SIAM J. COMPUT.
Vol. 27, No. 2, pp. 319–340, April 1998

ATOMIC SNAPSHOTS IN $O(n \log n)$ OPERATIONS*

HAGIT ATTIYA† AND OPHIR RACHMAN†

Abstract. The *atomic snapshot object* is an important primitive used for the design and verification of wait-free algorithms in shared-memory distributed systems. A snapshot object is a shared data structure partitioned into segments. Processors can either *update* an individual segment or instantaneously *scan* all segments of the object. This paper presents an implementation of an atomic snapshot object in which each high-level operation (scan or update) requires $O(n \log n)$ low-level operations on atomic read/write registers.

Key words. atomic read/write registers, single-reader multiwriter, snapshot objects, linearizability, asynchronous shared memory systems, wait-free computations

AMS subject classifications. 68P05, 68Q10, 68Q20, 68Q22

PII. S0097539795279463

1. Introduction.

Wait-free algorithms for shared-memory systems have attracted considerable attention during the past few years. The difficulty of synchronization and communication in such systems caused many of the algorithms that were developed to be quite intricate. A major research effort attempts to simplify the design and verification of efficient wait-free algorithms by defining convenient synchronization primitives and efficiently implementing them. One of the most attractive primitives is the *atomic snapshot object* introduced in [1, 2, 6].

An atomic snapshot object (in short, *snapshot object*) is a data structure shared by n processors. The snapshot object is partitioned into n segments, one for each processor. Processors can either *update* their own segment or instantaneously *scan* all segments of the object. By employing a snapshot object, processors obtain an instantaneous global picture of the system. This sidesteps the need to rely on "inconsistent" views of the shared memory and reduces the possible interleavings of the low level operations in the execution. Therefore, snapshot objects greatly simplify the design and verification of many wait-free algorithms. An excellent example is provided by comparing the recent proof of a bounded concurrent timestamp algorithm using snapshot objects [15] with the original intricate proof in [10].

Unfortunately, the great conceptual gain of using snapshot objects is often diminished by the actual cost of their implementation; the best snapshot implementation to date requires $O(n^2)$ read and write operations on atomic registers [1, 4]. Compared with the cost of simply reading n memory locations, this might seem a high price to pay for modularity and transparency. Thus, significant effort has been spent on avoiding snapshots and constructing algorithms directly from read and write operations.

This paper presents a snapshot object implementation in which each update or scan operation requires $O(n \log n)$ operations on single-writer multireader atomic reg-

*Received by the editors January 3, 1995; accepted for publication (in revised form) December 18, 1995. An extended abstract of this paper appeared in *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, Association for Computing Machinery, New York, 1993, pp. 29–40. This research was supported by grant 92-0233 from the United States–Israel Binational Science Foundation (BSF), Jerusalem, Israel, by the Technion V.P.R., Argentinian Research Fund, and by the fund for the promotion of research in the Technion.

http://www.siam.org/journals/sicomp/27-2/27946.html

†Department of Computer Science, The Technion, Haifa 32000, Israel (hagit@cs.technion.ac.il, fimfam@cs.technion.ac.il).

isters. Thus, we dramatically reduce the gap between the trivial lower bound of $\Omega(n)$ and the best known upper bound of $O(n^2)$ for atomic snapshots. Consequently, our snapshot object makes it feasible to design modular and easy to verify wait-free algorithms, without a great sacrifice in their efficiency.

We start with an algorithm for implementing an m-shot snapshot object, that is, a snapshot object to which up to m operations can be applied. The algorithm is simple and requires $O(n \log m)$ operations on single-writer multireader atomic registers. The algorithm is inspired by the algorithm presented in [7] for solving *lattice agreement* [4, 7, 11]. However, the algorithm of [7] uses atomic Test&Set operations, while the current algorithm uses only atomic read and write operations.

We then present ways to transform this algorithm to implement an ∞-shot snapshot object, that is, an object that supports an infinite number of operations.

