Faster than Optimal Snapshots (for a While)

Preliminary Version

James Aspnes[*]
Department of Computer Science, Yale University
aspnes@cs.yale.edu

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

Keren Censor-Hillel[‡]
Computer Science and Artificial Intelligence Lab, MIT
ckeren@csail.mit.edu

Faith Ellen[§]
Department of Computer Science, University of Toronto
faith@cs.toronto.edu

ABSTRACT

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.

Applications of this construction include an implementation of a limited-use generalized counter with polylogarithmic step complexity. This can be used, for example, to monitor the number of active processes, which is crucial to adaptive algorithms.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

[*]Supported in part by NSF grant CCF-0916389.

[†]Supported in part by the *Israel Science Foundation* (grant number 1227/10).

[‡]Supported by the Simons Postdoctoral Fellows Program.

[§]Supported in part by the Natural Science and Engineering Research Council of Canada.

PODC'12, July 16–18, 2012, Madeira, Portugal.

General Terms

Theory, Algorithms

Keywords

Concurrent objects, restricted-use objects, atomic snapshot, generalized counters

1. INTRODUCTION

Atomic snapshots [1] are fundamental data structures in shared memory computations. They allow processes to scan and update shared arrays so that the 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 the addition of arbitrary positive or negative integers. This is a very useful concurrent data structure: It can be used for keeping track of the number of participants in an algorithm, as is done for mutual exclusion [9], where processes join and leave the competition for the critical section. With atomic snapshots, each process can store its "contribution" (the sum of the values by which it has incremented and decremented) in its component. Using a scan, a process gets an instantaneous view of the contributions, which it sums to obtain the value of the counter.

The best previously-known algorithms for atomic snapshots using only reads and writes [7, 12] have step complexity that is linear in the number of processes n. For a long time, this was taken to be the inherent cost of atomic snapshots, in light of the linear lower bound proved by Jayanti, Tan, and Toueg [15].

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 reads and writes, assuming the number of increments is polynomial (in n) [4]. 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 $O(\log^3 n)$ step complexity, as long as the number of update operations that are performed is polynomial in n. Obtaining an implementation of a snapshot object with polylogarithmic step complexity using only reads and writes is particularly surprising, since all previous implementations had a process directly read a linear number of registers to perform an operation. Instead, 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, as is the case in many important applications. This implies implementations with polylogarithmic step complexity using only reads and writes for a wide variety of shared-memory objects, including generalized counters.

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 scan operation that returns the values of both components. The pairs (x_0, x_1) and (y_0, y_1) returned by different scans 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 first component at all levels of a tree of registers implementing the second component using the construction of [4].

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 sublinear time.

The reason our results do not contradict the linear lower bound [15] is because the proof uses executions that are exponentially long as a function of n. Specifically, it has been shown [5] 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.

Due to their importance, there have been other implementations of atomic snapshots, e.g., [2, 10, 16]. An interesting implementation of atomic snapshots using f-arrays takes one step for a scan and $O(\log n)$ steps for an update, but it uses LL/SC [13]. There are also atomic snapshot implementations using CAS [14, 18], which take one step for an update and $O(n)$ steps for a scan.

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. 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* [11] 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 r to v. $\texttt{Scan}(r)$ atomically reads the values of all m components. In a *single-writer snapshot*, 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* object of type $\texttt{MaxReg}_k$ is a max register whose values are restricted to $\{0, \ldots, k-1\}$; we say that it has *range* k.

A *2-component max array* consists of a pair of MaxReg objects, with an atomic operation that returns the values of both of them. Specifically, an object r of type $\texttt{MaxArray}_{k \times h}$ supports three linearizable operations: $\texttt{MaxUpdate0}(r, v)$, where $v \in \{0, \ldots, k-1\}$, $\texttt{MaxUpdate1}(r, v)$, where $v \in \{0, \ldots, h-1\}$, and $\texttt{MaxScan}(r)$, with the following properties:

- $\texttt{MaxUpdate0}(r, v)$ sets the value of the first component of r to v.
- $\texttt{MaxUpdate1}(r, v)$ sets the value of the second component of r to v.
- $\texttt{MaxScan}(r)$ returns the value of r, i.e. it returns a pair (v, v') such that v and v' are 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 MaxScan operations returning (u, u') and (v, v').

