Atomic snapshots in $O(\log^3 n)$ steps using randomized helping

James Aspnes[1] and Keren Censor-Hillel[2]

[1] Yale University, Department of Computer Science. aspnes@cs.yale.edu.
[2] Department of Computer Science, Technion. ckeren@cs.technion.ac.il.

Abstract. A randomized construction of unbounded snapshots objects from atomic registers is given. The cost of each snapshot operation is $O(\log^3 n)$ atomic register steps with high probability, where n is the number of processes, even against an adaptive adversary. This is an exponential improvement on the linear cost of the previous best known unrestricted snapshot construction [7, 8] and on the linear lower bound for deterministic constructions [9], and does not require limiting the number of updates as in previous sublinear constructions [4]. One of the main ingredients in the construction is a novel *randomized helping* technique that allows out-of-date processes to obtain up-to-date information without running into covering lower bounds.

1 Introduction

An **atomic snapshot** object allows processes to obtain the entire contents of a shared array as an atomic operation. The first known wait-free implementations of snapshot from atomic registers [1, 2, 6] required $\Theta(n^2)$ steps to carry out a snapshot with n processes; subsequent work [7, 8] reduced this cost to $O(n)$, which was shown to be optimal in the worst case for non-blocking deterministic algorithms by Jayanti *et al.* [9].

Limitations of the Jayanti *et al.* lower bound became apparent with the development of wait-free sublinear-complexity **limited-use** variants of objects to which the lower bound applied. These included deterministic implementations of **max registers** (which, when read, return the largest value written to them) and **counters** [3], and even snapshot objects [4], all with individual step complexity polylogarithmic in the number of operations applied to them.[3] These objects still have linear cost in the worst case, but the worst case is reached only after exponentially many operations.

The dependence on the number of operations was shown to be necessary initially for max registers [3], and later for a variety of objects satisfying a perturbability condition similar to that used in the Jayanti *et al.* lower bound [5].

[3] In the case of snapshot, this requires both registers large enough to hold a complete snapshot and the cooperation of updaters. The assumption of large registers may be avoidable for some applications of snapshot where only summary information is needed.

Curiously, for randomized implementations these lower bounds were not larger than $O(\log n)$ for any number of processes. This appeared to be a weakness of the particular proof technique used to obtain the randomized lower bounds.

We show that it is not the case that other techniques may produce larger lower bounds. Using a new randomized helping procedure along with a simple approximate max register implementation, it is possible to accelerate the max register implementation of [3] so that every operation finishes in $O(\log n)$ steps with high probability, regardless of the number of previous operations, provided the max register value does not change too quickly. Applying the same techniques to the **max array** of [4] (a pair of max registers supporting an atomic snapshot operation) yields a max array with $O(\log^2 n)$ step complexity with high probability, under the same restriction. This can be used in the snapshot implementation of [4] to obtain atomic snapshots with $O(\log^3 n)$ step complexity with high probability. Because the use of the max array within the atomic snapshot satisfies the restriction on changes in value, the complexity of the snapshot implementation holds without restrictions. The end result is a polylogarithmic snapshot implementation in which the cost of each operation does not depend on the number of operations but only on the number of processes.

1.1 Previous constructions

Before giving more detail on our construction, we give a quick review of the previous work on which it is based. The basic building block of the bounded snapshot construction in [4] is a 2-component max array. This object supports a write operation, which specifes a value and a component, and a read operation, which returns a pair of the maximal values written to the two components in all write operation linearized before it. To directly build an unbounded snapshot object we need an unbounded version of a max register, and an unbounded version of a 2-component max array.

The max register construction of [3] is based on a tree of *switches*, which are one-bit registers that initially hold the value 0 and can only be set to 1. Each leaf represents a value for the register. A write operation sets the switches on the path toward the respective leaf, while a read operation follows the rightmost path of set switches to get the largest value written. The problem with an unbounded max register according to this construction is that the length of an operation reading the rightmost path in the infinite tree construction is unbounded. This is because this operation is searching for the first node on the rightmost path whose switch is 0, and the depth of this node depends on the values that have been written, which are now unbounded. Even worse, such an operation is not guaranteed to be wait-free, as it might not terminate if new writes keep coming in with greater values, forcing it to continue moving down the tree to the right. To handle this, the tree is backstopped with a linear snapshot object that is used for larger values in order to bound the number of steps. Formally, this means that at some threshold level, the node on the rightmost path of switches no longer points to an infinite subtree of switches but rather to a single linear-time snapshot object, and all write operations set the switch at this node after

writing their value to the snapshot object, and all read operations accessing this node continue by reading the snapshot object. In total, this gives a complexity of $O(\min(\log v, n))$ steps per operation that reads or writes the value v.

