# Separating Lock-Freedom from Wait-Freedom

Hagit Attiya, Armando Castañeda, Danny Hendler, Matthieu Perrin

## To cite this version:

Hagit Attiya, Armando Castañeda, Danny Hendler, Matthieu Perrin. Separating Lock-Freedom from Wait-Freedom. The 37th ACM Symposium on Principles of Distributed Computing, Jul 2018, Egham, United Kingdom. pp.41-50. hal-02053248

**HAL Id: hal-02053248**
**https://hal.science/hal-02053248**

Submitted on 1 Mar 2019

# Separating Lock-Freedom from Wait-Freedom

Hagit Attiya[*]
Department of Computer Science, Technion
hagit@cs.technion.ac.il

Armando Castañeda[†]
Instituto de Matemáticas, UNAM
armando.castaneda@im.unam.mx

Danny Hendler[‡]
Department of Computer Science, Ben-Gurion University
hendlerd@cs.bgu.ac.il

Matthieu Perrin[§]
LS2N, Université de Nantes
matthieu.perrin@univ-nantes.fr

## ABSTRACT

A long-standing open question has been whether lock-freedom and wait-freedom are fundamentally different progress conditions, namely, can the former be provided in situations where the latter cannot? This paper answers the question in the affirmative, by proving that there are objects with lock-free implementations, but without wait-free implementations—using objects of any finite power.

We precisely define an object called *n-process long-lived approximate agreement* (*n-LLAA*), in which two sets of processes associated with two *sides*, 0 or 1, need to decide on a sequence of increasingly closer outputs. We prove that 2-LLAA has a lock-free implementation using reads and writes only, while $n$-LLAA has a lock-free implementation using reads, writes and $(n-1)$-process consensus objects. In contrast, we prove that there is no wait-free implementation of the $n$-LLAA object using reads, writes and specific $(n-1)$-process consensus objects, called $(n-1)$-window registers.

## CCS CONCEPTS

• **Theory of computation** → **Shared memory algorithms**; **Concurrent algorithms**;

## KEYWORDS

concurrency; shared memory; multi-core algorithms; wait-freedom; lock-freedom; nonblocking.

## 1 INTRODUCTION

Asynchronous shared-memory algorithms capture the behavior of concurrent systems, where failure-prone processes, each running at its own speed, communicate by applying primitives to shared base objects. Such algorithms are often used to *implement* higher-level objects, supporting ongoing invocations. These implementations withstand adverse system behavior, due to scheduling anomalies and failures.

Two main progress conditions have been studied for asynchronous shared-memory implementations [11]. The first, *wait-freedom*, ensures individual progress to each process, i.e., its operations complete as long as it takes an infinite number of steps. The second, *lock-freedom*, requires only global progress, namely, if a process takes an infinite number of steps then some (possibly other) processes complete their operations.[1] Clearly, every wait-free implementation is also lock-free.

Many wait-free implementations, starting with the universal construction [11] and recently [13], are derived by first presenting a fairly simple lock-free implementation, which is then made wait-free through sophisticated techniques. It is therefore natural to ask whether every object with a lock-free implementation also has a wait-free implementation. For *one-shot* (or *bounded*) objects, which each process accesses at most once (or a bounded number of times), the answer is trivially positive: Every lock-free algorithm is also wait-free, since once a process completes its operation (or a bounded number thereof), it takes no more steps, and the global progress condition implies that other processes are guaranteed to complete their operations.

The question, however, is challenging for long-lived objects, as there are objects with a lock-free implementation but without a known wait-free implementation *using the same primitives* (e.g., queues from test&set primitives [2]). The fundamental nature of wait-freedom and lock-freedom, and their use in numerous papers, makes this an important question.

The question was seemingly answered by Herlihy [10], who argued that there is an object for which there is a lock-free implementation for two processes using only reads and writes, and there is no wait-free algorithm in the same setting. Besides several inaccuracies in the proofs (see below), the answer provided by [10] is limited: it does not show a separation for more than two processes and it does not show a separation when primitives stronger than reads and writes can be used.

This paper paints a more complete picture of the separation between lock-freedom and wait-freedom. We precisely define an object called *n-process long-lived approximate agreement* (in short, *n-LLAA*), supporting a single operation called output(), which has no argument and returns a real number. Each of the $n$ processes is assigned a *side*, either 0 or 1. Let $S$ be a sequential execution with invocations of output(). The *current position* of side $i$ in $S$ is the output value of the last operation of a process in side $i$, or $i$ if there are no operations of processes in side $i$. The sequential specification of $n$-LLAA includes every sequential execution in which the distance between the current positions of the two sides is at most $\frac{1}{2^r}$, where $r$ is the total number of operations in the execution.

[*]Partially supported by the Israel Science Foundation (grant 1749/14).
[†]Partially supported by project PAPIIT UNAM IA102417.
[‡]Partially supported by the Israel Science Foundation (grant 1749/14).
[§]Part of this work was done while this author was at the Department of Computer Science, Technion, supported by the Israel Science Foundation (grant 1749/14).

[1] Implementations of this kind were initially called *nonblocking* [11], but we use the more contemporary terminology.

We present a lock-free linearizable implementation of 2-LLAA, a restriction of $n$-LLAA for two processes, using reads and writes only (Section 3). This implementation is extended to a lock-free linearizable implementation of $n$-LLAA, using reads, writes and $(n-1)$-process consensus objects (Section 4).[2]

A specific consensus object that can be used is a *window register*. These registers support read and write primitives: the write primitive is standard, but the read primitive, parameterized with an integer $k > 1$, returns a $k$-tuple composed of the last $k$ values written, in the order in which they were written. A $k$-window register supports read primitives with parameter $\leq k$; note that a 1-window register is a standard register supporting read and write primitives. This object was initially defined in [14] to illustrate causal consistency. A similar object was defined concurrently in the context of space lower bounds for concurrent objects [8]. It is proved in [1] that a $k$-window register can solve consensus among exactly $k$ processes, i.e., its *consensus number* is $k$.

Our main impossibility result shows that there is no wait-free implementation of the $n$-LLAA object, that uses only $k$-window registers, for $k < n$ (Section 5). Note that this includes read/write registers.

**Related Work:** The 2-LLAA object is a stronger version of the *iterated approximate agreement* object studied in [10], and its implementation is the same, except for a few small details. However, as explained in Section 3.2, 2-LLAA requires a more elaborate correctness proof, which determines the linearization points dynamically. Differently from 2-LLAA, the convergence requirement of the iterated approximate agreement object is that the $r$-th output operation of $p_i$ is at distance at most $1/2^r$ from the the current position of the other process. Furthermore, the impossibility result for wait-free implementations in [10] relies on an unstated assumption (in the second paragraph of the proof of Lemma 14) that an infinite number of steps by one process must eventually fix the decision of the other process. Although this behavior, often referred to as *helping*, is a common way to turn lock-free implementations into wait-free ones, it is not a priori known to be a necessary condition for wait-freedom. The notion of helping was recently formalized [5, 6] and was shown to be a nontrivial property, allowing to prove nontrivial results not known to hold otherwise. Additionally, the final step of the impossibility proof in [10] (in the proof of Theorem 15) needs a stronger convergence requirement than the one required by the iterated approximate agreement object. Our impossibility proof addresses these challenges and extends to the general case of systems with $(n-1)$-consensus objects.

Aspnes and Herlihy [4] defined a class of objects for which there are wait-free implementations from read/write objects, as well as lock-free ones. Another class of objects, called *Common2*, includes the set of objects that have a wait-free implementation from objects with consensus number 2, e.g., test&set [2, 3].

*Strong linearizability* [9] was introduced as a way to circumvent some anomalies when used with randomized implementations. It was shown that there are deterministic lock-free and strongly linearizable implementations of snapshot, max-register or counter from multi-writer multi-reader registers, while there are no such wait-free implementations [7].

[2] The separation question is trivial if $n$-process consensus objects are available since they support a wait-free implementation for $n$-processes of every object with a sequential specification [11].

## 2 MODEL OF COMPUTATION

We consider a standard shared memory system with $n$ asynchronous processes, $p_0, \ldots, p_{n-1}$, which may crash at any time during an execution. Processes communicate with each other by applying atomic *primitives* to *base objects*. The base objects we consider are *window registers* of size $k \in \{1, \ldots, n-1\}$ (in short, $k$-window registers) to which *read* and *write* primitives are applied. The *write* primitive has a single parameter, the value to be written. The *read* primitive returns a $k$-tuple composed of the last $k$ values written, in the order in which they were written. Missing values are replaced by a default value $\perp$. Each window register can be read and written to by all processes, i.e., they are *multi-reader multi-writer*. Note that a 1-window register is the standard register, supporting ordinary read and write operations.