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.

3. IMPLEMENTING A 2-COMPONENT MAX ARRAY

We begin with the description of the implementation of a $\texttt{MaxReg}_k$ object from registers [4], 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.

A max register r with range k consists of a single bit register, $r.\mathsf{switch}$, and two smaller max registers, $r.\mathsf{left}$ with range $m = \lceil k/2 \rceil$ 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. The construction results in a tree of depth $\lceil \log_2 k \rceil$.

Pseudocode for this implementation of a $\texttt{MaxReg}_k$ object is presented in Algorithm 1.

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 $\texttt{ReadMax}(r_0)$ followed by $\texttt{ReadMax}(r_1)$. For example, consider the execution in Figure 1.

In this execution, the steps of two scanners, p and p', are interleaved with those of an updater q, such that p returns (0,1) and p' returns (1,0), which are incomparable. Thus, it is impossible to linearize both these operations with this naive implementation.

However, since the only possible values are 0 and 1, there is a correct implementation of $\texttt{MaxScan}(r)$ that is only slightly more complicated: If 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 always return (1,1) and be linearized at its last step. When a process obtains either (0,1) or (1,0), it can return the pair of values resulting from performing $\texttt{ReadMax}(r_0)$ and $\texttt{ReadMax}(r_1)$ 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* [1], and the operation can be linearized between the two collects.

More generally, if r is a $\texttt{MaxArray}_{k\times h}$ object, then $\texttt{MaxScan}(r)$ can be performed by repeatedly performing $\texttt{ReadMax}(r_0)$ followed by $\texttt{ReadMax}(r_1)$ 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 values can change $k + h$ times.

The challenge in implementing a significantly faster $\texttt{MaxArray}_{k\times h}$ object is to ensure that, in each execution, all pairs returned by the $\texttt{MaxScan}$ operations are comparable. Our approach is to make the $\texttt{MaxScan}$ operations be responsible for this coordination. For the first component, we use the same binary tree as in the preceding implementation of a $\texttt{MaxReg}_k$ object. In addition, we insert a $\texttt{MaxReg}_h$ object for the second component at every node in the tree. To perform $\texttt{MaxUpdate0}(r, v)$, a process uses the algorithm for $\texttt{WriteMax}$, ignoring these additional objects. To perform $\texttt{MaxUpdate1}(r, v)$, a process simply performs $\texttt{WriteMax}$ on the $\texttt{MaxReg}_h$ object at the root of the tree, ignoring the rest of the $\texttt{MaxReg}_h$ objects at other nodes of the tree.

The $\texttt{MaxScan}$ operation uses a subtle helping mechanism that propagates values of the second component down the path in the tree, while it is being traversed to obtain the value of the first component. Specifically, a process performing $\texttt{MaxScan}(r)$ begins by performing $\texttt{ReadMax}$ 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 $\texttt{MaxScan}$ on the left subtree. If the bit at the root of the tree is 1, it repeats the $\texttt{ReadMax}$ on the $\texttt{MaxReg}_h$ object at the root of the tree before updating the $\texttt{MaxReg}_h$ object at the right child of the root with the value it receives and then recursively performs $\texttt{MaxScan}$ on the right subtree (and adds m to the first component of the result). Because the value of the $\texttt{MaxReg}_h$ object is nondecreasing, the value returned by the second $\texttt{ReadMax}$ is guaranteed to be at least as large as the value returned by the $\texttt{ReadMax}$ to any process that goes to the left subtree.

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

When $k > 1$, r consists of a $\texttt{MaxArray}_{m\times h}$ object $r.\mathsf{left}$, where $m = \lceil k/2 \rceil$, a $\texttt{MaxArray}_{(k-m)\times h}$ object $r.\mathsf{right}$, a binary register $r.\mathsf{switch}$, and a $\texttt{MaxReg}_h$ object $r.\mathsf{second}$.

Pseudocode for these operations is presented in Algorithm 2.

