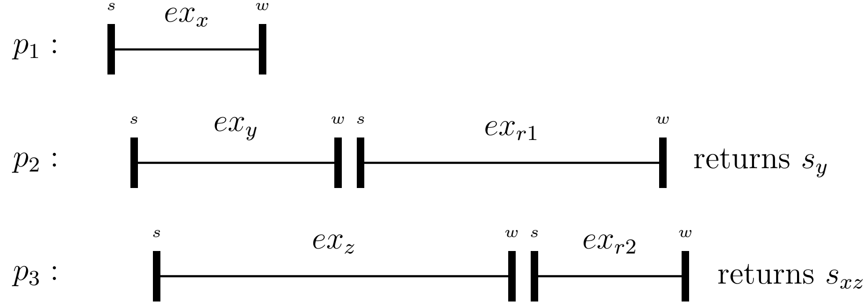

Figure 6.2: A diagram of the transcript T_D. The beginning of each interval represents the atomic *root.scan* operation from line 115 (labelled by s), while the end of each interval represents the atomic *root.update* operation from line 125 (labelled by w).

Consider the prefix T_{D_1} of T_D that ends with ex_{r1}^{115}. The graph constructed by ex_{r1} on line 116 is the precedence graph G_1 induced by $\mathcal{G}(T_{D_1})$. Notice that G_1 contains ex_x and ex_y. Additionally, notice that $time(ex_x^{115}) < time(ex_y^{125})$ and $time(ex_y^{115}) < time(ex_x^{125})$. Hence, by Observation 43, ex_x and ex_y are concurrent in G_1. Since y dominates x in the initial state, the linearization graph $lingraph(G_1)$ contains an edge from $oper(ex_x)$ to $oper(ex_y)$. Thus, ex_{r1} returns the state $\langle x \circ y \rangle = s_y$.

Now consider the prefix T_{D_2} of T_D that ends with ex_{r2}^{115}. The graph constructed by ex_{r2} on line 116 is the precedence graph G_2 induced by $\mathcal{G}(T_{D_2})$. Notice that G_2 contains ex_x, ex_y, and ex_z. Additionally, notice that ex_x, ex_y, and ex_z are pairwise concurrent in G_2. Since z commutes with both x and y in the initial state, in $lingraph(G_2)$ there are no edges between $oper(ex_x)$ and $oper(ex_z)$, and there are no edges between $oper(ex_y)$ and $oper(ex_z)$. Therefore, $oper(ex_z)$ has no incoming edges in $lingraph(G_2)$. Consider the computation of $getlinearization(G_2)$ by ex_{r2}; suppose that in the first iteration of the while-loop on line 161, p_3 selects $oper(ex_z)$ on line 163. During the second iteration of the loop, p_3 calculates $lingraph\big(G_2 - oper(ex_z)\big)$ on line 162. Since x dominates y in the state $\langle z \rangle = s_z$, there is a dominance edge from $oper(ex_y)$ to $oper(ex_x)$ in $lingraph\big(G_2 - oper(ex_z)\big)$. Hence, p_3 must

select $oper(ex_y)$ on line 163. Thus, the state returned by ex_{r2} is $\langle z \circ y \circ x \rangle = s_{xz}$.

We now attempt to calculate a linearization L_D of the history $\Gamma(T_D)$. Since ex_x and ex_y happen before ex_{r1} in T_D, ex_x and ex_y must happen before ex_{r1} in L_D. Similarly, ex_x, ex_y, and ex_z must all happen before ex_{r2} in L_D. We proceed by examining every possibility for L_D, based on the happens-before order of $\Gamma(T_D)$.

