Limited-Use Atomic Snapshots with Polylogarithmic Step Complexity

JAMES ASPNES, Yale University
HAGIT ATTIYA, Technion
KEREN CENSOR-HILLEL, Technion
FAITH ELLEN, University of Toronto

This paper presents a novel implementation of a snapshot object for n processes, with $O(\log^2 b \log n)$ step complexity for update operations and $O(\log b)$ step complexity for scan operations, where b is the number of updates. The algorithm uses only reads and writes.

For polynomially many updates, this is an exponential improvement on previous snapshot algorithms, which have linear step complexity. It overcomes the existing $\Omega(n)$ lower bound on step complexity by having the step complexity depend on the number of updates. The key to this implementation is the construction of a new object consisting of a pair of max registers that supports a scan operation.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; E.1 [**Data Structures**]: Distributed data structures; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms: Theory, Algorithms

Additional Key Words and Phrases: Concurrent objects, restricted-use objects, atomic snapshot, generalized counters

1. INTRODUCTION

An *atomic snapshot object* [Afek et al. 1993; Anderson 1993; Aspnes and Herlihy 1990] is a fundamental data structure for shared memory computation. It allows each process to update individual components of a shared array and to scan the entire array so that these operations seem to take effect atomically.

Atomic snapshots provide a crucial tool for many shared-memory algorithms, as they simplify coordination between processes. A typical example is a *generalized counter*, which supports a read operation that returns the value of the object and an operation that adds an arbitrary positive or negative integer to its value. With atomic snapshots, each process can store its "contribution" (the sum of the amounts by which it has incremented and decremented the value of the generalized counter) in a component that only it can update. Using a scan, a process gets an instantaneous view of all the contributions, which it sums to obtain the value of the counter.

Recently, it was shown that a counter, which only allows updates that add one to the counter value, can be implemented with polylogarithmic (in n) step complexity using only

A preliminary version of this paper appeared in Proceedings of the *31st Annual ACM Symposium on Principles of Distributed Computing*, 2012. James Aspnes is supported in part by NSF grant CCF-0916389. Hagit Attiya is partially supported by the *Israel Science Foundation* (grant number 1227/10) and Yad-Hanadiv foundation. Keren Censor-Hillel is a Shalon Fellow. Part of this work was done while at MIT and supported by the Simons Postdoctoral Fellows Program. Faith Ellen is supported in part by the Natural Science and Engineering Research Council of Canada.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 0000-0000/YYYY/01-ARTA $10.00
DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

reads and writes, assuming the number of increments is polynomial (in n) [Aspnes et al. 2012a]. This is indeed the case for many applications of a counter. The construction is based on an implementation of a bounded *max register*. It extends to other concurrent data structures, provided that they can be represented by monotone circuits. However, it critically depends on the facts that the value of the counter is monotonically increasing and that all increment operations have the same effect. Therefore, the construction cannot be used to implement a generalized counter or an atomic snapshot.

In this paper, we present a linearizable implementation of atomic snapshots with $O(\log n)$ step complexity for scan and $O(\log^3 n)$ step complexity for update, as long as the number of update operations that are performed is polynomial in n. Instead of having a process directly read a linear number of registers to perform an operation, our implementation allows processes performing scans and updates to cooperate to reduce the cost exponentially, provided the snapshot object is only updated polynomially many times. This implies implementations with polylogarithmic step complexity using only reads and writes for a wide variety of shared-memory objects, including generalized counters, when they are updated only a polynomial number of times. Many important applications, such as task allocation [Alistarh et al. 2014], renaming [Borowsky and Gafni 1993], approximate agreement [Attiya et al. 1994], and randomized consensus [Aspnes and Censor 2009] use objects satisfying this restriction.

The key technical development behind our results is the definition and implementation of a linearizable 2-*component max array*, a new data structure consisting of two components, each of which is a max register that may be updated independently, and which supports a `MaxScan` operation that returns the values of both components. The pairs (x_0, x_1) and (y_0, y_1) returned by different `MaxScan`s are always comparable in the sense that either $x_0 \leq y_0$ and $x_1 \leq y_1$ or $y_0 \leq x_0$ and $y_1 \leq x_1$. The implementation of the 2-component max array is based on inserting copies of the second component at all levels of a tree of registers implementing the first component using the construction of Aspnes et al. [2012a].

The 2-component max array is exactly the tool we need to coordinate the recursive construction of atomic snapshots. We use a binary tree of 2-component max arrays to manage the combination of increasingly wide snapshots of parts of an array of n values. The max registers store increasing indices into a table of partial snapshot values. The scan of a max array is used to guarantee that the two halves of a partial snapshot are consistent with each other. By requiring updaters to propagate their new values up the tree, we amortize the cost of constructing an updated snapshot of all n components across the updates that modify it. This allows a process to obtain a precomputed snapshot in a sublinear number of steps. Note that our implementation uses registers that can store $\Theta(n)$ values. Sublinear step complexity for scan is impossible only using objects that can store $O(1)$ values, since the output of a scan consists of n values.

It has been shown [Aspnes et al. 2012b] that collect objects and, hence, snapshot objects have $\Omega(\min(\log b, n))$ step complexity, where b is the number of updates performed. This indicates that our implementation is close to optimal.

A survey of implementations of atomic snapshot objects using only reads and writes, as well as lower bounds, appears in [Fich 2005]. The best previously-known algorithms for atomic snapshots using only reads and writes [Attiya and Fouren 2001; Inoue et al. 1994] have $\Theta(n)$ step complexity for both updates and scans. An interesting implementation of atomic snapshots using f-arrays takes one step for a scan and $\Theta(\log n)$ steps for an update, but it uses CAS or LL/SC objects [Jayanti 2002]. All of these implementations use objects that store $\Omega(n)$ values.

LL/SC objects storing w words can be implemented from single-word LL/SC objects [Anderson and Moir 1995] or CAS objects [Jayanti and Petrovic 2005] so that each implemented operation takes $O(w)$ steps. Likewise, a register storing w words can be implemented from single-word registers using indirection so that each implemented read or write takes $O(w)$ steps.

Atomic snapshots that take one step for an update and $\Theta(n)$ steps for a scan have been implemented using CAS or LL/SC objects [Jayanti 2005; Riany et al. 2001].

There is an $\Omega(n)$ lower bound on the step complexity of implementing an atomic snapshot from historyless objects (such as registers and swap objects) and resettable consensus objects [Jayanti et al. 2000], without any bound on the size of the objects used for the implementation. However, the proof uses executions that are exponentially long as a function of n.

2. MODEL AND PRELIMINARIES

Consider a deterministic asynchronous shared-memory system comprised of n processes, which communicate through shared registers that support read and write of arbitrarily large values. We assume that any number of processes can fail by crashing.

An *implementation* of a shared object in this system provides a representation of the object using shared registers and an algorithm for each type of operation supported by the object. The implementation is *linearizable* [Herlihy and Wing 1990] if, for every execution, there is a total order of all completed operations and a subset of the uncompleted operations in the execution that satisfies the sequential specifications of the object and is consistent with the real-time ordering of these operations (i.e. if an operation is completed before another operation begins, then the former operation occurs earlier in the total order).

There are a number of different shared objects we consider. A *counter*, r, supports two operations, $\texttt{Read}(r)$ and $\texttt{Increment}(r)$. If r is a *generalized counter*, then it also supports $\texttt{Add}(r, v)$, where $v \in \mathbb{Z}$, i.e. it allows the value of r to be atomically changed by an arbitrary integer, instead of simply being incremented by 1.

An *atomic snapshot* object consists of a finite array of m components. $\texttt{Update}(r, i, v)$ sets the value of component i of snapshot object r to v. $\texttt{Scan}(r)$ atomically reads the values of all m components. In a *single-writer snapshot object*, the number of components, m, is equal to the number processes, n, and only process i can update component i.

A *max register*, r, is an object that supports two operations, $\texttt{ReadMax}(r)$, which returns the value of r, and $\texttt{WriteMax}(r, v)$, which sets the value of r to $v \in \mathbb{N}$, if its value was less than v. Thus, a $\texttt{ReadMax}(r)$ operation returns the largest value of v in any $\texttt{WriteMax}(r, v)$ operation that is linearized before it. For any positive integer k, a *bounded max register* with *range* k is a max register whose values are restricted to $\{0, \ldots, k-1\}$. We say that it has type $\texttt{MaxReg}_k$.

A *2-component max array* consists of a pair of bounded max registers, with an atomic operation that returns the values of both of them and operations that update each of them. Specifically, an object, r, of type $\texttt{MaxArray}_{k \times h}$ supports three linearizable operations:

— $\texttt{MaxUpdate0}(r, v)$, which sets the value of the first component of r to $v \in \{0, \ldots, k-1\}$ if its value is less than v,
— $\texttt{MaxUpdate1}(r, v)$ sets the value of the second component of r to $v \in \{0, \ldots, h-1\}$ if its value is less than v, and
— $\texttt{MaxScan}(r)$, which returns the value of r, i.e. it returns a pair (v, v') such that v and v' are, respectively, the largest values in any $\texttt{MaxUpdate0}(r, v)$ and $\texttt{MaxUpdate1}(r, v')$ operations that are linearized before it.

The results of two $\texttt{MaxScan}(r)$ operations in a linearizable execution are never incomparable under the componentwise $\leq$ partial order, i.e., it is never the case that $u < v$ and $u' > v'$, for any pair of $\texttt{MaxScan}$ operations returning (u, u') and (v, v').