Algorithm 1 An implementation of a $\texttt{MaxReg}_k$ object

Shared data:
  switch: a single bit multi-writer register, initially 0
  left: a $\texttt{MaxReg}_m$ object, where $m = \lceil k/2 \rceil$, initially 0,
  right: a $\texttt{MaxReg}_{k-m}$ 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 ← 1

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

3.1 Linearizability

We show that our implementation is linearizable. We do this by showing that, in any execution, the pairs returned by $\texttt{MaxScan}(r)$ operations are comparable under componentwise $\leq$ and use this total ordering to linearize these operations. Then we linearize the $\texttt{MaxUpdate0}(r, v)$ and $\texttt{MaxUpdate1}(r, v)$

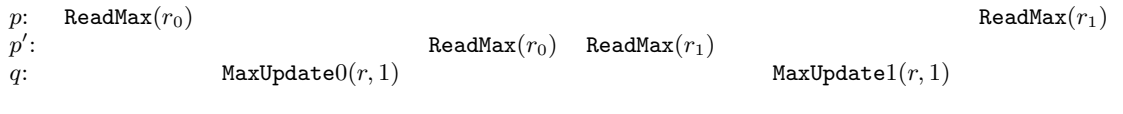

Figure 1: An execution of an incorrect implementation of a $\texttt{MaxArray}_{2\times 2}$ object

operations in a consistent manner before, after, and between them. We begin with some technical lemmas.

LEMMA 1. *For any execution, if v is the value of x the first time* ***WriteMax***$(r.\textsf{right.second}, x)$ *is performed on Line 18, then at all points in the execution, $r.\textsf{left.second} \leq v$.*

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

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

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

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

```
Shared data:
    switch: a 1-bit multi-writer register, initially 0
    left: a MaxArray_{m×h} object, where m = ⌈k/2⌉,
          initially (0,0)
    right: a MaxArray_{(k−m)×h} object, initially (0,0)
    second: a MaxReg_h object, initially 0

1:  MaxUpdate0(r, v):          // write to the first component
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 ← 1

8:  MaxUpdate1(r, v):          // write to the second component
9:    WriteMax(r.second, v)

10: MaxScan(r):
11:   x ← ReadMax(r.second)
12:   if r.switch = 0
13:     WriteMax(r.left.second, x)
14:     return MaxScan(r.left)
15:   else
16:     x ← ReadMax(r.second)
17:     WriteMax(r.right.second, x)
18:     return ((m, 0) + MaxScan(r.right))
```

LEMMA 2. *The second component of the pair returned by a* ***MaxScan***(r) *operation is at most the value of $r.\textsf{second}$.*

PROOF. By induction on the range 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 $\texttt{ReadMax}(r.\textsf{second})$, which is the value of $r.\textsf{second}$.

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

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 $\texttt{MaxScan}(r.\textsf{left})$, which, by the induction hypothesis, is at most the value of $r.\textsf{left.second}$. Similarly, 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 $\texttt{MaxScan}(r.\textsf{right})$, which, by the induction hypothesis, is at most the value of $r.\textsf{right.second}$.

Whenever $\texttt{WriteMax}(r.\textsf{left.second}, x)$ is performed on Line 13 or $\texttt{WriteMax}(r.\textsf{right.second}, x)$ is performed on Line 17, the value of x is the result of a preceding $\texttt{ReadMax}(r.\textsf{second})$ operation. Thus $r.\textsf{left.second}, r.\textsf{right.second} \leq r.\textsf{second}$. □

LEMMA 3. *The second component of the pair returned by a* ***MaxScan***(r) *operation is at least the value of $r.\textsf{second}$ when the operation was invoked.*

PROOF. By induction on the range 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 $\texttt{ReadMax}(r.\textsf{second})$. Then the claim follows from the fact that the value of the $\texttt{MaxReg}_h$ object $r.\textsf{second}$ does not decrease.

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

If op' performs $\texttt{WriteMax}(r.\textsf{left.second}, x)$ on Line 13, then the value of $r.\textsf{left.second}$ will be at least v' when op' invokes $\texttt{MaxScan}(r.\textsf{left})$ 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 $\texttt{ReadMax}(r.\textsf{second})$, which is still at least v'. Then op' performs $\texttt{WriteMax}(r.\textsf{right.second}, x)$ on Line 17. Hence, the value of $r.\textsf{right.second}$ will be at least v' when op' invokes $\texttt{MaxScan}(r.\textsf{right})$ 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 4. *The* $\texttt{MaxArray}_{k\times h}$ *implementation in Algorithm 2 is linearizable.*

PROOF. By induction on k. 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.left is a linearizable $\texttt{MaxArray}_{m \times h}$ object, and r.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 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 2, $x_1 \le r.\mathsf{left}.\mathsf{second}$. Let v be the value of x the first time that $\texttt{WriteMax}(r.\mathsf{right}.\mathsf{second}, x)$ is performed on Line 17 during the execution. Then, by Lemma 1, $r.\mathsf{left}.\mathsf{second} \le v$.