- Suppose ex_z happens before ex_{r1} in L_D. Then all of ex_x, ex_y, and ex_z must happen before ex_{r1} in L_D. Then $L_D = H_{xyz} \circ ex_{r1} \circ ex_{r2}$ or $L_D = H_{xyz} \circ ex_{r2} \circ ex_{r1}$, where H_{xyz} is some permutation of the operations ex_x, ex_y, and ex_z. However, none of these histories are valid, because any pair of adjacent *read* operations in a sequential history must return the same state (since *read* operations do not modify the state of the object). Hence, there is no linearization of $\Gamma(T_D)$ in which ex_z happens before ex_{r1}.
- Suppose ex_z happens after ex_{r1} in L_D. Then $L_D = ex_x \circ ex_y \circ ex_{r1} \circ ex_z \circ ex_{r2}$ or $L_D = ex_y \circ ex_x \circ ex_{r1} \circ ex_z \circ ex_{r2}$. But the response of ex_{r2} is not valid in either of these histories, since $\langle x \circ y \circ z \rangle = \langle y \circ x \circ z \rangle = s_{yz}$, but ex_{r2} returns s_{xz}. Hence, there is no linearization of $\Gamma(T_D)$ in which ex_z happens after ex_{r1}.

Hence, the history $\Gamma(T_D)$ has no linearization. This example demonstrates a general issue that arises with the "dynamic topological ordering" strategy used by Algorithm 10: some orderings of the input precedence graph may not be equivalent. In contrast, a key result of Aspnes and Herlihy is that every topological ordering of a linearization graph on a simple type is equivalent (Lemma 11 of [4]). Therefore, the sufficiency of the dynamic resiliency condition remains unproven.

6.1 Remarks

In this chapter we demonstrated that Anderson and Moir's notion of dynamic resiliency [24] is different from the definition of simple types by Aspnes and Herlihy [4]. If Anderson and Moir's claim was accurate, then we could conclude the following: a readable type T has

a wait-free strongly linearizable implementation from atomic snapshots if and only if T is dynamically resilient. However, our results suggest that there may be some dynamically resilient types that cannot be implemented from registers (while satisfying linearizability and wait-freedom). Additionally, if there exists a wait-free linearizable construction of implementations of dynamically resilient types, then we suspect that such a construction would differ considerably from Aspnes and Herlihy's construction (that is, we do not believe that there exists a trivial modification to Algorithm 8 that would accommodate dynamically resilient types). We would like to investigate the differences between dynamically resilient and simple types more closely, specifically in the domain of readable types. It would be especially exciting to learn that there is no non-simple type that has a wait-free linearizable implementation from registers, as this would immediately imply that any type with a wait-free linearizable implementation from registers also has a wait-free strongly linearizable implementation from snapshots.

7. Conclusion

We have provided a lock-free strongly linearizable implementation of a snapshot object using atomic multi-reader multi-writer registers as base objects. We used this implementation to demonstrate that any simple object also has a lock-free strongly linearizable implementation from atomic multi-reader multi-writer registers. The class of simple types seems to be a large subset of the types that have wait-free linearizable implementations from atomic registers. We are not aware of any classifications of non-simple types that are also known to have wait-free linearizable implementations from registers. Additionally, we have not yet explored the characteristics of objects that enable wait-free strongly linearizable implementations from registers.

As we briefly discussed in Chapter 5, the general construction defined by Aspnes and Herlihy [4] uses unbounded space. This is a result of the fact that each operation constructs a new node, and nodes added to the shared precedence graph are never reclaimed. The storage of unbounded precedence graphs also affects the liveness properties satisfied by the algorithm; while Algorithm 8 is wait-free, it is not *bounded* wait-free. That is, there is no constant that bounds the number of steps required by any operation. This is because each operation must calculate a linearization by topologically ordering an ever-expanding precedence graph. It would be interesting to know if this construction could be bounded, for example, by "pruning" the precedence graph at certain stages of the algorithm. This might be accomplished by storing *states*, rather than operations, in each entry of the shared snapshot object.

Regarding strong linearizability, little is known about the power of primitives with higher consensus numbers than registers and snapshots. As mentioned in Chapter 1, Golab, Higham, and Woelfel showed that standard universal constructions using consensus objects are strongly linearizable [1]. Therefore, it is possible to develop wait-free strongly linearizable implementations of any type in systems that have access to atomic compare-and-swap (CAS)

or load-linked/store-conditional (LL/SC) objects. We would like to know if there exist efficient wait-free strongly linearizable implementations of useful types from these powerful base objects. Perhaps many existing implementations of types from CAS or LL/SC objects are already strongly linearizable; in this case, it would be interesting to identify such implementations and prove that they are strongly linearizable.

