ABSTRACT

Recent work established that some restricted-use objects, such as max registers, counters and atomic snapshots, admit polylogarithmic step-complexity wait-free implementations using only reads and writes: when only polynomially-many updates are allowed, reading the object (by performing a ReadMax, CounterRead or Scan operation, depending on the object’s type) incurs $O(\log N)$ steps (where $N$ is the number of processes), which was shown to be optimal.

But what about the step-complexity of update operations? With these implementations, updating the object’s state (by performing a WriteMax, CounterIncrement or Update operation, depending on the object’s type) requires $\Omega(\log N)$ steps. The question that we address in this work is the following: are there read-optimal implementations of these restricted-use objects for which the asymptotic step-complexity of update operations is sub-logarithmic?

We present tradeoffs between the step-complexity of read and update operations on these objects, establishing that updating a read-optimal counter or snapshot incurs $\Omega(\log N)$ steps. These tradeoffs hold also if compare-and-swap (CAS) operations may be used, in addition to reads and writes.

We also derive a tradeoff between the step-complexities of read and update operations of $M$-bounded max registers: if the step-complexity of the ReadMax operation is $O(f(\min(N,M)))$, then the step-complexity of the WriteMax operation is $\Omega(\log f(\min(N,M)))$. It follows from this tradeoff that the step-complexity of WriteMax in any read-optimal implementation of a max register from read, write and CAS is $\Omega(\log \log \min(N,M))$. On the positive side, we present a wait-free implementation of an $M$-bounded max register from read, write and CAS for which the step complexities of ReadMax and WriteMax operations are $O(1)$ and $O(\log \min(N,M))$, respectively.

*Partially supported by the Israel Science Foundation (grant number 1227/10) and by the Lynne and William Frankel Center for Computing Science at Ben-Gurion University.

1. INTRODUCTION

Concurrent objects are often constructed only for a restricted use. Their use can be restricted either by limiting the number of operations applied to them (e.g., an atomic snapshot object that only supports a limited number of Update operations), or by passing bounded arguments to the operations applied to them (e.g., a max register that supports only writes of values below some threshold).

Recent work established that some restricted-use objects admit polylogarithmic step-complexity wait-free implementations using only reads and writes. Aspnes, Attiya and Censor-Hillel [2] presented a wait-free implementation of an $M$-bounded max register that supports a WriteMax operation that writes a value to the max register, and a ReadMax operation that returns the largest value previously written. Both these operations have step-complexity of $O(\log M)$. By using max register as a building block, they constructed an efficient wait-free counter. CounterRead operations on the counter incur a number of steps logarithmic in $N$, the number of processes sharing the implementation. If the counter is required to support only a limited number of CounterIncrement operations (polynomial in $N$), then the step complexity of CounterIncrement operations is $O(\log^2 N)$. Restricted-use max registers and counters have already been used for devising efficient randomized consensus [5] and mutual exclusion algorithms [7].

More recently, Aspnes et al. [3] presented an implementation of restricted-use atomic snapshots. For polynomially-many updates, the step-complexities of the Scan and Update operations in their implementation are $O(\log N)$ and $O(\log^3 N)$, respectively.

Are the step-complexities of these implementations asymptotically optimal? For read operations, it was shown that they are. Assuming (as we do in the rest of this paper) polynomially-many updates, Aspnes et al. [2] proved an $\Omega(\log M)$ step lower bound on ReadMax operations for obstruction-free implementations of $M$-bounded max registers. They also proved an $\Omega(\log N)$ step-complexity lower bound on CounterRead operations for obstruction-free counter implementations. This result was later generalized
by Aspnes et al. [4] for a class of objects that includes atomic snapshots, establishing an $Ω(\log N)$ step-complexity lower bound on $Scan$ operations for obstruction-free implementations of snapshots.

But what about the step-complexity of update operations on these objects? The $WriteMax$ operation of the max register algorithm presented in [2] has logarithmic step-complexity, and the counter $CounterIncrement$ and snapshot $Update$ operations of the algorithms presented in [2, 3] have poly-logarithmic step-complexities. Can we do better? Are there read-optimal implementations of these restricted-use objects for which the asymptotic step-complexity of update operations is sub-logarithmic or even constant? This is the question that we address in this paper.

Our Contributions:
We prove the following tradeoff for counters and atomic snapshots: if the step-complexity of the read operation is $O(f(N))$, then the step-complexity of the update operation is $Ω(\log \frac{1}{f(N)})$. The tradeoff holds for obstruction-free implementations, even if CAS may be used in addition to reads and writes. Setting $f(N) = \log N$ establishes that for any read-optimal implementation of a counter or snapshot object from reads and writes, the step-complexity of update operations is $Ω(\log N)$.

For $M$-bounded max registers, we were only able to obtain a weaker tradeoff: if the step-complexity of the $ReadMax$ operation is $O(f(\min(N,M)))$, then the step-complexity of the $WriteMax$ operation is $Ω(\log f(\min(N,M)))$. It follows from this tradeoff that the step-complexity of $WriteMax$ in any read-optimal implementation of a max register from read, write and CAS is $Ω(\log \log \min(N,M))$. On the positive side, we present a wait-free implementation of an $M$-bounded max register from read, write and CAS for which the step complexities of $ReadMax$ and $WriteMax$ operations are $O(1)$ and $O(\log \min(N,M))$, respectively.

2. PRELIMINARIES
We consider a standard model of an asynchronous shared memory system, in which processes communicate by applying operations to shared objects. An object is characterized by a domain of possible values and by a set of operations that provide the only means to manipulate it. An implementation of an object shared by a set $P = \{p_1, \cdots, p_N\}$ of $N$ processes provides a specific data-representation for the object from a set $B$ of shared base objects, each of which is assigned an initial value; the implementation also provides algorithms for each process in $P$ to apply each operation to the object being implemented.

A wait-free implementation of a concurrent object guarantees that any process can complete an operation in a finite number of its own steps. An obstruction-free [11] implementation guarantees only that if a process eventually runs by itself then it completes its operation within a finite number of its own steps. Each step consists of some local computation and one shared memory event, which is a primitive operation (or simply primitive) applied atomically to an object in $B$ that may return a response value. We say that the event accesses the object and that it applies the primitive to it. We say that an event is trivial, if it does not change the value of the base object to which it is applied.

An execution fragment is a (finite or infinite) sequence of events. An execution is an execution fragment that starts from the initial configuration, resulting when processes apply operations to the implemented object as they execute the implementation’s algorithm. We let $\bot$ denote the empty execution. For any finite execution fragment $E$ and any execution fragment $E'$, the execution fragment $EE'$ denotes the concatenation of $E$ and $E'$. Let $E$ and $F$ be two executions. We say that $F$ is an extension of $E$ if $F = EE'$ for some execution fragment $E'$. We say that executions $E$, $E'$ are indistinguishable to process $p$, if $p$ performs the same events and receives the same responses to these events in both executions. For an execution $E$ and a set of processes $P$, we let $E^{-P}$ denote the sequence of events obtained by removing all the events issued by the processes of $P$ from $E$.

If a process has not completed its operation, it has exactly one enabled event, which is the next event it will issue, as specified by the algorithm it is using to apply its operation to the implemented object. We say that a process $p$ is active after $E$ if $p$ has not completed its operation in $E$. Operation $Φ_1$ precedes operation $Φ_2$ in execution $E$, if $Φ_1$ completes in $E$ before the first event of $Φ_2$ has been issued in $E$.