The max array construction of [4] builds upon the above max register construction by combining the trees of the two components in a subtle manner. The data structure consists of a main tree, corresponding to the tree of the first component. The tree of the second component is embedded in the main tree at *every* node. That is, each switch of the main tree is associated with a separate copy of the tree of the second component. Writing to the first component is done by writing to the main tree, ignoring the copies of the second component at the switches. Writing to the second component is done by writing to the copy associated with the root of the main tree. The coordination between the pairs of values is left for the read operations. Such an operation travels the main tree in order to read the value of the first component, while dragging along the maximal value it reads for the second component along its path. It is proven in [4] that this implementation gives a linearizable 2-component max array.

1.2 Our Contributions

Our first contribution is an $O(\log n)$ construction of an unbounded max register, which overcomes the obstacle of the construction of [3] by combining a new **approximate max-register** with a novel technique of **randomized helping**. In essence, this technique allows an operation that is traveling down the tree to the right (we refer to the rightmost path of the tree as the **spine** of the tree) for too long to jump farther ahead to a point on the spine that is the correct one, that is, the first point on the spine for which the switch is unset. This is done by adopting a location in the spine used by another operation, with the challenge of making sure that this value is **fresh**—recent enough that the first operation can use it without violating linearizability. The only condition we place on the usage of the max register in order for this to work is that operations write values that are not increasing too fast. We need this condition in order to argue that once the operation found the correct node on the spine, it can safely continue to the left subtree without the worry that a new write operation is now writing a much larger value that is placed farther down the spine. While at first glance this might seem as a strong restriction, this is actually a very reasonable condition in applications that use max registers, and in particular it is satisfied by our implementation of an unbounded snapshot object.

Our second contribution is a 2-component max array that is unbounded, and whose cost per operation does not depend on the number of operations. The natural thing to try is embedding the unbounded max register construction in the 2-component max array construction of [4]. However, this does not work directly, since the main insight there is that values of the second component need to be propagated down while traveling the tree of the first component in order to guarantee that returned pairs are comparable. This cannot be done within our randomized helping technique because operations may jump down the spine without accessing each node along the way. We address this problem by

restructuring the 2-component max array implementation such that operations that go right on the spine re-read the value of the second component that is located at the root. The main observation here is that a single re-reading of the root is inexpensive, and that we do not care that this information skips the nodes between the root and the target node since the second component of these nodes will never be accessed again (because their switches are either set or skipped).

Plugging these two contributions into the snapshot implementation of [4] gives an implementation of an unbounded snapshot object with an $O(\log^3 n)$ step complexity (with high probability) for updating or scanning the object.

2 Unbounded max registers with bounded increments

A **max register** [3] supports operations `WriteMax`(v) and `ReadMax`(), where `WriteMax`(v) writes the value v to the max register and `ReadMax`() returns the largest value previously written. The purpose of a max register is typically to avoid lost updates, by ensuring that old values (tagged with smaller timestamps) cannot obscure newer values, regardless of the order in which they are written. In this section, we show how to construct an unbounded max register that is linearizable in all executions and wait-free with $O(\log n)$ step complexity with high probability in executions with bounded increments.

2.1 Bounded max registers

We begin by reviewing the max register implementation of Aspnes *et al.* [3]. The idea is to implement the register as a fixed binary tree of one-bit atomic registers, referred to as switch bits. Initially these bits are all 0, which is interpreted as pointing to the left child of the register, while a 1 points to the right child. Each value of the max register corresponds to a leaf of the tree (which does not get a register). A `ReadMax` operation follows the path determined by the values of the switch bits until it reaches a leaf; the number of leaves to the left of this leaf (its **rank**) gives the return value. (See Algorithm 1.)

An unbalanced tree backed by a linear-time snapshot implementation gives a cost of $O(\min(\log v, n))$ for an operation that read or writes the value v. Aspnes *et al.* [3] show that $O(\min(\log v, n))$ is optimal for deterministic obstruction-free max register implementations from atomic registers. For randomized implementations, they show a weaker lower bound of $O(\log n / \log\log n)$ steps for n-bounded max registers. This lower bound is obtained as a trade-off between the complexities of `ReadMax` and `WriteMax` operations.