A *(high-level) concurrent object*, or *data type*, is defined by a state machine consisting of a set of states, a set of operations, and a set of transitions between states. Such a specification is known as *sequential*. An *implementation* of a data type $T$ is a distributed algorithm $\mathcal{A}$ consisting of a local state machine $\mathcal{A}_p$, for each process $p$. For any process $p$, $\mathcal{A}_p$ specifies which primitives $p$ applies in order to return a response when it invokes an operation of $T$. Each of these primitive invocations is a step. For the rest of this section, fix an implementation $\mathcal{A}$ of some data type $T$.

A *configuration* $C$ of the system contains the states of all registers and processes. In an *initial* configuration, registers and processes are in their initial states. Given a configuration $C$ and process $p$, $p(C)$ is the configuration after $p$ takes its next step in $C$. Moreover, $p^0(C) = C$ and for every $n \in \mathbb{N}$, $p^{n+1}(C) = p(p^n(C))$.

An *execution* of $\mathcal{A}$ is a possibly infinite sequence of steps (namely, invocations of primitives) and invocations and responses of high-level operations in $\mathcal{A}$, with the following properties: (1) Each process first invokes a high-level operation, and only when it has a corresponding response, it can invoke another high-level operation, i.e., executions are *well-formed.* (2) A process takes steps only between an invocation and a response. (3) For any invocation of process $p$, the steps of $p$ between that invocation and the following response of $p$, if there is one, correspond to steps of $p$ that are specified by $\mathcal{A}$.

An operation in an execution is *complete* if both its invocation and response appear in the execution. An operation is *pending* if only its invocation appears in the execution. An implementation is *wait-free* if every process completes each of its operations in a finite number of its steps. Formally, if a process executes infinitely many steps in an execution, all its operations are complete. An implementation is *lock-free* if whenever processes execute steps, at least one of the operations terminates. Formally, in every infinite execution, infinitely many operations are complete. Thus, a wait-free implementation is lock-free but not necessarily vice versa.

An operation $OP$ *precedes* another operation $OP'$ if $OP$ completes before $OP'$ starts. $OP$ and $OP'$ are *concurrent* (or *overlapping*) if neither $OP$ precedes $OP'$ nor $OP'$ precedes $OP$. *$OP$ does not precede $OP'$* if either $OP$ is concurrent with $OP'$ or $OP'$ precedes $OP$.

*Linearizability* [12] is the standard notion used to identify a correct implementation. Roughly speaking, an implementation is linearizable if each operation appears to take effect atomically at some time between its invocation and response, hence operations' real-time order is maintained. Formally, let $\mathcal{A}$ be an implementation of a data type $T$. An execution $\alpha$ of $\mathcal{A}$ is *linearizable* if there is a sequential execution $S$ of $T$ (i.e., a sequence of matching invocation-response pairs, starting with an invocation) such that: (1) $S$ contains every complete operation of $\alpha$ and some pending operations. Hence, the output value in the matching responses of an invocation in $S$ and $\alpha$ are the same. (2) For every pair of operations *op* and *op′* in $\alpha$, if the response of *op* precedes the invocation of *op′* in $\alpha$, then *op* appears before *op′* in $S$; namely, $S$ respects the real-time order in $\alpha$. $\mathcal{A}$ is *linearizable* if each of its executions is linearizable.

## 3 READ/WRITE LOCK-FREE 2-LLAA

This section presents a read/write lock-free 2-LLAA implementation, as well as a sketch of its correctness proofs. Some proofs are deferred to the appendix.

Definition 1. *Let $S$ be a sequential execution with invocations of two processes $p_0$ and $p_1$ to an operation* output(), *each returning a real number. The* current position *of $p_i$ in $S$ is the output value of the last operation of $p_i$, or $i$ if there are no operations of $p_i$. We say that $S$ satisfies the* convergence requirement *if the distance between the current positions of the processes is at most $1/2^r$, where $r$ is the total number of operations in the execution. The sequential specification of the* 2-LLAA object *contains every sequential execution of* output() *operations by $p_0$ and $p_1$ such that each of its prefixes satisfies the convergence requirement.*

```
Shared variables:
    $M[0, 1] \leftarrow [\langle pos = 0, round = 0\rangle, \langle pos = 1, round = 0\rangle]$

Local variables:
    $myID \leftarrow i$
    $yourID \leftarrow 1 - i$
    $me.\langle pos, round\rangle \leftarrow \langle myID, 0\rangle$ %% My position and round
    $you.\langle pos, round\rangle \leftarrow \langle yourID, 0\rangle$ %% Other's position and round
    $prev, range \leftarrow [myID \pm 1]$

Function output():
(01) $me.round \leftarrow me.round + 1$
(02) $M[myID].round \leftarrow me.round$
(03) while true do
(04)     $M[myID].pos \leftarrow me.pos$
(05)     $you \leftarrow M[yourID]$
(06)     $rounds \leftarrow me.round + you.round$
(07)     $range \leftarrow [me.pos \pm 1/2^{rounds}]$
(08)     if $you.pos \in range \vee you.pos \notin prev$ then
(09)         $prev \leftarrow range$
(10)         return $me.pos$
(11)     elseif $me.pos < you.pos$ then
(12)         $me.pos \leftarrow me.pos + 1/2^{rounds}$
(13)     else
(14)         $me.pos \leftarrow me.pos - 1/2^{rounds}$
(15)     endif
(16) endwhile
endFunction
```

**Figure 1: A lock-free 2-LLAA algorithm for processes $p_0$ and $p_1$. Code of process $p_i$.**

The algorithm in Figure 1 is a lock-free implementation of 2-LLAA using read/write primitives. Each process $p_i$ locally stores its current position and round (number of complete and pending operations of $p_i$, so far) in its local variable *me*, and stores the position and round of the other process, denoted $p_j$, in *you*. The initial position and round of $p_i$ are $i$ and 0. Processes communicate through the shared array $M$ by writing and reading positions and rounds. Additionally, $p_i$ uses two (closed) intervals *prev* and *range*: *prev* stores the interval committed in $p_i$'s previous operation, which indicates where any future position of $p_i$ might be, and *range* stores the current interval the position of $p_j$ should belong to in order for $p_i$ to decide.

When executing output(), $p_i$ first increments its round counter and writes the new value in $M$ (lines 01 and 02) and then iterates the while loop (lines 03 to 15). Each iteration tries to get closer to the position of $p_j$ in $M$, until the condition in line 08 is satisfied. There are two (disjoint) cases that can make $p_i$'s current operation decided. In the first case, called *closeness rule*, the position of $p_j$ read by $p_i$ is in the interval *range* computed in line 07, hence the distance between the positions is at most $1/2^r$, where $r$ is the total number of operations $p_i$ is aware of. This ensures that the distance between the positions of the processes obeys the convergence requirement of 2-LLAA. In the second case, called *invalid position rule*, the position of $p_j$ is out of the previously-committed range *prev* of $p_i$. Observe that this can happen because of asynchrony: $p_i$ might have completed several operations getting closer to a position of $p_j$ in $M$ that is different from the position $p_j$ is about to write. The rationale behind the invalid position rule is that as $p_j$ was slow and its position is not in *prev* of $p_i$, it would have to "catch up" and decide by the closeness rule in its current operation. As shown below (Lemma 1), whenever $p_i$ decides by the invalid position rule, the current operation of $p_j$ is pending and $p_j$ decides by the closeness rule in every extension in which its operation is complete.

### 3.1 Correctness Proof

In the analysis of the algorithm, for simplicity and without loss of generality, the invocation of an operation is identified with its first shared-memory write operation outside the **while** loop of the output function (line 02), that is, it is executed atomically together with the local operations of line 01. The response of a decided operation is identified with its last shared-memory read step inside the loop (line 05). We also assume that the local computation inside the **while** loop (lines 05-15) is executed atomically together with the shared-memory read step in line 05.

For an execution $\alpha$, we use the following notation and terminology. The $k$-th operation of a process $p_i$ in $\alpha$ is denoted $P_i^k(\alpha)$, if there is such an operation. $P_i^0(\alpha)$ denotes a fictitious operation whose output is the initial position of $p_i$. An operation $P_i^k(\alpha)$ of $p_i$ is *decided* if it takes its response (read) step in $\alpha$. In this case, $v_i^k(\alpha)$ denotes the value of variable $v$ at the end of $P_i^k(\alpha)$. We let $rad(prev_i^k(\alpha))$ denote the radius of the interval $prev_i^k(\alpha)$ with center $me_i^k.pos(\alpha)$. A decided operation $P_i^k(\alpha)$ decides *by the closeness* *rule on* $P_j^m(\alpha)$ (with $j = 1 - i$) if $you_i^k.pos(\alpha) \in range_i^k(\alpha)$ and $you_i^k.round(\alpha) = m$. $P_i^k(\alpha)$ decides *by the invalid position rule on* $P_j^m(\alpha)$ if $you_i^k.pos(\alpha) \notin prev_i^k(\alpha)$ and $you_i^k.round(\alpha) = m$. We omit $\alpha$ from the above definitions when it can be understood from the context.