The compare-and-swap (CAS) operation is defined as follows: $CAS(v,expected,new)$ changes the value of variable $v$ to $new$ only if its value just before CAS is applied is $expected$; in this case, the CAS operation returns true and we say it is successful. Otherwise, CAS does not change the value of $v$ and returns false; in this case, we say that the CAS was unsuccessful. We assume that base objects support only the read, write, and CAS primitives.

A max register supports a $WriteMax(v)$ operation, which writes the value $v$ to the object, and a $ReadMax$ operation; in its sequential specification, $ReadMax$ returns the maximum value written by a $WriteMax$ operation instance preceding it. In the bounded version of these objects, the max register is only required to satisfy its specification if its associated value does not exceed a certain threshold $M$. A counter object supports a $CounterIncrement$ operation and a $CounterRead$ operation; the sequential specification of a counter requires that a $CounterRead$ operation instance returns the number of $CounterIncrement$ operation instances that precede it. An (atomic) single-writer snapshot object consists of an array of $N$ segments. An $Update$ operation by process $p_i$ atomically sets the value of segment $i$. The $Scan$ operation atomically reads the values of all segments.

3. TRADEOFF FOR COUNTERS AND SNAPSHOT OBJECTS
In this section, we prove the following theorem.

Theorem 1. Let $I$ be an $N$-process obstruction-free implementation of a counter from the read, write and CAS primitives. If the step-complexity of $CounterRead$ is $O(f(N))$, then the step-complexity of $CounterIncrement$ is $Ω(\log \frac{N}{f(N)})$.

By way of reduction, we show a similar result for single-writer snapshot objects. A key idea underlying our proofs, previously used by [13], is that the rate at which processes become aware of the existence of others is not too rapid. Our proofs extend a technique presented in [6], based on the following definitions which allow quantifying the extent of inter-process communication.

Definition 1. Let $e$ be an event applied by process $p$ to a base object $o$ in an execution $E$, where $E = E_1eE_2$. We say
that e is invisible in E, if either the value of o is not changed by e or if \( E_2 = E'e' \), \( e' \) is a write event to o, \( p \) does not take steps in \( E' \), and no event in \( E' \) is applied to o.

Informally, an invisible event is an event by some process that cannot be observed by other processes. If an event e is applied to some object o in an execution E is not invisible, we say that e is visible in E on o.

We next capture the extent by which processes are aware of the participation of other processes in an execution. Intuitively, a process p is aware of the participation of another process q in an execution if there is (either direct or indirect) information flow from q to p in that execution via shared memory. The following definitions formalize this notion.

**Definition 2.** Let \( e_p \) be an event by process q in an execution E, which applies a write or a CAS primitive to a base object o. We say that an event \( e_p \) in E by process p is aware of \( e_o \) if \( e_o \) accesses o and at least one of the following holds: 1) There is a prefix \( E' \) of E such that \( e_p \) is visible on o in \( E' \) and \( e_p \) is a read or CAS event that follows \( e_o \) in \( E' \), or 2) there is an event \( e_t \) that is aware of \( e_o \) in E and \( e_t \) is aware of \( e_o \) in E. If an event \( e_o \) of process p is aware of an event \( e_q \) of process q in E, we say that p is aware of \( e_q \) and that \( e_p \) is aware of q in E.

**Definition 3.** Process p is aware of process q after an execution E if either \( p = q \) or if p is aware of an event of q in E. The awareness set of \( p \) after E, denoted \( AW(p, E) \), is the set of processes that p is aware of after E.

**Definition 4.** Let \( E \) be an execution, o be a base object, and q be a process. We say that o is familiar with q after E if there is an event e, visible on o in E, such that \( E = E_1eE_2 \) and e is an application of a write or a CAS primitive to o by some process r such that q \( \in AW(r, E_2) \). The familiarity set of \( o \) after E, denoted \( F(o, E) \), contains all processes that o is familiar with after E.

A more general version of the following lemma for implementations that use read, write and k-CAS primitives appears in [6]. For the sake of presentation completeness, we provide here a simpler proof for our model.

**Lemma 1.** Let \( E \) be an execution. Let \( \mathcal{M}(E) = \max_{p,o}(\{|AW(p, E)|\mid p \in P\} \cup \{|F(o, E)|\mid o \in B\}) \) be the maximum size of all familiarity and awareness sets after E. Let S be a set of enabled events by processes that are active after E, each about to apply a read, write or CAS primitive. Then there is a schedule \( \sigma(E, S) \) of these events such that \( \mathcal{M}(E|\sigma(E, S)) \leq 3 \cdot \mathcal{M}(E) \).

**Proof.** First, we schedule all read, trivial CAS and trivial write events in an arbitrary order. Let \( \sigma_1 \) denote the resulting execution fragment. No event in \( \sigma_1 \) is visible, hence \( \forall o \in B : F(o, E_1) \leq F(o, E) \). Moreover, for every \( p \in P \) that takes a step in \( \sigma_1 : AW(p, E_1) \leq AW(p, E) + \max_o(|F(o, E)|) \leq 2 \cdot \mathcal{M}(E) \).

We next schedule all remaining write events (in an arbitrary order) and let \( \sigma_2 \) denote the resulting execution fragment. Consider all the events of \( \sigma_2 \) that access a specific base object o, if there are any. Only the last of them, denoted by \( e_t \), is visible in \( E_1|\sigma_2 \); let \( p_t \) be the process that issues \( e_t \). Consequently, \( F(o, E_1|\sigma_2) = F(o, E) + AW(p_t, E) \leq 2 \cdot \mathcal{M}(E) \). Moreover, for every \( p \in P \) that takes a step in \( \sigma_2 : AW(p, E_1) \leq AW(p, E) + \max_o(|F(o, E)|) \leq 2 \cdot \mathcal{M}(E) \).

Finally, we schedule all remaining CAS events (in an arbitrary order) and denote the resulting execution fragment by \( \sigma_3 \). Consider all events of \( \sigma_3 \) that access a specific base object o, if there are any. Let \( e_f \) be the first of these events and let \( p_f \) be a process that issues \( e_f \). We consider two cases.

1. If o was written in \( \sigma_2 \), then \( e_f \) is non-trivial and does changes the value of o, making all consequent CAS events in \( \sigma_3 \) trivial. Hence \( F(o, E_1|\sigma_2|\sigma_3) = F(o, E_1) + AW(p_f, E) \leq 2 \cdot \mathcal{M}(E) \). Moreover, for every \( p \in P \) that takes a step in \( \sigma_3 : AW(p, E_1) \leq AW(p, E) + F(o, E_1) + AW(p_f, E) \leq 3 \cdot \mathcal{M}(E) \).

2. If o was not written in \( \sigma_2 \), then \( e_f \) is non-trivial and does changes the value of o, making all consequent CAS events in \( \sigma_3 \) trivial. Hence \( F(o, E_1|\sigma_2|\sigma_3) = F(o, E_1) + AW(p_f, E) \leq 2 \cdot \mathcal{M}(E) \). Moreover, for every \( p \in P \) that takes a step in \( \sigma_3 : AW(p, E_1) \leq AW(p, E) + F(o, E_1) + AW(p_f, E) \leq 3 \cdot \mathcal{M}(E) \).