We will show that with randomization, the dependence on v can be eliminated. It is possible to build a snapshot object (and thus a max register), whose cost is polylogarithmic in n with high probability for all operations, regardless of the size of the values it contains.

```
1  Shared data:
2  switch: a single bit multi-writer register, initially 0
3  left: a MaxRegister_m object, where m = ⌈k/2⌉, initially 0,
4  right: a MaxRegister_{k−m} object, initially 0
5
6  procedure WriteMax(r, v)
7      if v < m then
8          if r.switch = 0 then
9              WriteMax(r.left, v)
10     else
11         WriteMax(r.right, v − m)
12         r.switch ← 1
13
14 procedure ReadMax(r)
15     if r.switch = 0 then
16         return ReadMax(r.left)
17     else
18         return ReadMax(r.right) + m
```

Algorithm 1: Implementation of $\texttt{WriteMax}(r, v)$ and $\texttt{ReadMax}(r)$ for a $\mathsf{MaxRegister}_k$ object called r.

2.2 An unbounded max register implementation

We now show how to extend the results of [3] to allow an unbounded max register that nonetheless has fixed cost per operation with high probability. The first step is to bound the cost of `WriteMax` operations. We will do this under the assumption of **k-bounded increments**, which we will define by the rule that each new `WriteMax` operation writes a value v that is at most k more than the largest input to any previously initiated `WriteMax` operation.[4] This assumption will be justified later by the details of our unbounded snapshot construction.

As in a standard max register, the core of our unbounded max register is a binary tree of switch bits. But now the tree is infinite, consisting of an infinite **spine** forming the rightmost path through the tree, each node of which has an m-valued max register (implemented as a balanced $\lceil \log m \rceil$-depth tree), where m is an integer that will be chosen later, rooted at its left child (see Figure 1). Using this tree with the original algorithm, a $\texttt{WriteMax}(v)$ operation must walk all the way from the root of the tree to the corresponding leaf, which will be found in the $\lceil v/m \rceil$-th m-valued max register. It must then walk back up to the root, setting switch bits as needed, giving a cost of $O(v/m + \log m)$.

In our algorithm, we assume that the tree is packed in memory so that a $\texttt{WriteMax}(v)$ operation can access the root of the $\lceil v/m \rceil$-th max register directly. Within this subtree, it executes the standard algorithm; but along the spine, it sets only as many switch bits as are needed to guarantee that all ancestors are set;

[4] Note that we do not require that this previous `WriteMax` operation finished.

this is checked by performing an embedded `ReadMax` operation. This optimization does not affect correctness, because setting switches that are already set farther up the spine has no effect. What it does give is an improvement to the step complexity under the assumption of k-bounded increments, since between the n processes v can have increased by at most kn above the value of the last complete `WriteMax`, meaning that only kn/m steps up the spine are needed.

Setting aside for the moment the cost of the `ReadMax`, this gives a cost for the `WriteMax` of $O(\log m)$ for updating the m-valued max register plus $O(kn/m)$ for updating the segment of the spine. We will later choose k and m in a way for which the above results in $O(\log n)$ steps per `WriteMax` operation. Note that assuming bounded increments, this procedure gives this complexity for `WriteMax` operations without dependence on the value being written and that this implementation is deterministic. However, the `ReadMax` operations still suffer from the problem mentioned earlier: they are not wait-free in the presence of concurrent `WriteMax` operations with increasing values. For this we add an additional mechanism of randomized helping. Algorithm 2 is a pseudo-code of our implementation, where $\texttt{WrapWriteMax}_i$ and $\texttt{WrapReadMax}_i$ are the operations for process i, which invoke `WriteMax` and `ReadMax` operations as in [3] on the m-values max registers (in which the process id does not matter).

We now provide a high-level description of the helping mechanism. Each `WriteMax` operation is wrapped with a $\texttt{WrapWriteMax}_i$ procedure, as follows. $\texttt{WrapWriteMax}_i$ operations by process i cycle over the PIDs, helping one process at a time. The operation then reads the *timestamp*, $\mathsf{TS}[s]$, associated with the current helped process, s, written to $\mathsf{TS}[s]$ by a $\texttt{WrapReadMax}_s$ operation. It then reads the value v' of the max register, and if the value v it needs to write is larger than v' it goes ahead and writes it into the max register. It then records the maximum between v and v' into a helping array, along with the timestamp it saw for s, and updates a random location in a pointer array with its pid. A $\texttt{WrapReadMax}_i$ operation first increments its timestamp and then takes a certain amount of steps reading the max register. If it does not finish within that number of steps, it tries to get help from a random process chosen from a random location in the pointer array. Getting help is done by checking whether the chosen helping process, j, holds the current timestamp of process i, performing the $\texttt{WrapReadMax}_i$ operation, and if so, taking its value from its helping array.