To see that Algorithm 1 is lock-free, we note that two operations $P_i^k$ and $P_j^m$ cannot both take an infinite number of steps without completing, since eventually they agree on $rounds = k + m$ and their positions get sufficiently close to each other. The formal proof is omitted for lack of space

Algorithm Linearize (Figure 2) specifies how we linearize a quiescent execution (i.e. an execution without pending operations) $\alpha$. Lemma 1 stated below (whose proof is in the appendix) forms the basis for showing that Algorithm Linearize produces correct linearizations. Linearize considers the operations of $\alpha$ in the order they decide. Let $P_i^k$ and $P_j^m$ be the operations considered at the beginning of the $\ell$-th iteration of the while loop of Linearize. If $P_j^m$ is decided, then $P_j^m$ is already in $seq_\ell$, and, as we shall see, $P_i^k$ decides by the closeness rule. This implies that $P_i^k$ and $P_j^m$ satisfy the convergence requirement. If $P_j^m$ is not decided (hence pending), then $P_j^m$ can be safely placed before or after $P_i^k$ because Lemma 1 implies that $P_j^m$ will decide by the closeness rule. Finally, as the proof of Lemma 3 argues, if none of $P_i^k$ and $P_j^m$ are in $seq_\ell$, then $P_i^k$ and $P_j^m$ start from a quiescent state of the algorithm (operations $P^2$ and $Q^1$ in Figure 3 are an example of this situation) and their order is decided considering the outputs of the previous operations.

Lemma 1. *Let $\alpha$ be a finite execution whose last step is a read (in line 05) of an operation $P_i^k$ of $p_i$ that makes that operation decided (by either the closeness or the invalid position rules) on some undecided operation $P_j^m$ of $p_j$, with $j = 1 - i$. Let $\alpha\beta$ be any extension of $\alpha$ in which $P_j^m$ is decided. Then, $P_j^m$ decides by the closeness rule in $\alpha\beta$ and $dist(me_i^k.pos, me_j^m.pos) \leq 1/2^{m+k}$.*

It follows directly from Algorithm Linearize that the latest point in which an operation is linearized is when it decides (by reading in line 05 of Figure 1). Moreover, once an operation is linearized, the operation (by the other process) it decides on is also linearized (if it was not linearized before).

Lemma 2. *Let $\alpha$ be a finite execution without pending operations. The sequential execution produced by* Linearize($\alpha$) *respects the real-time order in $\alpha$.*

Proof. Consider two operations $P_i^k$ and $P_j^m$ in $\alpha$, $k, m > 0$. Suppose that $P_i^k$ precedes $P_j^m$ in $\alpha$. The latest point at which Algorithm Linearize linearizes $P_i^k$ is when $\beta_{l+1}$ in line 03 of Algorithm Linearize ends in the read step which makes $P_i^k$ decide in $\alpha$. None of the steps of $P_j^m$ appear in $\beta_{l+1}$, hence $P_i^k$ does not decide on $P_j^m$ and consequently $P_j^m$ is not linearized in lines 10, 15 or 17 of Algorithm Linearize (which are the only places operations are linearized). Furthermore, no prior operation by $p_i$ could have decided on $P_j^m$ in $\beta_{l+1}$, hence it was not linearized before. □

```
Function Linearize(α):
(01) $seq_0 \leftarrow \epsilon$, $\ell \leftarrow 0$
(02) while $\alpha$ has more than $\ell$ operations do
(03)     $\beta_{\ell+1} \leftarrow$ shortest prefix of $\alpha$ with $\ell + 1$ decided operations
(04)     $P_i^k \leftarrow$ operation that decides in the last (read) step of $\beta_{\ell+1}$
(05)     $P_j^m \leftarrow$ operation that $P_i^k$ decides on in $\alpha$ (i.e. $you_i^k.round = m$)
(06)     if $P_i^k$ is in $seq_\ell$ then
(07)         if $P_j^m$ is in $seq_\ell$ then
(08)             $seq_{\ell+1} \leftarrow seq_\ell$
(09)         else
(10)             $seq_{\ell+1} \leftarrow seq_\ell \cdot P_j^m$
(11)     else
(12)         if $P_j^m$ is in $seq_\ell$ then
(13)             $seq_{\ell+1} \leftarrow seq_\ell \cdot P_i^k$
(14)         else if $dist(me_i^k.pos, me_j^{m-1}.pos)$
                      $\leq dist(me_j^m.pos, me_i^{k-1}.pos)$ then
(15)             $seq_{\ell+1} \leftarrow seq_\ell \cdot P_i^k \cdot P_j^m$
(16)         else
(17)             $seq_{\ell+1} \leftarrow seq_\ell \cdot P_j^m \cdot P_i^k$
(18)         endif
(19)     endif
(20)     $\ell \leftarrow \ell + 1$
(21) endwhile
(22) Remove the (fictitious) operations $P_0^0$ and $P_1^0$ from $seq_\ell$
(23) return $seq_\ell$
endFunction
```

**Figure 2: A sequential algorithm to linearize an execution $\alpha$ without pending operations.**

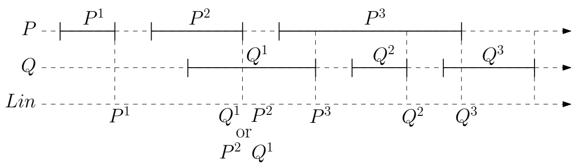

**Figure 3: An example of the linearization produced by Algorithm Linearize.**

Lemma 3, is the key technical lemma in the algorithm's correctness proof, establishing that Algorithm Linearize outputs correct linearizations.

Lemma 3. *Let $\alpha$ be any finite execution without pending operations. Then,* Lin($\alpha$) *is a sequential execution of the 2-LLAA object.*

Proof Sketch. We consider a linearization Lin($\alpha$) of an execution $\alpha$ which, as we assume towards a contradiction, is not a sequential execution of 2-LLAA, and consider the first operation $P_i^k$ that violates convergence. Let $P_j^{m-1}$ be the latest preceding operation of $p_j$ in Lin($\alpha$). Then, $dist(me_i^k.pos, me_j^{m-1}.pos) > 1/2^{m+k-1}$. Since Lin($\alpha$) respects the real-time order in $\alpha$ (Lemma 2), $P_i^k$ does not precede $P_j^{m-1}$ in $\alpha$.

If $P_j^{m-1}$ and $P_i^k$ are concurrent in $\alpha$, let $\beta$ be the shortest prefix of $\alpha$ in which the first of them, say $P_i^k$, is decided. Hence $P_j^{m-1}$ is pending and $P_i^k$ decides on $P_j^{m-1}$. By Lemma 1,

$$dist(me_i^k.pos, me_j^{m-1}.pos) \leq 1/2^{m+k-1},$$

and the convergence requirement holds.

We therefore know that $P_j^{m-1}$ precedes $P_i^k$ in $\alpha$. It can be shown that at the end of the previous (possibly fictitious) operation $P_i^{k-1}$ of $p_i$, $rad(prev_i^{k-1}) \geq 1/2^{m+k-2}$. Moreover, when $P_i^k$ starts, $prev_i = prev_i^{k-1}$, $me_i.pos = me_i^{k-1}.pos$ and $M[j].pos = me_j^{m-1}.pos$. Now, if no operation of $p_j$ overlaps $P_i^k$, then if $P_i^k$ decides by the invalid position rule, it does so in the very first iteration of the while loop, without any change to its position $me_i.pos$, and thus $me_i^k.pos = me_i^{k-1}.pos$. Hence, $me_j^{m-1}.pos \notin prev_i^{k-1}$ and $dist(me_i^{k-1}.pos, me_j^{m-1}.pos) > 1/2^{m+k-2}$, implying that either $P_i^{k-1}$ or $P_j^{m-1}$ (both preceding $P_i^k$ in $\mathsf{Lin}(\alpha)$) do not satisfy the convergence requirement, which is a contradiction. And if $P_i^k$ decides by the closeness rule, the code implies that

$$dist(me_i^k.pos, me_j^{m-1}.pos) \leq 1/2^{m+k-1},$$

and the convergence requirement holds.

We are left with the case that operation $P_j^m$ exists and overlaps with $P_i^k$ (and $P_j^{m-1}$ precedes $P_i^k$). We first prove that

$$\begin{aligned} dist(me_i^k.pos, me_j^{m-1}.pos) &\leq dist(me_j^m.pos, me_i^{k-1}.pos) \quad (1)\\ dist(me_i^k.pos, me_j^m.pos) &\leq 1/2^{m+k} \quad (2)\end{aligned}$$

To prove (1), note that $P_j^m$ cannot precede $P_i^{k-1}$ because it is concurrent with $P_i^k$. Also, it is easy to show that $P_i^{k-1}$ precedes $P_j^m$ in $\alpha$. Thus, $P_j^{m-1}$ precedes $P_i^k$ and $P_i^{k-1}$ precedes $P_j^m$, and hence *both $P_i^{k-1}$ and $P_j^{m-1}$ are complete when the first among $P_i^k$ and $P_j^m$ starts in $\alpha$.* Without loss of generality, suppose that $P_i^k$ decides before $P_j^m$, then, since they overlap, $P_i^k$ decides on $P_j^m$. Since $P_j^{m-1}$ precedes $P_i^k$ and $P_i^{k-1}$ precedes $P_j^m$, in the while iteration of Algorithm Linearize that linearizes $P_i^k$, the conditions of lines 06 and 12 are false, namely, $P_i^k$ and $P_j^m$ were not linearized yet, and they are linearized in lines 14-18. It can be shown that $P_j^m$ is not linearized before $P_i^k$, and the condition of line 14 holds, implying (1).

