Long-Lived Snapshots with Polylogarithmic Amortized Step Complexity

Ahad Mirza, Danny Hendler, Alessia Milani, Corentin Travers

Long-Lived Snapshots with Polylogarithmic Amortized Step Complexity

MIRZA AHAD MIRZA BAÏG*, IST Austria, Austria
DANNY HENDLER†, Ben-Gurion University, Israel
ALESSIA MILANI‡ and CORENTIN TRAVERS‡, LaBRI, U. Bordeaux, France

We present the first deterministic wait-free long-lived snapshot algorithm, using only read and write operations, that guarantees polylogarithmic amortized step complexity in all executions. This is the first non-blocking snapshot algorithm, using reads and writes only, that has sub-linear amortized step complexity in executions of arbitrary length. The key to our construction is a novel implementation of a 2-component max array object which may be of independent interest.

CCS Concepts: • **Theory of computation → Concurrent algorithms**.

Additional Key Words and Phrases: shared memory, atomic snapshot, max array, amortized step complexity

1 INTRODUCTION

The *snapshot* object [1, 2, 7] is a fundamental abstraction in distributed computing since it allows a process to obtain a consistent view of a collection of shared memory locations while other processes are concurrently updating them. Snapshot objects were used in the past for solving synchronization tasks such as consensus and approximate agreement [7, 11] and for implementing concurrent objects such as counters [8] and bounded timestamps [13]. In this paper we study the *single-writer atomic snapshot* (henceforth referred to simply as *snapshot*) object which allows each process to update its own component in a shared array (by invoking an `Update` operation) and to obtain an atomic view of all components (by invoking a `Scan` operation).

Wait-free snapshot can be implemented, using reads and writes only, in step complexity linear in the number of processes n [9, 16]. A well-known result by Jayanti, Tan and Toueg [17] showed that this is tight, by proving a linear lower bound on the worst-case step complexity of obstruction-free implementations of a large class of shared objects that includes snapshots from operations in a set that includes (among some other operations) read and write.

Aspnes, Attiya and Censor-Hillel [3] observed that the lower bound of [17] holds only when numerous operations are applied to the object and does not rule out the possibility of obtaining algorithms whose wort-case step complexity is sub-linear when the number of operations is bounded. Leveraging this observation, they presented constructions of several concurrent objects for which an operation's step complexity is polylogarithmic in n as long as the object's value is polynomial in n.

Their constructions are based on a new abstraction they introduced called a max register. A *max register* r supports a `WriteMax`(r,v) operation that writes a non-negative integer v to r and a `ReadMax`(r) operation that returns the maximum value previously written to r. An *m-bounded max register* can assume values from $\{0, \ldots, m-1\}$, for some integer m.

Building upon the m-bounded max register algorithm introduced in [3], Aspnes et al. [4] presented the first wait-free snapshot algorithm with sub-linear worst-case step complexity. Specifically, their algorithm has $O(\log n)$ step complexity for `Scan` operations and $O(\log^3 n)$ step complexity for `Update` operations, in executions in which the number of update

*Research done while Ahad was an intern at LaBRI supported by UMI Relax.
†Supported in part by ISF grant 380/18.
‡Supported in part by ANR projects DESCARTES and FREDDA

Authors' addresses: Mirza Ahad Mirza Baïg, IST Austria, Austria, mirzaahad.baig@ist.ac.at; Danny Hendler, Ben-Gurion University, Beer-Sheva, Israel, hendlerd@cs.bgu.ac.il; Alessia Milani; Corentin Travers, LaBRI, U. Bordeaux, Talence, France, [milani,travers]@labri.fr.

operations is polynomial in n. However, both the worst-case and the amortized step complexities of their snapshot algorithm deteriorate as the number of `Update` operations increases. For executions in which the number of `Update` operations is exponential in n, both the worst-case and the amortized step complexities become linear in n.

Our contribution. The lower bound of [17] leaves open the question of whether there exists a snapshot algorithm with sub-linear *amortized* step complexity. In this paper, we answer this question in the affirmative. We present a wait-free, linearizable atomic snapshot algorithm whose amortized complexity is polylogarithmic in all executions, regardless of their length. This is the first wait-free snapshot algorithm that provides sub-linear amortized step complexity in all executions.

Our unbounded snapshot algorithm is largely inspired by the bounded snapshot algorithm of Aspnes et al. [4]. The latter is based on a new data structure they defined and implemented called a *2-component max array*. It consists of a pair of (left and right) *bounded* max registers and supports a `MaxUpdate`($side, v$) operation that writes value v to either the left or the right max register, and a `MaxScan`() operation that returns a view consisting of the values of both components.[1] Since their snapshot algorithm uses bounded max registers and bounded max arrays, once the number of update operations exceeds this bound, it falls back to a linear step-complexity algorithm.

We replace every instance of a bounded max register used by their algorithm by an instance of an unbounded max register implementation with $O(\log n)$ amortized step complexity, recently presented by Baig et al in [12]. We also replace every instance of a bounded 2-component max array they use with an instance of a novel unbounded 2-component max array implementation that we present in this paper. Our unbounded 2-component max array implementation provides $O(\log^2 n)$ amortized step complexity in all executions. This algorithm and its analysis are our key technical contribution.

Aspnes and Censor-Hillel [5, 6] presented a *randomized* snapshot algorithm in which each operation incurs $O(\log^3 n)$ step complexity with high probability. Their implementation is based on randomized constructions of an unbounded max register and an unbounded 2-component max array, whose operation step complexities are, respectively, $O(\log n)$ and $O(\log^2 n)$ with high probability. Our unbounded 2-component max array algorithm differs substantially from their conference version [5] but is similar to the algorithm in their updated version [6]. The key difference is that, whereas their algorithm employ a randomized helping mechanism, ours uses a deterministic helping mechanism, and is consequently deterministic. This, as well as additional differences between the two algorithms, result in much simpler linearizability proofs.

The rest of this paper is organized as follows. We present the system model we assume and additional required definitions in Section 2. In Section 3, we present our key technical contribution – an unbounded 2-component max array algorithm, and prove that it guarantees linearizability and polylogarithmic amortized step complexity when the values of its components are not increased "too quickly". In Section 4, we prove that by "plugging" our unbounded 2-component max array and the unbounded max register of [12] into the snapshot algorithm of [4], we obtain a linearizable snapshot with $O((\log^3 n))$ amortized step complexity. The paper is concluded with a discussion in Section 5.

2 PRELIMINARIES

We consider a standard shared-memory model, where n crash-prone asynchronous processes communicate via shared *registers*, supporting only atomic read and write operations. A concurrent object *implementation* specifies the object's state representation and the algorithms processes follow when they perform operations supported by the object.

[1] Aspnes et al. [4] refer to these components at 0 and 1 and define different operations for updating each of them, but other than this syntactic difference, our definition of a max array is identical to theirs.

An *execution* is a series of *steps* performed by processes as they follow their algorithms, in each of which a process applies at most a single read or write operation to a register (possibly in addition to some local computation). The execution interval of a high-level operation is the interval between its invocation step and response steps.

Linearizability [15] ensures that, for every completed operation in the execution E and some of the uncompleted operations, there is a point within the execution interval of the operation called its *linearization point*, such that the responses returned by operations in E are the same as the responses returned if all these operations were executed sequentially in the order determined by their linearization points. An object implementation is *linearizable* if all its executions are linearizable. An implementation is *wait-free* [14] if, in every execution, each *correct process* (i.e., a process that does not crash) completes each operation it performs within a finite number of steps.

A **max register** r supports two operations : a `WriteMax`(r,v) operation that writes a non-negative integer $v \geq 0$ to r and a `ReadMax`(r) operation that returns the maximum value previously written to r. A max register can be either bounded or unbounded. A *bounded max register* can assume values from $\{0, \ldots, m-1\}$, for some integer m. An *unbounded max register* can store any non-negative integer.

A **2-component max array** consists of a left and a right max registers and supports two operations. The `MaxUpdate`(*side*,v) operation receives two arguments: *side* specifies whether the left or the right array component should be written, and v is the value to be written to that component. The `MaxScan` operation returns a *view* of the max array – a pair of values where the left (resp. right) pair-value contains the maximum value written to the left (resp. right) component by a `MaxUpdate` operation that was linearized before the `MaxScan`. An *(m_1,m_2)-bounded 2-component max array* can assume values from $\{0, \ldots, m_1 - 1\}$ in its first component and values from $\{0, \ldots, m_2 - 1\}$ in its second component, for some positive integers m_1, m_2. Each component of an *unbounded 2-component max array* can store any non-negative integer.

A **single-writer snapshot** object consists of n locations and supports two operations : `Update`(v) allows a process to store a value v in its location and `Scan` takes no input and returns n values (one for each location), such that the value corresponding to each location i is the value of the last `Update` operation by process i that has been linearized before the `Scan` operation. We say that a snapshot object is *b-limited-use* if the total number of `Update` operations that can be applied to it in any execution is at most b. If there is no such bound b, we say that the object is an *unlimited-use* snapshot object.