The idea behind the proof is that if a `ReadMax` operation takes too many steps trying to read the max register without finishing, it must be that there are many concurrent `WriteMax` operations that keep sending it down the spine. But in such a case, the $\texttt{WrapReadMax}_i$ operation finds a value in one of the helping arrays that it may use, in the sense that it was updated by one of these concurrent $\texttt{WrapWriteMax}_j$ operations – specifically, after the $\texttt{WrapReadMax}_i$ operation started.

Next, we proceed with the formal proof. Let *spine* be the array induced by the switch bits on the spine of the tree. Let M_i be the m-valued max register whose root is $spine[i]$.

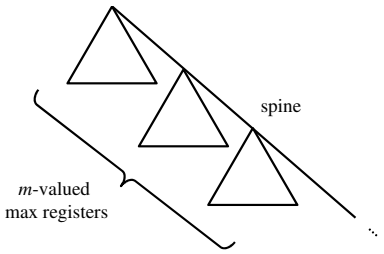

Fig. 1. An unbounded max register

```
1  Shared Data:
2  array TS[1..n] where TS[i] = timestamp for process i
3  array pointer[1..n^3]; each entry is a pid
4  array help[i]; each entry consists of
5     value = integer, most recent value seen by a WrapWriteMax_i operation
6     TS[j] = integer, most recent timestamp of p_j seen by a WrapWriteMax_i
   operation
7  procedure WrapWriteMax_i(v)
8      s ← s + 1 mod n // initialized to 0
9      t ← TS[s]
10     v' ← WrapReadMax_i()
11     if v > v' then
12         WriteMax(M_{⌊v/m⌋}, v mod m) // Write to the corresponding m-valued
           max register
13         for j = ⌊v/m⌋ to ⌊v'/m⌋ do
14             spine[j] ← 1
15     help[i].value ← max(v, v')
16     help[i].TS[s] ← t
17     pointer[random()] ← i
18 procedure WrapReadMax_i()
19     TS[i] ← TS[i] + 1
20     while true do
21         for t = 1 to c' log m // For a constant c', to be fixed in the step
           complexity proof do
22             Take a step of ReadMax()
23         if finished (initially false) then
24             return value
25         else
26             j ← pointer[random(1, ..., n^3)]
27             if help[j].TS[i] = TS[i] then
28                 return help[j].value
```

Algorithm 2: Max register with randomized helping; code for process i.

We linearize a $\texttt{WrapWriteMax}_i$ operation writing a value v at the first time in which all the relevant switches on the path from the root to the leaf corresponding to v are set. We linearize a $\texttt{WrapReadMax}_i$ operation that returns in Line 24 at the time the corresponding original `ReadMax` is linearized. We linearize a $\texttt{WrapReadMax}_i$ operation that returns in Line 28 at the linearization point of the $\texttt{WrapReadMax}_j$ operation by p_j that is part of the $\texttt{WrapWriteMax}_j$ operation that wrote to $\mathsf{help}[j].TS[i]$ the value read by $\texttt{WrapReadMax}_i$ in Line 27.

It is worth mentioning that, as the proof below shows, we do not need the assumption of k-bounded increments for linearizability of the construction. This assumption is used only for bounding the step complexity.

Lemma 1. *Algorithm 2 is a linearizable implementation of an unbounded max register.*

Proof. We base our proof on the correctness proof of the max register construction in [3]. We need to address two issues that differ in our implementation. First, we need to address $\texttt{WrapWriteMax}_i$ operations and show that the switches leading to a written value are indeed set by the time it terminates, showing that our linearization is well defined. The second issue is that we need to address $\texttt{WrapReadMax}_i$ operations that return in Line 28.

We use an induction on the order of linearization points to prove the correctness of the linearization. We add to the inductive claim the invariant that all switches on the path from the root to a leaf corresponding to a value v written by a `WriteMax` operation *op* are set if the path descends to their right child on the tree, by the time *op* finishes. This clearly holds for the base case, when no operation has yet been performed.