**Lemma 2.** Let \( E \) be an execution and let \( p \in P \) be a process that issues events in E. Let \( E' \) be an execution obtained from E in the following manner. First, all the events issued by p are removed from E. Then, for every process \( q \neq p \), all the events of q that are aware of p are removed. Then \( E' \) is an execution.

**Proof.** Let \( e_q \in E' \) be an event by process q such that \( E' = E_1e_2E_2' \). Since \( e_q \in E \) we can write \( E = E_1e_2E_2 \). If \( |E_1| = k \) we denote \( E_1 \) by \( \sigma_k \) and \( E_2' \) by \( \sigma'_k \). By induction on k = 1, \ldots |E'| we prove that \( \sigma'_k \) is an execution.

**Base:** \( \sigma'_0 \) is empty and is obviously an execution.

**Hypothesis:** \( \sigma'_k \) is an execution.

**Step:** \( \sigma'_{k+1} = \sigma'_k e_q \). From induction hypothesis, \( \sigma_k \) is an execution. Assume towards a contradiction that \( \sigma_k e_q \) is not an execution, i.e. \( e_q \) returns different responses when scheduled after \( \sigma_k \) and after \( \sigma'_k \). Then \( e_q \) is aware of some event \( e_i \) that was removed from \( \sigma_k \). However, from Definition 2 and from the way \( E' \) is constructed from E, it follows that there is a finite sequence of events \( e_i, e_{i+1}, \ldots, e_j \), each of which is aware of its successor, such that \( e_j \) is an event of p. It follows from Definition 2 that \( e_q \) is aware of p and cannot appear in \( \sigma'_{k+1} \). This is a contradiction.

**Lemma 3.** Let \( I \) be a linearizable obstruction-free implementation of a counter and let \( EE_1 \) be an execution of I such that each of the processes of \( P' = \{p_1, \ldots, p_{n-1}\} \) completes a CounterIncrement operation in E, and \( EE_1 \) is an extension of E, in which \( p_n \) performs to completion a single CounterRead operation. Then \(|AW(p_n, EE_1)| = N \).

**Proof.** Assume towards a contradiction that \(|AW(p_n, EE_1)| < N \). Hence, \( p_n \) is not aware of some process \( p_i \in P' \). We construct from \( EE_1 \) a new execution \( E' \) in the following manner. First, we remove from \( EE_1 \) all the events of \( p_i \). Then, for each \( p_k \in P' \), if any of p_k’s events in E in aware of \( p_i \), then we remove all the events of \( p_k \) starting from the first event of \( p_k \) that is aware of \( p_i \). Since \( p_N \) is unaware of \( p_i \), all its events are preserved in \( E' \). From Lemma 2, \( E' \) is an execution.

From Definitions 2-3, since \( p_N \) is unaware of \( p_i \) in \( EE_1 \), \( EE_1 \) and \( E' \) are indistinguishable to \( p_N \). From linearity, \( p_N \)’th CounterRead operation returns \( N - 1 \) in \( EE_1 \) but
must return a smaller value in \( E' \), since at most \( N - 2 \) CounterIncrement operations were completed in \( E' \). This is a contradiction. □

**Proof of Theorem 1.** We iteratively construct an execution \( E = \sigma_1 \sigma_2, \ldots, \sigma_r \), in which each of the processes of \( P' = \{p_1, \ldots, p_{N-1}\} \) completes a CounterIncrement operation. For \( j \in \{1, \ldots, r\} \), we let \( E_j = \sigma_1 \sigma_2, \ldots, \sigma_j \). Our construction maintains the invariant that for all \( j \in \{1, \ldots, r\} \) and \( o \in B \), \( |F(o, E_j)| \leq 3' \).

In the initial configuration, the familiarity set of all base objects is empty; the awareness set of each process contains only itself, and each of the processes of \( P' \) has an enabled event. Let \( S \) denote the set of these events. From Lemma 1, there is a schedule \( \sigma_i = \sigma(1, S) \) such that \( M(\sigma_i) = M(E_i) \leq 3 \).

Assume we have constructed execution \( E_i \). If a subset \( Q \subseteq P' \) of processes did not complete their CounterIncrement operations in \( E_i \), then we construct execution \( E_{i+1} = E_i \sigma_{i+1} \) as follows. Let \( S \) denote the set of the events of the processes of \( Q \) that are enabled after \( E_i \). From Lemma 1, there is a schedule \( \sigma_{i+1} = \sigma(E, S) \) such that \( M(E_{i+1}) \leq 3M(E_i) \leq 3^{i+1} \). We proceed in this manner until all the processes of \( P' \) complete their operations or until we complete \( \left\lceil \log_3 \left( \frac{N}{f(N)} \right) \right\rceil \) iterations, whatever occurs first.

Let \( p_i \) be a process that does not terminate its CounterIncrement operation in \( E_{i-1} \). There must be such a process, as otherwise the construction of \( E \) would have stopped after the \( (r - 1) \)th iteration. Clearly from our construction, \( p_i \) issues \( r \) events in \( E \). To prove the theorem, we will show that \( r = \Omega \left( \log_3 \left( \frac{N}{f(N)} \right) \right) \).

Assume towards a contradiction that \( r = o \left( \log_3 \left( \frac{N}{f(N)} \right) \right) \), implying that all the processes of \( P' \) completed their CounterIncrement operations in \( E \). From our construction, the following holds:

\[
\forall o \in B : |F(o, E)| \leq 3' = o \left( \frac{N}{f(N)} \right). \tag{1}
\]

Let \( E_1 \) be an extension of \( E \) in which \( p_N \) performs a CounterRead operation. Since \( p_N \) performs \( O(f(N)) \) steps in \( E_1 \), it accesses at most \( O(f(N)) \) base objects. Therefore, it follows from Equation 1 that \( |AW(p_N, E_1)| = o(N) \). This is a contradiction to Lemma 3. □

Given an \( N \)-component single-writer snapshot object, it is straightforward to implement a counter as follows. To perform a CounterIncrement operation, process \( p_i \) increments the value of the \( i \)th component by performing a single Update operation. To read the counter, a process performs a single Scan operation and returns the sum of all components. We get the following:

**Corollary 1.** Let \( I \) be an \( N \)-process obstruction-free implementation of a single-writer snapshot object from the read, write and CAS primitives. If the step-complexity of Scan is \( O(f(N)) \), then the step-complexity of Update is \( \Omega \left( \log_3 \left( \frac{N}{f(N)} \right) \right) \).

Jayanti presented an \( f \)-arrays-based \([14]\) construction of counters and snapshots where CounterRead and Scan operations have constant step complexity, and CounterIncrement and Update operations have logarithmic step complexity. His construction uses the load-link/store-conditional primitives in addition to reads and writes, but can be made to work also using CAS instead.

It follows from Theorem 1 and Corollary 1 that no constant-read implementation for these objects from read, write and CAS can have update operations with sub-logarithmic step complexity. This follows also from another work of Jayanti \([13]\) where he shows that the sum of steps performed by a CounterIncrement operation followed by a CounterRead operation (or an Update operation followed by a Scan operation) is logarithmic. The following theorem, however, does not follow from \([13]\).

**Theorem 2.** Let \( I \) be an obstruction-free \( N \)-process implementation of a counter (respectively single-writer snapshot) object from reads and writes that supports only a limited (polynomial in \( N \)) number of CounterIncrement (respectively Update) operations. If the worst-case step complexity of \( I \)'s CounterRead (respectively Scan) operations is \( O(1) \) (that is, \( O(\log N) \)), then the worst-case step-complexity of its CounterIncrement (respectively Update) operations is \( \Omega(\log N) \).