Lemma 1 implies (2), since $P_i^k$ are $P_j^m$ are concurrent, similarly to the case when $P_i^k$ and $P_j^{m-1}$ are concurrent, handled before.

Finally, consider the outputs $me_i^{k-1}.pos$ and $me_j^{m-1}.pos$ of $P_i^{k-1}$ and $P_j^{m-1}$. Suppose that $me_i^{k-1}.pos \leq me_j^{m-1}.pos$ (the other case is similar). Let $\mathcal{I} = [me_i^{k-1}.pos, me_j^{m-1}.pos]$. It can be shown that $me_i^k.pos, me_i^m.pos \in \mathcal{I}$. By assumption, $P_i^{k-1}$ and $P_j^{m-1}$ do satisfy the convergence requirement, namely,

$$dist(me_i^{k-1}.pos, me_j^{m-1}.pos) \leq 1/2^{m+k-2},$$

and hence $|\mathcal{I}| \leq 1/2^{m+k-2}$. The interesting case is when $|\mathcal{I}| = 1/2^{m+k-2}$. Lemma 12 (in the appendix) implies that the first to decide among $P_i^k$ and $P_j^m$ must decide a position (in $\mathcal{I}$) that is

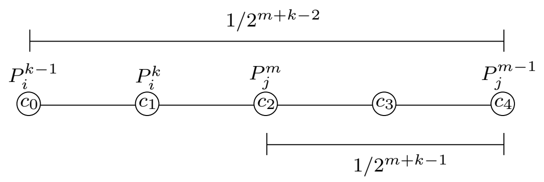

**Figure 4: An example of the last case in the proof of Lemma 3: $P_i^k$ decides before $P_j^m$ and outputs $c_1$, whose distance to the output of $P_j^{m-1}$ is strictly larger than $1/2^{m+k-1}$. By (2), the output of $P_j^m$ is in the interval $[c_0, c_2]$. This is a contradiction since the distance between any point in $[c_0, c_2]$ and the output of $P_i^{k-1}$ is at most $1/2^{m+k-1}$, and thus $dist(me_i^k.pos, me_j^{m-1}.pos) > dist(me_j^m.pos, me_i^{k-1}.pos)$.**

an integer multiple of $1/2^{m+k}$. There are only five integer multiples of $1/2^{m+k}$ in $\mathcal{I}$.[3] A case by case analysis shows that if $dist(me_i^k.pos, me_j^{m-1}.pos) > 1/2^{m+k-1}$, then inequality (2) implies that $dist(me_i^k.pos, me_j^{m-1}.pos) > dist(me_j^m.pos, me_i^{k-1}.pos)$, which is a contradiction, given inequality (1) (see an example in Figure 4). □

Consider any finite execution $\alpha$ of the algorithm from Figure 1. Let $\alpha\beta$ be an extension of $\alpha$ in which all operations in $\alpha$ are complete (and no new operation is started). This extension exists since the algorithm is lock-free. By Lemma 2, the sequential execution $seq(\alpha\,\beta)$ produced by $\mathsf{Lin}(\alpha\beta)$ respects the real-time order of $\alpha\,\beta$; by Lemma 3, it is a sequential execution of the 2-LLAA object. Thus, $seq(\alpha\,\beta)$ is a linearization of $\alpha\,\beta$, and of $\alpha$ as well, implying the main theorem of this section:

Theorem 4. *Algorithm 1 is a lock-free linearizable implementation of the 2-LLAA object, using reads and writes.*

## 3.2 A Simpler Linearization Proof?

This section discussed whether a simpler linearization argument exits. The linearization proof in the previous subsection is nontrivial, partly because Algorithm Linearize may place operations that are pending in $\alpha$ before operations that are complete. Furthermore, as we described previously, to linearize a non-quiescent execution $\alpha$, we first complete all pending operations in $\alpha$ and then apply Algorithm Linearize to the resulting quiescent execution. Consequently, it is possible that the linearization $S'$ we obtain for an extension $\alpha'$ of $\alpha$ is not an extension of the linearization $S$ we obtain for $\alpha$, implying that our implementation is not necessarily strongly linearizable [9].[4]

In contrast, the linearizability order used in the correctness proof of the lock-free algorithm [10, Figure 5, Section 4] is simple and shows that the algorithm is strongly linearizable. Below we argue that this simple approach does not work for our stronger 2-LLAA object.

[3] It is useful to consider the initial configuration, where $m = k = 1$ and $me_i^{k-1}.pos = 0$, $me_j^{m-1}.pos = 1$.

[4] An algorithm is *strongly linearizable* if there is a function $f$ from executions of the algorithm to sequential executions such that (1) for every execution $\alpha$, $f(\alpha)$ is a linearization of $\alpha$ and (2) for every execution $\alpha\beta$, $f(\alpha)$ is a prefix of $f(\alpha\beta)$.

The linearization order used in [10] (right above Lemma 11) is the following. For every operation $OP$ in an execution, let $rounds(OP)$ be the first value the operation sets to the local variable $rounds$ (that variable is called $i$ in [10]). Then, if $rounds(OP) < rounds(OP')$, $OP$ is linearized before $OP'$, and if $rounds(OP) = rounds(OP')$, $OP$ and $OP'$ can be linearized in any order.

Consider the following execution of the algorithm in Figure 1: (1) $p_0$ starts an invocation to output(), completes one iteration of the loop (note that it does not decide as $1/2^{rounds} = 1/2^1$) and stops before writing its new preference at the first step of the second iteration of the loop. Note that $p_0$ is about to write $1/2$ and the position of $p_0$ in $M$ is still 0. (2) $p_1$ executes, until completion, two consecutive invocations to output() ($p_0$ takes no steps while $p_1$ completes the two invocations). Observe that $p_1$ obtains $1/4$ and $1/8$ from the two invocations, respectively. (3) $p_0$ runs solo until it completes its pending output() invocation. Note that $p_0$ obtains $1/4$ from the invocation.

It is easy to see that, for the execution $\alpha$ defined above, $rounds(P_0^1) = 1$, $rounds(P_1^1) = 2$ and $rounds(P_1^2) = 3$, and thus, using the linearization order defined in [10], $\alpha$ would be linearized as follows: $\langle P_0^1 : 1/4\rangle\langle P_1^1 : 1/4\rangle\langle P_1^2 : 1/8\rangle$. However, this linearization is incorrect because $\mathsf{lin}(\alpha)$ is not a sequential execution of 2-LLAA: the prefix $\langle P_0^1 : 1/4\rangle$ of the sequential execution is invalid, because the distance between the current position of $p_0$ and the current (initial) position of $p_1$, which are respectively $1/4$ and $1$, is larger than $1/2^1$. Consequently, $\mathsf{lin}(\alpha)$ is not a valid sequential execution of 2-LLAA either.

Finally, it can be verified that Algorithm Linearize produces the linearization $\langle P_1^1 : 1/4\rangle\langle P_0^1 : 1/4\rangle\langle P_1^2 : 1/8\rangle$ from $\alpha$, which is a sequential execution of 2-LLAA.

## 4 LOCK-FREE $n$-LLAA, USING $(n-1)$-CONSENSUS OBJECTS

We formally define the $n$-LLAA object and explain how it can be implemented in a lock-free manner using $(n-1)$-process consensus objects; full correctness proofs are omitted for lack of space.