Assume that the linearization is correct up to some operation $t-1$ in the total order it induces. Let *op* be the t-th operation, and assume it is a $\texttt{WrapWriteMax}_i$ operation. By construction, all appropriate switches inside $M(\lfloor v/m \rfloor)$ are set in Line 12. By the induction hypothesis, all spine switches from the root down to location $\lfloor v'/m \rfloor$, where v' is the value read by *op* in Line 10, are set. The loop in Line 13 then shows that the invariant still holds.

Next, since correctness for $\texttt{WrapReadMax}_i$ operations that return in Line 24 now follows from the proof in [3], let *op* be a $\texttt{WrapReadMax}_i$ operation that returns in Line 28. Let op' be a $\texttt{WrapWriteMax}_j$ operation by p_j that writes to $\mathsf{help}[j].\mathsf{TS}[i]$ the timestamp read by *op* in Line 27. Let op'' be the $\texttt{WrapReadMax}_j$ operation performed by op' in Line 10. Since op'' is performed after *op* writes to $\mathsf{TS}[i]$ and before op' writes to $\mathsf{help}[j].\mathsf{TS}[i]$, the linearization point of op'' is within the execution interval of *op*. By the correctness of the linearization points of the construction in [3], the value returned by *op*, which is the maximum between the value returned by op'' and the value written by op', is the largest value written by operations that are linearized before *op*.

Having shown that this implementation is linearizable, we turn to prove its logarithmic step complexity. Here we choose $m = 3cn^3 \log n \leq O(n^4)$ and $k = O(n^2 \log^2 n)$ for some fixed constant c that is required by the proof.

Lemma 2. *The step complexity of operations in Algorithm 2 is $O(\log n)$ with high probability, when taking $m = O(n^3 \log n)$ and assuming k-bounded increments for $k = O(n^2 \log^2 n)$.*

Proof. Let op_i be a $\texttt{WrapReadMax}_i$ operation by p_i. We say that a process p_j is **current for operation** op_i if $\mathsf{help}[j].\mathsf{TS}[i] = \mathsf{TS}[i]$, where $\mathsf{TS}[i]$ is the timestamp written by op. Every process p_j can perform at most n $\texttt{WrapWriteMax}_j$ operations before it becomes current for op_i, since j iterates over the processes to help.

By a coupon collector argument (see, e.g., [10, Chapter 2]), there is a constant c such that after $cn^3 \log(n^3)$ executions of Line 26, op_i covers all elements of the array *pointer*. Suppose that $3cn^3 \log n$ wrapped `WriteMax` operations begin after $\mathsf{TS}[i]$ is incremented. Then, at most n^2 of these operations are by processes that are not current for op_i. There can be at most n^2 different locations in the *pointer* array written by such process, plus at most $n-1$ locations that have operations by current processes pending to write them, but still contain previous values. The rest of the $\Theta(n^3)$ locations hold values written by processes that are current for op_i. This implies that the probability of op_i choosing a random location in *pointer* that holds a value written by a process that is current for it is at least $1 - (n + n^2)/n^3 = 1 - O(1/n)$.

Assume now that op_i does not complete its `ReadMax` operation in Line 21 within $c' \log m$ steps, where the constant is such that the number of steps is enough to read a spine segment and an m-valued max register covering km values. For this to happen, op_i takes at least $O((c'-1)\log m)$ steps down the spine (otherwise, it goes down some m-valued max register and terminates within another $O(\log m)$ steps). By the k-bounded increments assumption, there are at least m values being written for this to happen. Taking m to be $3cn^3 \log n \leq O(n^4)$ now gives that the probability of op_i choosing a random location in *pointer* written by a process current for it is at least $1 - O(1/n)$. Therefore, with high probability, op_i finishes within $O(\log m) = O(\log n)$ steps.

A $\texttt{WrapWriteMax}_i$ operation op_i takes $O(\log m + kn/m)$ steps in addition to calling $\texttt{WrapReadMax}_i$. We choose $k = O(n^2 \log^2 n)$ such that $kn/m = O(\log n)$ and therefore the number of steps required for this operation is also $O(\log n)$, completing the proof.

3 Unbounded max arrays with bounded increments

To present our unbounded 2-component max array, we first describe the implementation in [4] and then show how to overcome the obstacles that arise when embedding our unbounded max register in that construction. The [4] 2-component max array roughly works as follows. It has a main tree for the max register of the first component, where each of the switches is associated with a MaxRegister variable tail, that holds copy of the max register of the second component. A write operation to the first component simply ignores these copies, and travels up the main tree from the relevant leaf to the root, setting the required switches along the way. A write operation to the second component writes

only to the tail copy associated with the root of the main tree. A read operation travels down the main tree reading the first component, while reading the tail copy of the second component at every switch and updating it if it saw a greater value earlier up the tree.