Algorithm 1 An implementation of a $\texttt{MaxReg}_k$ object, for $k > 1$

```
Shared data:
    switch: a single bit multi-writer register, initially 0
    left: a MaxReg_m object, where $m$ is typically $\lceil k/2 \rceil$, initially 0,
    right: a MaxReg_{k-m} object, initially 0

1:  WriteMax($r, v$):
2:      if $v < m$
3:          if $r$.switch $= 0$
4:              WriteMax($r$.left, $v$)
5:      else
6:          WriteMax($r$.right, $v - m$)
7:          $r$.switch $\leftarrow 1$

8:  ReadMax($r$):
9:      if $r$.switch $= 0$
10:         return ReadMax($r$.left)
11:     else
12:         return ReadMax($r$.right) $+ m$
```

A *b-limited-use* object limits the total number of update operations (e.g. `Increment`, `Add`, or `Update`) that can be applied to it during an execution to at most b. Operations that do not change the value of the object can be applied an unlimited number of times.

The *step complexity* of an operation in an implementation of an object is the worst-case number of accesses to shared memory by a process while performing a single instance of the operation.

3. IMPLEMENTING A 2-COMPONENT MAX ARRAY

We begin with a description of the implementation of a $\texttt{MaxReg}_k$ object from registers [Aspnes et al. 2012a], since our implementation of a $\texttt{MaxArray}_{k\times h}$ object is based on it. The smallest max register, the trivial $\texttt{MaxReg}_1$ object, requires no reads or writes and uses no space: $\texttt{WriteMax}(r, 0)$ does nothing and $\texttt{ReadMax}(r)$ simply returns 0. To get larger max registers, smaller ones are combined recursively.

Pseudocode for the implementation of a basic $\texttt{MaxReg}_k$ object, r, with range k appears in Algorithm 1. It consists of a single bit register, $r.\mathsf{switch}$, and two smaller max registers, $r.\mathsf{left}$, with range $m < k$ and $r.\mathsf{right}$, with range $k - m$. When $r.\mathsf{switch} = 0$, the value of r is the value of $r.\mathsf{left}$; when $r.\mathsf{switch} = 1$, the value of r is m plus the value of $r.\mathsf{right}$. This gives a simple recursive algorithm for $\texttt{ReadMax}$. If $v \geq m$, a process performs $\texttt{WriteMax}(r, v)$ by recursively calling $\texttt{WriteMax}(r.\mathsf{right}, v - m)$ and then setting $r.\mathsf{switch}$ to 1. Otherwise, it first checks that $r.\mathsf{switch} = 0$ and, if so, recursively calls $\texttt{WriteMax}(r.\mathsf{left}, v)$. If $r.\mathsf{switch} = 1$, the value of r is already at least m, so no recursive call is needed. When $m = \lceil k/2 \rceil$ at each step of the recursion, the construction results in a balanced tree of depth $\lceil \log_2 k \rceil$. Both `ReadMax` and `WriteMax` then have $O(\log k)$ step complexity.

As observed by Aspnes et al. [2012a], it is also possible to use an unbalanced tree to make the cost of each operation depend on the value written or read, giving a cost of $O(\log v)$ to read or write v. We will show that a similar strategy works for max arrays.

Next, we turn attention to the implementation of a $\texttt{MaxArray}_{2\times 2}$ object, r. Suppose we use two $\texttt{MaxReg}_2$ objects, r_0 and r_1, one storing the value of each component. Then $\texttt{MaxUpdate0}(r, v)$ can be performed by performing $\texttt{WriteMax}(r_0, v)$ and $\texttt{MaxUpdate1}(r, v)$ can be performed by performing $\texttt{WriteMax}(r_1, v)$. However, it is incorrect to perform $\texttt{MaxScan}(r)$

by simply collecting the values of both components, i.e., by performing `ReadMax`(r_0) followed by `ReadMax`(r_1). For example, consider an execution in which processes p and p' each perform `MaxScan`(r) by performing `ReadMax`(r_0) followed by `ReadMax`(r_1), interleaved with an update to the first component and an update to the second component, both performed by process q. If the execution occurs as in Figure 1, p returns (0,1) and p' returns (1,0), which are incomparable. Thus, it is impossible to linearize both their operations.

```
p:   ReadMax(r0)                                                          ReadMax(r1)
p':                                    ReadMax(r0)   ReadMax(r1)
q:                 MaxUpdate0(r,1)                                 MaxUpdate1(r,1)
```

Fig. 1. An execution of an incorrect implementation of a `MaxArray`$_{2\times 2}$ object

However, if the only possible values are 0 and 1, there is a correct implementation of `MaxScan`(r) that is only slightly more complicated: When a process obtains (0,0) from a collect, it can return (0,0) and its operation can be linearized at its first step. Similarly, a process that obtains (1,1) can return (1,1) and be linearized at its last step. If a process obtains either (0,1) or (1,0), it can return the pair of values resulting from performing `ReadMax`(r_0) and `ReadMax`(r_1) again. Since the value of each component is nondecreasing, its second collect will either return (1,1) or the same pair as its first collect. In the latter case, we have an identical *double collect* [Afek et al. 1993], and the operation can be linearized between the two collects.

More generally, if r is a `MaxArray`$_{k\times h}$ object, then `MaxScan`(r) can be performed by repeatedly performing `ReadMax`(r_0) followed by `ReadMax`(r_1) until the result is either (0,0), (k,h), or the same pair twice in a row. Unfortunately, the worst case step complexity of this implementation is $\Theta((k+h)(\log k + \log h))$, since the value of the first component can change $k-1$ times and the value of the second component can change $h-1$ times.

The challenge in implementing a significantly faster `MaxArray`$_{k\times h}$ object is to ensure that, in each execution, all pairs returned by the `MaxScan` operations are comparable. Our approach is to make the `MaxScan` operations be responsible for this coordination. For the first component, we use the same binary tree as in the preceding implementation of a `MaxReg`$_k$ object. In addition, we insert a `MaxReg`$_h$ object for the second component at every node in the tree. The `MaxReg`$_h$ object at the root of the tree corresponds to the second component, while the copies at the rest of the nodes are used for coordination.

Formally, our implementation of a `MaxArray`$_{k\times h}$ object r is recursive. When $k = 1$, we use a single `MaxReg`$_h$ object, r.second. `MaxScan`(r) returns $(0, x)$, where x is the result of performing `ReadMax`$(r.\mathsf{second})$. `MaxUpdate1`(r, v) performs `WriteMax` on this object. `MaxUpdate0`(r, v) does nothing.

When $k > 1$, r consists of a `MaxArray`$_{m\times h}$ object r.left, a `MaxArray`$_{(k-m)\times h}$ object r.right, a binary register r.switch, and a `MaxReg`$_h$ object r.second. In the simplest case, $m = \lceil k/2 \rceil$, but the value of m can be adjusted, as with max registers, to make the cost of max array operations depend on the stored values.

Pseudocode is presented in Algorithm 2.

To perform `MaxUpdate0`(r, v), a process uses the algorithm for `WriteMax`, ignoring `MaxReg`$_h$ objects. To perform `MaxUpdate1`(r, v), a process simply performs `WriteMax` on the `MaxReg`$_h$ object at the root of the tree, ignoring the rest of the `MaxReg`$_h$ objects at other nodes of the tree.

The `MaxScan` operation obtains the value of the first component by traversing a path from the root to a leaf, as in `ReadMax`. The second component is obtained from the `MaxReg`$_h$ object at this leaf. A subtle helping mechanism propagates values of the second component down the path in the tree, while it is being traversed. Specifically, a process performing

Algorithm 2 An implementation of a $\texttt{MaxArray}_{k\times h}$ object for $k > 1$

```
Shared data  for a $\texttt{MaxArray}_{k\times h}$ object $r$:
    switch: a 1-bit multi-writer register, initially 0
    left: a $\texttt{MaxArray}_{m\times h}$ object, where $m$ is typically $\lceil k/2 \rceil$,
          initially (0,0)
    right: a $\texttt{MaxArray}_{(k-m)\times h}$ object, initially (0,0)
    second: a $\texttt{MaxReg}_h$ object, initially 0

1:  MaxUpdate0($r, v$):           // write to the first component  of a $\texttt{MaxArray}_{k\times h}$ object $r$
2:    if $v < m$
3:      if $r$.switch $= 0$
4:        MaxUpdate0($r$.left, $v$)
5:    else
6:      MaxUpdate0($r$.right, $v - m$)
7:      $r$.switch $\leftarrow 1$

8:  MaxUpdate1($r, v$):           // write to the second component  of a $\texttt{MaxArray}_{k\times h}$ object $r$
9:    WriteMax($r$.second, $v$)

10: MaxScan($r$):                                          //  for a $\texttt{MaxArray}_{k\times h}$ object $r$
11:   $x \leftarrow$ ReadMax($r$.second)
12:   if $r$.switch $= 0$
13:     WriteMax($r$.left.second, $x$)
14:     return MaxScan($r$.left)
15:   else
16:     $x \leftarrow$ ReadMax($r$.second)
17:     WriteMax($r$.right.second, $x$)
18:     return MaxScan($r$.right) + $(m, 0)$
```

`MaxScan`(r) begins by performing `ReadMax` on the $\texttt{MaxReg}_h$ object at the root of the tree. If the switch bit at the root of the tree is 0, it updates the $\texttt{MaxReg}_h$ object at the left child of the root with the value it obtained from the $\texttt{MaxReg}_h$ object at the root and recursively performs `MaxScan` on the left subtree. If the bit at the root of the tree is 1, it repeats the `ReadMax` on the $\texttt{MaxReg}_h$ object at the root of the tree and updates the $\texttt{MaxReg}_h$ object at the right child of the root with the value it receives. In this case, it then recursively performs `MaxScan` on the right subtree and adds m to the first component of the result. It is important that, at each internal node, the `ReadMax` of second is performed before switch is read. Together with the fact that the value of second is nondecreasing, this ensures that the result of the `ReadMax` by a process that goes to the left subtree is never larger than the result of the second `ReadMax` by a process that goes to the right subtree.