Definition 2. *Each of the n processes belongs to a* side*, either 0 or 1. The n-LLAA object supports a single operation called* output()*. Let S be a sequential execution with invocations of* output()*, each returning a real number. The* current position *of side i in S is the output value of the last operation of a process in side i, or i if there are no operations of processes in side i. We say that S satisfies the* convergence requirement *if the distance between the current positions of the two sides is at most $\frac{1}{2^r}$, where r is the total number of operations in the execution. The sequential specification of the* $n$-LLAA object *contains every sequential execution of* output() *operations by the n processes such that each of its prefixes satisfies the convergence requirement.*

The lock-free implementation of $n$-LLAA is based on the lock-free 2-LLAA algorithm. Consider first the case in which there is at least one process in each side. Then, using $(n-1)$-process consensus objects, the $\leq n-1$ processes in side $i$ simulate operations of process $p_i$ in Algorithm 1; each time an operation terminates, the processes decide who takes this output value, also using consensus objects.

```
Shared variables:
    $M[0,1][0,\ldots,n-1] \leftarrow$ each entry of $M[j]$, $j \in \{0,1\}$,
initialized to $\langle pos = j, round = 0, tmst = 0\rangle$
    $SIDE[0,\ldots,n-1] \leftarrow$ each entry initialized to $\perp$
    $READS[0,1][\infty] \leftarrow$ bi-dimensional array
with infinitely many $(n-1)$-process consensus objects
    $OPS[0,1][\infty] \leftarrow$ bi-dimensional array
with infinitely many $(n-1)$-process consensus objects

Local variables:
    $mySide \leftarrow$ initialized to 0 or 1
    $otherSide \leftarrow 1 - mySide$
    $me.\langle pos, round, tmst\rangle \leftarrow \langle mySide, 0, 0\rangle$
%% My side's position, round and timestamp
    $you.\langle pos, round, tmst\rangle \leftarrow \langle otherSide, 0, 0\rangle$
%% Other's position, round and timestmp
    $prev, range \leftarrow [mySide \pm 1]$
    $count\_reads, count\_ops \leftarrow 0$

Function output():
(01) $SIDE[ID] \leftarrow mySide$
(02) $s \leftarrow$ snapshot($SIDE$)
(03) if $|\{ID : s[ID] = mySide\}| = n$ then
(04)     return $otherSide$
(05) while true do
(06)     $me.round \leftarrow me.round + 1$
(07)     $me.tmst \leftarrow me.tmst + 1$
(08)     $M[mySide][ID] \leftarrow me$
(09)     $decided \leftarrow$ false
(10)     while $\neg decided$ do
(11)         $M[mySide][ID] \leftarrow me$
(12)         $t \leftarrow$ snapshot($M[otherSide]$)
(13)         $max \leftarrow$ max($\{tmst' : t[ID].tmst = tmst'\}$)
(14)         $prop \leftarrow$ any $t[ID]$ such that $t[ID].tmst = max$
(15)         $count\_reads \leftarrow count\_reads + 1$
(16)         $you \leftarrow READS[mySide][count\_reads]$.decide($prop$)
(17)         $rounds \leftarrow me.round + you.round$
(18)         $range \leftarrow [me.pos \pm 1/2^{rounds}]$
(19)         if $you.pos \in range \vee you.pos \notin prev$ then
(20)             $prev \leftarrow range$
(21)             $count\_ops \leftarrow count\_ops + 1$
(22)             $owner \leftarrow OPS[mySide][count\_ops]$.decide($ID$)
(23)             if $owner = ID$ then
(24)                 return $me.pos$
(25)             $decided \leftarrow$ true
(26)         elseif $me.pos < you.pos$ then
(27)             $me.pos \leftarrow me.pos + 1/2^{rounds}$
(28)             $me.tmst \leftarrow me.tmst + 1$
(29)         else
(30)             $me.pos \leftarrow me.pos - 1/2^{rounds}$
(31)             $me.tmst \leftarrow me.tmst + 1$
(32)         endif
(33)     endwhile
(34) endwhile
endFunction
```

**Figure 5: A lock-free $n$-process LLAA object algorithm.**

When there are $n$ processes in the same side, the processes can safely decide on the initial value of the other side.

Figure 5 shows how processes in the same side $i$ simulate process $p_i$ of Algorithm 1. Lines 05–31 simulate a single operation output() of $p_i$; each time this loop starts, a new operation of $p_i$ is simulated. Lines 08 and 11 correspond to the write operations of Algorithm 1 and Lines 12–16 simulate the read operation of Algorithm 1, with the help of the consensus objects in *READS*. Each process first reads the pair ($pos, round$) of the processes in the other side and considers the most current one (from its point of view), with the help of the timestamps in the local variables *me*. This is its proposal to the consensus which decides the next read value of the simulated process $p_i$. Once the output value of the current simulated operation of $p_i$ has been computed (the condition in Line 19 is true), the processes in the same side $i$ compete for that output value in Line 22, with the help of the consensus objects in *OPS*. The winning process outputs that value for its current operation, while the others start a new simulated operation of $p_i$. The goal of Lines 01–04 is to handle the case in which all processes belong to the same side: when a process discovers that all processes are in its side, it knows that the other side is not moving (it has not moved since the beginning of the execution) hence it can safely decide the initial position of the other side.

The way processes advertise their side in Line 01, and then check the others' side in Line 03, ensures that in every execution, at most $n-1$ processes invoke the same consensus object (in Lines 16 and 22). The following property directly follows from the agreement property of consensus.

Lemma 5. *Consider an execution of Algorithm 5. Then, for every pair of processes $p_\ell$ and $p_j$ in the same side, if both $p_\ell$ and $p_j$ assign the $k$th value to a local variable $v$, which is either $me, you, prev$ or $range$, then $v_j^k = v_\ell^k$.*

Using this lemma, we can now precisely define the notion of a *simulated operation*. Let $\alpha$ be a (finite or infinite) execution of Algorithm 5. For every side $i$ and integer $k \geq 1$, the *$k$-th simulated operation of side i*, denoted $sOP_i^k$, is the subsequence of $\alpha$ containing all steps corresponding to steps in Lines 08, 11, 12, 16 or 22 of any process such that when it executes the step, its *me.round* local variable is equal to $k$. The following lemma relates the executions of Algorithms 1 and 5, where the two processes in Algorithm 1 are denoted $q_0$ and $q_1$.

Lemma 6. *For every finite or infinite execution $\alpha$ of Algorithm 5, there is an execution $\alpha^*$ of Algorithm 1 such that:*

*(1) There is a function $f$ from the set of operations of $\alpha$ onto the set of operations in $\alpha^*$; moreover, for each operation $OP$ in side i, $f(OP)$ is an operation of $q_i$.*

*(2) An operation $OP$ is pending if and only if $f(OP)$ is pending; moreover, $OP$ has infinitely many steps if and only if $f(OP)$ has infinitely many steps too.*

*(3) The outputs of a completed operation $OP$ and its associated completed operation $f(OP)$ are the same.*

*(4) If $OP$ precedes $OP'$ in $\alpha$ then $f(OP)$ precedes $f(OP')$ in $\alpha^*$.*

Proof Sketch. By Lemma 5, every pair of processes performing the steps of the same simulated operation *sOP*, write the same sequence of values (possibly at distinct times, due to asynchrony). Thus, for each simulated operation *sOP* in $\alpha$, we can map an operation $g(sOP)$ in an execution of $\alpha^*$ of Algorithm 1 by removing redundant steps from $\alpha$ (essentially, we keep the first $\ell$th simulated step of *sOP*, for every $\ell$). For any operation *OP* of process $p_i$ in $\alpha$, let $h(OP)$ be the simulated operation in $\alpha$, the last step of *OP* by $p_i$ belongs to. Then, if *OP* is completed, $h(OP)$ is the simulated operation $p_i$ is simulating when it wins in the consensus at Line 22. Finally, we let $f(OP) = g(h(OP))$ □

Theorem 7. *Algorithm 5 is a lock-free linearizable implementation of the n-LLAA object, using reads, writes and $(n-1)$-process consensus objects.*

Proof. Lemma 6 implies that if there is an infinite execution $\alpha$ of Algorithm 5 in which only finitely many operations are completed, then there is an infinite execution $\alpha^*$ of Algorithm 1 in which only finitely many operations are completed, in contradiction to the lock-freedom of Algorithm 1. This implies that Algorithm 5 is lock-free as well.

Next, consider any finite execution $\alpha$ and let $\alpha\beta$ be an extension in which all operations in $\alpha$ are completed. Let $\lambda$ be an execution of Algorithm 1 associated with $\alpha\beta$, as in Lemma 6. The output of a completed operation *OP* is equal to the output of its associated completed operation $f(OP)$, and if *OP* precedes $OP'$ in $\alpha\beta$ then, by Lemma 6, $f(OP)$ precedes $f(OP')$ in $\lambda$. Thus, by replacing each *OP* with $f(OP)$, we can obtain a linearization $\text{lin}(\alpha\beta)$ of $\alpha\beta$ from any linearization $\text{lin}(\lambda)$ of $\lambda$, since the convergence requirement in both 2-LLAA and $n$-LLAA is a function of the number of all previous operations. Thus, Algorithm 5 is linearizable. □

## 5 IMPOSSIBILITY OF WAIT-FREE $n$-LLAA USING $(n-1)$-WINDOW REGISTERS

An execution $\beta$ is an *extension* of a finite execution $\alpha$ if $\alpha$ is a prefix of $\beta$. A configuration $C$ is *reachable* if there is a finite execution that ends with all processes and variables in the same states as in $C$. A configuration $C'$ is *reachable* from a configuration $C$ if there is a finite execution $\alpha$ that ends with $C'$ and a prefix of the execution $\alpha$ ends with $C$. Two configurations $C$ and $C'$ are *indistinguishable* to process $p$, denoted $C \overset{p}{\sim} C'$, if the state of every shared variable and the state of $p$ are the same in $C$ and $C'$.

Fix an implementation $\mathcal{A}$ of the n-LLAA object. Below, we consider only executions of $\mathcal{A}$ in which process $p_0$ is on side 0, performing a single output() operation, and all other processes are on side 1.