Attiya, Castañeda, and Hendler showed that a wait-free strongly linearizable implementation of an n-process queue or stack can be used to solve n-consensus [11]. Their proof is quite simple; suppose we have access to a wait-free strongly linearizable queue. To solve consensus, each process first writes its value to a single-writer register, then enqueues its identifier to the queue, and finally takes a snapshot of the shared memory locations used by the queue to obtain a local copy of the object. Following this, a process simulates a dequeue on its local copy of the queue to obtain the process identifier p, and then decides on the value proposed by p. Since the queue is strongly linearizable, at some point every process agrees on the "head" of the queue, which implies that each process obtains the same identifier from its dequeue operation. This result immediately implies that there is no strongly linearizable implementation of an n-process queue or stack using base objects with consensus number less than n. We would like to know if a similar result holds for other types with consensus number 2, such as read-modify-write types.

In Section 1.1 we mentioned that strong linearizability is a form of strong observational refinement, a result presented by Attiya and Enea [7]. In this same paper, it is also shown that strong observational refinement is equivalent to the well-studied concept of forward simulations, originally formalized by Lynch and Vaandrager [27]. A forward simulation is a relation between the states of two types, where every outgoing transition of one type may be simulated by some sequence of steps of the other. Since strong observational refinement and forward simulations are equivalent, an object O of type $\mathscr{T}$ is strongly linearizable if and only if there exists a forward simulation from O to an atomic object of type $\mathscr{T}$ [7]. It would be interesting to know whether this equivalence admits simpler proof methodologies.

Additionally, in cases where strongly linearizable implementations are either impossible or inefficient, we would like to study the feasibility of developing implementations which are strong refinements of non-atomic (i.e. concurrent) specifications.

Our research was initially motivated by the following conjecture: every type that has a wait-free linearizable implementation from atomic registers has a wait-free strongly linearizable implementation from atomic snapshot objects. While we have not definitively proven/disproven this conjecture, we believe that we have made significant progress. For readable types, a solution seems to be near. Anderson and Moir's [24] assertion that dynamic resiliency is necessary for the existence of a wait-free linearizable implementation of a readable type from registers is accurate; therefore, dynamic resiliency is also necessary for the existence of a wait-free strongly linearizable implementation of a readable type from atomic snapshot objects. There is one final gap left to fill: the space between simple and dynamically resilient types. As mentioned in Chapter 6, all simple types are dynamically resilient, but not all dynamically resilient types are simple. It is possible that dynamically resilient readable types that are not simple have no wait-free linearizable implementations from registers; at the very least, we suspect that implementations of such types would differ considerably from the scan-compute-write strategy used by Aspnes and Herlihy. A summary of relevant knowns/unknowns is provided in Table 7.1.

Base Objects	S.L. Object	Liveness Property	Implementation
Atomic registers	ABA-detecting register	Lock-freedom	Bounded implementation by Theorem 1
Atomic registers	Snapshot	Wait-freedom	Proven impossible [2]
Atomic registers	Snapshot	Lock-freedom	Unbounded implementation in [2], bounded implementation by Theorem 2
S.L. max-register, versioned object of type $\mathscr{T}$	Type $\mathscr{T}$	Lock-freedom	Unbounded implementation in [2]
Atomic snapshots	Simple types	Wait-freedom	Unbounded implementation in [4], proof of S.L. Chapter 5
S.L. snapshots	Simple types	Lock-freedom	Unbounded implementation by Theorem 3
Atomic snapshots	Dynamically resilient readable types (as defined in Chapter 6 and [24])	Lock/Wait-freedom	Unknown
Atomic snapshots	Any type that has a wait-free linearizable implementation from atomic registers	Lock/Wait-freedom	Unknown
Consensus object	Any type	Wait-freedom	Standard universal constructions [1]

Table 7.1: A summary of relevant results and further directions. The table should be interpreted as follows: the first column contains a set of base objects B, the second column contains a type T, and the third column contains a liveness property L. The fourth column contains an answer to the following question: does type T have a strongly linearizable implementation from B that satisfies L? In all cases above "atomic registers" refers to atomic *multi-writer* registers. S.L. stands for either strong linearizability or strongly linearizable.

Bibliography

[1] W. Golab, L. Higham, and P. Woelfel, "Linearizable implementations do not suffice for randomized distributed computation," in *Proceedings of the 2011 ACM symposium on Theory of Computing*, pp. 373–382, 2011.