Note that a $\texttt{MaxArray}_{k\times 1}$ object is essentially a $\texttt{MaxReg}_k$ object: `WriteMax` is identical to `MaxUpdate0` and `ReadMax`can be obtained from `MaxScan` by deleting lines 11, 13, 16, and 17 and changing $(m, 0)$ to m on line 18.

3.1. Linearizability of Algorithm 2

We show that our implementation is linearizable. We do this by showing that, in any execution, the pairs returned by `MaxScan`(r) operations are comparable under componen-twise $\leq$ and use this total ordering to linearize these operations. Then we linearize the `MaxUpdate0`(r, v) and `MaxUpdate1`(r, v) operations in a consistent manner before, after, and between them. We begin with some technical lemmas.

LEMMA 3.1. *For any execution, if v is the value of the local variable x the first time* $\texttt{WriteMax}(r.\textit{right.second}, x)$ *is performed on line 17, then, at all points in the execution,* $r.\textit{left.second} \leq v$.

PROOF. Consider the $\texttt{MaxScan}(r)$ operation *op* that first performs $\texttt{WriteMax}(r.\mathsf{right.second}, x)$ on line 17. Prior to this step, *op* read $r.\mathsf{switch} = 1$ on Line 12 and then received some value v when it performed $\texttt{ReadMax}(r.\mathsf{second})$ on Line 16.

The value of $r.\mathsf{left.second}$ is initially 0 and is changed only when a $\texttt{MaxScan}(r)$ operation op' performs $\texttt{WriteMax}(r.\mathsf{left.second}, x)$ on Line 13 and $r.\mathsf{left.second} < x$. The value of x at this step is the value v' that op' obtained by performing $\texttt{ReadMax}(r.\mathsf{second})$ on Line 11, prior to reading $r.\mathsf{switch} = 0$ on Line 12.

Since $r.\mathsf{switch}$ only changes from 0 to 1, the $\texttt{ReadMax}(r.\mathsf{second})$ by op' on Line 11 occurred before the $\texttt{ReadMax}(r.\mathsf{second})$ by *op* on line 16. Since $r.\mathsf{second}$ is a max register, $v' \leq v$. Thus, at all points in the execution, $r.\mathsf{left.second} \leq v$. □

LEMMA 3.2. *Let v be the value of the second component of the pair returned by a* $\texttt{MaxScan}(r)$ *operation. Then the value of* $r.\textit{second}$ *is at least v when the operation returns.*

PROOF. By induction on the range, k, of the first component. If r is a $\texttt{MaxArray}_{1\times h}$ object, then the second component returned by a $\texttt{MaxScan}(r)$ operation is the result of $\texttt{ReadMax}(r.\mathsf{second})$, which is the value of $r.\mathsf{second}$.

Now let r be a $\texttt{MaxArray}_{k\times h}$ object, where $k > 1$. Suppose the claim is true for $r.\mathsf{left}$ and $r.\mathsf{right}$.

There are two cases. If $r.\mathsf{switch} = 0$, then the second component of the pair returned by a $\texttt{MaxScan}(r)$ operation on Line 14 is the second component of the pair returned by $\texttt{MaxScan}(r.\mathsf{left})$, which, by the induction hypothesis, is at most the value of $r.\mathsf{left.second}$. Otherwise, $r.\mathsf{switch} = 1$ and the second component of the pair returned by a $\texttt{MaxScan}(r)$ operation on Line 18 is the second component of the pair returned by $\texttt{MaxScan}(r.\mathsf{right})$, which, by the induction hypothesis, is at most the value of $r.\mathsf{right.second}$.

Whenever $\texttt{WriteMax}(r.\mathsf{left.second}, x)$ is performed on Line 13 or $\texttt{WriteMax}(r.\mathsf{right.second}, x)$ is performed on Line 17, the value of x is the result of a preceding $\texttt{ReadMax}(r.\mathsf{second})$ operation. Since $r.\mathsf{second}$ is a max register, its value never decreases, so $r.\mathsf{left.second}, r.\mathsf{right.second} \leq r.\mathsf{second}$. □

LEMMA 3.3. *Let v be the value of the second component of the pair returned by a* $\texttt{MaxScan}(r)$ *operation. Then the value of* $r.\textit{second}$ *is at most v when the operation is invoked.*

PROOF. By induction on the range, k, of the first component. If $k = 1$, and r is a $\texttt{MaxArray}_{1\times h}$ object, then the second component returned by a $\texttt{MaxScan}(r)$ operation is the result of $\texttt{ReadMax}(r.\mathsf{second})$. Then the claim follows from the fact that the value of the $\texttt{MaxReg}_h$ object $r.\mathsf{second}$ does not decrease.

Now let r be a $\texttt{MaxArray}_{k\times h}$ object, where $k > 1$. Suppose the claim is true for $r.\mathsf{left}$ and $r.\mathsf{right}$. Let v' be the value of $r.\mathsf{second}$ when a $\texttt{MaxScan}(r)$ operation op' is invoked. Then the value of x immediately after op' performs $\texttt{ReadMax}(r.\mathsf{second})$ on Line 11 is at least v'.

If op' performs $\texttt{WriteMax}(r.\mathsf{left.second}, x)$ on Line 13, then the value of $r.\mathsf{left.second}$ will be at least v' when op' invokes $\texttt{MaxScan}(r.\mathsf{left})$ on Line 14. Then, by the induction hypothesis, the second component of the pair returned by this operation (and, hence by $\texttt{MaxScan}(r)$) is at least v'.

Otherwise, on Line 16, op' sets x to the result of $\texttt{ReadMax}(r.\mathsf{second})$, which is still at least v'. Then op' performs $\texttt{WriteMax}(r.\mathsf{right.second}, x)$ on Line 17. Hence, the value of $r.\mathsf{right.second}$ will be at least v' when op' invokes $\texttt{MaxScan}(r.\mathsf{right})$ on Line 18. By the induction hypothesis, the second component of the pair returned by this operation (and, hence by $\texttt{MaxScan}(r)$) is at least v'. □

THEOREM 3.4. *The* $\texttt{MaxArray}_{k\times h}$ *implementation in Algorithm 2 is linearizable.*

PROOF. By induction on the range, k, of the first component. The linearizability of the $\texttt{MaxArray}_{1\times h}$ implementation follows immediately from the linearizability of the $\texttt{MaxReg}_h$ object that represents it.

Now let $k > 1$. Suppose that $1 \le m < k$, $r.\mathsf{left}$ is a linearizable $\texttt{MaxArray}_{m\times h}$ object, and $r.\mathsf{right}$ is a linearizable $\texttt{MaxArray}_{(k-m)\times h}$ object. We will show that r is a linearizable $\texttt{MaxArray}_{k\times h}$ object.

Consider any execution and let (x_0, x_1) and (x_0', x_1') be the pairs returned by two $\texttt{MaxScan}(r)$ operations *op* and *op*′. If both are the result of $\texttt{MaxScan}(r.\mathsf{left})$ on Line 14, then, by the induction hypothesis, they can be ordered in a consistent manner. The same is true if both are $(m, 0)$ plus the result of $\texttt{MaxScan}(r.\mathsf{right})$ on Line 18. Otherwise, one of the pairs, say (x_0, x_1), is the result of $\texttt{MaxScan}(r.\mathsf{left})$ on Line 14 and (x_0', x_1') is equal to $(m, 0)$ plus the result of $\texttt{MaxScan}(r.\mathsf{right})$ on Line 18.

The only instruction that updates the first component of $r.\mathsf{left}$ is $\texttt{MaxUpdate0}(r.\mathsf{left}, v)$ on Line 4. By the test on Line 2, $v < m$. Hence $x_0 < m$. Initially, $r.\mathsf{right} = 0$, so, by Line 18, $x_0' \ge m$. Thus $x_0 < x_0'$.

By Lemma 3.2, $x_1 \le r.\mathsf{left.second}$. Let v be the value of x the first time during the execution that $\texttt{WriteMax}(r.\mathsf{right.second}, x)$ is performed on Line 17. Then, by Lemma 3.1, $r.\mathsf{left.second} \le v$.

Since $r.\mathsf{right.second}$ is a $\texttt{MaxReg}_h$ object, which never decreases in value, $r.\mathsf{right.second} \ge v$ when *op*′ invokes Line 18. By Lemma 3.3, $x_1' \ge v$. Hence $x_1 \le x_1'$ and *op* is linearized before *op*′.

The only step performed by a $\texttt{MaxUpdate1}(r, v)$ operation is $\texttt{WriteMax}(r.\mathsf{second}, v)$ on Line 9. It follows from Lemmas 3.2 and 3.3 that it can be linearized among the $\texttt{MaxScan}(r)$ operations.

Provided $r.\mathsf{switch} = 0$, the $\texttt{MaxUpdate0}(r, v)$ operations with $v < m$ can be linearized where the $\texttt{MaxUpdate0}(r.\mathsf{left}, v)$ operations on Line 4 are linearized, which, by the induction hypothesis, can be linearized among the $\texttt{MaxScan}(r.\mathsf{left})$ operations. When $r.\mathsf{switch} = 1$, the $\texttt{MaxUpdate0}(r, v)$ operations with $v < m$ have no effect and they can be linearized when they return.