One way is based on general-purpose transformations. In [7], the snapshot object was proved to be reducible to the lattice agreement problem. By employing the transformation of [7], the restriction of our algorithm to solve lattice agreement immediately implies an ∞-shot snapshot object in which each operation requires $O(n \log n)$ read and write operations on atomic registers. Unfortunately, this implementation requires an unbounded amount of memory. The bounded rounds abstraction of [13] can be used to bound the memory requirements of this implementation.

An alternative path is a direct implementation of an ∞-shot snapshot object, with $O(n \log n)$ operations for each scan or update. This implementation uses a bounded amount of memory and is based on recycling a single copy of the m-shot object. This recycling combines in a novel way synchronization techniques such as handshake bits [6], borrowing views [1] and traceable use techniques [14], and we believe it is interesting on its own.

The bounded algorithm uses atomic operations on registers that may contain up to $O(n(\log n + |V|))$ bits, where $|V|$ is the number of bits needed to represent a value of the snapshot object. (There are also operations on registers of size $O(n^4 \log n)$, but these occur infrequently.)

Besides the conceptual contribution to the design of future wait-free algorithms, our snapshot object immediately yields improvements to existing algorithms that use the snapshot object by plugging in our more efficient one. These include randomized consensus [3, 6], approximate agreement [8], bounded timestamping [15], and general constructions of wait-free concurrent objects [4, 17].

A *multiwriter* snapshot object is a generalized snapshot object in which any processor can update any segment. There is a transformation of Anderson's [2] which uses any snapshot object as a black box to construct a multiwriter snapshot object; this transformation requires a linear number of read and write operations. This transformation can be used to turn our algorithm into an algorithm for a multiwriter snapshot object with the same complexity.

Deterministic snapshot implementations have been proposed by Anderson [2] (bounded memory and exponential number of operations), by Aspnes and Herlihy [4] (unbounded memory and $O(n^2)$ operations), and by Afek et al. [1] (bounded memory and $O(n^2)$ operations). Attiya, Herlihy, and Rachman [7] give an $O(n \log^2 n)$ implementation that uses *Test&Set* registers, and an $O(n)$ implementation that uses dynamic *Test&Set* registers. Israeli, Shaham, and Shirazi [23] give a general technique to transform any snapshot implementation that requires $O(f(n))$ operations per scan or update into an (unbounded) implementation that requires $O(f(n))$ operations per scan and only a linear number of operations per update (or vice versa). Constructions of multiwriter snapshot objects appear in [1, 2].

Introduced in [7] are randomized implementations of the snapshot object ($O(n \log^2 n)$ using single-writer multireader registers, and $O(n)$ using dynamic single-writer multireader registers). Chandra and Dwork [9] also give a randomized implementation that requires $O(n \log^2 n)$ operations on atomic single-writer multireader registers. Weaker variants of the snapshot object were implemented by Kirousis, Spirakis, and Tsigas [24] (single-scanner snapshot object), and by Dwork et al. [12] (nonlinearizable snapshot object).

Independent of our work, Israeli and Shirazi [21] constructed a deterministic snapshot object that requires $O(n^{3/2} \log^2 n)$ operations and uses unbounded memory. Also, they showed a lower bound of $\Omega(\min\{w, r\})$ low-level operations for any update operation, where w is the number of updaters and r is the number of scanners [22].

As is made clear by the above review, our $O(n \log n)$ deterministic snapshot implementation significantly improves all known deterministic implementations that use only atomic registers and even improves almost all the existing randomized implementations. Note that by the general technique of [23], our snapshot implementation can be improved to require $O(n \log n)$ operations per update and only $O(n)$ operations per scan (or vice versa).

Following the original publication of our algorithm, Inoue et al. [19] presented an atomic snapshot object that requires only a linear number of read and write operations. However, this algorithm requires multiwriter registers, that is, each processor can write to each register. In contrast, our algorithms use only single-writer registers.

The rest of the paper is organized as follows. Section 2 includes definitions of the model and of the snapshot object. In section 3, we present the implementation of the m-shot snapshot object, which is then used to construct the ∞-shot snapshot object in section 4. We conclude in section 5 with a discussion of our results.