Propagating the values of the second component down the main tree is the key ingredient in guaranteeing that returned pairs are comparable. The main invariant that needs to be maintained is that a reader does not go right at a switch of the main tree returning a value for the second component that is smaller than that returned by a reader who goes left at that switch. In [4], this is guaranteed by having the reader re-read the tail copy of a switch that is set, and propagating this fresher value down to the right subtree.

However, embedding our max register in this construction does not work: in our max register implementation, a read operation does not travel all the way down from the root to the leaf, therefore it cannot drag the value of the second component with it. This causes gaps in the values of the tail copies of the second component along the tree, violating the required invariant.

To solve this, our observation is that we can re-read the tail copy of the second component associated with the root of the main tree, instead of reading the tail component of the current spine node, which may not have been updated. This guarantees that the value returned for the second component is always updated to the largest one written. Notice that we can only do this with read operations that go down the rightmost path of the main tree, that is, the spine. Otherwise, an operation that started early and goes left at some switch of the main tree might read a value for the second component that is too large: larger than the one read by a quicker operation that goes right. But the fact that we can do this only for the spine fits our goals, and our approach to handle the above issue is to re-read the tail variable at the root only when traveling the spine. At other switches the reader copies the values down the tree as in the original construction, which is unaffected by our max register implementation since gaps in switches can only occur on the spine, as a process going down some m-valued max register travels an entire path from its root to a leaf. Algorithms 3 and 4 show the pseudo-code.

Instead of repeating the linearizability proof of the 2-component max array in [4] (denoted by Alg hereafter), we reduce the algorithm in Algorithms 3 and 4 to Alg. In particular, we show that any execution of the algorithm can be translated to an execution of Alg in a way which preserves returned values, implying that the linearization of Alg also applies to the algorithm in Algorithms 3 and 4. The intuition is that whenever a `ReadMaxArray` operation goes down the spine of the main tree, just before it is about to read the copy of the second component again before going right, we imagine that a very quick `ReadMaxArray` operation in Alg starts and runs solo, going down the spine of the main tree, propagating the value of the second component that is at the copy of the second component associated with the root. If we then let the first `ReadMaxArray` operation do its read then it gets exactly the value associated with the root at that time. Hence, it cannot distinguish between these two executions, and we can take its

```
1  Shared Data:
2  switch: a 1-bit multi-writer register, initially 0
3  left, right: two MaxArray objects with an unbounded second component, initially
     (0,0); at the spine, left has an m-bounded first component and right has an
     unbounded first component; at a MaxArray with a b-bounded first component
     for any integer b, the first component of both left and right is b/2-bounded
4  tail: an unbounded MaxRegister object, initially 0
5  array TS[1..n] where TS[i] = timestamp for process i
6  array pointer[1..n^3]; each entry is a pid
7  array help[i]; each entry consists of
8    value = most recent value seen by p_i
9    TS[j] = most recent timestamp seen by p_i for p_j
10 procedure WriteMaxArray0(r, v) // Write to the first component
11   s ← s + 1 mod n // initialized to 0
12   t ← TS[s]
13   (v', v'') ← ReadMaxArray(r)
14   if v > v' then
15     WriteMax(M_⌊v/m⌋, v mod m)
16     for j = ⌊v/m⌋ to ⌊v'/m⌋ do
17       spine[j] ← 1
18   help[i].value ← max(v, v')
19   help[i].TS[s] ← t
20   pointer[random(1, ..., n^3)] ← i
21
22 procedure WriteMaxArray1(r, v) // Write to the second component
23   WrapWriteMax_i(r.tail, v)
```

Algorithm 3: Writing to the 2-component max array; code for process i.

linearization point as that of its corresponding operation in Alg. Following is the formal proof of the above argument.

Theorem 1. *The algorithm in Algorithms 3 and 4 is a linearizable implementation of a 2-component max array. It has a step complexity of $O(\log^2 n)$ per operation with high probability, when taking $m = O(n^3 \log n)$ and assuming k-bounded increments for $k = O(n^2 \log^2 n)$.*

Proof. Let α be an execution of the algorithm in Algorithms 3 and 4 with processes $\{p_0, \ldots, p_{n-1}\}$. We construct a sequence of executions $\alpha_0, \alpha_1, \ldots, \alpha'$, which ends in an execution α' of Alg, for which the return values of all operations are the same as in α.