**Proof.** Immediate from Theorem 1 and Corollary 1. □

### 4. Tradeoff for Max Registers
In this section, we prove the following tradeoff.

**Theorem 3.** Let \( I \) be an \( N \)-process obstruction-free implementation of an \( M \)-bounded maxRegister from the read, write and CAS primitives. Let \( K = \min\{M, N\} \). If the step complexity of ReadMax is \( O(f(K)) \), then there is an execution of \( I \) in which each of \( O(f(K)) \) processes takes \( \Omega(\log \frac{\log K}{\log f(K)}) \) steps as it performs a single WriteMax operation.

Our proofs combine and extend techniques from \([1, 6, 15]\). We start by describing our proof strategy. This is followed by formal proofs.

We construct an execution involving \( \Theta(f(K)) \) processes, each performing \( \Omega(\log \frac{\log K}{\log f(K)}) \) steps. The construction proceeds iteratively, where iteration \( i \) constructs execution \( E_i \). Initially, we have a set \( P' = \{p_1, \ldots, p_K\} \) of \( K - 1 \) processes such that, for \( i \in \{1, \ldots, K - 1\} \), \( p_i \) is about to perform a WriteMax \((i)\) operation.

A key concept in our construction is that of an essential set. Each execution \( E_i \) is associated with an essential set \( E_i \subseteq P' \) (the essential set of \( E_i \)) of size \( \Omega(\sqrt{K}) \) that satisfies the following properties: 1) Each process in \( E_i \) performs exactly \( i \) steps in \( E_i \). 2) If \( p \) is in \( E_i \), then \( q \neq p \) is aware of \( p \) after \( E_i \). 3) No base object is familiar with more than a single process in \( E_i \). 4) The identifiers of the processes of \( E_i \) are larger than those of all \( M \) other processes that issue events in \( E_i \).

The properties stated above guarantee that not too many of the processes of \( E_i \) may complete their operations in \( E_i \). To see why, consider a set \( F \subseteq E_i \) of \( m \) such processes. As we prove, a ReadMax operation \( \Phi \) by \( p_K \), scheduled after \( E_i \), must access \( m \) distinct base objects in the course of its execution - those base objects that are familiar with the processes of \( F \). If \( \Phi \) fails to read even one such base object (say, the one familiar with \( p_j \)), then we can remove all the events issued by \( E_i \backslash \{p_j\} \) from \( E_i \). In this case, \( \Phi \) will fail to return \( j \), which is the largest value written prior to \( \Phi \)'s execution. It follows that at most \( O(f(n)) \) processes of \( E_i \) can terminate in \( E_i \).
Let us denote by \( \mathcal{E}_i^+ \) the set of those processes of \( \mathcal{E}_i \) that remain active after \( E_i \). Our goal is to pick a relatively large subset \( \mathcal{E}_{i+1} \subseteq \mathcal{E}_i^+ \), each of whose processes can perform an additional step, while limiting information flow so that properties 1)-4) above hold for \( \mathcal{E}_{i+1} \). We now describe how we select \( \mathcal{E}_{i+1} \) and construct \( E_{i+1} \).

First, all the events of the processes of \( \mathcal{E}_i \) are removed. We say that these processes are erased from the execution (or simply erased). Then, we consider the next event that will be issued by the processes of \( \mathcal{E}_i \). Specifically, we consider the number \( j \) of distinct base-objects that these events will access. The following two cases exist.

**Case 1 (Low Contention):** If \( j > \sqrt{|\mathcal{E}_i|} \), we pick an arbitrary set \( A' \) of \( \sqrt{|\mathcal{E}_i|} \) processes accessing distinct base objects. We then pick a subset \( A \subseteq A' \) such that no step by a process of \( A \) is about to access a base object that has another process of \( A \) in its familiarity set. As we show, a constant fraction of the processes of \( A' \) remain in \( A \). The set \( A \) is defined as \( \mathcal{E}_{i+1} \).

To construct \( E_{i+1} \), the processes of \( \mathcal{E}_i \setminus A \) are also erased. \( E_{i+1} \) is obtained by extending the remaining execution by allowing each of the processes of \( \mathcal{E}_{i+1} \) to perform an additional step. We refer to this case as the low-contention scenario. Figure 1 illustrates the construction of \( \mathcal{E}_{i+1} \) in the low-contention scenario.

**Figure 1: Low contention case for iteration \( i \)**

The processes that are not erased from the execution form the essential set \( \mathcal{E}_{i+1} \).

**Case 2 (High Contention):** Otherwise \( (j \leq \sqrt{|\mathcal{E}_i|}) \), we pick an arbitrary base object \( o \) accessed by a subset (denoted \( P^o \)) of at least \( \sqrt{|\mathcal{E}_i|} \) processes from \( \mathcal{E}_i \). We erase all the processes of \( \mathcal{E}_i \setminus P^o \) from the execution. As for the processes of \( P^o \), there are two possibilities. Either we can schedule their steps so that none of them becomes visible on \( o \) (in which case they will all belong to \( \mathcal{E}_{i+1} \)); or we schedule them so that a single process, say \( p_k \), becomes visible and a constant fraction of the processes of \( P^o \) have bigger identifiers. These are the processes that will form \( \mathcal{E}_{i+1} \). As for \( p_k \), we say that it is halted. It will not issue additional events in later iterations, nor will it be a part of essential sets of later iterations. The rest of the processes of \( P^o \) are erased from the execution. We refer to this case as the high contention scenario. Figure 2 illustrates the construction of \( \mathcal{E}_{i+1} \) in the high-contention scenario.

**Figure 2: High contention case for iteration \( i \)**

The processes not in \( P^o \) are erased from the execution. Process \( p_2 \) becomes halted. The rest of the processes in \( P^o \) form the essential set \( \mathcal{E}_{i+1} \).

**Figure 3: The construction of execution \( E \)**

Erased processes do not appear in \( E \). Halted processes issue an event in each construction iteration until they become halted. Essential set processes issue an event in every construction iteration.

denote an execution of a max register implementation \( I \) and we let \( P \) be the set of processes taking steps in \( E \).

**Definition 5.** Let \( O \) be the set of all base objects used by \( I \). We say that a process \( p \in P \) is hidden after \( E \), if \( \forall p' \in P : p \in \text{AW}(p', E) \rightarrow p = p' \) (informally, no process except \( p \) is aware of \( p \) after \( E \)). We say that \( P' \subset P \) is a hidden set after \( E \), if the processes of \( P' \) are hidden after \( E \) and if \( \forall o \in O : |F(o, E) \cap P| \leq 1 \) (informally, each base object in \( O \) is familiar with at most a single process in \( P \) after \( E \)).

**Claim 1.** Let \( P' \) be a set of processes hidden after \( E \), then \( E' = E^{P'} \) is an execution.

**Proof:** From Definition 5, for all \( p \in P' \), no process but \( p \) is aware of \( p \) after \( E \), hence the claim follows from Lemma 2.

**Definition 6.** We say that \( P' \subset P \) is a supreme set, if \( \forall p_i \in P' : \forall p_j \in P \setminus P' : i > j \) holds (informally, the processes of \( P' \) have the highest indices out of all processes that issue events in the execution).
Definition 7. We say that $P' \subseteq P$ is an $i$-step essential set of $E$, if $P'$ is a supreme set hidden after $E$, such that every process in it issues exactly $i \ events$ in $E$.