Amortized step complexity is defined as the worst case (taken over all possible executions) average number of steps performed by operations. It measures the performance of an implementation as a whole rather than the performances of individual operations. More precisely, given a finite execution E, an operation Op *appears* in E if it is initiated in E. We denote by $Steps(Op, E)$ the number of steps performed by Op in E and by $Ops(E)$ the number of operations that appear in E. The amortized step complexity of an implementation A is then:

$$AvgSteps(A) = \max_{E\text{: finite execution of } A} \frac{\sum_{Op \in Ops(E)} Steps(Op, E)}{|Ops(E)|}.$$

The unbounded 2-component max array algorithm we present in Section 3 uses two instances of an unbounded max register due to Baig et al. [12], whose amortized step complexity is $O(\log n)$. However, the correctness and complexity of this unbounded max register algorithm are guaranteed only in executions in which its value is increased in increments of at most n. This notion is formalized by the following definition.

*Definition 2.1 (ℓ-**Bounded-Increment Execution**).* Let E be an execution and let M be an `UnboundedMaxReg` object. We say that E is an ℓ-bounded-increment execution for M if for each operation op = `WriteMax`(v) on M in E, with $v > \ell$, there exists an operation op' = `WriteMax`(v') on M in E that precedes op, such that $v - \ell \leq v' < v$.

3 UNBOUNDED 2-COMPONENT MAX ARRAY ALGORITHM

The pseudo-code of our unbounded 2-component max array implementation is presented by Algorithm 1. A `UB-MaxArray` object contains the following data components.

- $\mathtt{MR}_{left}, \mathtt{MR}_{right}$ are two shared unbounded max registers [12], initialized to 0, representing the left and right max array component, respectively. Each such register encapsulates an array of bounded max registers and the subscript in the type $\mathtt{UnboundedMaxReg}_{n^2}$ indicates that the bound on each of them is n^2. As proven in [12, Theorem 10], the correctness of $\mathtt{UnboundedMaxReg}_{n^2}$ is guaranteed only in n-bounded executions, in which they have logarithmic amortized step complexity.
- An infinite number of shared $(m \times m)$-bounded 2-component max array objects [4], denoted $\mathtt{MA}_j$, for all $j \in \mathbb{N}_0$. All max arrays are initialized to $\langle 0, 0\rangle$. $\mathtt{MA}_j$ stores the modulo m residues of the left and right components of the implemented max array. For simplicity of presentation and without loss of generality, we assume that m is an integral power of 2. The left (resp. right) component of $\mathtt{MA}_j$ stores the $\log m$ low-order bits of a value v_l previously written to $\mathtt{MR}_{left}$ (resp. value v_r previously written to $\mathtt{MR}_{right}$), whereas j is the sum of the high-order bits of v_l and v_r.[2]
- A shared helping array $\mathtt{H}[1..n]$. Entry $\mathtt{H}[i]$ is used by process i for helping other processes to complete their `MaxScan` operations. Each entry consists of two fields. The first is a monotonically increasing timestamp incremented by process i before each write to $\mathtt{H}[i]$. The second is a (2-component) view computed by i.
- ts_i is an integer timestamp, accessed by process i only, whose value persists throughout the execution.

We now describe the `MaxScan` operation. A `MaxScan` operation S, performed by some process i, simply invokes the `CompScan` function and returns the view computed by the latter. `CompScan` consists almost entirely of the loop of lines 9-29. In each loop iteration, process i attempts to obtain a view $\langle l^*, r^*\rangle$ by itself. If i succeeds, we say that $\langle l^*, r^*\rangle$ is a *direct view*. As we prove, if `CompScan` fails in obtaining a direct view, then eventually a helping mechanism built into the algorithm will make available to it a view computed by another process; in this case, we call the view returned by `CompScan` (and then by S) an *indirect view*.

Each loop iteration starts by reading $\mathtt{MR}_{right}$ (line 10), then $\mathtt{MR}_{left}$ (line 11), and then $\mathtt{MR}_{right}$ again (line 12). Lines 10-12 attempt to compute a snapshot of the high-level bits of $\mathtt{MR}_{left}$ and $\mathtt{MR}_{right}$, hence we name their execution a *high-level bits snapshot attempt*. We say that the *high-level bits snapshot attempt succeeds*, if the high-order bits of the two values read from $\mathtt{MR}_{right}$ are equal; otherwise we say that the *high-level bits snapshot attempt fails*.

Successful high-level bits snapshot attempt: In this case, the condition of line 13 is satisfied and process i computes (line 14) an index h of a bounded 2-component max array object by summing up the high-order bits of the values read (in lines 10-12) from the two unbounded max registers. Then, i writes the low-order bits of value l, read from $\mathtt{MR}_{left}$ (line 11), to the left component of $\mathtt{MA}_h$ (line 15) and the low-order bits of the 2nd value, r_2, read from $\mathtt{MR}_{right}$ (line 12), to the right component of $\mathtt{MA}_h$ (line 16). As we prove, when $\mathtt{MR}_{left}$ is read in line 11, the sum of the high-order bits of $\mathtt{MR}_{left}$ and $\mathtt{MR}_{right}$ is h; this will be used to prove the correctness of the algorithm. Next, i invokes a `MaxScan` operation on $\mathtt{MA}_h$ (line 17) and obtains a view $\langle \ell', r'\rangle$ of the low-level bits. It then verifies (line 18) that the high-order bits still sum to h. If this is the case, then a direct view was successfully computed, so i proceeds to compute the left and right values by concatenating the high-order and low-order bits (line 19), writes the direct view together with a fresh value of i'th

[2] In the initial configuration, both $\mathtt{MR}_{left}$ and $\mathtt{MR}_{right}$ are 0 and so are the values of both components of each $\mathtt{MA}_j$ object.

Algorithm 1 Unbounded 2-component Max Array `UB-MaxArray`, code for process i

Shared objects:

$\mathtt{MR}_{left}, \mathtt{MR}_{right}$: $\mathtt{UnboundedMaxReg}_{n^2}$ registers, initially 0 ▷ Unbounded max registers [12] for left & right components

$\mathtt{MA}_j$: an $(m \times m)$-bounded 2-component max array for each $j \in \mathbb{N}_0$, initially $\langle 0, 0 \rangle$ ▷ 2-component max array objects [4]

$\mathtt{H}[1..n]$: array of n registers, initially $[(0, \langle 0, 0\rangle), \ldots, (0, \langle 0, 0\rangle)]$ ▷ Each entry consists of a timestamp and a values-pair

Persistent local variables for process i:

ts_i: integer, initially 0 ▷ Timestamp used for helping

```
1:  function MaxScan( )
2:      return CompScan()                                   ▷ Invoke utility function, return the view it returns
3:  function MaxUpdate(side, v)
4:      WriteMax(MR_side, v)                                ▷ Write input to the MR object corresponding to side
5:      CompScan()                                          ▷ Invoke utility function, disregard its response value
6:      return
7:  function CompScan( )                                    ▷ Utility function implementing most algorithm logic
8:      c ← 0                                               ▷ Initialize iterations number counter
9:      while true do                                       ▷ Loop until a view is obtained
10:         r1 ← ReadMax(MR_right)                          ▷ First read of the MR object corresponding to the right side
11:         l ← ReadMax(MR_left)                            ▷ Single read of the MR object corresponding to the left side
12:         r2 ← ReadMax(MR_right)                          ▷ Second read of the MR object corresponding to the right side
13:         if ⌊r1/m⌋ = ⌊r2/m⌋ then                         ▷ If this was a successful high-level bits snapshot attempt
14:             i1 ← ⌊l/m⌋, i2 ← ⌊r2/m⌋, h ← i1 + i2        ▷ Compute the high-level bits of left and right sides and their sum
15:             MaxUpdate(MA_h, left, l mod m)              ▷ Write the low-level bits of MR_left to the left side of MA_h
16:             MaxUpdate(MA_h, right, r2 mod m)            ▷ Write the low-level bits of MR_right to the right side of MA_h
17:             ⟨ℓ′, r′⟩ ← MaxScan(MA_h)                    ▷ Obtain from MA_h a view of the low-level bits of both components
18:             if ⌊ReadMax(MR_left)/m⌋ = i1 ∧ ⌊ReadMax(MR_right)/m⌋ = i2 then   ▷ If the sum of the high-level bits of MR_left,MR_right is still h
19:                 l* ← i1 · m + ℓ′, r* ← i2 · m + r′      ▷ Compute view by concatenating high-order and low-order bits
20:                 ts_i ← ts_i + 1                         ▷ Increment process timestamp used when helping
21:                 Write(H[i], (ts_i, ⟨l*, r*⟩))           ▷ Write timestamp and view to i'th entry in the helping array
22:                 return ⟨l*, r*⟩                         ▷ Return direct view
23:         if c = ⌊n/log² m⌋ then                          ▷ If performed a linear number of steps since CompScan's execution started
24:             h0 ← Collect(H)                             ▷ Perform first collect of helping array
25:         else if (c mod ⌊n/log² m⌋ = 0) then             ▷ If performed a linear number of steps since previous collect of helping array
26:             h1 ← Collect(H)                             ▷ Perform an additional collect
27:             if ∃j : (h0[j] = (ts0, s0)) ∧ (h1[j] = (ts1, s1)) ∧ (ts1 > ts0 + 1) then   ▷ A view that can be used is available
28:                 return s1                               ▷ return an indirect view
29:         c ← c + 1                                       ▷ Increment iterations number counter
```

timestamp (a local variable initialized to 0 in line 8) to its entry in the helping array (lines 20, 21) and returns the direct view (line 22).