Since $r.\mathsf{right}.\mathsf{second}$ is a $\texttt{MaxReg}_h$ object, which never decreases in value, $r.\mathsf{right}.\mathsf{second} \ge v$ when op' invokes Line 18. By Lemma 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 2 and 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 5. *For the $\texttt{MaxArray}_{k\times h}$ implementation in Algorithm 2, 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 $T(1) = 1$ 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, performs one $\texttt{MaxUpdate1}(r', v')$ operation, 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 $T_h(k) = T_h(\lceil k/2 \rceil) + 1 + 3 \cdot O(\log h)$. Since $T_h(1)$ is $O(\log h)$, it follows that $T_h(k)$ is $O(\log k \log h)$. □

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

We build a $(b-1)$-limited-use snapshot object from $\texttt{MaxArray}_{b\times b}$ objects, registers, and a $\texttt{MaxReg}_b$ object. To do so, we construct a strict, balanced, binary tree in which each leaf holds a pointer to the value of one component and each internal node holds a pointer to a partial snapshot containing the values of all the components in the subtree of which it is the root. The pointers held by each pair of siblings in the tree are stored at their parent in a 2-component max array. The pointer held by the root is stored in a max register. Each pointer is a nondecreasing index into a different array of b registers. The initial value of component j is stored in $\mathsf{leaf}_j.\mathsf{view}[0]$, for $j = 0, \ldots, n-1$. The concatenation of these values, for each leaf in the substree rooted at an internal node u, is stored in $u.\mathsf{view}[0]$. Figure 2 depicts this structure, with an $\texttt{Update}(4, s)$ operation in progress.

To perform a $\texttt{Scan}$, a process simply takes the result of a $\texttt{ReadMax}$ of the $\texttt{MaxReg}_b$ stored at the root and, if nonzero, uses it to index the array at the root. The step complexity of $\texttt{Scan}$ is dominated by the step complexity of $\texttt{ReadMax}$, 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 $\texttt{MaxScan}$ 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 $\texttt{MaxScan}$ operation takes $O(\log^2 b)$ steps and the tree has $O(\log n)$ height, the step complexity of $\texttt{Update}$ is $O(\log^2 b \log n)$.

Pseudocode for our implementation is given in Algorithm 3.

The resulting algorithm is similar to the lattice agreement procedure of Inoue *et al.* [12], except that we use $\texttt{MaxScan}$ 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

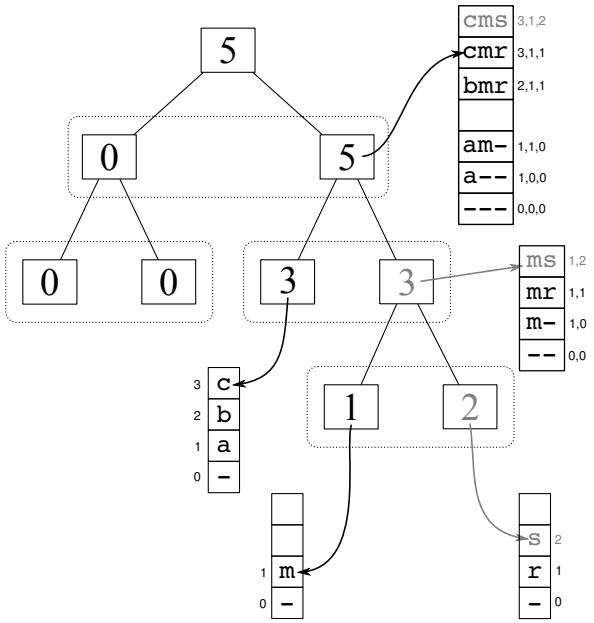

Figure 2: A limited-use single-writer snapshot object shared by 5 processes. Grayed values correspond to an update operation in progress. Sequences outside the view arrays represent entries of the seq arrays from the proof of correctness. Not all array locations are shown.