Lemma 4. Let $E_i$ be an execution and let $E_i'$ be an $i$-step essential set of $E_i$. Let $E_i' \subseteq E_i$, be the set of those processes of $E_i$ that have an enabled event after $E_i$ and let $m = |E_i'|$ denote its size. If $m \geq 81$, then there exists an execution $E_{\delta+1}$ with an $(i+1)$-step essential set of size at least $\sqrt{m - 2}$.

Proof. Let $S = \{e_1, \ldots, e_m\}$ be the set of events that are to be scheduled, and let $E_i' \subseteq E_i$ be the set of all processes that are about to access an object $o$. Let $E_i' \subseteq E_i$ denote the set of all processes from $E_i'$ that are about to access $o$. There are two cases to consider.

Case 1 (Low Contention): For each base object $o \in O : |P_o| \leq \sqrt{m}$. In this case, the steps by $E_i'$ access $k \geq \sqrt{m}$ distinct base objects. For every accessed object, we arbitrary pick a process $p_i \in P_o$ and denote the resulting set by $P_{\delta+1}$. We build an undirected graph $G = (V, E)$ such that $V = \{v_p \in P | p \in P_o\}$ and $E = \{<v_i, v_o> | p_i^\delta \in F(o, E_i)\}$. It follows immediately from the construction that $|V| = |P| = k$.

Case 2 (High Contention): There is a base object $o \in O$ such that $|P_o| > \sqrt{m}$. We consider the processes of $P_o$ according to the type of operation they are about to apply to $o$ after $E_i$: we let $P' \subseteq P_o$ denote those processes about to apply a CAS event that will change the value of $o$ (if applied immediately after $E_i$); we let $P' \subseteq P_o$ denote those processes about to apply a write event, and we let $P' \subseteq P_o$ denote those processes about to apply a read or a CAS event that will not change the value of $o$. We need to consider the following three sub-cases.

Sub-case 1: $|P_o| \leq \sqrt{m}$. Let $S = F(o, E_i) \cap E_i'$. After execution $E_i$, object $o$ is familiar with at most a single process of $E_i'$, thus $|S| \leq 1$. Let $p_i$ be the process with the smallest identifier of all the processes of $P_o$, and let $E_i + 1 = P_o \setminus ((p_i) \cup S)$ (since $m \geq 81$, $E_i + 1$ is non-empty). Let $\sigma$ be a sequence of events in which we schedule all the CAS events by the processes of $E_i + 1$ (in some arbitrary order), preceded by the non-trivial CAS by $p_i$. Let $K = (E_i \cap P_o) \cup S$. From Claim 4, $E_i + 1 \subseteq E_i + 1$ is an execution. We now show that $E_i + 1$ is an $(i+1)$-step essential set of $E_i + 1$.

Sub-case 2: $|P_o| > \sqrt{m}$. Let $p_i$ be the process with the smallest identifier of all the processes of $P_o$, and let $E_i + 1 = P_o \setminus \{p_i\}$ (since $m \geq 81$, $E_i + 1$ is non-empty). Let $\sigma$ be a sequence of events in which we schedule all the write events (in some arbitrary order) by the processes of $E_i + 1$ followed by the write by $p_i$. Let $K = (E_i \setminus P_o)$. Since $E_i$ is hidden after $E_i$, it follows from Claim 4 that $E_i + 1 = E_i - K$ is an execution. We now show that $E_i + 1$ is an $(i+1)$-step essential set of $E_i + 1$.

Sub-case 3: $|P_o| > \sqrt{m}$. Let $S = F(o, E_i) \cap E_i'$. After execution $E_i$, object $o$ is familiar with at most a single process of $E_i'$, thus $|S| \leq 1$. Let $E_i + 1 = P_o \setminus S$ (since $m \geq 81$, $E_i + 1$ is non-empty) and let $\sigma$ be a sequence of events in which we schedule all the read and trivial CAS events by the processes of $E_i + 1$ in some arbitrary order. Let $K = (E_i \setminus P_o) \cup S$. Since $E_i$ is hidden after $E_i$, it follows from Claim 4 that $E_i + 1 = E_i - K$ is an execution. We now show that $E_i + 1$ is an $(i+1)$-step essential set of $E_i + 1$.
$\mathcal{E}_i$ is an essential set of $E_i$, $\mathcal{E}_{i+1} \subset \mathcal{E}_i$ and all the processes of $\mathcal{E}_i \setminus \mathcal{E}_{i+1}$ are erased, hence $\mathcal{E}_{i+1}$ is a supreme set of $\mathcal{E}_{i+1}$. Since $\mathcal{E}_i$ is an $i$-step essential set of $E_i$, every process of $\mathcal{E}_i$ issues exactly $i$ events in $E_i$. It follows immediately from our construction of $\mathcal{E}_{i+1}$ that every process of $\mathcal{E}_{i+1}$ issues exactly $i + 1$ events in $E_{i+1}$. Thus, $\mathcal{E}_{i+1}$ is an $(i + 1)$-step essential set. Finally, it is immediate from our construction that $|\mathcal{E}_{i+1}| \geq \sqrt{|\mathcal{E}_i|} - 1$. □

**Lemma 5.** Let $E$ be an execution in which a process $p_i$ that is hidden after $E$ completes its $\text{WriteMax}$ operation. Assume also that $p_i$ wrote a (unique) maximum value in $E$. Let $p_j$ be a process that issued no events in $E$ and let $\Phi$ be an execution of a $\text{ReadMax}$ operation by $p_j$, immediately after $E$. Then $p_j$ must access in $\Phi$ an object familiar with $p_i$.

**Proof.** Assume towards a contradiction that $p_j$ accesses no such object. Consider execution $E'$ obtained from $E$ by removing all the events by $p_j$ from $E$. From Claim 1, $E'$ is an execution. Since $p_i$ is hidden in $E$, $E'$ is indistinguishable from $E$ for $p_j$. Consequently, $p_j$ performs $\Phi$ also after $E'$ and returns the same response in both $E\Phi$ and $E'\Phi$. This is a contradiction since the maximum values written in $E$ and $E'$ differ. □

**Lemma 6.** Let $E$ be an execution in which each process $p_i \in P \subseteq \{p_1, \cdots, p_K - 1\}$ performs a $\text{WriteMax}(i)$ operation. Let $A$ be a hidden and supreme set of $E$ and let $E^* \subset A$ denote the set of those processes of $A$ that complete their operation in $E$. If the step complexity of $\mathcal{E}_1$ operations is at most $m$, then $|E^*| \leq m$.

**Proof.** We iteratively construct an execution $E'$ such that, if $|E^*| > m$, then process $p_{K}$ must miss the maximum value written to the max object when it performs its $\text{WriteMax}$ operation after $E'$. We denote the execution obtained after $r$ iterations as $E'_r$. Our construction starts with $E'_0 = E^{-A}E\cdot$. From Claim 1, $E'_0$ is an execution.