Failed high-level bits snapshot attempt: A high-level bits snapshot attempt fails if the condition in line 13 is false or, if it is true, but the condition in line 18 is false. In either of these cases, i attempts to obtain an indirect view via the helping array. In order to amortize the steps incurred by the helping mechanism against those taken by the attempts to obtain a direct view, a collect of the helping array (which consists of copying all of its entries and incurs a linear number of steps) occurs only when the number of failed attempts is an integral multiple of $\lfloor n/\log^2 m \rfloor$ (incurring altogether, as we show, a linear number of steps). If exactly $\lfloor n/\log^2 m \rfloor$ failed attempts were performed by the current instance of `CompScan` (line 23), then it performs a first collect of the helping array `H` and stores the result in a local variable (line 24). It then increments the number of failed attempts (line 29) and proceeds to the next loop iteration. Otherwise, this is not the first collect of the helping array done by the current instance of `CompScan` (line 25), so it stores the result of

the new collect to a second local variable (line 26) and then checks if there exists an entry H[j] that was written to (by process j) at least twice since the first collect was computed (line 27). If this is the case then, as we prove, the value stored in the second component of H[j] of the new collect is a view that can be used by S, so CompScan returns it (in line 28). Otherwise, i increments the number of failed attempts (line 29) and proceeds to the next loop iteration.

The MaxUpdate operation is very simple. It receives two argument – a value v to write to the implemented max array and the *side* to which v should be written. It first writes v to $\mathtt{MR}_{side}$ (line 4) and then invokes CompScan (line 5). Once CompScan returns, so does MaxUpdate (line 6), disregarding the view returned by CompScan.

In what follows, unless stated otherwise, we only consider executions that are n-bounded-increment for both $\mathtt{MR}_{left}$ and $\mathtt{MR}_{right}$. As we prove in Section 4.1, this is the case for all executions in which our unbounded 2-component max array is used by the snapshot algorithm of [4].

3.1 Linearizability

We refer to the instance of CompScan invoked by an instance of a MaxScan or a MaxUpdate operation Op as Op's CompScan instance. For presentation simplicity, we sometimes refer to steps taken by Op's CompScan instance as the steps of Op.

Operations are linearized according to the following *linearization rules*.

(1) A MaxScan operation that returns a direct view is linearized when its last invocation of $\mathtt{MaxScan}(\mathtt{MA}_h)$ (line 17) before it returns occurs .
(2) Let S be a MaxScan operation that returns an indirect view s_1 (line 28) after reading it from some entry j of the helping array (line 26). Let S' be the CompScan function instance (invoked by a MaxScan or a MaxUpdate operation) by process j that wrote this view to H[j] (line 21). Then the linearization point of S is set at the last invocation of $\mathtt{MaxScan}(\mathtt{MA}_h)$ (line 17) in S' before S' returns. If several MaxScan operations are linearized at the same step, then their internal order is set arbitrarily.
(3) Let E be an execution and let W be a $\mathtt{MaxUpdate}_{side}$ operation that appears in E. Let $E = E_0 \circ E_1$ be such that the last step of E_0 is W's write to $\mathtt{MR}_{side}$ (line 4). Let Ops be the set of operations (which may or may not include W itself) that read $\mathtt{MR}_{side}$ in E_1 (in line 11 or line 12) and then write in E_1 to side *side* of an MA object (in line 15 or line 16). If Ops is empty, then W is not linearized in E. Otherwise, let $Op \in Ops$ be the first operation in Ops to write to side *side* of an MA object after reading $\mathtt{MR}_{side}$ in E_1 and let s be the first step by Op in E_1 in which such a write occurs, then W is linearized at s. If there are several write operations that are linearized at s, then their internal order is the order in which they wrote to $\mathtt{MR}_{side}$.

Claim 1. *Let S be a* CompScan *instance that returns an indirect view s_1. Then there is a* CompScan *instance S' such that s_1 is a direct view of S' and its last execution of lines 10-19 is within the execution interval of S.*

Proof. Let S be a CompScan instance by process i that returns an indirect view s_1 (line 28). It follows that there is a CompScan instance S' by some process $j \neq i$ that wrote s_1 to H[j] (line 21) before returning a direct view (line 22). From lines 23-28, the value of the first component of H[j] (j's timestamp) increased by at least 2 between when S read it in line 24 and when it read it in line 26. From the code of the CompScan function, process j executes lines 10-19 between every two consecutive increments of its timestamp (line 20). It follows that the last execution of lines 10-19 by S' is contained within the execution interval of S. □

Claim 2. *Let E be an execution and let W be a* $\texttt{MaxUpdate}_{side}$ *operation that appears in E. Then if W completes in E, it is linearized within its execution interval.*

Proof. From linearization rule 3, W cannot be linearized before it writes to $\texttt{MR}_{side}$ (line 4), hence it cannot be linearized before it starts. Assume towards a contradiction that W completes in E but is not linearized before it completes. If W completes in E after its CompScan instance returns a direct view, then it reads $\texttt{MR}_{side}$ (line 11 or line 12) and then writes to $\texttt{MA}_h$ with side *side* (line 15 or 16) in E after it writes to $\texttt{MR}_{side}$ (line 4). It follows from linearization rule 3 that W is linearized either when it executes line 15 or line 16, or before that – a contradiction. Assume, then, that W's CompScan instance returns an indirect view s_1. In this case, from Claim 1, there is a CompScan instance S' such that s_1 is a direct view of S' and the last execution of lines 10-19 by S' is within the execution interval of W's CompScan instance, hence it follows W's write to $\texttt{MR}_{side}$ (line 4). Thus, from linearization rule 3, W is linearized either when S' writes to $\texttt{MA}_h$ with side *side* (line 15 or 16) for the last time in E or before that, hence it is linearized within its execution interval. This is a contradiction. □

Lemma 3.1. *The order specified by linearization rules 1-3 respects the real-time order of operations that appear in the execution.*

Proof. We prove the lemma by showing that every operation that completes in the execution is linearized at some point within its execution interval. The following cases exist.

(1) Let S be a MaxScan operation that returns a direct view (line 22). Then S is linearized according to rule 1 when its CompScan instance invokes $\texttt{MaxScan}(\texttt{MA}_h)$ (line 17). The lemma follows since this is a step performed by a function called by *Op*.
(2) Let S be a MaxScan operation by process i that returns an indirect view s_1 (line 28). Then the lemma follows from Claim 1.
(3) Let W be a $\texttt{MaxUpdate}_{side}$ operation that appears in the execution. If W completes, the claim follows from Claim 2. Otherwise, from linearization rule 3, W cannot be linearized before it writes to $\texttt{MR}_{side}$ (line 4), hence it cannot be linearized before it starts. This completes the proof. □

Recall that we refer to the value of the $\log m$ low-order bits of a value v as v's *low-order bits*. We refer to the value of the binary representation of $\lfloor v/m \rfloor$ as v's *high-order bits*.

Observation 1. *The high-order bits of the values of* $\texttt{MR}_{left}$ *and* $\texttt{MR}_{right}$ *are monotonically increasing.*

Proof. The values of $\texttt{MR}_{left}$ and $\texttt{MR}_{right}$ are monotonically increasing because of the semantics of a max register, hence the high-order bits of their values are monotonically increasing as well. □

Observation 2. *The values i_1 and i_2 computed in line 14 are, respectively, the high-order bits of* $\texttt{MR}_{left}$ *and* $\texttt{MR}_{right}$ *when l is read in line 11.*

Proof. From the semantics of a max register and the computation in line 14, i_1 is the high-order bits of $\texttt{MR}_{left}$ when it is read in line 11. i_2 is the value of the high-order bits of $\texttt{MR}_{right}$ when l is read in line 11, because it is read from $\texttt{MR}_{right}$ both before and after line 11 (in lines 10, 12), from Observation 1, and from the its computation in line 14. □

Observation 3. *Whenever an index h is computed in line 14 as the sum of i_1 and i_2, the same values of i_1 and i_2 are used.*

Proof. Follows immediately from Observations 1, 2. □

Based on Observation 3, in what follows we refer to the values i_1,i_2 used for computing index h in line 14 as the *left and right high-order bits corresponding to h*, respectively. Also, we let O denote the implemented max array object. We let $val(\texttt{MA}_j.left)$ (resp. $val(\texttt{MA}_j.right)$) denote the value of O's left (resp. right) component of $\texttt{MA}_j$.