[2] O. Denysyuk and P. Woelfel, "Wait-freedom is harder than lock-freedom under strong linearizability," in *International Symposium on Distributed Computing*, pp. 60–74, 2015.

[3] Z. Aghazadeh and P. Woelfel, "On the time and space complexity of ABA prevention and detection," in *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pp. 193–202, 2015.

[4] J. Aspnes and M. Herliny, "Wait-free data structures in the asynchronous PRAM model," in *Proceedings of the 1990 ACM Symposium on Parallel algorithms and architectures*, pp. 340–349, 1990.

[5] J. McLean, "A general theory of composition for trace sets closed under selective interleaving functions," in *Proceedings of 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pp. 79–93, 1994.

[6] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

[7] H. Attiya and C. Enea, "Putting strong linearizability in context: Preserving hyperproperties in programs that use concurrent objects," in *arXiv preprint arXiv:1905.12063*, 2019.

[8] N. Lynch and F. Vaandrager, "Forward and backward simulations part i: Untimed systems (replaces tm-486)," tech. rep., Cambridge, MA, USA, 1994.

[9] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza, "Tractable refinement checking for concurrent objects," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, (New York, NY, USA), pp. 651–662, ACM, 2015.

[10] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 1, pp. 124–149, 1991.

[11] H. Attiya, A. Castañeda, and D. Hendler, "Nontrivial and universal helping for wait-free queues and stacks," *J. Parallel Distrib. Comput.*, vol. 121, pp. 1–14, 2018.

[12] M. Helmi, L. Higham, and P. Woelfel, "Strongly linearizable implementations: possibilities and impossibilities," in *Proceedings of the 2012 ACM symposium on Principles of Distributed Computing*, pp. 385–394, 2012.

[13] J. Aspnes, H. Attiya, and K. Censor, "Max registers, counters, and monotone circuits," in *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009*, pp. 36–45, 2009.

[14] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic snapshots of shared memory," in *Journal of the Association for Computing Machinery*, pp. 873–890, 1993.

[15] K. Abrahamson, "On achieving consensus using a shared memory," in *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, (New York, NY, USA), pp. 291–302, ACM, 1988.

[16] Y. Afek and E. Weisberger, "The instancy of snapshots and commuting objects," *Journal of Algorithms*, vol. 30, no. 1, pp. 68–105, 1999.

[17] E. Borowsky and E. Gafni, "Immediate atomic snapshots and fast renaming," in *Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*, pp. 41–51, ACM, 1993.

[18] R. Gawlick, N. Lynch, and N. Shavit, "Concurrent timestamping made simple," in *Theory of Computing and Systems*, pp. 171–183, Springer, 1992.

[19] M. Herlihy and N. Shavit, "The asynchronous computability theorem for t-resilient tasks," in *STOC*, vol. 93, pp. 111–120, 1993.

[20] J. Aspnes and K. Censor-Hillel, "Atomic snapshots in o (log 3 n) steps using randomized helping," in *International Symposium on Distributed Computing*, pp. 254–268, Springer, 2013.

[21] H. Attiya, M. Herlihy, and O. Rachman, "Atomic snapshots using lattice agreement," *Distributed Computing*, vol. 8, no. 3, pp. 121–132, 1995.

[22] H. Attiya and O. Rachman, "Atomic snapshots in o (n log n) operations," *SIAM Journal on Computing*, vol. 27, no. 2, pp. 319–340, 1998.

[23] M. Inoue, T. Masuzawa, W. Chen, and N. Tokura, "Linear-time snapshot using multi-writer multi-reader registers," in *International Workshop on Distributed Algorithms*, pp. 130–140, Springer, 1994.

[24] A. James and M. Mark, "Towards a necessary and sufficient condition for wait-free synchonization," in *Proceedings of the 1993 International Workshop on Distributed Algorithms*, 1993.

[25] M. Herlihy and J. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[26] Z. Aghazadeh, *Efficient Shared Memory Algorithms for Bounding Space*. PhD thesis, University of Calgary, 2018.

[27] N. Lynch and F. Vaandrager, "Forward and backward simulations," *Information and Computation*, vol. 121, no. 2, pp. 214–233, 1995.