We construct $E'_r$ from $E'_{r-1}$ as follows. Let $e_r$ be the $r$-th event about to be issued by $p_K$ after $E'_{r-1}$ and let $o$ be the object it will access. We let $I = F(o, E'_{r-1}) \cap E^*$ and $E'_r = E'_{r-1} \cdot e_r$. Since $E^*$ is a hidden set, $|I| \leq 1$, hence $E'_r$ is either $E'_{r-1} \cdot e_r$ or $E'_{r-1} \cdot (p_i) \cdot e_r$ for some $p_i \in E^*$. Since $E'_{r-1}$ is an execution and since $p_i$ (if it exists) is hidden, it follows from Claim 1 that $E'_r$ is an execution. Since none of the events of $p_K$ access an object that is familiar with a process of $A$, $E'_r \cdot e_1 \cdot e_2 \cdots$ is an execution as well. The construction stops after $p_K$ takes its last step, hence it stops after at most $m$ iterations.

Assume towards a contradiction that $|E^*| > m$. Let $K$ be the set of processes erased in the course of the construction of $E'$. Since $|K| \leq m$, the set $E^* \setminus K$ is non-empty. Let $p_{\text{max}}$ be the process in $E^* \setminus K$ with maximum ID. Since $p_{\text{max}} \in E^* \subset A$ and from construction, $p_{\text{max}}$ is the process that performed in $E'$ the $\text{WriteMax}$ operation with the maximum operand. Moreover, $p_{\text{max}}$ completed its operation in $E'$.

From Lemma 5, $p_{K}$ must access an object familiar with $p_{\text{max}}$. However, our construction of $E'$ ensures that none of the objects accessed by $p_{K}$ is familiar with a process of $E^*$. This is a contradiction. □

**Proof of Theorem 3.** We construct an execution $E$ with the required properties. The construction proceeds in iterations. In iteration $i$, we construct an execution $E_i$ that has an $i$-step essential set $\mathcal{E}_i$ of size $\Omega(K^{1/3})$. Initially, we have a set $P' = \{p_1, \cdots, p_{K-1}\}$ of $K - 1$ processes such that, for $i \in \{1, \cdots, K - 1\}$, $p_i$ is about to perform a $\text{WriteMax}(i)$ operation.

For the base case, we let $E_0$ be the empty execution and claim that $P'$ is a 0-step essential set. In the initial configuration, all processes are aware only of themselves and all base objects have empty familiarity sets. Hence, all processes are hidden after the empty execution. Moreover, $P'$ is supreme and all processes in it issue 0 events in $E_0$. It follows from Definition 7 that $P'$ is a 0-step essential set of $E_0$.

The construction stops after the first iteration $i^*$, such that at least half of the processes of $\mathcal{E}_i$ terminate in $E_{i^*}$, or $|\mathcal{E}_{i^*+1}| < f(K)$, whichever happens first. In the first case, from Lemma 6:

$$|\mathcal{E}_{i^*}| \leq 2 \cdot f(K).$$

In the second case, we get from Lemma 4 and from our construction:

$$|\mathcal{E}_{i^*}| = O(|\mathcal{E}_{i^*+1}|) = O(f(K)^2).$$

On the other hand, from Lemma 4, we have:

$$\forall i \in \{1, \cdots, i^*\} : |\mathcal{E}_i| \geq \frac{1}{2} \left( |\mathcal{E}_{i-1}| \right)^{1/2} - 2 = \Omega(|\mathcal{E}_{i-1}|^{1/3})$$

$$\Rightarrow |\mathcal{E}_{i^*}| = \Omega(K^{1/3}).$$

Combining Equations 2, 3 and the right-hand equality of Equation 4, we get $i^* = \Omega(\log \log K)$. The theorem now follows from the fact that each process in $\mathcal{E}_i$ issues $i^*$ events in $E_{i^*}$ and the fact that, from our construction, $|\mathcal{E}_{i^*}| \geq f(K)$. □

The following is immediate from Theorem 3.

**Theorem 4.** Let $I$ be a obstruction-free $N$-process implementation of an $M$-bounded max register from read, write and CAS. If the worst-case step complexity of $I$’s $\text{ReadMax}$ operation is $O(\log M)$, then the worst-case step-complexity of its $\text{WriteMax}$ operation is $O(\log \log \min(N, M))$.

5. MAX REGISTER IMPLEMENTATIONS USING READ/WRITE/CAS

We now present a wait-free max$\text{Register}$ algorithm, using read, write and CAS primitives. The step-complexity of a max$\text{Register}$ operation is constant and the step complexity of a $\text{maxRegister}(v)$ operation is $O(\log \min(N, v))$. The algorithm uses a binary tree $T$, every node of which stores an integer value (initialized to $-\infty$) and pointers to its child nodes and parent node. The left sub-tree of $T$, denoted as $T_L$, is an unbalanced binary tree with $N - 1$ leaves such that its $i$-th leaf is at depth $O(\log i)$ (the construction of such a tree, referred to as a $B_1$ tree, is introduced in [8]). The right sub-tree of $T$, denoted as $T_R$, is a complete binary tree with $N$ leaves. An illustration of the data structure for $N = 4$ processes is depicted in Figure 4.

The pseudo-code appears in Algorithm A. To perform a $\text{ReadMax}$, a process simply reads the value stored in the root. To perform a $\text{WriteMax}(v)$ operation, process $p_i$ writes $v$ to a leaf $\mathcal{L}$ and attempts to propagate it up to the root of $T$ in a manner we soon explain. $\mathcal{L}$ is selected as follows: if $v < N$
Figure 4: The data structure for \texttt{maxRegister} shared by \( N = 4 \) processes

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{maxRegister}
\caption{The data structure for \texttt{maxRegister} shared by \( N = 4 \) processes.}
\end{figure}

\begin{algorithm}[h]
\caption{Max Register}
\begin{algorithmic}[1]
\Function {Read} \ alum \end{algorithmic}
\end{algorithm}

then \( L \) is the \( v \)-th leaf of \( T_L \), otherwise it is the \( i \)-th leaf of \( T_R \).

Propagating a value given as operand to a \texttt{WriteMax} operation up the tree is implemented by the \texttt{Propagate} procedure, which works similarly to the Tree Algorithm of Jayanti [14]. At each level of the tree along the path from the leaf to the root, a process computes the maximum of the value of its current node and its sibling and attempts to write this maximum to the parent node by using \texttt{CAS}. Since the \texttt{CAS} may fail, the computation of the maximum value and the \texttt{CAS} are performed twice at each level. This ensures that if the \texttt{CAS} failed, then \( T_L \) by another process must have succeeded in updating the parent node based on the new value.

Now we prove that Algorithm A is correct and wait-free, that the step-complexity of the \texttt{ReadMax} operation is constant and that the step complexity of the \texttt{WriteMax}(\( v \)) operation is \( O(\min(\log N, \log v)) \).

\textbf{Linearizability.}

In the following, we prove that Algorithm A is linearizable [12]. For every operation \( \Phi \), in an execution \( E \) of \( A \), we uniquely identify a linearization point, i.e. an event \( e, \in \mathcal{C}(\Phi) \) within the time interval of \( \Phi \). For simplicity and \text{WLOG}, we assume that no process tries to write a value which is smaller than the values it has written earlier.

\textbf{Definition 8.} We say that a node \( N \) counts an operation \( \Phi \) after \( E \), if \( E = E' \varepsilon E'' \), \( e \) is a visible event on \( N \)’s value and either \( e \) is issued in \( \Phi \) or it was issued by some process \( p \) such that

1. \( E' = E_1 \varepsilon E_2 \), where \( \varepsilon' \) is a read by \( p \) of \( N' \)’s value.
2. \( N'' \) is a child of \( N \) and \( N'' \) counts \( \Phi \) after \( E_1 \).