Definition 3.2. Let E be an execution. Let $\texttt{MA}_h$ be O's bounded 2-component max array object with the highest index such that a `MaxUpdate` operation to at least one of $\texttt{MA}_h.left$ or $\texttt{MA}_h.right$ appears in E, or $\texttt{MA}_0$ if no `MaxUpdate` operations to any `MA` object appear in E. We say that $\texttt{MA}_h$ is O's *active two-component max array* (or simply *active array*) *after E*. We let $active(O, E)$ denote index h. We define *O's value after E* as follows:

- If no `MaxUpdate` operations to any `MA` object appear in E then O's value after E is $\langle 0, 0\rangle$.
- Otherwise, O's value after E is $\langle val(\texttt{MA}_h.left) + i_1 \cdot m, val(\texttt{MA}_h.right) + i_2 \cdot m\rangle$, where i_1 and i_2 are the left and right high-order bits corresponding to h.

Observation 4. *The value of $active(O, E)$ is monotonically increasing.*

Proof. Immediate from Definition 3.2 and Observations 1, 3. □

Claim 3. *Let $E = E_0 \circ s_0 \circ E_1 \circ s_1 \circ E_2 \circ s_2$ be an execution, where:*

- *s_2 is a step in which a* `CompScan` *function instance S returns a direct view $\langle l^*, r^*\rangle$ (line 22).*
- *s_1 is the step in which S executes (for the last time before s_2) a* `MaxScan` *in line 17 and received a view $\langle \ell', r'\rangle$.*
- *s_0 is the step in which S executes (for the last time before s_1) line 11.*

Then $\langle l^*, r^*\rangle$ is O's value when s_1 occurs.

Proof. From lines 10-14 and Observations 1,2, no write to an $\texttt{MA}_{h'}$ object, for $h' > h$, appears in E before s_0. From the code, `CompScan` can only reach lines 19-22 if the condition of line 18 is satisfied. It follows, once again from Observations 1,2, that no write to an $\texttt{MA}_{h'}$ object, for $h' > h$, appears in E before s_1. Since S writes to $\texttt{MA}_h$ (lines 15-16) in E before s_1, it follows that h is O's active array when s_1 occurs. The claim now follows from Definition 3.2, line 19 and the semantics of a max array `MaxScan` operation. □

Claim 4. *Let E be an execution and let $\langle l^*, r^*\rangle$ be O's value after E. Then l^* (resp. r^*) is the maximum value written by a* `WriteMax` *instance to $\texttt{MR}_{left}$ (resp. $\texttt{MR}_{right}$) that was linearized in E, or 0 (resp. 0) if no such* `WriteMax` *instance exists.*

Proof. If no `WriteMax` instance (to neither $\texttt{MR}_{left}$ or $\texttt{MR}_{right}$) was linearized in E then it follows immediately from Definition 3.2 and linearization rule 3 that O's value after E is $\langle 0, 0\rangle$. Assume otherwise. From Definition 3.2, $l^* = val(\texttt{MA}_h.left) + i_1 \cdot m$ (resp. $r^* = val(\texttt{MA}_h.right) + i_2 \cdot m$), where h is $active(O, E)$ and i_1 (resp. i_2) is the (unique) value of the left (resp. right) high-order bits associated with h. From lines 10-16, l^* (resp. r^*) is the largest value read in E from $\texttt{MR}_{left}$ (resp. $\texttt{MR}_{right}$) whose low-order bits were then written to the left (resp. right) side of $\texttt{MA}_h$. Moreover, from the definition of h, l^* (resp. r^*) is the largest value read from $\texttt{MR}_{left}$ (resp. $\texttt{MR}_{right}$) in E whose low-order bits were later written in E to the left (resp. right) side of *any* `MA` object. Since l^* (resp. r^*) was first written in E in line 4, then read in E from $\texttt{MR}_{left}$ in line 11 (resp. from $\texttt{MR}_{right}$ in lines 10,12) and then its low-order bits were written in E to the left (resp. right) side of $\texttt{MA}_h$ in line 15 (resp. line 16), from linearization rule 3, a `WriteMax` operation that wrote l^* (resp. l^*) to $\texttt{MR}_{left}$ (resp. $\texttt{MR}_{right}$) was linearized in E and l^* (resp. r^*) is the maximum such value. □

Lemma 3.3. *The order specified by linearization rules 1-3 respects the semantics of the 2-max array object.*

Proof. From Claim 3, every CompScan instance that returns a direct view returns O's value at some point during its execution interval. From Claim 1, this is also the case for a CompScan instance that returns an indirect view. Consequently, from Claim 3, every view returned by a MaxScan instance is O's value at some point during its execution interval. The lemma now follows from Claim 4. □

3.2 Wait-freedom

A MaxScan operation invokes CompScan and returns the response of that invocation. A MaxUpdate operation writes once to $\mathtt{MR}_{left}$ or $\mathtt{MR}_{right}$ and then invokes CompScan. To show wait-freedom, it therefore suffices to prove that the CompScan function is wait-free.

An instance of a CompScan operation consists of one or more iterations of the while loop (lines 9-29). We say that a loop iteration is *unsuccessful* if either the test on line 13 or on line 18 fails. Otherwise, we say that the iteration is *successful*. In a successful iteration that completes, the conditions of both tests are satisfied, a view is returned at line 22, and the instance of CompScan terminates. Note that a view may also be returned in an unsuccessful iteration, if help is received (line 28).

Lemma 3.4. *Let E be a b-bounded-increment execution for both $\mathtt{MR}_{left}$ and $\mathtt{MR}_{right}$. Let S be a CompScan instance in E. If $K \geq 2$ unsuccessful loop iterations are performed in S, there are at least $\left\lceil \frac{(K-2)m}{b} \right\rceil$ instances of MaxUpdate operations that start during the execution interval of S.*

Proof. For each $i, 1 \leq i \leq K$, let R_i (respectively L_i) denote the value of max register $\mathtt{MR}_{right}$ (respectively $\mathtt{MR}_{left}$) the first time it is read in iteration i at line 10 (respectively, line 11). Similarly, let R'_i (respectively L'_i) denote the value of max register $\mathtt{MR}_{right}$ (respectively $\mathtt{MR}_{left}$) the *last* time it is read in iteration i, at line 12 or line 18 (respectively, at line 11 or line 18). As iteration i is unsuccessful and since successive values read from a max register are monotonically increasing, we have $\left\lfloor \frac{L_i}{m} \right\rfloor < \left\lfloor \frac{L'_i}{m} \right\rfloor$ or $\left\lfloor \frac{R_i}{m} \right\rfloor < \left\lfloor \frac{R'_i}{m} \right\rfloor$. Moreover, since iteration $i+1$ starts after iteration i, for every $i, 1 \leq i \leq K-1$, $L'_i \leq L_{i+1}$ and $R'_i \leq R_{i+1}$.

Let ℓ denote the number of iterations i for which $\left\lfloor \frac{L_i}{m} \right\rfloor < \left\lfloor \frac{L'_i}{m} \right\rfloor$. Hence, $\left\lfloor \frac{L_1}{m} \right\rfloor + \ell \leq \left\lfloor \frac{L'_K}{m} \right\rfloor$, from which it follows that $(\ell - 1) \cdot m + 1 \leq L'_K - L_1$. Similarly, denoting the number of iterations i for which $\left\lfloor \frac{R_i}{m} \right\rfloor < \left\lfloor \frac{R'_i}{m} \right\rfloor$ by r, we have that $(r-1) \cdot m + 1 \leq R'_k - R_1$. Note that $\ell + r \geq K$.

For each side $s \in \{left, right\}$, the value stored in $\mathtt{MR}_s$ may be modified only by an instance of a MaxUpdate(s, v) operation. Specifically, each instance of MaxUpdate(s, v) may change the value of $\mathtt{MR}_s$ to v when performing line 4. As the value of $\mathtt{MR}_{left}$ increased by at least $(\ell - 1) \cdot m + 1$ during these K unsuccessful iterations, it follows from our assumption that E is b-bounded-increment for $\mathtt{MR}_{left}$ that at least $\left\lceil \frac{(\ell-1)m+1}{b} \right\rceil$ instances of WriteMax operations on $\mathtt{MR}_{left}$ occur during S. As WriteMax operations on $\mathtt{MR}_{left}$ are only invoked in MaxUpdate$(left, v)$ operation instances and each such instance invokes WriteMax operations on $\mathtt{MR}_{left}$ only once, it thus follows that at least $\left\lceil \frac{(\ell-1)\cdot m+1}{b} \right\rceil$ instances of MaxUpdate$(left, -)$ operations have started during S. Similarly, the number of instances of MaxUpdate$(right, -)$ operations is at least $\left\lceil \frac{(r-1)\cdot m+1}{b} \right\rceil$. Since $r + \ell \geq K$, denoting the number of instances of MaxUpdate that start during S by W, we get:

$$W \geq \left\lceil \frac{(\ell-1)\cdot m+1}{b} \right\rceil + \left\lceil \frac{(r-1)\cdot m+1}{b} \right\rceil \geq \left\lceil \frac{(\ell-1+r-1)\cdot m+2}{b} \right\rceil \geq \left\lceil \frac{(K-2)\cdot m}{b} \right\rceil$$

□

In the following, for each i, $1 \leq i \leq n$, we denote by $H[i].ts$ and $H[i].view$ the first and second member of the couple stored in $H[i]$.