Let $\alpha$ be a finite execution of $\mathcal{A}$; $\alpha$ is *$v$-potent* if there is an extension $\alpha'$ of $\alpha$ in which the operation of $p_0$ is complete and returns $v$; $\alpha$ is *$v$-univalent* (or just *univalent* when $v$ is irrelevant) if $\alpha$ is $v$-potent and for all $w \neq v$, $\alpha$ is not $w$-potent; $\alpha$ is *bivalent* if it is both $v$-potent and $w$-potent, for two values $v \neq w$.

Definition 3. *An execution $\alpha$ is* ambivalent *for process $q \neq p_0$ (or simply* ambivalent *when $q$ is immaterial), if $\alpha$ is $v$-univalent and for some $w \neq v$, there is a $w$-univalent execution $\alpha'$ that is indistinguishable from $\alpha$ to process q. If $\alpha$ ends with a configuration C, we say that C* is ambivalent *for process $q$.*

Definition 4. *An execution $\alpha$, ending with configuration $C$, has a* critical $q$-extension *(or simply* critical extension *when $q$ is immaterial) for a process $q \neq p_0$, if there is a process $q' \notin \{p_0, q\}$ such that, for all $\ell \geq 0$, $p_0(q^\ell(C))$ is $v$-univalent and $p_0(q'(q^\ell(C)))$ is $w$-univalent with $v \neq w$.*

Definition 5. *An execution $\alpha$, ending with configuration $C$, is* $(v, q, v')$-critical *(or simply* critical *if $v$, $q$ and $v'$ are immaterial), if $p_0(C)$ is $v$-univalent, there is a process $q \neq p_0$ such that $p_0(q(C))$ is $v'$-univalent, for $v' \neq v$, and for all finite extensions $\alpha\beta$ of $\alpha$ ending in configuration $C'$, $p_0(C')$ is univalent.*

Lemma 8. *If $\alpha$ is a $(v, p_1, v')$-critical execution of $\mathcal{A}$ ending in configuration $C$, then either some extension of $\alpha$ is ambivalent or some extension of $\alpha$ has a critical extension.*

Proof. The proof considers the next steps that $p_0$ and $p_1$ are about to take in $C$. If both $p_0$ and $p_1$ access different window registers, or $p_1$'s step is a read, then $p_1(p_0(C))$ and $p_0(p_1(C))$ are indistinguishable to $p_0$. Hence, a solo execution of $p_0$ after both configurations returns the same value, contradicting the fact that $\alpha$ is $(v, p_1, v')$-critical, that is, $p_1(p_0(C))$ is $v$-univalent and $p_0(p_1(C))$ is $v'$-univalent.

If $p_0$'s next step is a read of a window register, then $p_1(p_0(C))$ and $p_0(p_1(C))$ are indistinguishable to all processes other than $p_0$, $p_1(p_0(C))$ is $v$-valent and $p_0(p_1(C))$ is $v'$-valent, so $p_1(p_0(C))$ and $p_0(p_1(C))$ are both ambivalent for all processes other than $p_0$.

The remaining case is when both steps are writes to the same window register $R$. If $n = 2$, then $R$ is an ordinary register, so $p_0(p_1(C))$ and $p_0(C)$ are indistinguishable to $p_0$. Since $\alpha$, ending with $C$, is $(v, p_1, v')$-critical, $p_0(C)$ is $v$-valent and $p_0(p_1(C))$ is $v'$-valent, which is a contradiction.

For $n > 2$, we can prove the following claim:

Claim 9. *Let $\alpha\beta$ be a finite extension of $\alpha$ ending in configuration $C'$, such that $p_0$ and $p_1$ do not take steps in $\beta$ and no process writes to $R$ in $\beta$. Then $p_0(C')$ is $v$-univalent and $p_0(p_1(C'))$ is $v'$-univalent.*

Proof. The proof is by induction on the length of $\beta$, with a trivial base case when $\beta$ is empty. Suppose the lemma holds for when the length of $\beta$ is $\ell$, and let us consider an extension $\alpha\beta\gamma$ of $\alpha$, where $\gamma$ is a single step of some process $q \notin \{p_0, p_1\}$. Let $C'$ be the configuration at the end of $\alpha\beta$, and $C''$ be the configuration at the end of $\alpha\beta\gamma$. By the induction hypothesis, $p_0(C')$ is $v$-univalent and $p_0(q(C'))$ is $v'$-univalent. Note that no process writes to $R$ in $\beta\gamma$.

If $q$ applies a primitive to a window register different than $R$, then $p_0(C'') = p_0(q(C')) = q(p_0(C'))$ is $v$-univalent and $p_0(p_1(C'')) = p_0(p_1(q(C'))) = q(p_0(p_1(C')))$ is $v'$-univalent.

Otherwise, since no process writes to $R$ in $\beta\gamma$, $q$ reads $R$. Then $p_0(C'') = p_0(q(C'))$ and $p_0(C')$ are indistinguishable to $p_0$. As $p_0(C')$ is $v$-univalent, $p_0(C'')$ is $v$-potent, since the solo executions by $p_0$ starting from $C''$ and from $C'$ are the same, hence both must return $v$. Moreover, since $\alpha$ is critical, $p_0(C'')$ is univalent. Therefore, $p_0(C'')$ is $v$-univalent. Similarly, $p_0(p_1(C'')) = p_0(p_1(q(C')))$ and $p_0(p_1(C'))$ are indistinguishable to $p_0$ and $p_0(p_1(C'))$ is $v'$-univalent by the induction hypothesis. Hence, $p_0(p_1(C''))$ is $v'$-univalent. □

Assume first that we can let each process $p_2, \ldots, p_{n-1}$, in turn, take steps until it is about to write to $R$, reaching a configuration $D$ in which the next step of every process is a write to $R$. By Claim 9, $p_0(D)$ is $v$-valent and $p_0(p_1(D))$ is $v'$-valent. However, since $R$ is an $(n-1)$-window register, $p_{n-1}(\ldots(p_2(p_0(D)))\ldots)$ and $p_{n-1}(\ldots(p_2(p_0(p_1(D))))\ldots)$ are indistinguishable to all processes except $p_1$, contradicting the fact that $p_{n-1}(\ldots(p_2(p_0(D)))\ldots)$ is $v$-valent, whereas $p_{n-1}(\ldots(p_2(p_0(p_1(D))))\ldots)$ is $v'$-valent.

Otherwise, for some process $q \neq p_1$, there is an infinite extension $\alpha\beta\gamma$ of $\alpha$ such that $q$ runs solo in $\gamma$ and no process writes to $R$ in $\beta\gamma$. Let $C'$ be the configuration at the end of $\alpha\beta$. By Claim 9, for all $\ell \geq 0$, $p_0(q^\ell(C'))$ is $v$-univalent and $p_0(p_1(q^\ell(C')))$ is $v'$-univalent, i.e., $\gamma$ is a critical $q$-extension of $\alpha\beta$. □

We next use a standard valency argument to prove that if $\mathcal{A}$ is wait-free, then it has a critical execution.

Lemma 10. *If $\mathcal{A}$ is a wait-free linearizable implementation of the $n$-LLAA object, then it has a critical execution.*

Proof. Since $\mathcal{A}$ is linearizable, in a solo execution by $p_0$ starting from $C_0$ and completing one output operation, the current position of side 0 must be at least $\frac{1}{2}$. On the other hand, in the execution from $C_0$ in which $p_1$ first executes solo two complete output operations and then $p_0$ runs solo and executes its output operation to completion, the current position of side 0 can be at most $\frac{3}{8}$. It follows that the empty execution (from $C_0$) is bivalent.

Starting with the empty execution as $\alpha$, we repeatedly extend $\alpha$ to a bivalent execution $\alpha\alpha''$, such that $p_0$ executes a step in $\alpha''$ ($\alpha''$ may include steps by other processes). Since $\mathcal{A}$ is wait-free, eventually we reach a bivalent execution $\alpha\alpha'$ ending with configuration $C'$ such that for any extension $\alpha\alpha'\beta$ of $\alpha\alpha'$ ending with configuration $C''$, $p_0(C'')$ is univalent. In particular, this holds for the empty $\beta$.