Every execution α_j in the sequence is an execution with $n+1$ processes, such that every process $p_i \in \{p_0, \ldots, p_{n-1}\}$ invokes the same operations as in α, and process p_n is an extra process that performs only `ReadMaxArray` operations. If in α the process p_i reads the copy of the second component associated with the root in Line 8, then starting from some α_j it reads the copy associated with the

```
1  procedure ReadMaxArrayDirect(r)
2      x ← WrapReadMax_i(r.tail)
3      if r.switch = 0 then
4          WrapWriteMax_i(r.left.tail, x)
5          return ReadMaxArrayDirect(r.left)
6      else
7          if on spine then
8              x ← WrapReadMax_i(root.tail)
9          else
10             x ← WrapReadMax_i(r.tail)
11         WrapWriteMax_i(r.right.tail, x)
12         return ReadMaxArrayDirect(r.right) + (m, 0)
13
14 procedure ReadMaxArray(r)
15     TS[i] ← TS[i] + 1
16     while true do
17         for t = 1 to c' log m // For a constant c' as in Algorithm 2 do
18             Take a step of ReadMaxArrayDirect(r)
19         if finished then
20             return pair
21         else
22             j ← pointer[random(1, ..., n^3)]
23             if help[j].TS[i] = TS[i] then
24                 firstComponent ← help[j].value
25                 return ReadMaxArrayDirect(spine[firstComponent/m])
26
```

Algorithm 4: Reading the 2-component max array; code for process i.

current switch (notice that this difference only occurs when reading locations on spine).

Even though p_i reads different locations in α and α_j, steps by p_n are used to make it obtain the same values. We define the behavior of p_n by induction. In α_0 the process p_n is not used, therefore it is the execution described above. Assume executions $\alpha_0, \ldots, \alpha_j$ are defined and define execution α_{j+1} as follows. Let p_i be the first process in α_j that reads root.tail in Line 8 corresponding to some location x on the spine. Denote $\alpha_j = \alpha'_j s_i \alpha''_j$ such that s_i is that step of p_i (note that we can assume an operation on a max register is an atomic operation). We define $\alpha_{j+1} = \alpha'_j \sigma s'_i \alpha''_j$, where in σ process p_n performs a read operation and s'_i is a step by p_i reading the copy of the second component associated with location x.

Our claim is that all operations return the same values in α_j and in α_{j+1}. The reason is that p_n reads the copy of the second component associated with the root of the main tree and copies it down the spine at least until location x since it starts after p_i reaches x and hence all switches toward it are set.

Therefore, when p_i reads the copy in x in s'_i in α_{j+1} it gets the same value it reads from the root in s_i in α_j. Finally, for some j we reach an execution $\alpha' = \alpha_j$ of Alg, for which all returned values of processes $\{p_0, \ldots, p_{n-1}\}$ are the same as in α. This execution α' is linearizable by the proof of [4]. Because p_n performs only `ReadMaxArray` operations, removing these operations from the linearization of α' does not affect the return values of any other operations; this reduced linearization is thus a linearization of α.

4 Unbounded snapshots

Given our unbounded 2-component max array implementation, we can now obtain an unbounded snapshot object.

We use the construction from [4], which for convenience we restate here in Algorithm 5 The shared data is:

- leaf_j, for $j \in \{0, \ldots, n-1\}$: the leaf node corresponding to process j, with fields:
  - parent: the parent of this leaf in the tree
  - view$[0, 1, \ldots]$: an infinite array, each of whose entries contains a partial snapshot, view$[0]$ contains the initial value of component j and view$[\ell]$ contains the ℓ-th value of component j
  - root: the root of the tree
- Each internal node has the fields:
  - left: the left child of the node in the tree
  - right: the right child of the node in the tree
  - view$[0, 1, \ldots]$: an infinite array, each of whose entries contains a partial snapshot, view$[0]$ contains the concatenation of leaf_j.view$[0]$ for all leaves leaf_j in the subtree rooted at this node, and view$[\ell]$ contains the concatenation of views of the leaves after ℓ updates
  - ma: an infinite MaxArray object, initially (0,0)
- The root also has the field mr: an infinite MaxRegister object, initially 0
- Each non-root internal node also has the field parent: the parent of the node in the tree