Similarly, each $\texttt{MaxUpdate0}(r, v)$ operation with $v \ge m$ performs a $\texttt{MaxUpdate0}(r.\mathsf{right}, v - m)$ operation on Line 6. By the induction hypothesis, these operations can be linearized among the $\texttt{MaxScan}(r.\mathsf{right})$ operations, each of which corresponds to a $\texttt{MaxScan}(r)$ operation that reads $r.\mathsf{switch} = 1$ on Line 12. The $\texttt{MaxScan}(r.\mathsf{right})$ operations all occur after $r.\mathsf{switch}$ becomes 1. Any $\texttt{MaxUpdate0}(r, v)$ operation with $v \ge m$ that performs Line 6 when $r.\mathsf{switch} = 0$ can be linearized when $r.\mathsf{switch}$ is changed to 1, which occurs at or before it performs Line 7. □

3.2. Step complexity

Our $\texttt{MaxArray}_{k\times h}$ implementation has step complexity that is polylogarithmic in h and k.

LEMMA 3.5. *For the* $\texttt{MaxArray}_{k\times h}$ *implementation in Algorithm 2, using a balanced tree for both the max array and the embedded max registers, the step complexity of* $\texttt{MaxUpdate0}$ *is* $O(\log k)$*, the step complexity of* $\texttt{MaxUpdate1}$ *is* $O(\log h)$*, and the step complexity of* $\texttt{MaxScan}$ *is* $O(\log k \log h)$*.*

PROOF. A $\texttt{MaxUpdate1}(r, v)$ operation performs one $\texttt{WriteMax}$ operation on a $\texttt{MaxReg}_h$ object, which has step complexity $O(\log h)$. A $\texttt{MaxUpdate0}(r, v)$ operation accesses the binary register $r.\mathsf{switch}$ once and performs one $\texttt{MaxUpdate0}(r', v')$ operation, where r' is a $\texttt{MaxArray}_{m\times h}$ object or a $\texttt{MaxArray}_{(k-m)\times h}$ object, and $m = \lceil k/2 \rceil$. If $T(k)$ is the step complexity of $\texttt{MaxUpdate0}(r, v)$ for a $\texttt{MaxArray}_{k\times h}$ object r, it follows that $T(1) = 0$ and $T(k) = T(\lceil k/2 \rceil) + 1$. Hence $T(k)$ is $O(\log k)$.

A $\texttt{MaxScan}(r)$ operation reads $r.\mathsf{switch}$ once, performs at most two $\texttt{ReadMax}(r.\mathsf{second})$ operations, each taking $O(\log h)$ steps, performs one $\texttt{MaxUpdate1}(r', v')$ operation, which also takes $O(\log h)$ steps, and performs one $\texttt{MaxScan}(r')$ operation, where $m = \lceil k/2 \rceil$ and r' is a $\texttt{MaxArray}_{m\times h}$ object or a $\texttt{MaxArray}_{(k-m)\times h}$ object. If $T_h(k)$ is the step complexity of a $\texttt{MaxScan}$ operation on a $\texttt{MaxArray}_{k\times h}$ object, then

$$T_h(k) = T_h(\lceil k/2 \rceil) + O(\log h).$$

Since $T_h(1)$ is $O(\log h)$, it follows that $T_h(k)$ is $O(\log k \log h)$. $\square$

The structure of the proof suggests an alternative interpretation of the cost: each $\texttt{MaxUpdate0}(v)$ operation traverses a path through the tree corresponding to the bits in v, performing $O(1)$ register and max register operations for each node; so does a $\texttt{MaxScan}$ operation returning $(-, v)$. Because the correctness of the max array implementation does not depend on the structure of the tree, we can replace the balanced tree with an unbalanced tree (as described by Aspnes et al. [2012a]) where each value v is encoded by a string of $O(\log v)$ bits. This requires traversing $O(\log v)$ nodes to perform a $\texttt{WriteMax1}(v)$ or $\texttt{ReadMax}$ returning $(-, v)$. By similarly using an unbalanced-tree implementation of a max register at each node, we get a cost of $O(\log v_1 \log v_2)$ for any operation on a max array that leaves it in state (v_1, v_2).

It is worth noting that, while this construction allows the cost of the max array to adapt to the size of the stored values, it is still necessary for the values to be bounded in order to avoid starvation by slow readers. Later (in Section 6), we show that for the particular application of building snapshots, we can overcome this limitation by combining a fast (but limited-use) implementation of a snapshot based on bounded max registers with a standard, unlimited-use snapshot, obtaining a graceful increase in the cost of a snapshot over time that eventually converges to the cost of a standard snapshot.

It is also worth noting that by modifying the algorithm, it is possible to swap the step complexities of the $\texttt{MaxUpdate}$ and $\texttt{MaxScan}$ operations: instead of paying $O(\log b)$ for $\texttt{MaxUpdate}$ and $O(\log^2 b)$ for $\texttt{MaxScan}$ on a max array of size $b \times b$, we can pay $O(\log^2 b)$ for $\texttt{MaxUpdate}$ and only $O(\log b)$ for $\texttt{MaxScan}$. This may improve the complexity of algorithms that perform $\texttt{MaxScan}$ operations more often.

The basic idea is that the $\texttt{MaxUpdate}$ operations now do the work of coordinating the pairs. The new $\texttt{MaxArray}_{b\times b}$ object r' consists of r and a $\texttt{MaxReg}_{b^2}$ object, view, that holds the current view stored in r. We can use a max register to store an ordered pair in $\{0, b-1\} \times \{0, b-1\}$ by extending the component-wise partial order on $\{0, b-1\} \times \{0, b-1\}$ to a total order, for example, lexicographically. To perform $\texttt{MaxScan}(r')$, a process simply performs $\texttt{ReadMax}(\mathit{view})$ and is linearized at the same point. To perform $\texttt{MaxUpdate}i(r', v)$, a process performs $\texttt{MaxUpdate}i(r, v)$ followed by $v' \leftarrow \texttt{MaxScan}(r)$ and then $\texttt{WriteMax}(\mathit{view}, v')$. A $\texttt{MaxUpdate}i(r', v)$ operation is linearized at the first point following its invocation at which view contains a pair whose value is at least v in component i. This implies that any $\omega(\log b)$ lower bound on the step complexity of a 2-component max array has to allow a trade-off between the step complexities of the $\texttt{MaxScan}$ and $\texttt{MaxUpdate}$ operations.

For $c > 2$, our 2-component max array implementation easily extends to a c-component max array in a recursive manner, by having $r.\mathsf{second}$ be a $(c-1)$-component max array, instead of a max register, and having $r.\mathsf{left}$ and $r.\mathsf{right}$ be c-component max arrays, instead of 2-component max arrays. Then the complexity of $\texttt{MaxUpdate}i$ is $i + O(\log k_i)$, where k_i is the range of component i, and the complexity of $\texttt{MaxScan}$ is $O(\prod_{i=0}^{c-1} \log k_i)$.

4. SINGLE-WRITER SNAPSHOTS FROM 2-COMPONENT MAX ARRAYS

Recall that a single-writer snapshot has one component per process and only process i can update component i. We build a $(b-1)$-limited-use single-writer snapshot object from $\texttt{MaxArray}_{b\times b}$ objects, registers, and a $\texttt{MaxReg}_b$ object. An example of our implementation,

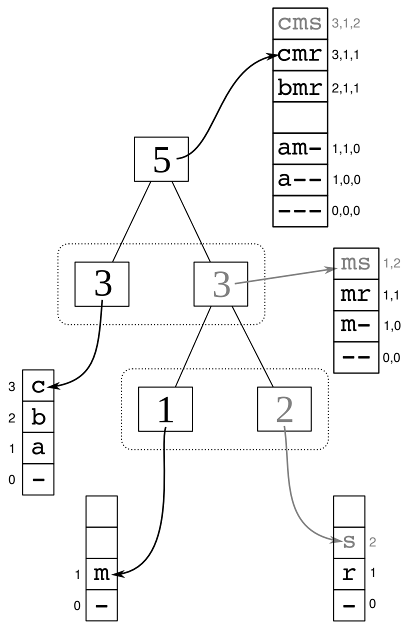

Fig. 2. A limited-use single-writer snapshot object shared by 3 processes. Grayed values correspond to an update operation in progress. Sequences outside the view arrays represent entries of the seq arrays from the proof of correctness. Not all array locations are shown.

with an Update$(3, s)$ operation in progress, is depicted in Figure 2. We use a strict, balanced, binary tree with n leaves, one for each process. Each leaf leaf_i has an array, leaf_i.view, of b registers, which contains the sequence of values that have been used for updates to component i. The initial value of component i is stored in leaf_i.view[0]. Each process i has a local variable, count_i, which is nondecreasing and persistent (i.e. its value is maintained across invocations, rather than re-initialized each invocation), that is used as a pointer (or index) into leaf_i.view.

Similarly, each internal node, u, has an array, view_u, of b registers, each containing a partial snapshot of the components corresponding to the leaves in the subtree rooted at u. The concatenation of leaf_i.view[0] for all leaves, leaf_i, in the subtree rooted at an internal node u, is stored in u.view[0]. For each (leaf or internal) node u in the tree, we keep track of the number of Update operations performed by processes whose leaves are in the subtree rooted at u. This number is used as a pointer (or index) into u.view. The pointer associated with the root is stored in the max register root.mr. The pointer associated each other node is stored in one component of the 2-component max array, ma, contained in its parent. This pointer is stored in component 0 if the node is the left child of its parent and in component 1 if the node is the right child of its parent. Having the pointers associated with each pair of siblings stored in a 2-component max array at their parent guarantees that pairs of views (and eventually the entire view) seen by different processes are comparable.

Pseudocode for our implementation is given in Algorithm 3.

To perform a Scan, a process simply takes the result of a ReadMax of the MaxReg_b stored at the root and uses it to index the array at the root. The step complexity of Scan is dominated by the step complexity of ReadMax, which is $O(\log b)$.