Observation 5. *Let S be an instance of a CompScan function by process i that completes. Let h and h' denote the value of the array H when S starts and ends, respectively, then the following holds: $h[i].ts + 1 = h'[i].ts$ or $\exists j, 1 \leq j \leq n : h[j].ts + 2 \leq h'[j].ts$.*

Proof. We first note that any write to $H[j].ts$ increments it (lines 20-21), thus, for all j, the value of $H[j].ts$ is monotonically increasing. Consider the last loop iteration of S. If S returns a direct view, i.e. it returns at line 22, then $H[i].ts$ is incremented at line 20 and this is the only write to $H[i].ts$ during the execution interval of S, hence the observation follows. Otherwise, S returns an indirect view at line 28. In that case, the condition of line 27 is satisfied, so there exists j such that $h_0[j].ts + 2 \leq h_1[j].ts$. $h_0[j].ts$ and $h_0[j].ts$, $h_1[j].ts$ are the values of $H[j].ts$ at two different points during the execution interval of S. The observation now follows from the monotonicity of $H[j]$. □

Lemma 3.5. *In any b-bounded increment execution, Algorithm 1 is wait-free.*

Proof. It suffices to prove that CompScan functions are wait-free. Let E be a b-bounded increment execution and let K be an integer such that

$$\left\lceil \frac{(K-2)m}{b} \right\rceil \geq 2n + 1.$$

We show that every instance of the CompScan operation in E, performed by a correct process, terminates after performing at most $K + 2\left\lceil \frac{n}{\log^2 m} \right\rceil$ loop iterations. Assume towards a contradiction that there is an instance of CompScan, by some process i, that performs $K + 2\left\lceil \frac{n}{\log^2 m} \right\rceil$ iterations and does not terminate. We partition the loop iterations of S to three intervals: I_1 that consists of the first $\left\lceil \frac{n}{\log^2 m} \right\rceil$ iterations, I_2 that consists of the next K iterations and I_3 that consists of the $\left\lceil \frac{n}{\log^2 m} \right\rceil$ remaining iterations.

As S does not terminate, every iteration in I_2 is unsuccessful. Therefore, by the choice of K, it follows from lemma 3.4 that at least one process j starts 3 instances of MaxUpdate in this interval. Hence, at least two instances of MaxUpdate by the same process start and complete in I_2. Therefore, by Observation 5, there exists j, $1 \leq j \leq n$, $j \neq i$, such that $h[j].ts + 2 \leq h'[j].ts$, where h and h' respectively denote the values of the help array H at the beginning and at the end of I_2.

In the last iteration of I_1, each entry of array H is copied to the local array $h0$ (line 24). Since the value of each entry $H[\ell].ts$ is monotonically increasing, $h0[j].ts \leq h[j].ts$. Furthermore, the array H is copied to the local array $h1$ every $\left\lceil \frac{n}{\log^2 m} \right\rceil$ iterations (lines 25-26). Thus, this occurs in some iteration in interval I_3, since I_3 consists of $\left\lceil \frac{n}{\log^2 m} \right\rceil$ consecutive iterations. In that iteration, once the local array $h1$ has been updated (line 26), we have $h'[j].ts \leq h_1[j].ts$. Hence, $h0[j].ts + 2 \leq h1[j].ts$ and S terminates (lines 27-28). □

3.3 Step Complexity Analysis

Let E be a finite n-bounded-increment execution in which processes execute operations on an unbounded 2-max array object O. We now prove an $O(\log^2 n)$ upper bound on the amortized step complexity of E, formally defined as follows :

$$AmtSteps(E) = \frac{\sum_{ops \in Ops(E)} nsteps(op, E)}{|Ops(E)|},$$

where $Ops(E)$ is the set of instances of MaxUpdate or MaxScan operations on O that take at least one step in E and $nsteps(op, E)$ is the number of steps performed in E by the instance op.

Before proving the main Lemma 3.6, we need to introduce some notation and observations. We partition $Ops(E)$ to two sets, W the set of instances of MaxUpdate operations and S, the set of instances of MaxScan operations. Each instance of MaxUpdate or MaxScan invokes the CompScan function. An instance of a CompScan consists in one or several loop iterations (lines 9-29). We denote by L the set of all loop iterations in E in which at least one step is taken. We further partition $L = L_u \cup L_s \cup L_{nt}$ into three sets: Recall that a completed iteration is said to be successful if a direct view is returned at line 22, and unsuccessful otherwise.

- L_s is the subset of iterations that complete and are successful;
- L_u is the subset of iterations that complete and are unsuccessful and;
- L_{nt} is the subset of iterations that do not complete.

For every iteration $I \in L_s \cup L_u$, we define:

- $proc(I)$, the process that performs I.
- $index(I) = \left\lfloor \frac{r_1}{m} \right\rfloor + \left\lfloor \frac{\ell}{m} \right\rfloor$, where r_1 and ℓ respectively denote the values read from $\mathtt{MR}_{right}$ and from $\mathtt{MR}_{left}$ at lines 10 and 11 during I. This is well-defined, since every complete iteration applies these read operations to $\mathtt{MR}_{right}$ and $\mathtt{MR}_{left}$. Also, from lines 10-17, if I accesses an underlying bounded 2-component max array $\mathtt{MA}_h$, then $index(I) = h$ holds.

We first observe that different unsuccessful iterations performed by the same process never have the same index:

Observation 6. $\forall I \neq I' \in L_u$ *s.t.* $proc(I) = proc(I') : index(I) \neq index(I')$.

Proof. Suppose WLOG that iteration I precedes iteration I' and both are performed by process i. As I is unsuccessful, the high-order bits of the value stored in $\mathtt{MR}_{right}$ or $\mathtt{MR}_{left}$ (or both) change between when they are read in I and when they are read in I'. From Observation 1, it follows that $index(I) < index(I')$. □

For any given $op \in Ops(E)$, in the corresponding instance of CompScan (if any), at most a single loop iteration does not complete or is successful. This proves the following observation 7.

Observation 7. $|L_s| + |L_{nt}| \leq |W| + |S|$.

To compute $AmtSteps(E)$, we need to count the number of steps performed in instances of operations on the low level max registers and m-bounded 2-max array objects. We denote by $nsteps(\mathtt{O}, op, \sigma)$, where σ is a sequence of steps and op is an operation supported by object $\mathtt{O}$, the number of steps in σ that are performed in the instances of op. Similarly, we denote by $Ops(\mathtt{O}, op, \sigma)$ the set of instances of operation op on $\mathtt{O}$ for which at least one step is a step of σ.

For n-bounded increment executions, the amortized complexity of the max register implementation in [12] is $O(\log n)$. Therefore, for each $side \in \{left, right\}$,

$$\frac{nsteps(\mathtt{MR}_{side}, \mathtt{WriteMax}, E) + nsteps(\mathtt{MR}_{side}, \mathtt{ReadMax}, E)}{|Ops(\mathtt{MR}_{side}, \mathtt{WriteMax}, E)| + |Ops(\mathtt{MR}_{side}, \mathtt{ReadMax}, E)|} = O(\log n).$$

Hence,

$$\frac{\sum_{\mathtt{O} \in \{\mathtt{MR}_{left}, \mathtt{MR}_{right}\}, \mathtt{op} \in \{\mathtt{WriteMax}, \mathtt{ReadMax}\}} nsteps(\mathtt{O}, \mathtt{op}, E)}{\sum_{\mathtt{O} \in \{\mathtt{MR}_{left}, \mathtt{MR}_{right}\}, \mathtt{op} \in \{\mathtt{WriteMax}, \mathtt{ReadMax}\}} |Ops(\mathtt{O}, \mathtt{op}, E)|} = O(\log n). \quad (1)$$

We will thus count together steps performed in WriteMax and ReadMax instances on $\mathtt{MR}_{left}$ and $\mathtt{MR}_{right}$. For $\mathtt{op} \in \{\mathtt{WriteMax}, \mathtt{ReadMax}\}$, let $nsteps(\mathtt{MR}, \mathtt{op}, \sigma) = nsteps(\mathtt{MR}_{left}, \mathtt{op}, \sigma) + nsteps(\mathtt{MR}_{right}, \mathtt{op}, \sigma)$ and $Ops(\mathtt{MR}, \mathtt{op}, \sigma) = Ops(\mathtt{MR}_{left}, \mathtt{op}, \sigma) \cup$

$Ops(\mathsf{MR}_{right}, \mathsf{op}, \sigma)$. Equation (1) can thus be rewritten as follows:

$$\frac{\sum_{\mathsf{op}\in\{\mathtt{WriteMax},\mathtt{ReadMax}\}} nsteps(\mathsf{MR}, \mathsf{op}, E)}{\sum_{\mathsf{op}\in\{\mathtt{WriteMax},\mathtt{ReadMax}\}} |Ops(\mathsf{MR}, \mathsf{op}, E)|} = O(\log n). \tag{2}$$