We denote the set of all operations counted by \( N \) after \( E \) as \( \mathcal{C}(N, E) \).

Clearly, if \( N \) counts \( \Phi \) after \( E \), then it counts \( \Phi \) in all extensions of \( E \); and if \( N \) does not count \( \Phi \) after \( E \), then it does not count \( \Phi \) in any prefix of \( E \).

\textbf{Definition 9.} For \texttt{ReadMax} operation \( \Phi \), in \( E \) we define the linearization point as the read event on \( r . v a l u e \) (line 2). For \texttt{WriteMax} operation \( \Phi \), in \( E \) we define the linearization point as the first event \( e \) after which \( T \) counts \( \Phi \), i.e. if \( E = E_1 \varepsilon E_2 \) then \( \Phi \) does not count \( \Phi \) in \( E \). Moreover, we say that event \( e \) precedes (is preceded by) operation \( \Phi \) in \( E \) if \( e \) is issued before the first (after the last) event of \( \Phi \) and denote this by \( e \prec \Phi \) (\( \Phi \prec e \)). We now prove the following:

1. \( H \) preserves the partial order of operations in \( E \). It is sufficient to prove that the linearization point \( e \) of every operation \( \Phi \) is defined uniquely within the time interval of \( \Phi \), i.e. \( -e \prec \Phi \) and \(-\Phi \prec e \).
2. \( H \) is legal sequential history of \texttt{maxRegister}. It is sufficient to prove that every \texttt{ReadMax} operation in \( H \) returns the maximum value written before it in \( H \).

\textbf{Observation 1.} By Definition 9, the linearization point \( e \) of \texttt{ReadMax} operation \( \Phi \) is a read issued by \( \Phi \) on \texttt{T.root}’s value. Obviously, \( e \) is within the time interval of \( \Phi \) and is defined uniquely.

\textbf{Observation 2.} By Definition 9, the linearization point \( e \) of \texttt{WriteMax} operation \( \Phi \) is the first and thus unique event, after which \( \Phi \) is counted by \texttt{T.root}.

\textbf{Lemma 7.} Let \( \Phi \) be a \texttt{WriteMax} operation in \( E \) such that its linearization point \( e \), as specified by Definition 9, is also in \( E \). Then \( e \) does not precede \( \Phi \) in \( E \).

\textbf{Proof.} Assume towards a contradiction that \( e \prec \Phi \), i.e. \( \Phi \in \mathcal{C}(\texttt{T.root}, E') \), where \( E' \) is a prefix of \( E \). Since no events of \( \Phi \) are issued in \( E' \), from Definition 8, some child of \( \texttt{T.root} \) counts \( \Phi \) in \( E' \). We apply this argument recursively until it implies that some leaf \( L \) should count \( \Phi \) in \( E' \). Consequently, from Definition 8, \( L \) must be written by \( \Phi \) in \( E' \). On the other hand, \( \Phi \) issues no events in \( E' \). This is a contradiction. □
Let \( \mathcal{N} \) be some internal node of \( T \) and let \( p_i \) be a process that accesses \( \mathcal{N} \) when executing a \texttt{WriteMax} operation \( \Phi \). For \( i \in \{1, 2\} \) we denote the \( i \)-th read of \( \mathcal{N} \) by \( r_i \) in line 7 as \( \text{read}^i_1(\mathcal{N}, \Phi) \), the \( i \)-th read of \( \mathcal{N} \cdot \text{left} \cdot \text{child} \cdot \text{value} \) by \( p_i \) in line 8 as \( \text{read}^i_2(\mathcal{N}, \Phi) \), the \( i \)-th read of \( \mathcal{N} \cdot \text{right} \cdot \text{child} \cdot \text{value} \) by \( p_i \) in line 8 as \( \text{read}^i_3(\mathcal{N}, \Phi) \) and the \( i \)-th \text{CAS} of \( N \cdot \text{value} \) by \( p_i \) in line 9 as \( \text{cas}^i_1(\mathcal{N}, \Phi) \). The following lemma will help us prove that a \texttt{WriteMax} operation does not precede its linearization point.

**Lemma 8.** In execution \( E \), the sequence of values stored in every node of \( T \) is non-decreasing.

**Proof.** First, we consider values stored in the leaves of \( T \). The \( v \)-th leaf of \( T_v \) stores either \( -\infty \) (initial value) or \( v \geq 0 \) (written in line 17), hence its values are non-decreasing. The \( i \)-th leaf of \( T_e \) is written only by \( p_i \), hence it only stores the operands of \texttt{WriteMax} operations by \( p_i \), that are always non-decreasing.

We now consider the internal nodes of \( T \). Assume towards a contradiction that the claim of the lemma is violated in \( E \) and let event \( e \), issued by some process \( p_i \), be the first event violating the claim, by changing \( N \cdot \text{value} \), for some node \( N \), from value \( x \) to a smaller value \( y \). Obviously \( e \) is a non-trivial \text{CAS} event issued by \( p_i \), when executing some \texttt{WriteMax} operation \( \Phi \). WLOG, \( e \) is a \text{cas}^1_1(\mathcal{N}, \Phi) \) and \( y \) is the maximum of values obtained by \( \text{read}^1_1(\mathcal{N}, \Phi) \) and \( \text{read}^1_3(\mathcal{N}, \Phi) \).

Since \( e \) changes the value of \( N \cdot \text{value} \), according to the algorithm \( A \), the value is obtained by \( \text{read}^1_2(\mathcal{N}, \Phi) \). Let \( p_i \) be the process that wrote \( x \) to \( N \cdot \text{value} \), when executing a \texttt{WriteMax} operation \( \Phi \). WLOG, \( p_i \) obtained \( x \) from \( \text{read}^1_2(\mathcal{N}, \Phi) \) and wrote it to \( N \cdot \text{value} \) in \text{cas}^1_1(\mathcal{N}, \Phi) \). It holds that:

\[
\text{read}^1_1(\mathcal{N}, \Phi) \preceq \text{cas}^1_1(\mathcal{N}, \Phi) \preceq \text{read}^1_2(\mathcal{N}, \Phi) \preceq \text{read}^1_3(\mathcal{N}, \Phi).
\]

On the other hand, \( \text{read}^1_2(\mathcal{N}, \Phi) \) returns \( x \), while \( \text{read}^1_3(\mathcal{N}, \Phi) \) returns \( y \). Hence the maximal value stored in \( N \cdot \text{value} \) decreases in the execution prior to \( E \). Thus \( e \) is not the first event violating the claim. This is a contradiction.

**Lemma 9.** Let \( \Phi \) be a \texttt{WriteMax} operation in \( E \) such that its linearization point \( e \), as specified by Definition 9, is also in \( E \). Then \( \Phi \) does not precede \( e \) in \( E \).

**Proof.** If \( \Phi \) does not complete in \( E \) then obviously \( \neg (e \preceq \Phi) \). Hence we only consider operations that complete in \( E \). Let \( E' \) be a prefix of \( E \) such that \( \Phi \) completes in \( E' \). We prove that \( \Phi \in \mathcal{C}(T \cdot \text{root}, E') \). We consider this claim as a special case of the following invariant.