When a process updates its component of the snapshot object, it writes the new value to the first empty location in the array at its leaf and increases the value of the pointer held in its leaf to point to the location of this new value. Then it propagates this new value up the tree, combining partial snapshots. Specifically, at an internal node, a process performs a `MaxScan` of `ma`, the 2-component max array containing the pointers held at its children, and reads the array elements to which they point to obtain a partial snapshot. Its new pointer is the sum of the two pointers held at its children. The process stores the partial snapshot at the location in the array to which it points. The 2-component max arrays ensure linearizability. Since each `MaxScan` operation takes $O(\log^2 b)$ steps and the tree has $O(\log n)$ height, the step complexity of `Update` is $O(\log^2 b \log n)$.

The resulting algorithm is similar to the lattice agreement procedure of Inoue et al. [1994], except that we use `MaxScan` in place of double collects and we allow processes to update their values more than once.

The length of the array at a node is one greater than the total number of updates that can be performed by processes whose components are in the subtree rooted at that node. The pointer to this array is initially 0 and its maximum value is one less than the length of the array. Thus, if the arrays at a pair of siblings have length k and h, respectively, a $\texttt{MaxArray}_{k\times h}$ object can be used to store the pointers held by those nodes.

The size of each register in an array is the sum of the maximum sizes of the components in the partial snapshot it stores. This may be impractical, unless it is possible to represent the important information in a partial snapshot in a condensed manner. For example, a generalized counter can be implemented using a single-writer snapshot in which component i contains the sum of the values process i has added to the counter. Then each partial snapshot stored in a register (in an array) can be replaced by the sum of its components. The upper bound on the number of `Add` operations that can be performed by each process in the generalized counter is the number of times that process can update its component in the single-writer snapshot. This construction is similar to the f-array of Jayanti [2002] for efficient computation of aggregate functions (such as max and sum) of the elements of an array. Because the pointers are nondecreasing, we can use 2-component max arrays instead of the more powerful primitives used in that paper.

4.1. Linearizability of Algorithm 3

Now we show that our implementation is linearizable. A `Scan` operation is linearized when it performs ReadMax(root.mr) on Line 20. If $ptr = d$ when an `Update` operation performs Line 16 with $u =$ root, then the `Update` operation is linearized the first time any process performs `WriteMax`(root.mr, ptr) on Line 18 with $ptr \geq d$. The `Update` operation performs Line 18 with $ptr = d$ before it returns, so its linearization point occurs before it returns. The following lemma shows that its linearization point occurs after it begins.

LEMMA 4.1. *If d is the index stored at root.mr when an* `Update` *operation begins, then $ptr > d$ when the operation perform* `WriteMax`*(root.mr, ptr) on Line 18.*

PROOF. We also prove that, when an `Update` operation tries to update a pointer stored in a component of a `MaxArray` to ptr on Line 8 or 10, ptr is greater than the index stored at the component when the `Update` began.

The proof is by induction. The claim is true for a pointer held at a leaf. This is because only one process updates the pointer, it is intially 0, and count_i is incremented on Line 2 before it is assigned to ptr on Line 4.

Suppose the claim is true for a pointer held at a non-root node. The pointer held at its sibling never decreases. Since ptr is the sum of these two indices, the claim is true at the parent of this node, whether or not it is the root. □

Now, we prove that our linearization satisfies the specifications of a snapshot object.

For the purpose of the proof, we introduce an auxiliary array, $\mathsf{seq}[0..b-1]$, stored at each node. We imagine that, when Line 4 is performed, $\mathsf{leaf}_i.\mathsf{seq}[ptr] \leftarrow ptr$ is performed at the same time and, when Line 15 is performed, $u.\mathsf{seq}[ptr] \leftarrow u.\mathsf{left}.\mathsf{seq}[lptr] \cdot u.\mathsf{right}.\mathsf{seq}[rptr]$ is

Algorithm 3 An implementation of a $(b-1)$-limited-use single-writer snapshot object s, code for process i.

```
Shared data:
    $\mathsf{leaf}_j$, for $j \in \{0, \ldots, n-1\}$:
      the leaf node corresponding to process $j$, with field:
      $\mathsf{view}[0..b-1]$: an array, each of whose entries contains  a value of component $j$
          $\mathsf{view}[0]$ contains the initial value of component $j$
    $\mathsf{root}$: the root of the tree
    Each internal node has the fields:
      $\mathsf{left}$: the left child of the node in the tree
      $\mathsf{right}$: the right child of the node in the tree
      $\mathsf{view}[0..b-1]$: an array, each of whose entries contains a partial snapshot
           of the components with leaves in the subtree rooted at this node
           $\mathsf{view}[0]$ contains the concatenation of the initial values of these components
      $\mathsf{ma}$: a $\mathtt{MaxArray}_{b \times b}$ object, initially (0,0)
    The root also has the field:
    $\mathsf{mr}$: a $\mathtt{MaxReg}_b$ object, initially 0
    Each  non-root node also has the field:
      $\mathsf{parent}$: the parent of the node in the tree
Persistent local data: $\mathsf{count}_i$, initially 0.

1:  $\mathtt{Update}(s, i, v)$
2:    $\mathsf{count}_i \leftarrow \mathsf{count}_i + 1$
3:    $u \leftarrow \mathsf{leaf}_i$
4:    $ptr \leftarrow \mathsf{count}_i$
5:    $u.\mathsf{view}[ptr] \leftarrow v$
6:    repeat
7:      if $u = u.\mathsf{parent.left}$
8:        $\mathtt{MaxUpdate0}(u.\mathsf{parent.ma}, ptr)$
9:      if $u = u.\mathsf{parent.right}$
10:       $\mathtt{MaxUpdate1}(u.\mathsf{parent.ma}, ptr)$
11:     $u \leftarrow u.\mathsf{parent}$
12:     $(lptr, rptr) \leftarrow \mathtt{MaxScan}(u.\mathsf{ma})$
13:     $lview \leftarrow u.\mathsf{left.view}[lptr]$
14:     $rview \leftarrow u.\mathsf{right.view}[rptr]$
15:     $ptr \leftarrow lptr + rptr$                  // concatenate the views from the left and right
16:     $u.\mathsf{view}[ptr] \leftarrow lview \cdot rview$
17:   until $u = \mathsf{root}$
18:   $\mathtt{WriteMax}(\mathsf{root.mr}, ptr)$

19: $\mathtt{Scan}(s)$
20:   $ptr \leftarrow \mathtt{ReadMax}(\mathsf{root.mr})$
21:   return $\mathsf{root.view}[ptr]$
```

performed at the same time. Thus, each element of $u.\mathsf{seq}$ is a sequence of pointers, one into the array at each leaf of the subtree rooted at u. The following invariants are maintained:

— ptr is the sum of the elements in the sequence $u.\mathsf{seq}[ptr]$,
— if $ptr \leq ptr'$, then each component of $u.\mathsf{seq}[ptr]$ is less than or equal to the corresponding component of $u.\mathsf{seq}[ptr']$, and
— the j -th component of $u.\mathsf{view}[ptr]$ is equal to the element of view in the j -th leaf of the subtree rooted at node u pointed to by the j -th component of $u.\mathsf{seq}[ptr]$, i.e. $(u.\mathsf{view}[ptr])_j = \ell.\mathsf{view}[(u.\mathsf{seq}[ptr])_j]$, where ℓ is the j -th leaf of the subtree rooted at node u.

The second of these follows inductively from Line 12 and the fact that $u.\mathsf{ma}$ is a linearizable max array.

Consider an `Update` operation by process i that is linearized when process j performs Line 18. Suppose that $ptr = c$ when process j performs Line 18 and suppose that $ptr = d$ when the `Update` operation by process i performs Line 18. By the definition of the linearization points, $c \geq d$. Hence $(\mathsf{root.seq}[c])_i \geq (\mathsf{root.seq}[d])_i$. Only process i modifies the pointer at leaf_i (setting it to count_i) and its operation is linearized before it returns, so $(\mathsf{root.seq}[c])_i \leq \mathsf{count}_i \leq (\mathsf{root.seq}[d])_i$. Therefore $(\mathsf{root.seq}[c])_i = (\mathsf{root.seq}[d])_i = \mathsf{count}_i$. Similarly, any other `Update` operation that is linearized after this `Update` operation by process i is linearized, but before any other `Update` operation by process i is linearized, has $(\mathsf{root.seq}[ptr])_i = \mathsf{count}_i$ when it performs Line 18.

Consider any linearized `Scan` operation op. Suppose that $\mathsf{root.seq}[ptr] = (f_0, \ldots, f_{n-1})$ when it performs Line 20. Then $\mathsf{root.view}[f] = (v_0, \ldots, v_{n-1})$ is the view it returns, where $f = f_0 + \cdots + f_{n-1}$ and $v_j = \mathsf{leaf}_j.\mathsf{view}[f_j]$ for $j = 0, \ldots, n-1$. We need to show that v_j is the value written by process j in its last `Update` operation, op_j, linearized before op. Suppose that $ptr = c$ when op_j is linearized. From the preceding paragraph, it follows that $(\mathsf{root.seq}[f])_j = (\mathsf{root.seq}[c])_j = \mathsf{count}_j$. Since every `Update` by process j sets count_j to a new value on Line 2, op_j updated component j with value v_j in $\mathsf{leaf}_j.\mathsf{view}[\mathsf{count}_j]$. Similarly, if there is no `Update` operation by process j that is linearized before op, $\mathsf{count}_j = 0$ and $v_j = \mathsf{leaf}_j.\mathsf{view}[0]$ is the initial value of component j.