Lemma 3.6. *If $m = \Theta(n^c)$, where $c \geq 2$ is a constant, then the* UnLimMaxArray *implementation of Algorithm 1 has amortized step complexity of $O(\log^2 n)$ in any execution E that is n-bounded-increment for both MR_{left} and MR_{right}.*

Proof. Since E is n-bounded-increment for both MR_{left} and MR_{right}, Equations (1) and (2) hold. We thus have:

$$\sum_{op\in Ops(E)} nsteps(op, E) = O\Bigg(\sum_{w\in W} nsteps(\mathsf{MR}, \mathtt{WriteMax}, w) + \sum_{I\in L} \Big(nsteps(\mathsf{MR}, \mathtt{ReadMax}, I) + nsteps(\mathsf{MA}_{index(I)}, \mathtt{WriteMax}, I) + nsteps(\mathsf{MA}_{index(I)}, \mathtt{MaxScan}, I) \Big) + \frac{|L|}{\left\lfloor \frac{n}{\log^2 m} \right\rfloor} n \Bigg). \tag{3}$$

Indeed, an instance of a MaxUpdate operation on the high-level unbounded 2-max array consists in an instance of a WriteMax operation applied to either MR_{left} or MR_{right}, followed by an instance of CompScan in which one or more iterations of the while loop are performed. Similarly, an instance of a MaxScan operation consists in one or more loop iterations of an instance of CompScan. Each step in iteration I is (1) either performed in an instance of a ReadMax on MR_{left} or MR_{right}, or (2) performed in an instance of WriteMax or MaxScan on $\mathsf{MA}_{index(I)}$, or (3) is a Write to an entry of the shared helping array. Additionally, at most every $\left\lfloor \frac{n}{\log^2 m} \right\rfloor$ iterations, an instance of Collect is performed on H, which incurs $O(n)$ additional steps for every such iteration.

The m-bounded 2-max array algorithm in [4] has $O(\log m)$ and $O(\log^2 m)$ worst case complexities for WriteMax and MaxScan operations, respectively. Hence,

$$\sum_{op\in Ops(E)} nsteps(op, E) = O\left(\sum_{w\in W} nsteps(\mathsf{MR}, \mathtt{WriteMax}, w) + \sum_{I\in L} nsteps(\mathsf{MR}, \mathtt{ReadMax}, I) + |L|(\log^2 m + \log m) \right). \tag{4}$$

Each instance of MaxUpdate calls WriteMax on an unbounded max register once, and in each iteration $I \in L$, the number of invocations of ReadMax operation on unbounded max registers is upper-bounded by a constant. It thus follows from Equation (2) that:

$$\begin{aligned} \sum_{w\in W} nsteps(\mathsf{MR}, \mathtt{WriteMax}, w) + \sum_{I\in L} nsteps(\mathsf{MR}, \mathtt{ReadMax}, I) &= O\left(\left(|Ops(\mathsf{MR}, \mathtt{WriteMax}, E)| + \Big| \bigcup_{I\in L} Ops(\mathsf{MR}, \mathtt{ReadMax}, I) \Big| \right) \log n \right) \\ &= O\big((|W| + |L|) \log n \big). \end{aligned} \tag{5}$$

Let $h^* = \left\lfloor \frac{r^*}{m} \right\rfloor + \left\lfloor \frac{\ell^*}{m} \right\rfloor$, where r^* and ℓ^* are respectively the value of MR_{right} and MR_{left} at the end of E. As E is n-bounded-increment for both MR_{right} and MR_{left}, at least $\frac{mh^*}{n}$ MaxUpdate instances appear in E. Hence,

$$h^* \leq \frac{n|W|}{m}. \tag{6}$$

Recall that L_u is the set of unsuccessful iterations $I \in L$ that complete. From Observation 6, unsuccessful iterations with the same index are performed by distinct processes. Hence, $|L_u| \leq nh^*$, from which it follows by combining Observation 7, Equation (6) and the fact that $|L| = |L_u| + |L_s| + |L_{nt}|$:

$$|L| \leq \frac{n^2|W|}{m} + |W| + |S|. \tag{7}$$

Therefore, by combining Equations (4) and (5), the bound on $|L|$ of Equation (7), and recalling that $m = \Theta(n^c)$, for some constant $c \geq 2$, we obtain:

$$\sum_{op \in Ops(E)} nsteps(op, E) = O\left((|W| + |L|)\log n + |L|\log^2 m\right) = O\left((|W| + |S|)\log^2 n\right).$$

Hence,

$$AmtSteps(E) = O\left(\frac{(|W| + |S|)\log^2 n}{|Ops(E)|}\right) = O\left(\frac{(|W| + |S|)\log^2 n}{|W| + |S|}\right) = O(\log^2 n),$$

which concludes the proof. □

Theorem 3.7. *If $m \geq n^2$, then Algorithm 1 is a wait-free linearizable n-process implementation of an unbounded 2-component max array, with amortized step complexity of $O(\log^2 m)$, in any execution E that is n-bounded-increment for both $\mathtt{MR}_{left}$ and $\mathtt{MR}_{right}$.*

Proof. Follows from Lemmas 3.1, 3.3, 3.5 and 3.6. □

4 WAIT-FREE SNAPSHOT WITH POLYLOGARITHMIC AMORTIZED STEP COMPLEXITY

Aspnes et al. present implementations of both a limited-use and an unlimited-use single-writer snapshot [4]. We obtain an *unlimited-use* single-writer snapshot with polylogarithmic amortized step complexity by making the following simple changes in their *limited-use* algorithm: We replace every instance of a bounded max register used by their algorithm by an instance of an unbounded max register implementation of [12]; we also replace every instance of a bounded 2-component max array they use with an instance of the unbounded 2-component max array implementation of Algorithm 1.

The resulting pseudo-code is presented in Algorithm 2. For simplicity, we assume that n is an integral power of 2. An `UnLimSnapshot` object encapsulates a balanced binary tree with n leaves. Each process i is uniquely associated with a leaf, denoted $\mathtt{leaf}_i$. Each node u is associated with an infinite array u.`views`, each entry of which is intended to store a partial snapshot. More precisely, a non-$\perp$ entry stores a partial snapshot of the components of the processes whose associated leaves are the leaves of the tree rooted at u. In particular, $\mathtt{leaf}_i$.`views` stores the sequence of inputs of the updates performed by process i. The index of the entry of u.`views` containing the most recent partial snapshot is stored in the parent node. To that end, each internal node v contains an unbounded 2-component max array v.`MA`, whose implementation is described by Algorithm 1. The left component of v.`MA` stores the index of the most recent partial snapshot on the array `views` of the left child. Similarly, the right component stores the index for the array `views` of the right child of v. As the root has no parent, an additional unlimited max register `root.MR` stores the index associated with the root.

For each array u.`views`, the index of the most recent partial snapshot is actually the number of instances of Update performed by the processes whose associated leaves are in the sub-tree rooted at u. When process i performs an instance of Update, it first increments a persistent local variable $count_i$, which keeps track of the number of Update instances performed by i (line 2). The new value is then written to the first $\perp$-entry of $\mathtt{leaf}_i$.`views` whose index is $count_i$ (line 4). Process i then propagates the new value of its component in the partial snapshots stored in the nodes on the path from its leaf to the root. For each node u in this path, process i obtains by performing a MaxScan on u.`MA` the indexes $lptr$ and $rptr$ of the most recent partial snapshots stored in the left and right children of u (line 9). It then reads the corresponding entries of the `views` arrays of the children (lines 10-11) to form a new partial snapshot, which is stored at index $s = lptr + rptr$ in u.`views` (line 13). The new index s is then reported into the appropriate component of the

max array associated with u's parent (line 8) (or, if u is the root, into the max register `root.MR`, in line 14). Consistency of partial snapshots (and also the full snapshots stored in the root) follows from the semantics of max arrays.

A Scan instance first obtains the index ptr of the last fully propagated snapshot by reading the max register `root.MR` (line 16) and then returns the snapshot stored in the corresponding entry of `root.views` (line 17).