It follows that $p_0(C')$ is univalent, so it is $v$-valent for some $v$. Since $\alpha\alpha'$ is bivalent, it has an extension $\alpha\alpha'\beta_1$, ending with a configuration $C_1$, such that $p_0(C_1)$ is $v'$-valent, for some $v' \neq v$. Let $\beta_2$ be the longest prefix of $\beta_1$ such that $\alpha\alpha'\beta_2$ ends with a configuration $C_2$ for which $p_0(C_2)$ is $v$-valent. Let $q$ be the process that takes the next step in $\alpha\alpha'\beta_1$ after the execution of $\alpha\alpha'\beta_2$. From our construction, $p_0(C_2)$ is $v$-valent, $p_0(q(C_2))$ is $v'$-valent for $v' \neq v$, and for any finite extension $\alpha\alpha'\beta_3$ of $\alpha\alpha'\beta_2$, ending with a configuration $C_3$, $p_0(C_3)$ is univalent. It follows from Definition 5 that $\alpha\alpha'\beta_2$ is $(v, q, v')$-critical. □

Theorem 11. *There is no wait-free linearizable implementation of the $n$-LLAA object, which uses only $(n-1)$-window registers. (Note that this includes read and write registers.)*

Proof. Assume, by way of contradiction, that $\mathcal{A}$ is a wait-free linearizable implementation of the $n$-LLAA object. By Lemma 10, $\mathcal{A}$ has a critical execution $\alpha$. By Lemma 8, we have two cases:

**Case 1:** Some extension $\alpha'$ of $\alpha$, ending with a configuration $C'$, is ambivalent for a process $q \neq p_0$. By Definition 3, $\alpha'$ is $v$-valent for some $v$, and there is a $w \neq v$ and an execution $\alpha_w$, ending with a configuration $C_w$, such that $\alpha_w$ is $w$-univalent and configurations $C'$ and $C_w$ are indistinguishable to $q$.

Pick an integer $k$ such that $\frac{1}{2^k} < |v-w|$. Let $\alpha'\beta_v^q$ be an extension of $\alpha'$ such that, in $\beta_v^q$, $q$ runs solo and executes to completion $k+3$ output operations. Let $x$ be the value returned by the last operation of $q$. Let $\alpha'\beta_v^q\beta_v^{p_0}$ be an extension of $\alpha'\beta_v^q$ such that, in $\beta_v^{p_0}$, $p_0$ runs solo until it completes its output operation. Since $\alpha'$ is $v$-univalent, $p_0$'s operation returns $v$.

Consider a linearization of $\alpha'\beta_v^q\beta_v^{p_0}$. It contains at least $k+3$ operations of processes in side 1, including all the ones by $q$, and one operation by $p_0$. Suppose first that the last operation of $q$ is linearized after the operation of $p_0$. By Definition 2, $|x-v| \leq \frac{1}{2^{k+4}} < \frac{1}{2^{k+1}}$.

Suppose now that the last operation of $q$ is linearized before the operation of $p_0$. Let $k'$ denote the number of operations in side 1 linearized between the last operation of $q$ and the operation of $p_0$ and let $y$ be the value returned by the operation linearized right before the operation of $p_0$. By Definition 2,

$$\begin{array}{rcccccc} |x-v| & \leq & |x-0| & + & |0-y| & + & |y-v| \\ & \leq & 2^{-(k+3)} & + & 2^{-(k+3+k')} & + & 2^{-(k+3+k'+1)} \\ & < & 2^{-(k+1)} . & & & & \end{array}$$

Let $\alpha_w\beta_w^q$ be an extension of $\alpha_w$ such that, in $\beta_w^q$, $q$ runs solo after $\alpha_w$ and executes to completion $k+3$ output operations. Since $\alpha'$ and $\alpha_w$ are indistinguishable to $q$, $\beta_w^q = \beta_v^q$ and so $q$'s $(k+3)$'th output operation in $\beta_w^q$ also returns $x$. Now consider the extension $\alpha_w\beta_w^q\beta_w^{p_0}$ such that, in $\beta_w^{p_0}$, $p_0$ runs solo until it completes its output operation. Since $\alpha_w$ is $w$-univalent, $p_0$'s output operation returns $w$. Consider a linearization of $\alpha_w\beta_w^q\beta_w^{p_0}$. Using a similar reasoning as above, we can show that $|w-x| < 2^{-(k+1)}$. Therefore,

$$|w-v| \leq |w-x| + |x-v| < \frac{2}{2^{k+1}} = \frac{1}{2^k},$$

which is a contradiction.

**Case 2:** Some extension $\alpha'$ of $\alpha$, ending with configuration $C'$, has a critical $q$-extension. That is, there are two processes $q \neq q'$, different from $p_0$, and two values $v \neq w$, such that for all $\ell \geq 0$, $p_0(q^\ell(C'))$ is $v$-univalent and $p_0(q'(q^\ell(C')))$ is $w$-univalent.

The rest of the reasoning is similar to the previous case. Pick an integer $k$ such that $\frac{1}{2^k} < |v-w|$. Let $\alpha'\beta$ be an extension of $\alpha'$ ending in a configuration $C''$, such that in $\beta$, $q$ runs solo and executes $k+3$ output operations to completion. Let $x$ be the value returned by $q$'s last operation.

Let $\alpha'\beta\gamma_v$ and $\alpha'\beta\gamma_w$ be two extensions of $\alpha'\beta$ such that, in $\gamma_v$, $p_0$ runs solo until it completes its output operation, returning $v$, since $p_0(C'')$ is $v$-univalent; and in $\gamma_w$, $q'$ executes a single step and then $p_0$ runs solo until it completes its output operation, returning $w$, since $p_0(q'((C'')))$ is $w$-univalent. Considering a linearization of $\alpha'\beta\gamma_v$ and a linearization of $\alpha'\beta\gamma_w$, it can be shown that

$$|v-w| \leq |v-x| + |w-x| < \frac{2}{2^{k+1}} = \frac{1}{2^k},$$

which is a contradiction. □

## 6 SUMMARY AND FUTURE WORK

This paper shows a computational separation between lock-freedom and wait-freedom, two dominant progress conditions of concurrent algorithms: while there is no wait-free implementation of the $n$-LLAA object from $k$-window registers, with $k < n$ (whose consensus number is exactly $k$), there is a lock-free implementation of the $n$-LLAA object using read/write primitives and $(n-1)$-process consensus objects. To the best of our knowledge, this is the first time such a separation is shown.

An interesting open question is whether there exists an $n$-LLAA implementation from $x$-process consensus objects for $x < n-1$, e.g., reads and writes, which would strengthen the separation. Noting that the $n$-LLAA object is non-deterministic, a second avenue for future research is to investigate whether or not the separation that we have shown holds also for *deterministic* long-lived objects. Finally, the $n$-LLAA specification assumes that the object "knows" the ID of the process invoking an operation on it. It would be interesting to see if the separation holds also for objects that do not assume this.

## REFERENCES

[1] M. R. Achour Mostéfaoui, Matthieu Perrin. A simple object that spans the whole consensus hierarchy. *Parallel Processing Letters*, 2018 (in press).

[2] Y. Afek, E. Gafni, and A. Morrison. Common2 extended to stacks and unbounded concurrency. *Distributed Computing*, 20(4):239–252, 2007.

[3] Y. Afek, E. Weisberger, and H. Weisman. A completeness theorem for a class of synchronization objects. In *PODC*, pages 159–170, 1993.

[4] J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous pram model. In *SPAA*, pages 340–349, 1990.

[5] H. Attiya, A. Castañeda, and D. Hendler. Nontrivial and universal helping for wait-free queues and stacks. In *OPODIS*, 2015.

[6] K. Censor-Hillel, E. Petrank, and S. Timnat. Help! In *PODC*, pages 241–250, 2015.

[7] O. Denysyuk and P. Woelfel. Wait-freedom is harder than lock-freedom under strong linearizability. In *DISC*, pages 60–74, 2015.

[8] F. Ellen, R. Gelashvili, N. Shavit, and L. Zhu. A complexity-based hierarchy for multiprocessor synchronization. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 289–298, 2016.

[9] W. Golab, L. Higham, and P. Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *STOC*, pages 373–382, 2011.

[10] M. Herlihy. Impossibility results for asynchronous pram. In *SPAA*, pages 327–336, 1991.

[11] M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[12] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–492, July 1990.

[13] A. Kogan and E. Petrank. A methodology for creating fast wait-free data structures. In *PPoPP*, pages 141–150, 2012.

[14] M. Perrin, A. Mostefaoui, and J. Claude. Causal consistency: beyond memory. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016.

## A ADDITIONAL PROOFS FOR SECTION 3

We present additional details for the correctness analysis of Algorithm 1; some technical proofs are omitted for lack of space.

For an execution $\alpha$, we use the following notation and terminology. If $\alpha$ is finite, $v_i^\alpha$ denotes the value of local variable $v$ of a process $p_i$ at the end of $\alpha$, while $M^\alpha$ denotes the value of the shared array $M$ at the end of $\alpha$. $I_i^\alpha$ denotes the interval $[M^\alpha[i].pos \pm 1/2^{me_i^\alpha.round+you_i^\alpha.round}]$. If $\alpha$ is finite and ends with a read step of $p_i$ that makes its current operation decided, then $rad(prev_i^\alpha)$ is the radius of the interval $prev_i^\alpha$ with center $me_i^\alpha.pos$.

The following lemma can be easily proved by induction on the length of the execution $\alpha$.

Lemma 12. *In every finite execution $\alpha$, for $i \in \{0,1\}$, $M^\alpha[i].pos = y/2^{me_i^\alpha.round+you_i^\alpha.round}$, for some integer $y \geq 0$.*

Lemma 13. *Let $\lambda = \alpha e_1 e_2 \cdots e_\ell$ be a finite execution. Then, $I_i^\lambda \subseteq prev_i^\alpha$.*

Proof. The proof of the lemma is based on the following two claims.