Thus, we have proved:

THEOREM 4.2. *The $(b-1)$-limited-use single-writer snapshot implementation in Algorithm 3 is linearizable.*

4.2. Step Complexity

LEMMA 4.3. *For the $(b-1)$-limited-use single-writer snapshot implementation in Algorithm 3, the step complexity of* `Scan` *is $O(\log b)$ and the step complexity of* `Update` *is $O(\log^2 b \log n)$.*

PROOF. A `Scan` operation performs one `ReadMax` on a $\mathtt{MaxReg}_b$ object and reads one entry from the array $\mathsf{root.view}$. Hence it has step complexity $O(\log b)$.

An `Update` operation performs at most $\lceil \log_2 n \rceil$ iterations, one for each ancestor of leaf_i. In each iteration, there is one `MaxUpdate` operation and one `MaxScan` operation applied to a $\mathtt{MaxArray}_{b \times b}$ object and a constant number of accesses to entries of view arrays. Finally, one `WriteMax` operation is performed on the $\mathtt{MaxReg}_b$ at root. This implies the claimed step complexity of $O(\log^2 b \log n)$. □

For simplicity, we have been assuming that the max array at each node has range b, which gives a fixed cost for each max array operation. Because the correctness of the snapshot does not depend on the implementation of the max array, using unbalanced max arrays instead gives $O(\log^2 s \log n)$ cost for each snapshot operation, where s is the number of updates that are linearized before the operation.

Many objects can be implemented from single-writer snapshot objects so that each of their operations uses only $O(1)$ snapshot operations. In addition to generalized counters (and counters), this is true for any object that only supports read operations, which do not change the state of the object, and update operations, which do not return a value, provided that each pair of update operations either commutes or one overwrites the other [Anderson and Moir 1993; Aspnes and Herlihy 1990]. Combining such an implementation with our implementation of a single-writer snapshot object gives an implementation of the object from registers in which each operation has $O(\log^3 n)$ step complexity, provided only a polynomial number of operations are performed.

5. MULTI-WRITER SNAPSHOTS

The previous section considered a single-writer snapshot object, in which each component can only be updated by a single process. Here, we extend this to implement a c-component multi-writer snapshot object, s', where each component can be updated by every process. This is done by using a single-writer snapshot object, s, and having each process record its own updates to each multi-writer component along with a *timestamp*. When these records are scanned, the value for each multi-writer component is the value written with the largest timestamp. The local function $\texttt{CombineScan}(c, \mathit{view})$ takes a view resulting from a scan and returns a vector of c pairs, one for each component j, containing the largest timestamp that any process has used to update component j, together with its associated value. Pseudocode for our implementation is given in Algorithm 4.

We use a $(b-1)$-limited-use max register to implement the generation of a timestamp. The step complexity to perform $\texttt{GetTS}$ is $O(\log b)$. To ensure that two different processes don't ever get the same timestamp, we multiply the result by n and then add the process id. Equivalently, we could append the process id to the end of the timestamp. Both approaches enable the process that generated a specific timestamp to be easily computed from the value of the timestamp (by taking the remainder when dividing by n or $2^{\lceil \log_2 n \rceil}$, respectively). This timestamp implementation is linearizable, with $\texttt{GetTS}$ operations linearized in increasing order of the timestamps they return.

THEOREM 5.1. *The $(b-1)$-limited-use multi-writer snapshot implementation in Algorithm 4 is linearizable. The step complexity of* $\texttt{MW-Scan}(s)$ *is* $O(\log b)$ *and the step complexity of* $\texttt{MW-Update}(s, j, v)$ *is* $O(\log^2 b \log n)$.

PROOF. The step complexity of $\texttt{MW-Scan}(s')$ is the same as that of $\texttt{Scan}(\mathsf{s})$, and the step complexity of $\texttt{MW-Update}(s', j, v)$ is the sum of the step complexities of $\texttt{GetTS}(\mathsf{timestamp})$ and $\texttt{Update}(\mathsf{s}, i, \mathsf{record}_i)$.

We linearize $\texttt{MW-Scan}(s')$ when it performs $\texttt{Scan}(\mathsf{s})$ on Line 5. Consider an instance *op* of $\texttt{MW-Update}(s', j, v)$ by process i that uses timestamp t. If no other process i' performs an $\texttt{Update}$ on Line 3 with $\mathsf{record}_{i'}[j].\mathsf{ts} > t$ before *op* performs Line 3, we linearize *op* when it performs Line 3. Otherwise, we linearize *op* at the first such step by any process i'. If multiple $\texttt{MW-Update}$ operations are linearized at the same step, they are ordered by their timestamps. Since $\texttt{GetTS}$ operations can be linearized in increasing order of the timestamps they return, *op* is always linearized after it performs Line 2. Since every operation is linearized after it begins and before it returns, the order of non-overlapping operations is preserved.

Note that, for each component j, if an instance of $\texttt{MW-Update}(s', j, v)$ with timestamp t completes before an instance of $\texttt{MW-Update}(s', j, v')$ with timestamp t' begins, then $t < t'$. Our choice of linearization points ensures that $\texttt{MW-Update}$ operations on component j are linearized in order of their timestamps. Since $\texttt{MW-Update}$ operations are linearized at $\texttt{Update}$ steps, it follows that the largest timestamp for component j in the view produced by a $\texttt{Scan}$ was the result of the $\texttt{Update}$ performed by the latest $\texttt{MW-Update}$ operation on component j linearized before that $\texttt{Scan}$. Thus, if $(v_0, \ldots, v_{c-1})$ is the view returned by an instance *op* of

Algorithm 4 An implementation of a $(b-1)$-limited-use c-component multi-writer snapshot object s', code for process i.

```
Shared data:
    s: a single-writer snapshot object,
        each component is an array of $c$ pairs (val,ts),
        each pair is initialized to $(-, 0)$
     timestamp: a $(b-1)$-limited-use max register
Persistent local data:
    record$_i[0..c-1]$: an array of pairs (val,ts),
        each is initialized to $(-, 0)$

1:  MW-Update$(s', j, v)$
2:      record$_i[j] \leftarrow (v,$ GetTS(timestamp)$)$
3:      Update(s, $i$, record$_i$)

4:  MW-Scan$(s')$
5:      $result \leftarrow$ CombineScan($c$, Scan(s))
6:      return$(result_0$.val$, \ldots, result_{c-1}$.val$)$

7:  CombineScan$(c, view)$
8:      for $0 \le j < c$ do                    // find pair with largest timestamp for component $j$
9:          $k \leftarrow \max\{view_k[j]$.ts $\mid 0 \le k < n\}$ rem $n$
10:         $result_j \leftarrow view_k[j]$
11:     return $result$

12: GetTS(timestamp)
13:     $t \leftarrow 1 +$ ReadMax(timestamp)
14:     WriteMax(timestamp, $t$)
15:     return$(nt + i)$
```

MW-Scan(s'), then, for each component j, the last MW-Update(s', j, v) operation linearized before *op* has $v = v_j$. Therefore, the linearization satisfies the specifications of a multi-writer snapshot object. □

6. UNLIMITED-USE SNAPSHOTS

We can build an unlimited-use snapshot object that behaves like the limited-use snapshot object as long as its operations are not too expensive, and falls back to a snapshot object with linear step complexity otherwise. The construction uses a switch bit to choose between the two objects. As long as the switch bit is 0, all operations use the limited-use snapshot object. Once the switch is set to 1, new operations use the linear-complexity snapshot object, while first reading the limited-use snapshot object and copying any old values from it. The switch gets set by `Update` operations when they observe that the number of `Update` operations that have been performed has reached some threshold T, which will be specified as part of the complexity analysis.

Algorithm 5 is an implementation of an unlimited-use single-writer snapshot object. It uses a $(T + n - 1)$-limited-use single-writer snapshot object, `LimitedSnapshot`, and an unlimited-use single-writer snapshot object, `LinearSnapshot`, with linear step complexity. The code uses the number of `Update` operations that have been performed on `LimitedSnapshot` to decide when to switch. This can be obtained from the `MaxReg` at its root. The switch is changed from 0 to 1 when this quantity reaches T.

Algorithm 5 An implementation of an unlimited-use single-writer snapshot object, s, code for process i

```
Shared data:
    switch: a 1-bit multi-writer register, initially 0
    LimitedSnapshot: a $(T + n - 1)$-limited-use single-writer snapshot object
        each component is a pair (val, ts), initialized to $(-, 0)$
    LinearSnapshot: a snapshot object with linear step complexity
        each component is an array of $n$ pairs (val, ts), initialized to $(-, 0)$
Persistent local data:
    $\mathsf{count}_i$, initially 0
    $\mathsf{value}_i$, initially $-$

1:  Update($s, i, v$):
2:      $\mathsf{count}_i \leftarrow \mathsf{count}_i + 1$
3:      $\mathsf{value}_i \leftarrow v$
4:      if switch
5:          $view \leftarrow$ Scan(LimitedSnapshot)
6:          $view_i \leftarrow (v, \mathsf{count}_i)$
7:          Update(LinearSnapshot, $i$, $view$)
8:      else
9:          Update(LimitedSnapshot, $i$, $(v, \mathsf{count}_i)$)
10:         if at least $T$ Updates have been performed then switch $\leftarrow 1$
11:     Scan($s$)

12: Scan($s$):
13:     $view \leftarrow$ Scan(LimitedSnapshot)
14:     if switch
15:         $view \leftarrow$ Scan(LimitedSnapshot)
16:         $view_i \leftarrow (\mathsf{value}_i, \mathsf{count}_i)$
17:         Update(LinearSnapshot, $i$, $view$)
18:         $views \leftarrow$ CombineScan($n$, $Scan$(LinearSnapshot))
19:     return $(view_0.\mathsf{val}, \ldots, view_{n-1}.\mathsf{val})$
```

When the switch is 1, a process i performing Update(s, i, v) first scans LimitedSnapshot and propagates the information from all components $j \neq i$, together with its new value, v, and sequence number for component i, to LinearSnapshot. This is used to ensure that updates performed on LimitedSnapshot just prior to when the switch changed will not be lost. A process that performs Scan(s) and sees that the switch is 1 also scans LimitedSnapshot and propagates the information from all components $j \neq i$, together with the last value and sequence number it used for updating component i of s, to LinearSnapshot. Then it scans LinearSnapshot and uses the information it receives to determine the most recent value for each component.