**Invariant 1.** Let \( e'^\mathcal{N} \) be the last write or \text{CAS} event of \( \Phi \) that accesses some node \( \mathcal{N} \) in \( E \). Let \( E = E_1^{e'_1} E_2^{e'_2} \cdot \). Then \( \Phi \in \mathcal{C}(\mathcal{N}, E_2^{e'_2} E_2^{e'_2}) \)

Let \( N_1, \ldots, N_d \), where \( d \) is the depth of the leaf \( \mathcal{L} \) written by \( \Phi \), denote the nodes in the propagation path of \( \Phi \), such that \( N_1 \) is \( \mathcal{L} \) and \( N_d \) is \( T \cdot \text{root} \). We prove by induction on \( r = 1, \ldots, d \) that the invariant holds for every node \( N_r \).

For the induction base, note that \( N_1 \) is a leaf and thus \( e_{1}^\mathcal{N} \) is written by \( \Phi \). Hence, \( \Phi \in \mathcal{C}(N_1, E_1^{e'_1} E_2^{e'_2}) \) and the invariant holds for \( N_1 \).

For the induction step, assume that \( \Phi \in \mathcal{C}(N_r, E_r^{e'_r} E_2^{e'_2}) \). Let \( \Phi \) be an operation by \( p_i \) that issues \( e_{r+1}^\mathcal{N} \). WLOG, \( N_r \) is the left child of \( N_{r+1} \) and \( e_{r+1}^\mathcal{N} = \text{cas}^1_1(N_{r+1}, \Phi) \).
According to Algorithm A, $e^{N_E}$ is read by $\text{read}_{\text{LC}}(N_{r+1}, \Phi') \equiv \text{cas}^1(N_{r+1}, \Phi') \equiv e^{N_{r+1}}$. Since $\Phi \in C(N_r, E_{N_{r+1}}^{N_r})$, from Definition 8, $\Phi \in C(N_{r+1}, E_{N_{r+1}}^{N_{r+1}})$ and the invariant holds for $N_{r+1}$. This is a contradiction.

**Lemma 11.** Let $\Phi$ be a $\text{WriteMax}$ operation by $p_i$ with operand $v$. Let event $e$ be the linearization point of $\Phi$ in $E$.

If $E = E'eE''$, then $T$.root.value $\geq v$ after $E'e$.

**Proof.** From Definition 9, $\Phi \in C(T$.root$, E'e)$. Let $L$ be the leaf accessed by $\Phi$. Let $N_1, \ldots, N_d$, where $d$ is the depth of $L$, denote the nodes in the path of $\Phi$, such that $N_1$ is $L$ and $N_d$ is $T$.root. For $r = 1, \ldots, d$, let $e^{N_r}$ denote the first event in $E$ such that $E' = E^{N_1}e^{N_r}E''$ and $\Phi \in C(N_r, E_{N_{r+1}}^{N_r})$.

From Definition 9, $E'e \equiv E_{N_1}^{N_d}e^{N_r}$. We prove that after $E_{N_1}^{N_d}e^{N_r}$ the value of $T$.root $\geq v$, considering this as a special case of the following invariant: for $r = 1, \ldots, d$, $N_r$.value $\geq v$ after $E_{N_1}^{N_d}e^{N_r}$. The proof is by induction on $r$.

For induction base, note that $N_1$ is a leaf and thus $e^{N_1}$ is the write of value $v$ on $N_1$.value (in line 17). Hence $N_1$.value $= v$ after $E_{N_1}^{N_d}e^{N_r}$ and the invariant holds for $N_1$.

For induction step, assume that $N_r$.value $\geq v$ after $E_{N_1}^{N_d}e^{N_r}$. Let $\Phi'$ be a $\text{WriteMax}$ operation by $p_j$ that issues $e^{N_{r+1}}$. WLOG, $N_r$ is the left child of $N_{r+1}$ and $e^{N_{r+1}}$ is cas$^1(N_{r+1}, \Phi')$. We consider the following cases.

1. If $\Phi = \Phi'$, then by Invariant 1, $N_r$.count $\Phi'$ in execution prior to $\text{read}_{\text{LC}}(N_{r+1}, \Phi') \equiv \text{read}_{\text{LC}}(N_{r+1}, \Phi')$.

2. If $\Phi \neq \Phi'$, then obviously $p_i \neq p_j$ and from Definition 9, $N_r$.count $\Phi$ in the execution prefix prior to $\text{read}_{\text{LC}}(N_{r+1}, \Phi')$.

According to Algorithm A, the value $v'$ written by cas$^1(N_{r+1}, \Phi') \equiv e^{N_{r+1}}$ is greater than or equal to the value read by $\text{read}_{\text{LC}}(N_{r+1}, \Phi')$. Since $e^{N_r} \prec \text{read}_{\text{LC}}(N_{r+1}, \Phi')$, $N_r$.value $\geq v$ prior to $\text{read}_{\text{LC}}(N_{r+1}, \Phi')$ and thus $v' \geq v$. Consequently, $N_{r+1}$.value $\geq v$ after $E_{N_1}^{N_r}e^{N_{r+1}}$ and the invariant holds for $N_{r+1}$. \( \blacksquare \)

**Lemma 12.** Let $\Phi$ be a $\text{ReadMax}$ operation in $H$, such that $H = H_1\Phi H_2$. Then $\Phi$ returns the last value written in $H_1$ (or $-\infty$, if no $\text{WriteMax}$ appears in $H_1$).

**Proof.** Let $e$ be the linearization point of $\Phi$ in $E$, such that $E = E'eE''$. From Definition 9, $e$ is a read of $T$.root.value and according to Algorithm A, $\Phi$ returns the value obtained by $e$.

If no $\text{WriteMax}$ appears in $H_1$, then no event is visible on $T$.root.value after $E_1$. Hence $e$ reads the initial value of $T$.root.value, which is $-\infty$.

Let $v$ be the largest value written in $H_1$ and let $\Phi_w$ be the $\text{WriteMax}$ operation that wrote $v$. We prove that $T$.root.value $= v$ after $E_1$. Let $e_w$ be the linearization point of $\Phi_w$, such that $E_1 = E_{E_w}E_2$. From Lemma 11, $T$.root.value $\geq v$ after $E_{E_w}$ and thus after $E_1$. Moreover, from Lemma 10, $T$.root.value $\leq v$ after $E_1$, otherwise $v$ was not the largest value written in $H_1$. \( \blacksquare \)

From Observations 1 - 2 and from Lemmas 7 - 9, for every operation $\Phi$ in $E$ its linearization point is uniquely defined within the time interval of $\Phi$, and thus $H$ preserves a partial precedence order of operations in $E$. Moreover, from Lemma 12, $H$ is a legal sequential history of $\text{maxRegister}$. This immediately leads us to the following theorem.

**Theorem 5.** Algorithm A is a linearizable implementation of $N$-process unbounded $\text{maxRegister}$.

**Wait-freedom and Complexity.**

Clearly, Algorithm A is wait-free. The number of events issued by $\text{ReadMax}$ is constant. The number of events issued by $\text{WriteMax}(v)$ is proportional to the depth $d$ of leaf $L$:

1. if $v < N$, then $L \in T_d$ and from the definition of $B_1$, $d = O(\log v) = O(\min(\log N, \log v))$.
2. if $v \geq N$, then $L \in T_R$ and from the definition of complete binary tree, $d = O(\log N) = O(\min(\log N, \log v))$

**Theorem 6.** Algorithm A is wait-free; the step-complexity of the $\text{ReadMax}$ operation is $O(1)$ and the step-complexity of the $\text{WriteMax}(v)$ is $O(\min(\log N, \log v))$.

6. REFERENCES