Claim 14. *For every finite execution $\alpha$, $I_i^{\alpha} \subseteq prev_i^{\alpha}$.*

Proof Sketch. By induction on the length of the prefixes of $\alpha$, one can prove that $me_i^{\alpha}.pos \in prev_i^{\alpha}$ and the distance between $me_i^{\alpha}.pos$ and any extreme of $prev_i^{\alpha}$ is at least $1/2^{me_i^{\alpha}.round+you_i^{\alpha}.round}$. This invariant proves the lemma when $M^{\alpha}[i].pos = me_i^{\alpha}.pos$, because then $I_i^{\alpha} \subseteq prev_i^{\alpha}$ by definition. Otherwise, it must be that the last step of $p_i$ in $\alpha$ is a read step in line 05 that modifies its position $me_i.pos$ after executing lines 11-15, in which case we consider the execution $\alpha'$ obtained from $\alpha$ by removing the last step of $p_i$ and proceed as in the first case. □

Claim 15. *For every finite execution $\alpha e$, $prev_i^{\alpha e} \subseteq prev_i^{\alpha}$.*

Proof. We prove the claim by induction on the length of the prefixes of $\alpha e$. Note that the first step $f$ in $\alpha e$ must be a write in line 02, which does not change the value of any local variable, and hence $prev_i^{f} = prev_i^{\epsilon}$, where $\epsilon$ is the empty execution. This is the base of the induction.

Suppose that $prev_i^{\beta f} \subseteq prev_i^{\beta}$, for a prefix $\beta f$ of $\alpha$. We show that $prev_i^{\beta f g} \subseteq prev_i^{\beta f}$, where $g$ is the next step in $\alpha e$. If $g$ is not a step of $p_i$ or is a write step of $p_i$ (in line 02 or line 04), then such a step does not change the local variables of $p_i$, and $prev_i^{\beta f g} = prev_i^{\beta f}$.

Otherwise, $g$ is a read step of $p_i$ (Line 05) and the local variables of $p_i$ might change when $p_i$ executes lines 07–15. The claim clearly holds if $p_i$ does not decide, since it executes line 07 and then lines 11–15; therefore, $prev_i$ is not modified, namely, $prev_i^{\beta f g} = prev_i^{\beta f}$.

If $p_i$ decides, that is, the condition of line 08 is satisfied, then note that $p_i$ does not modify its position after the read step, hence $me_i^{\beta f g}.pos = me_i^{\beta f}.pos = M^{\beta f}[i].pos$. By Claim 14, $I_i^{\beta f} \subseteq prev_i^{\beta f}$ holds. Also, following the execution of line 07, $range^{\beta f g} = I_i^{\beta f g} \subseteq I_i^{\beta f}$ holds (note that the containment may be proper, because $you_i^{\beta f g}.round \geq you_i^{\beta f}.round$) and $prev^{\beta f g} = range_i^{\beta f g}$ (line 09), and thus $prev^{\beta f g} \subseteq prev^{\beta f}$. □

By Claim 14, $I_i^{\lambda} \subseteq prev_i^{\lambda}$. By repeatedly applying Claim 15, $prev^{\alpha e_1 e_2 \cdots e_\ell i} \subseteq prev^{\alpha e_1 e_2 \cdots e_{\ell-1} i} \subseteq \ldots \subseteq prev^{\alpha}$. Therefore, $I_i^{\alpha e_1 e_2 \cdots e_\ell} \subseteq prev_i^{\alpha}$ and the lemma follows. □ □

Lemma 16. *Let $\alpha$ be a finite execution. Then, for each $i \in \{0, 1\}$ and $j = 1 - i$, either $M^{\alpha}[i].pos \in prev_j^{\alpha}$ or $M^{\alpha}[j].pos \in prev_i^{\alpha}$.*

Proof Sketch. By strong induction on the length of the executions, we can show that either $me_i^{\alpha}.pos \in prev_j^{\alpha}$ or $me_j^{\alpha}.pos \in prev_i^{\alpha}$. This proves the lemma because if neither $M^{\alpha}[i].pos \in prev_j^{\alpha}$ nor $M^{\alpha}[j].pos \in prev_i^{\alpha}$, then we consider the execution $\alpha'$ obtained from $\alpha$ by removing, if necessary, the last read step of each process so that $M^{\alpha'}[i].pos = me_i^{\alpha'}.pos$ and $M^{\alpha'}[j].pos = me_j^{\alpha'}.pos$, hence reaching a contradiction. □

Proof of Lemma 1. First, we show that if $M^{\alpha}[i].pos \in prev_j^{\alpha}$, then $P_j^m$ must decide by the closeness rule in $\alpha\beta$. Note that $me_i^{\alpha}.pos = M^{\alpha}[i].pos$. Assume otherwise towards a contradiction, then $P_j^m$ decides by the invalid position rule, hence at some point in the execution $p_j$ evaluates $you_j.pos \notin prev_j$ as true in line 08. Thus, there must exist a prefix $\lambda$ of $\alpha\beta$ such that $p_i$'s last read in $\lambda$ (in line 05) results in it executing lines 11–15 and writing to $me_i^{\lambda}.pos$ a value such that $me_i^{\lambda}.pos \notin prev_j^{\lambda}$ (with $\alpha$ a prefix of $\lambda$ and $prev_j^{\alpha} = prev_j^{\lambda}$). Observe that $p_i$ changes $me_i^{\lambda}.pos$ in order to get closer to $p_j$'s position, hence $you_i^{\lambda}.pos = M^{\lambda}[j].pos \notin prev_j^{\lambda}$, implying that $I_j^{\lambda} \nsubseteq prev_j^{\lambda}$, contradicting Lemma 13. Thus, $P_j^m$ cannot decide by the invalid position rule.

We prove that $M^{\alpha}[i].pos \in prev_j^{\alpha}$, by considering two possibilities. (1) $P_i^k$ decides by the closeness rule Note that $me_i^{\alpha}.pos = M^{\alpha}[i].pos$ and $you_i^{\alpha}.pos = M^{\alpha}[j].pos$. The fact that $P_i^k$ decides by the closeness rule implies that at the end of $\alpha$, when $p_i$ executes line 08, it sees that $M^{\alpha}[j].pos = you_i^{\alpha}.pos \in range_i^{\alpha}$. Since the round counter of each process increases monotonically, we have that $rad(range_i^{\alpha}) \leq rad(I_j^{\alpha})$, hence $M^{\alpha}[i].pos = me_i^{\alpha}.pos \in I_j^{\alpha}$. By Lemma 13, $I_j^{\alpha} \subseteq prev_j^{\alpha}$, from which follows that $M^{\alpha}[i].pos \in prev_j^{\alpha}$. (2) $P_i^k$ decides by the invalid position rule. Hence, at the end of $\alpha$, when $p_i$ executes Line 08, it sees that $you_i^{\alpha}.pos = M^{\alpha}[j].pos \notin prev_i^{\alpha}$. Lemma 16 directly implies that $M^{\alpha}[i].pos \in prev_j^{\alpha}$. In both cases, $M^{\alpha}[i].pos \in prev_j^{\alpha}$ and hence $P_j^m$ decides by the closeness rule.

Now, since the last step in $\alpha$ is a read by $p_i$ in line 05 that makes $P_i^k$ decided, $p_i$ executes lines 07–10 at the end of $\alpha$, and so we have that $me_i^{\alpha}.pos$ is the center of $prev_i^{\alpha}$, $rad(prev_i^{\alpha}) = 1/2^{m+k}$, $me_i^{k}.pos = me_i^{\alpha}.pos$ and $prev_i^{k} = prev_i^{\alpha}$. We prove that $me_j^{m}.pos \in prev_i^{\alpha}$, which implies that $dist(me_i^{k}.pos, me_j^{m}.pos) \leq 1/2^{m+k}$.

Let $\lambda$ be the shortest prefix of $\alpha\beta$ in which $P_j^m$ is decided (hence $\alpha$ is a prefix of $\lambda$). The last step of $\lambda$ is a read step by $p_j$ in line 05 that makes $P_j^m$ decided. Let $x = you_j^{\lambda}.round$. Thus, $P_j^m$ decides (by the closeness rule) on the operation $P_i^x$, for $x \geq k$. At the end of $\lambda$, after $p_j$ executes lines 07–10, we have that $me_j^{m}.pos = me_j^{\lambda}.pos$.

By Lemma 13, $I_i^{\lambda} \subseteq prev_i^{\alpha}$. Since $P_j^m$ decides by the closeness rule, at the end of $\lambda$ the condition of line 08 is satisfied when $p_j$ checks it, and thus $dist(me_j^{\lambda}.pos, you_j^{\lambda}.pos) \leq 1/2^{m+x}$. Notice that $you_j^{\lambda}.pos = M^{\lambda}[i].pos$, and hence $me_j^{m}.pos \in I_i^{\lambda} \subseteq prev_i^{\alpha}$, which concludes the proof. □