The i -th component of LimitedSnapshot stores the pair $(\mathsf{value}_i, \mathsf{count}_i)$ containing the last value with which process i performed an Update and a sequence number, which is the number of Update operations process i has performed. Since count_i is nondecreasing, the sequence number stored in the i -th component fo LimitedSnapshot is also nondecreasing. When any process performs Scan(LimitedSnapshot), we say that it *sees* this pair in component i.

The i -th component of LinearSnapshot stores an array of such pairs. The i -th entry of this array is $(\mathsf{value}_i, \mathsf{count}_i)$. The j -th entry of this array, for $j \neq i$, is the pair process

i saw in component j when it last performed `Scan`(`LimitedSnapshot`). If $Views$ is the result when a process performs `Scan`(`LinearSnapshot`), we say that the process *sees* the pair (v, c) in component i if $Views_j[i] = (v, c)$ for some process j.

OBSERVATION 1. *The sequence number in each array entry j in each component i of* `LinearSnapshot` *is nondecreasing.*

PROOF. Component i of `LinearSnapshot` changes when process i performs `Update`(`LinearSnapshot`, i, $view$) on Line 7 or 17. Prior to this, process i sets $view$ to the result of `Scan`(`LimitedSnapshot`) on Line 5 or 15 and then sets $view_i.\mathsf{ts}$ to count_i on Line 6 or 16. Since count_i is nondecreasing, entry i in component i is nondecreasing. If $j \neq i$, then entry j in component i is also nondecreasing, because component j of `LimitedSnapshot` is nondecreasing. □

LEMMA 6.1. *The unlimited-use single-writer snapshot implementation in Algorithm 5 is linearizable.*

PROOF. Consider any execution. We will first give linearization points for all `Scan` and `Update` operations peformed on the unlimited use snapshot object s. Then we will show that, for each component, i, each `Scan`(s) operation returns the value of the last `Update` operation on s by process i that was linearized before it. Since `LimitedSnapshot` and `LinearSnapshot` are linearizable objects, we treat all `Scan` and `Update` operations peformed on them as if they occur atomically.

All `Scan`(s) operations that see $\mathsf{switch} = 1$ are linearized when they perform `Scan`(`LinearSnapshot`) on Line 18. All `Scan`(s) operations that return on Line 24 are linearized when they perform `Scan`(`LimitedSnapshot`) on Line 13.

Now consider an `Update`(s, i, v) operation U_i by process i. Suppose it assigns value c to count_i on Line 2. If $\mathsf{switch} = 0$ on Line 4, it performs `Update`(`LimitedSnapshot`, i, (v, c)) on Line 9, followed by `Scan`(s) on Line 11. The latter begins with `Scan`(`LimitedSnapshot`) on Line 13 in which process i sees (v, c) in component i. This is because it is the only process that can update component i of `LimitedSnapshot`.

If $\mathsf{switch} = 1$ on Line 4, U_i performs `Update`(`LinearSnapshot`, i, $view$) with $view_i = (v, c)$ on Line 7, instead. Next it performs `Scan`(s) on Line 11, which performs `Update`(`LinearSnapshot`, i, $view$) with $view_i = (v, c)$ followed by `Scan`(`LinearSnapshot`) on Line 18. Note that process i sees (v, c) in component i, since it is the only process that can update component i of `LinearSnapshot`.

In both cases, we linearize U_i immediately before the first linearized `Scan`(s) operation (by any process) that sees (v, c) in component i. Note that this happens before U_i completes. Also, it cannot happen until after U_i performs an `Update` operation on `LimitedSnapshot` or `LinearSnapshot` on Line 7 or Line 9. Therefore, each operation is linearized within its execution interval and the order of non-overlapping operations is preserved.

Let S be a `Scan`(s) operation that returns value v in component i. Then, from the code, it saw (v, c) in component i during a `Scan`(`LimitedSnapshot`) on Line 13 or a `Scan`(`LinearSnapshot`) on Line 18, for some sequence number c. By definition, the `Update`(s, i, v) operation , U_i, with $\mathsf{count}_i = c$ is linearized before S.

First, suppose that S is linearized when it performed Line 13. Then $\mathsf{switch} = 0$ when it performed Line 14. Consider any `Update` operation U_i' on s by process i that is linearized before S. It is linearized after performing Line 7 or Line 9, which is after it read switch on Line 4. Thus $\mathsf{switch} = 0$ when U_i' performed Line 4. Hence, U_i' called `Update` on `LimitedSnapshot` on Line 9. When S performs `Scan`(`LimitedSnapshot`) on Line 13, the pair , (v, c), it sees in component i is from the last preceding `Update` operation on `LimitedSnapshot` by process i, which is performed during the last preceding `Update` operation on s by process i. Hence, U_i is the last `Update` operation linearized before S that is performed on s by process i.

Now suppose that S is linearized when it performs $\texttt{Scan}(\texttt{LinearSnapshot})$ on Line 18. If S is performed by process i, then, on Line 17, it performed $\texttt{Update}(\texttt{LinearSnapshot}, i, view)$ with $view_i = (\mathsf{value}_i, \mathsf{count}_i)$. Since component i has not been updated with a timestamp greater than count_i, S returns value_i, which is the value of the last Update operation linearized before S that is performed on s by process i.

Otherwise, S is performed by some other process $j \neq i$. Let U_i' be the last Update operation on s by process i that is linearized before S, let v' be the value with which it updates s, and let c' be the value U_i' assigns to count_i on Line 2. If U_i' performed $\texttt{Update}(\texttt{LinearSnapshot}, i, view)$ on Line 7, then S sees (v', c') in component i. This is because U_i' performed Line 7 before it was linearized and S was linearized when it performed $\texttt{Scan}(\texttt{LinearSnapshot})$ on Line 18, which is after U_i' was linearized. In this case, S returns v' in component i and $U_i' = U_i$.

If not, U_i' performed $\texttt{Update}(\texttt{LimitedSnapshot}, i, (v', c'))$ on Line 9 after reading $\mathsf{switch} = 0$ on Line 4. Thus, every Update operation on s by process i that occurred before U_i' read $\mathsf{switch} = 0$ on Line 4 and performed an Update to LimitedSnapshot on Line 9.

If S performed $\texttt{Scan}(\texttt{LimitedSnapshot})$ on Line 15 after U_i' performed Line 9, then S saw (v', c') in component i and performed $\texttt{Update}(\texttt{LinearSnapshot}, j, view)$ with $view_i = (v', c')$ on Line 17. After performing $\texttt{Scan}(\texttt{LinearSnapshot})$ on Line 18, $Views_j[i] = (v', c')$, since process j is the only process that can update component j of LinearSnapshot. Thus S sees (v', c') in component i and returns v' in component i. Therefore $U_i' = U_i$.

Finally, suppose that S performed $\texttt{Scan}(\texttt{LimitedSnapshot})$ on Line 15 before U_i' performed Line 9. Since U_i' is linearized before S, some $\texttt{Scan}(s)$, S', linearized at or before S saw (v', c') in component i. This happened when S' performed $\texttt{Scan}(\texttt{LimitedSnapshot})$ on Line 13 or Line 15 or when it performed $\texttt{Scan}(\texttt{LinearSnapshot})$ on Line 18.

If S' saw (v', c') when it performed $\texttt{Scan}(\texttt{LimitedSnapshot})$ on Line 13, U_i' must have previously performed $\texttt{Update}(\texttt{LimitedSnapshot})$ on Line 9. Since this occurred after S performed Line 15, which occurred after S read $\mathsf{switch} = 1$ on Line 14, S' read $\mathsf{switch} = 1$ on Line 14 at its next step. Then S' performed $\texttt{Scan}(\texttt{LimitedSnapshot})$ again on Line 15. Since all updates to component i of LimitedSnapshot are performed by process i using successively larger sequence numbers, the sequence number S' saw when it performed Line 15 is at least c'. It cannot be larger than c': Otherwise another update by process i would be linearized after U_i' and before S', and, hence, before S. This contradicts the choice of U_i'. Therefore, in this case, S' also saw (v', c') when it performed Line 15.

If S' saw (v', c') in component i when it performed $\texttt{Scan}(\texttt{LimitedSnapshot})$ on Line 15, it next performed an Update to LinearSnapshot on Line 17 with $view_i = (v', c')$ and, hence, saw (v', c') in component i when it performed $\texttt{Scan}(\texttt{LinearSnapshot})$ on Line 18. Thus, no matter when S' first saw (v', c') in component i, it saw (v', c') in component i, when it performed $\texttt{Scan}(\texttt{LinearSnapshot})$ on Line 18.

S cannot have seen any pair with larger sequence number in component i: Otherwise, another update by process i would be linearized between U_i' and S, contradicting the choice of U_i'. Since S and S' are linearized when they performed Line 18 and S' is linearized at or before S, it follows from Observation 1 that S also saw (v', c') in component i when it performed $\texttt{Scan}(\texttt{LinearSnapshot})$ on Line 18. Therefore, S returns v' in component i and $U_i' = U_i$.

In all cases, component i of the result of a Scan operation on s is the value of the last Update operation on s performed by process i that is linearized before the Scan. $\square$

The number of steps required by any operation on our unlimited-use single-writer snapshot object in Figure 5 depends on the choice of the threshold T.