Algorithm 2 Unlimited-Use Snapshot `UnLimSnapshot`, code for process i

Shared objects:

for each internal node u of a balanced binary tree with n leaves:
- u.MA UB-MaxArray, initially $(0, 0)$ ▷ Unbounded 2-component max array (Algorithm 1)
- u.views[0...] unbounded array, initially $[\perp, \ldots]$ ▷ Each entry is intended to store a partial snapshot
- u.parent, u.left, u.right ▷ Parent, left and right child of node u

for each leaf u:
- u.views[0...] unbounded array, initially $[\perp, \ldots]$
- u.parent ▷ Parent node of node u

root ▷ The root of the tree

root.MR: UnboundedMaxReg$_{n^2}$ register, initially 0 ▷ Unbounded max register [12] associated with the root of the tree

Persistent local variables for process i:

$count_i$: integer, initially 0 ▷ Number of Updates performed by i

$\texttt{leaf}_i$: (constant) unique leaf of the tree associated with process i

```
1:  function Update(v)
2:      count_i ← count_i + 1                          ▷ Keep track of the number of Updates by process i
3:      u ← leaf_i                                     ▷ Dedicated leaf for process i
4:      ptr ← count_i; u.view[count] ← v;              ▷ Store input in a fresh entry
5:      while u ≠ root do                              ▷ Loop until root is reached
6:          Let side ∈ {left, right} be s.t. u.parent.side = u   ▷ Identify on which side of the parent the current node is
7:          u ← u.parent                               ▷ Climb to parent node
8:          MaxUpdate(u.MA, side, ptr)                 ▷ Report the index of the most recent partial snapshot
9:          (lptr, rptr) ← MaxScan(u.MA)               ▷ Get the indexes of the most recent partial snapshot stored at the children
10:         lview ← u.left.views[lptr]                 ▷ Read partial snapshot from left child
11:         rview ← u.right.views[rptr]                ▷ Read partial snapshot from right child
12:         ptr ← lptr + rptr                          ▷ New index
13:         u.views[ptr] ← lview ∘ rview               ▷ Concatenate left and right child partial snapshots
14:     WriteMax(root.MR, ptr)                         ▷ Report index of last fully propagated snapshot
15: function Scan( )
16:     ptr ← ReadMax(root.MR)                         ▷ Get index of most recent full snapshot
17:     return root.views[ptr]                         ▷ Return most recent full snapshot
```

4.1 Correctness and complexity Analysis

We first prove that every execution E of Algorithm 2 is n-bounded increment for all the unbounded max registers used by the algorithm, i.e., the components of the unbounded max-array and the unbounded max register at the root.

Lemma 4.1. *Let u be an internal node in the tree and $h \geq 1$ denote its height. The 2-component max register u.MA is accessed by at most 2^h processes, and for each side, increments of the unbounded max register of u.MA of that side are 2^{h-1}-bounded.*

Proof. For the first part of the lemma, we note that the only processes that perform MaxUpdate or MaxScan operations on u.MA are those processes whose associated leaf is in the tree rooted at u. There are at most 2^h such leafs.

The proof of the second part is by induction on the height of the nodes. For the base case, let u be a node whose height is 1. Let $side \in \{left, right\}$. Since the height of u is 1, there is only a single process i that performs instances

of MaxUpdate(u.MA, *side*, v) operations. In each of these instances, the value written is the initial value of the local variable *ptr* when the loop starts (line 5). Initially, *ptr* is set to the value of the persistent local variable $count_i$ (line 4), which is incremented by 1 in each invocation of MaxUpdate(u.MA, *side*, v) by i (line 2). Therefore, increments on the components of the 2-component max array u.MA are 1-bounded.

For the induction case, let $h \geq 1$ and suppose that the lemma holds for every node whose height is at most h. Let x be a node whose height is $h+1$. Let us consider a MaxUpdate instance with input v on the *left* side of x.MA (the proof for the *right* side is similar). We denote by ins_x this instance. We first introduce some notations:

- For a node u of the tree, let T_u be the tree rooted at u.
- Given a node u of tree, let $P(u)$ be the set of processes i whose associated leaf $\texttt{leaf}_i$ is a leaf of T_u.
- For each process $j \in P(x.\texttt{left})$, let C_j the be set of instances of Update operations performed by process j that have completed before ins_x starts.

We establish the following claims:

(1) $v \leq \sum_{j \in P(x.\texttt{left})} |C_j| + 2^{h-1}$.

(2) There is a value v', such that $\sum_{j \in P(x.\texttt{left})} |C_j| \leq v'$ and v' was written to the left side of x.MA before ins_x starts.

From these claims, $v - 2^{h-1} \leq v'$ follows. If in addition $v' < v$ or $v \leq 2^{h-1}$, increments on the left side of x.MA are 2^{h-1}-bounded. Otherwise, let W denote the set of values $w \geq v$ that were written to the left side of x.MA before ins_x starts. Among these values, let $w*$ be the value whose write starts first. $w*$ is well-defined since $W \neq \emptyset$. By applying claims (1) and (2) to $w*$, there exists a value $w*'$ such that $w* - 2^{h-1} \leq w*'$ and whose write to the left side of x.MA precedes the write of $w*$. Hence, $w*' \notin W$. Therefore, as $v \leq w*$, we have $v - 2^{h-1} \leq w* - 2^{h-1} \leq w*' < v$, from which it follows that increments of left side of x.MA are 2^{h-1}-bounded.

We observe that, for every non-root node u in T_x, the Lemma follows from IH. Hence, for every such node u, u.MA is a linearizable 2-component max array. We can thus consider each instance of MaxScan or MaxUpdate performed on it as atomic. To prove claim (1), we note that it follows from the code that v is at most the sum of the values of the $count_j$ variables when ins_x starts, for all processes j that are associated with a leaf of the tree rooted at $x.\texttt{left}$. For each such j, $\texttt{count}_j$ is at most the number of Update instances by j that have completed before ins_x starts, together with possibly a single on-going instance. Hence, $v \leq \sum_{j \in P(x.\texttt{left})} count_j \leq \sum_{j \in P(x.\texttt{left})} |C_j| + 2^{h-1}$.

To prove claim (2), let $C = \bigcup_{j \in P(x.\texttt{left})} C_j$ and let $\mathcal{I}$ denote the set of MaxUpdate instances performed in an Update instance belonging to C on a 2-component max array u.MA where u is a non-root node in T_x. Finally, for any non-root node u of T_x, let (L_u, R_u) be the value of u.MA immediately after the last MaxUpdate instance on u.MA in $\mathcal{I}$.

Claim 5. *For any non-root node u of T_x, $\sum_{j \in P(u.\texttt{left})} |C_j| \leq L_u$ and $\sum_{j \in P(u.\texttt{right})} \leq R_u$.*

Proof. The proof is by induction on the height of the node.

- Base case. Let u be a non-root node in T_x whose height is 1. Let i_ℓ and i_r be the processes associated with its left and right child, respectively. After the last MaxUpdate instance in $\mathcal{I}$ on u.MA, by i_ℓ, the value stored in the left side is $|C_{i_\ell}|$. Similarly, the value stored in the right side after the last instance in $\mathcal{I}$ of MaxUpdate on u.MA is $|C_{i_r}|$. Hence, as the value stored on each side never decreases, $|C_{i_\ell}| \leq L_u$ and $|C_{i_r}| \leq R_u$.
- Induction step. Let u be a non-root node in T_x whose height is $h \geq 1$. We show that $\sum_{j \in P(u.\texttt{left})} |C_j| \leq L_u$. The proof for the right side is similar. Let u' denote the left child of u, and let w'_ℓ and w'_r be respectively the last instance in $\mathcal{I}$ of MaxUpdate performed on the left and right side of u'.MA, respectively. Assume that w'_r occurs after w'_ℓ (the other case is symmetric). By definition of $\mathcal{I}$, w'_r is performed in an instance of Update that

completes. Hence, it is followed by an instance of MaxScan (at line 9), and in the following iteration of the loop (lines 12 and line 8) the sum s of the values in the scan is written to the left side of u.MA. As the instance of MaxScan follows the last instance of MaxUpdate in $\mathcal{I}$ performed on u'.MA, $s \geq L_{u'} + R_{u'}$. Therefore, by the induction hypothesis, $s \geq \sum_{j \in P(u'.\texttt{left})} |C_j| + \sum_{j \in P(u'.\texttt{right})} |C_j| = \sum_{j \in P(u.\texttt{left})} |C_j|$. As the values stored on the left side of u.MA never decrease, it follows that $L_u \geq \sum_{j \in P(u.\texttt{left})} |C_j|$. □

Let $w \in \mathcal{I}$ be the last MaxUpdate instance performed on x.left.MA. As w is performed in an Update instance that terminates before ins_x starts, it is followed by a MaxScan instance on x.left.MA, and the sum σ of the pair of values returned by the scan is then written to the left side of x.MA. From Claim 5, $\sigma = L_{x.\texttt{left}} + R_{x.\texttt{left}} \geq \sum_{j \in P(x.\texttt{left})} |C_j|$. Moreover, the write of σ on the left side of x.MA terminates before ins_x starts, as it is performed in an Update instance that terminates before ins_x starts. Taking $v' = \sigma$ completes the proof of claim (2). □

Lemma 4.2. *The increments of the max register* root.MR *are n-bounded.*

Proof. Consider the implementation of Algorithm 2 for $2n$ processes. Suppose that only processes $1, \ldots, n$ perform instances of Update and Scan. In the $2n$-processes implementation, the values written on the left side on the 2-component max array at the root are sums of values of scans of the max array of the left child. By applying Lemma 4.1 to the max array at the root, increments of the left side are n-bounded.