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 Jayanti's f-arrays [13] for efficient computation of aggregate functions (such as max and sum) of the elements of an array. Because the pointers are non-decreasing, we can use 2-component max arrays instead of the more powerful primitives used in that paper.

4.1 Linearizability

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 6. *If d is the index stored at root.mr when an `Update` operation begins, then $ptr > d$ when the operation performs `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 5.

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, $u.\mathsf{seq}[ptr] \leftarrow u.\mathsf{left.seq}[lptr] \cdot u.\mathsf{right.seq}[rptr]$ is 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 7. *The $(b-1)$-limited-use single-writer snapshot implementation in Algorithm 3 is linearizable.*

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

```
Shared data:
    leaf_j, for j ∈ {0,...,n−1}:
      the leaf node corresponding to process j, with fields:
      parent: the parent of this leaf in the tree
      view[0..b−1]: an array, each of whose entries contains
            a partial snapshot,
            view[0] contains the initial 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..b−1]: an 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
      ma: a MaxArray_{b×b} object, initially (0,0)
    The root also has the field:
    mr: a MaxReg_b object, initially 0
    Each non-root internal node also has the field:
      parent: the parent of the node in the tree
Persistent local data: count_i, initially 0.

1:  Update(s, v)
2:    count_i ← count_i + 1
3:    u ← leaf_i
4:    ptr ← count_i
5:    u.view[ptr] ← v
6:    repeat
7:      if u = u.parent.left
8:        MaxUpdate0(u.parent.ma, ptr)
9:      if u = u.parent.right
10:       MaxUpdate1(u.parent.ma, ptr)
11:     u ← u.parent
12:     (lptr, rptr) ← MaxScan(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:   until u = root
18:   WriteMax(root.mr, ptr)

19: Scan(s)
20:   ptr ← ReadMax(root.mr)
21:   return root.view[ptr]
```

4.2 Step Complexity

LEMMA 8. *The step complexity of a `Scan` operation is $O(\log b)$ and the step complexity of an `Update` operation is $O(\log^2 b \log n)$, where $b-1$ is an upper bound on the number of `Update` operations it supports.*

PROOF. A `Scan` operation performs one `ReadMax` on a $\mathtt{MaxReg}_b$ object and reads one entry from the array 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 $\mathtt{MaxArray}_{b \times b}$ objects 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 $O(\log^2 b \log n)$. □

This immediately gives us an $O(\log^3 n)$ implementation of any object that can be built from a snapshot object, including counters, generalized counters, and (by [3, 6]) any object with the property that for each pair of operations, either the operations always commute or one always overwrites the other—provided we only want to use the implementation polynomially many times.

5. MULTI-WRITER SNAPSHOTS

The previous section considered a single-writer snapshot object, that is, each component can be updated by a single process. Here, we extend this to implement a multi-writer snapshot object, where each component can be updated by every process. This is done by using a single-writer snapshot object 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. Ties are broken using process ids. To produce a timestamp, Lamport's linearizable timestamp algorithm [17] is used: a process scans the single-writer snapshot object and adds one to the largest timestamp for the corresponding multi-writer component. The creation of timestamps can be linearized in increasing order, with ties broken using process ids.

It is easy to see that the step complexity of `MW-Scan`(s) is the same as that of `Scan`(snp), and that the step complexity of `MW-Update`(s, j, v) is the sum of the step complexities of `Scan`(snp) and `Update`(snp, v).

We linearize a MW-Scan(s) operation at Line 7, which is the linearization point of the Scan(snp) operation. We linearize a MW-Update(s, j, v) operation which uses timestamp t the first time after it performs Line 2 at which the timestamp of the j'th pair of the component of some process is updated on Line 5 with a timestamp $t' \geq t$, with ties broken by process id. This is at or before the linearization point of the Update(snp, record) operation. Since every operation is linearized after it begins and before it returns, the order of non-overlapping operations is preserved.

If a MW-Scan(s) operation *op* returns a view $(v_0, \ldots, v_{c-1})$, then, for $0 \leq j < c$, the last MW-Update(s, j, v) operation op_j linearized before *op* has $v = v_j$. This follows because we are using a linearizable single-writer snapshot object, because the timestamps are linearized in increasing order, and because of the linearization points we chose for MW-Update operations. Thus, the linearization satisfies the specifications of a multi-writer snapshot object.