THEOREM 6.2. *If $T = O(2^{\sqrt{n/\log n}})$, then the step complexity of every* Scan *and* Update *operation is $O(n)$. Each* Scan *operation has $O(\log(T + n))$ step complexity and each* Update *operation has $O(\log^2(T+n)\log n)$ step complexity, provided that it completes before T* `Update` *operations have begun.*

PROOF. When fewer than T `Update` operations have begun, fewer than T `Update` operations have been performed on `LimitedSnapshot`, so switch $= 0$. While switch $= 0$, each `Scan` operation just performs a `Scan` on the $(T+n-1)$-limited-use single-writer snapshot object `LimitedSnapshot`, which has $O(\log(T+n))$ step complexity. Furthermore, while switch $= 0$, each `Update` operation performs an `Update` and a `Scan` on `LimitedSnapshot`, plus a constant number of additional steps, so it has $O(\log^2(T+n)\log n)$ step complexity. Both of these are $O(n)$, since $T = O(2^{\sqrt{n/\log n}})$.

Each `Scan` and `Update` operation accesses `LimitedSnapshot` and `LinearSnapshot` at most a constant number of times, so it has $O(n)$ step complexity. □

Using balanced max arrays, Theorem 6.2 allows T to be tuned to optimize the trade-off between minimizing the cost of each operation before T is reached and delaying the time at which the algorithm switches to the more expensive $O(n)$-step snapshot. This sudden jump in complexity can be ameliorated to some extent by using unbalanced max arrays instead. In this case, after s updates, the cost of a max array operation is either $O(\log^2 s \log n)$ when $s \le T$ or $O(n)$ when $s > t$. Because of the upper bound on T, this cannot eliminate the discontinuity, but it can allow for adaptive behavior over an exponentially long sequence of updates before resorting to the linear-time backup object.

6.1. An Unlimited-use Multi-writer Snapshot Object

A combination of the constructions used in Algorithms 4 and 5 can be used to obtain an unlimited-use c-component multi-writer snapshot object with $O(n)$ step complexity that has substantially smaller step complexity when it has not been updated too many times. The idea is to have a switch choose between our limited-use single-writer snapshot implementation, `LimitedSnapshot`, and an unlimited-use single-writer snapshot implementation, `LinearSnapshot`, with $O(n)$ step complexity, both using the same timestamp object, implemented from a $(T+n-1)$-limited-use max register. Processes carry values from `LimitedSnapshot` to `LinearSnapshot` when the switch is on. However, if its last update to a component has a larger timestamp than those recorded in `LimitedSnapshot`, a process uses that instead. Pseudocode for our implementation is presented in Algorithm 6. The proof of correctness is similar to the proof for Algorithm 5.

7. DISCUSSION

This paper gives linearizable implementations of a snapshot object with $O(\log^3 n)$ step complexity, as long as the number of update operations is at most polynomial in the number of processes, n. This is an exponential improvement over the best previously known algorithms, which have step complexity linear in n. In addition, we showed how to combine our implementations with existing, unlimited-use implementations so that an unlimited number of update operations can be performed, with the step complexity growing gracefully from $O(\log^3 n)$ for the first polynomially-many operations, to the $O(n)$ complexity of previously-known snapshot implementations after exponentially-many operations.

The key component of our implementations is a new object, the bounded 2-component max array. Improved implementations of it would imply corresponding improvements to our snapshot implementations. Therefore, improved implementations or lower bounds on the tradeoff between the step complexities of `MaxScan` and `MaxUpdate` would be interesting.

At the end of Section 3, we briefly explained how to obtain a bounded c-component max array for $c > 2$. As with snapshots, an unbounded c-component max array implementation can be constructed from our bounded implementation and an unbounded implementation based on a single-writer snapshot object, by having processes carry values from the former

Algorithm 6 An implementation of an unlimited-use c-component multi-writer snapshot object, s, code for process i

```
Shared data:
    switch: a 1-bit multi-writer register, initially 0
    LimitedSnapshot: a $(T + n - 1)$-limited-use $n$-component single-writer snapshot object
      each component is an array of $c$ pairs (val, ts), initialized to $(-, 0)$
    LinearSnapshot: a single-writer snapshot object with linear step complexity
      each component is an array of $c$ pairs (val, ts), initialized to $(-, 0)$
    timestamp: a $(T + n - 1)$-limited-use max register
Persistent local data:
    record$[0..c - 1]$: an array of pairs (val,ts),
      each is initialized to $(-, 0)$

1:  MW-Update$(s, j, v)$:
2:    record$[j] \leftarrow (v,$ GetTS(timestamp)$)$
3:    if switch
4:      $view \leftarrow$ CombineScan$(c,$ Scan(LimitedSnapshot)$)$
5:      for $0 \le j < c$ do
6:        if record$[j]$.ts $> view_j$.ts then $view_j \leftarrow$ record$[j]$
7:      Update(LinearSnapshot$, i, view)$
8:    else
9:      Update(LimitedSnapshot$, i,$ record$)$
10:     if at least $T$ Update operations have been performed then switch $\leftarrow 1$
11:   MW-Scan$(s)$

12: MW-Scan$(s)$:
13:   $view \leftarrow$ CombineScan$(c,$ Scan(LimitedSnapshot)$)$
14:   if switch
15:     $view' \leftarrow$ CombineScan$(c,$ Scan(LimitedSnapshot)$)$
16:     $view \leftarrow$ CombineScan$(c,$ Scan(LinearSnapshot)$)$
17:     for $0 \le j < c$ do
18:       if $view'_j$.ts $> view_j$.ts then $view_j \leftarrow view'_j$
19:     Update(LinearSnapshot$, i, view)$
20:   return $(view_0$.val$, \ldots, view_{c-1}$.val$)$
```

implementation to the latter when the switch is on. Are more efficient implementations possible?

Our constructions use multi-writer registers. A very intriguing question is to extend them to obtain a snapshot object with $O(n)$ step complexity using only *single-writer* registers, improving on the $O(n \log n)$ best previously-known upper bound [Attiya and Rachman 1998].

Finally, like many previous implementations of snapshot objects, our implementations of snapshot objects use registers that can store $\Theta(n)$ values. It would be interesting to obtain implementations in which each register contains only one value (or a small constant number of values). This would require the result of a scan be represented implictly, rather than explictly.

REFERENCES

AFEK, Y., ATTIYA, H., DOLEV, D., GAFNI, E., MERRITT, M., AND SHAVIT, N. 1993. Atomic snapshots of shared memory. *J. ACM 40,* 4, 873–890.

ALISTARH, D., ASPNES, J., BENDER, M., GELASHVILI, R., AND GILBERT, S. 2014. Dynamic task allocation in asynchronous shared memory. In *Proceedings of the 25th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 416–435.

ANDERSON, J. H. 1993. Composite registers. *Distributed Computing 6,* 3, 141–154.

ANDERSON, J. H. AND MOIR, M. 1993. Towards a necessary and sufficient condition for wait-free synchronization. In *Proceedings of the 7th International Workshop on Distributed Algorithms (WDAG)*. Lecture Notes in Computer Science Series, vol. 725. Springer, 39–53.

ANDERSON, J. H. AND MOIR, M. 1995. Universal constructions for large objects. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG)*. Lecture Notes in Computer Science Series, vol. 972. Springer, 168–182.

ASPNES, J., ATTIYA, H., AND CENSOR-HILLEL, K. 2012a. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM 59,* 1, 2:1–2:24.

ASPNES, J., ATTIYA, H., CENSOR-HILLEL, K., AND HENDLER, D. 2012b. Lower bounds for resricted-use objects. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 172–181.

ASPNES, J. AND CENSOR, K. 2009. Approximate shared-memory counting despite a strong adversary. In *SODA '09: Proceedings of the Nineteenth Annual ACM -SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 441–450.

ASPNES, J. AND HERLIHY, M. 1990. Wait-free data structures in the asynchronous PRAM model. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 340–349.

ATTIYA, H. AND FOUREN, A. 2001. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput. 31,* 2, 642–664.

ATTIYA, H., LYNCH, N., AND SHAVIT, N. 1994. Are wait-free algorithms fast? *J. ACM 41,* 4, 725–763.

ATTIYA, H. AND RACHMAN, O. 1998. Atomic snapshots in $O(n \log n)$ operations. *SIAM J. Comput. 27,* 2, 319–340.

BOROWSKY, E. AND GAFNI, E. 1993. Immediate atomic snapshots and fast renaming. In *Proceedings of the 12th annual ACM symposium on Principles of Distributed Computing*. 41–51.

FICH, F. E. 2005. How hard is it to take a snapshot? In *Proceedings of 31st Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM)*. Lecture Notes in Computer Science Series, vol. 3381. Springer, 27–35.

HERLIHY, M. P. AND WING, J. M. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12,* 3, 463–492.

INOUE, M., MASUZAWA, T., CHEN, W., AND TOKURA, N. 1994. Linear-time snapshot using multi-writer multi-reader registers. In *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG)*. Lecture Notes in Computer Science Series, vol. 857. Springer, 130–140.

JAYANTI, P. 2002. f-arrays: implementation and applications. In *Proceedings of the 21st annual symposium on Principles of Distributed Computing (PODC)*. ACM, New York, NY, USA, 270–279.

JAYANTI, P. 2005. An optimal multi-writer snapshot algorithm. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC)*. 723–732.

JAYANTI, P. AND PETROVIC, S. 2005. Efficient wait-free implementation of multiword LL/SC variables. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, 59–68.

JAYANTI, P., TAN, K., AND TOUEG, S. 2000. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput. 30,* 2, 438–456.

RIANY, Y., SHAVIT, N., AND TOUITOU, D. 2001. Towards a practical snapshot algorithm. *Theor. Comput. Sci. 269,* 1-2, 163–201.