The root of the n-process implementation can be identified with the left child of the root of the $2n$-processes implementation. In the n-processes implementation, values written to the max-register are sums of values of scans of the max array at the root. Hence, they are the same values written to the left side of max array at the root of the $2n$-processes implementation. It thus follows that increments of the max register root.MR are n-bounded as well in the n-processes implementation. □

Lemma 4.3. *The* UnLimSnapshot *implementation of Algorithm 2 has amortized step complexity of $O(\log^3 n)$ in any finite execution E.*

Proof. Our goal is to bound:

$$AmtSteps(E) = \frac{\sum_{ops \in Ops(E)} nsteps(ops, E)}{|Ops(E)|},$$

where $Ops(E)$ is the set of instances of Update and Scan operations that appear in E, and for each operation instance op, $nsteps(op, E)$ is the number of steps taken in this instance.

We partition $Ops(E)$ as follows:

$$Ops(E) = S \cup \bigcup_{1 \leq i \leq n} U_i,$$

where S is the set of instances of Scan and, for each i, $1 \leq i \leq n$, U_i is the set of Update instances initiated by process i. Given an object X, we denote by $nsteps(\texttt{X})$ the total number of steps taken by all the instances of operations on X. We thus have:

$$\sum_{ops \in Ops(E)} nsteps(ops, E) = nsteps(\texttt{root.MR}) + \sum_{u\text{: node of the tree}} nsteps(u.\texttt{MA}) + nsteps(u.\texttt{views}). \tag{8}$$

From Lemma 4.2, increments of the max register root.MR are n-bounded. The max register implementation in [12] has $O(\log n)$ amortized complexity in n-bounded-increment executions. Since in each instance of Update or Scan at most a single instance of an operation on root.MR is performed (at line 14 or at line 16), we have:

$$nsteps(\texttt{root.MR}) = O(|Ops(E)| \log n). \tag{9}$$

Let u be a non-leaf node in the tree, and let h_u denote its height. Recall that $P(u)$ is the set of processes i whose associated leaf $\texttt{leaf}_i$ is in the tree rooted in u. From the pseudo-code, each process $i \in P(u)$ performs a constant number of operation instances on $u.\texttt{MA}$ in each instance of Update. By Lemma 4.1, for each side, increments of $u.\texttt{MA}$ are 2^{h_u-1} bounded. Since the max register implementation of Algorithm 1 has $O(\log^2 n)$ amortized complexity in n-bounded-increment executions (Lemma 3.6), we have:

$$nsteps(u.\texttt{MA}) = O\left(\Big|\bigcup_{i\in P(u)} U_i\Big| \log^2 n\right)$$

Therefore,

$$\begin{aligned}\sum_{u\text{: node of the tree}} nsteps(u.\texttt{MA}) &= O\left(\sum_{1\le h\le \log n}\ \sum_{u:h_u=h} \Big|\bigcup_{i\in P(u)} U_i\Big| \log^2 n\right)\\ &= O\left(\Big|\bigcup_{1\le i\le n} U_i\Big| \log^3 n\right). \end{aligned} \tag{10}$$

In each Update instance by process i, for each visited node u in the tree, a constant number of accesses to the array $u.\texttt{views}$ are made by process i. In a Scan instance, only the array $\texttt{views}$ associated with the root is accessed. Hence, as each Update instance visits $O(\log n)$ nodes:

$$\sum_{u\text{: node of the tree}} nsteps(u.\texttt{views}) = O\left(\Big|\bigcup_{1\le i\le n} U_i\Big| \log n + |S|\right). \tag{11}$$

It thus follows from equations (9), (10) and (11) that

$$\begin{aligned}\sum_{op\in Ops(E)} nsteps(op, E) &= nsteps(\texttt{root.MR}) + \sum_{u\text{: node of the tree}} nsteps(u.\texttt{views}) + nsteps(u.\texttt{MA})\\ &= O\left((|S| + \Big|\bigcup_{1\le i\le n} U_i\Big|)(\log n + \log^3 n)\right).\end{aligned}$$

Therefore,

$$AmtSteps(E) = O(\log^3 n).$$

□

Theorem 4.4. *The* `UnLimSnapshot` *implementation of Algorithm 2 is an unlimited-use, wait-free and linearizable implementation of a single-writer snapshot with* $O(\log^3 n)$ *amortized step complexity.*

Proof. Lemmas 4.1-4.2 show that in any execution of Algorithm 2, increments to each side of each 2-max array or to the max register $\texttt{root.MR}$ are n-bounded. Hence, the unbounded max register implementation of [12] and the unbounded 2-max array implementation of Algorithm 1 are linearizable when used by Algorithm 2. The correctness proof of the limited-use single-writer snapshot algorithm of [4] (Section 4.1) does not rely on the fact that the max registers and max arrays it uses are bounded and thus holds also for Algorithm 2, where they are unbounded. Thus, Algorithm 2 is a linearizable implementation of an unbounded single-writer snapshot. The implementation is also wait-free, since each base object it uses is wait-free and each instance of Scan or Update accesses a bounded number of such objects. Finally, the amortized complexity follows from Lemma 4.3. □

5 DISCUSSION

We presented an n-process deterministic wait-free single-writer snapshot algorithm from reads and writes. We proved that the algorithm is linearizable and has $O(\log^3 n)$ amortized step-complexity. This is the first deterministic non-blocking read/write snapshot algorithm with sub-linear amortized step complexity in all executions. Since $\Omega(\log n)$ is a lower bound on the amortized complexity of counters [10, 12], and as counters can be easily implemented using a single-writer snapshot object, $\Omega(\log n)$ is also a lower bound on the amortized complexity of snapshot. Closing the gap between the lower and the upper bounds is an open question for future research.

REFERENCES

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1993. Atomic snapshots of shared memory. *Journal of the ACM (JACM)* 40, 4 (1993), 873–890.

[2] James H Anderson. 1993. Composite registers. *Distributed Computing* 6, 3 (1993), 141–154.

[3] James Aspnes, Hagit Attiya, and Keren Censor-Hillel. 2012. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM* 59, 1 (2012), 2:1–2:24. https://doi.org/10.1145/2108242.2108244

[4] James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. 2015. Limited-Use Atomic Snapshots with Polylogarithmic Step Complexity. *J. ACM* 62, 1 (2015), 3:1–3:22. https://doi.org/10.1145/2732263

[5] James Aspnes and Keren Censor-Hillel. 2013. Atomic Snapshots in $O(\log^3 n)$ Steps Using Randomized Helping. In *27th International Symposium on Distributed Computing (DISC) (Lecture Notes in Computer Science)*, Vol. 8205. Springer, 254–268. https://doi.org/10.1007/978-3-642-41527-2

[6] James Aspnes and Keren Censor-Hillel. 2018. Atomic Snapshots in $O(\log^3 n)$ Steps Using Randomized Helping. (2018). http://www.cs.yale.edu/homes/aspnes/papers/randomized-snapshot-abstract.html.

[7] James Aspnes and Maurice Herlihy. 1990. Fast randomized consensus using shared memory. *Journal of algorithms* 11, 3 (1990), 441–461.

[8] James Aspnes and Maurice Herlihy. 1990. Wait-Free Data Structures in the Asynchronous PRAM Model. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, (SPAA)*, Frank Thomson Leighton (Ed.). ACM, 340–349. https://doi.org/10.1145/97444.97701

[9] Hagit Attiya and Arie Fouren. 2001. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.* 31, 2 (2001), 642–664.

[10] Hagit Attiya and Danny Hendler. 2010. Time and Space Lower Bounds for Implementations Using k-CAS. *IEEE Trans. Parallel Distrib. Syst.* 21, 2 (2010), 162–173. https://doi.org/10.1109/TPDS.2009.60

[11] Hagit Attiya, Nancy Lynch, and Nir Shavit. 1994. Are wait-free algorithms fast? *Journal of the ACM (JACM)* 41, 4 (1994), 725–763.

[12] Mirza Ahad Baig, Danny Hendler, Alessia Milani, and Corentin Travers. 2019. Long-Lived Counters with Polylogarithmic Amortized Step Complexity. In *33rd International Symposium on Distributed Computing, (DISC) (LIPIcs)*, Vol. 146. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:16.

[13] Rainer Gawlick, Nancy A. Lynch, and Nir Shavit. 1992. Concurrent Timestamping Made Simple. In *Israel Symposium Theory of Computing and Systems (ISTCS) (Lecture Notes in Computer Science)*, Danny Dolev, Zvi Galil, and Michael Rodeh (Eds.), Vol. 601. Springer, 171–183.

[14] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 1 (1991), 124–149.

[15] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[16] Michiko Inoue and Wei Chen. 1994. Linear-Time Snapshot Using Multi-writer Multi-reader Registers. In *8th International Workshop on Distributed Algorithms (WDAG) (Lecture Notes in Computer Science)*, Gerard Tel and Paul M. B. Vitányi (Eds.), Vol. 857. Springer, 130–140.

[17] Prasad Jayanti, King Tan, and Sam Toueg. 2000. Time and Space Lower Bounds for Nonblocking Implementations. *SIAM J. Comput.* 30, 2 (April 2000), 438–456. https://doi.org/10.1137/S0097539797317299