THEOREM 9. *The $(b-1)$-limited-use multi-writer snapshot implementation in Algorithm 4 is linearizable. The step complexity of a MW-Scan(s) operation is $O(\log b)$ and the step complexity of a MW-Update(s, j, v) operation is $O(\log^2 b \log n)$.*

6. DISCUSSION

This paper gives a linearizable implementation 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 [4], an implementation is given for an *unbounded* max register that can support an unbounded number of values and has a step complexity of $O(\min(\log v, n))$, where v is the value written or read. It may be possible to implement 2-component max arrays that support unbounded values and use them to construct unbounded snapshot objects whose step complexity is both $O(n)$ and polylogarithmic in the number of update operations. However, direct use of the unbounded tree construction from [4] seems to give worse complexity bounds. We leave this for future research.

Our 2-component max array implementation easily extends to c-component max arrays in a recursive manner, by having r.second be a $(c-1)$-component max array. The complexity of a MaxUpdate operation is then $O(\log k)$, where k is the range of each component, and the complexity of a MaxScan operation is $O(\log^c k)$. As with 2-component max arrays, this can also support components with different ranges, as well as unbounded ranges, with corresponding step complexities.

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 [8].

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

```
Shared data:
    snp: a single-writer snapshot object,
      each component is an array of c pairs (val,ts),
      each pair is initialized to (−, 0)
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:    view ← Scan(snp)
3:    t ← 1 + max{view_k[j].ts | 0 ≤ k < n}
4:    record[j] ← (v, t)
5:    Update(snp, record)

6:  MW-Scan(s)
7:    view ← Scan(snp)
8:    for 0 ≤ j < c do
9:      k ← argmax{(view_k[j].ts) × n + k | 0 ≤ k < n}
                  // find process with largest timestamp for
              // component j, use process id to break ties
10:     result_j ← view_k[j].val
                              // use its value for component j
11:   return result
```

7. REFERENCES

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

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

[3] J. H. Anderson and M. Moir. Towards a necessary and sufficient condition for wait-free synchronization (extended abstract). In A. Schiper, editor, *Distributed Algorithms, 7th International Workshop, WDAG '93, Lausanne, Switzerland, September 27-29, 1993, Proceedings*, volume 725 of *Lecture Notes in Computer Science*, pages 39–53. Springer, 1993.

[4] J. Aspnes, H. Attiya, and K. Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):2:1–2:24, Mar. 2012.

[5] J. Aspnes, H. Attiya, K. Censor-Hillel, and D. Hendler. Lower bounds for resricted-use objects. In *The 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2012. to appear.

[6] J. Aspnes and M. 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] H. Attiya and A. Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, 2001.

[8] H. Attiya and O. Rachman. Atomic snapshots in $O(n \log n)$ operations. *SIAM J. Comput.*, 27(2):319–340, 1998.

[9] M. A. Bender and S. Gilbert. Mutual exclusion with $O(\log^2 \log n)$ amortized work. In *IEEE 52nd Annual Symposium on Foundations of Computer Science, (FOCS)*, pages 728–737, 2011.

[10] F. E. Fich. How hard is it to take a snapshot? In *Proceedings of 31st Annual Conference on Current Trends in Theory and Practice of Informatics*, volume 3381, pages 27–35. LNCS, 2005.

[11] M. P. Herlihy and J. M. Wing. Linearizability: a

correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[12] M. Inoue and W. 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.

[13] P. Jayanti. f-arrays: implementation and applications. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 270–279, New York, NY, USA, 2002. ACM.

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

[15] P. Jayanti, K. Tan, and S. Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.*, 30(2):438–456, 2000.

[16] L. Kirousis, P. Spirakis, and P. Tsigas. Reading many variables in one atomic operation: Solutions with linear or sublinear complexity. *IEEE Trans. on Parallel and Distributed Systems*, 5(7):688–696, July 1994.

[17] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, Aug. 1974.

[18] Y. Riany, N. Shavit, and D. Touitou. Towards a practical snapshot algorithm. *Theor. Comput. Sci.*, 269(1-2):163–201, 2001.