We use this algorithm with our implementations of unbounded max registers and unbounded max arrays from the previous sections. Loosely speaking, the construction is based on a balanced binary tree with n leaves, one for each process. Each intermediate node holds a 2-component max array object for its two children, that counts the number of update operations performed on each. It also stores the (unique) view corresponding to this number. A process that updates its location does so by updating the nodes from its leaf to the root, and a process scans the object by reading the view held by the root. We emphasize that correctness is always guaranteed in the above implementation, therefore the proof from [4] shows that this gives an unbounded snapshot object. It remains to show the step complexity of our construction. For this, we only need to show that the k-bounded increment assumption holds, and use the complexity analysis of

```
1  procedure Update(s, i, v)
2      count_i ← count_i + 1
3      u ← leaf_i
4      ptr ← count_i
5      u.view[ptr] ← v
6      while u ≠ root do
7          if u = u.parent.left then
8              WriteMaxArray0(u.parent.ma, ptr)
9          if u = u.parent.right then
10             WriteMaxArray1(u.parent.ma, ptr)
11         u ← u.parent
12         (lptr, rptr) ← ReadMaxArray(u.ma)
13         lview ← u.left.view[lptr]
14         rview ← u.right.view[rptr]
15         ptr ← lptr + rptr
16         u.view[ptr] ← lview · rview
17     WriteMax(root.mr, ptr)
18 procedure Scan(s)
19     ptr ← ReadMax(root.mr)
20     return root.view[ptr]
```

Algorithm 5: Unbounded snapshot object; code for process i.

the previous sections. Intuitively, this is because every MaxRegister is used only to store the number of operations observed in the subtree of processes that it represents. If the difference between two values written to a MaxRegister is more than n, then some processes completed a `WriteMax` operation between these two `WriteMax` operations, implying that the maximal difference was smaller to begin with. Formally, we prove this claim in the following lemma.

Lemma 3. *In Algorithm 5, all* MaxRegister *and* MaxArray *objects are accessed according to the n-bounded increments assumption.*

Proof. A process that performs `WriteMaxArray` on u.ma for some node u writes the value of its ptr variable. We show that ptr holds a value which is at most the number of `Update` operations invoked by processes corresponding to this subtree, hence a value being written to u.ma is larger by at most n than the largest value previously written to it. The claim follows by a simple induction on the height of the node that holds the object. When accessing a leaf, ptr holds the value of count_i, which is the number of operations performed by process p_i. For an intermediate node u, ptr holds the sum of the values of its two children, which, by the induction hypothesis are the number of `Update` operations invoked by processes corresponding to these subtrees, which proves the claim. Finally, the same holds for the value of ptr when the root is accessed, implying the claim also for the MaxRegister object there.

Combining Lemma 3 with Theorem 1 gives our main theorem.

Theorem 2. *Algorithm 5 is an implementation of an unbounded snapshot object, with a step complexity of $O(\log^3 n)$ per operation with high probability.*

5 Discussion

This paper gives the first sub-linear unbounded snapshot implementation from atomic read/write registers. It is a randomized algorithm, with a step complexity of $O(\log^3 n)$ with high probability for each operation, where n is the number of processes. The main component of the construction is a new randomized implementation of an unbounded max register with a complexity of $O(\log n)$ steps per operation with high probability. The novelty of the construction is a randomized helping technique, which allows slow processes to obtain fresh information from other processes. The use of randomization avoids in most cases the linear worst-case lower bound based on covering of Jayanti *et al.* [9], because the adversary cannot predict what locations a process will read from the helper array and thus cannot guarantee to cover those locations with old values. Conversely, the lower bound shows that some use of randomization is necessary.

Acknowledgement: The authors thank the anonymous reviewers for careful comments and suggestions that helped improve the presentation of this work. The first author is supported in part by NSF grant CCF-0916389. The second author is a Shalon Fellow.

References

1. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.
2. James H. Anderson. Multi-writer composite registers. *Distributed Computing*, 7(4):175–195, 1994.
3. James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):2:1–2:24, March 2012.
4. James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. Faster than optimal snapshots (for a while). In *2012 ACM Symposium on Principles of Distributed Computing*, pages 375–384, July 2012.
5. James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Danny Hendler. Lower bounds for restricted-use objects. In *Twenty-Fourth ACM Symposium on Parallel Algorithms and Architectures*, pages 172–181, June 2012.
6. James Aspnes and Maurice Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Second Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 340–349, July 1990.
7. Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, 2001.
8. Michiko Inoue and Wei Chen. Linear-time snapshot using multi-writer multi-reader registers. In *WDAG '94: Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 130–140, London, UK, 1994. Springer-Verlag.
9. Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2):438–456, 2000.
10. Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.