2. The snapshot object. Our model of computation is standard and follows, e.g., [8, 16].

An *atomic snapshot* object is partitioned into n segments, $S_1, \ldots, S_n$, where only processor p_i may write to the ith segment. The snapshot object supports two operations, *scan* and *update*(v). The scan operation allows a processor to obtain an instantaneous view of the segments, as if all n segments are read in a single atomic step. A scan operation returns a *view*, which is a vector $V[1, \ldots, n]$, where $V[i]$ is the value for the ith segment. The update operation with parameter v allows a processor to write the value v into its segment.

An implementation of the snapshot object should be *linearizable* [18]. That is, any execution of *scan* and *update* operations should appear as if it was executed sequentially in some order that preserves the real time order of the operations.

In more detail, each scan or update is implemented as a sequence of primitive operations. The nature of the primitive operations depends on the low-level objects used; in our case, read and write operations of atomic registers. An execution is a sequence of primitive operations, each executed by some processor as part of some scan or update operation. We assume that each processor has at most one (high-level) operation in progress at a time; that is, it does not start a new operation before the preceding one has completed.

Define a partial order $\rightarrow$ on (high-level) operations in an execution such that $op_1 \rightarrow op_2$ if (and only if) the operation op_1 has terminated before the operation op_2 has started; that is, all low-level operations that comprise op_1 appear in the execution before any low-level operation that is part of op_2. The partial order $\rightarrow$ reflects the external real time order of nonoverlapping operations in the execution.

For the snapshot implementation to be correct, we require that *scan* and *update* operations can be *linearized*. That is, there is a sequence that contains all *scan* and *update* operations in the execution that

a. extends the real time order of operations as defined by the partial order $\rightarrow$; and
b. obeys the sequential semantics of the snapshot operations; that is, if *view* is returned by some *scan* operation, then for every segment i, $view[i]$ is the value written by the last update to the ith segment which precedes the scan operation in the sequence.

In this paper, we define one operation that combines both scan and update, denoted $scate(v)$. Executing a $scate(v)$ operation by p_i both writes v into S_i and returns an instantaneous view of the n segments.[1] Intuitively, to perform $update(v)$ a processor invokes $scate(v)$ and simply ignores the view it returns; to perform a scan the processor invokes $scate(v)$, where v is the current value of its segment.

Another property that we require is *wait-freedom*; that is, every processor completes its execution of a scan or an update within a bounded number of its own (low-level) operations, regardless of the execution of other processors.

3. Implementation of an m-shot snapshot object. In this section we construct an m-shot snapshot object, which is a degenerate instance of the general snapshot object. Namely, an m-shot snapshot object is defined exactly as the general snapshot object, except that the total number of scate operations that may be performed by all processors is at most m.

3.1. Preliminaries. For the construction of the m-shot snapshot object, we modify each segment of the snapshot object to contain both the value of the segment in the field *value*, and some additional information that indicates the number of times p_i performed an operation. The additional fields *seq* and *counter* are incremented with each operation performed by p_i. Although the *seq* and *counter* fields contain exactly the same information, they have different roles in the implementation. The *seq* field determines which of two values written by p_i is more up to date. The *counter* field simply counts the number of operations performed by p_i. When we present the general implementation of the snapshot object, we shall see that the information in these two fields is maintained differently; this is why we separate them here as well.

Note that each *scate* operation returns a *view*, which is a vector with three fields in each entry. All segments are initially $(\perp, 0, 0)$. We now introduce some terminology.

The *size* of a view V, denoted by $|V|$, is $\sum_i V[i].counter$. The size of a view reflects the "amount of knowledge" that this view contains; that is, the size of a view counts the total number of operations performed on the snapshot object before this view was obtained.

A view V_1 *dominates* a view V_2, if for all i, $V_1[i].seq \geq V_2[i].seq$. Two views V_1 and V_2 are *comparable* if either V_1 dominates V_2 or V_2 dominates V_1. Two views V_1 and V_2 are *unanimous*, if for all i, $V_1[i].seq = V_2[i].seq$ implies that $V_1[i] = V_2[i]$. A set $\{V_1, \ldots, V_l\}$ of views is *unanimous* if any pair of views in the set are unanimous. The *union* of a unanimous set $\{V_1, \ldots, V_l\}$ of views, denoted by $\cup\{V_1, \ldots, V_l\}$, is the minimal view that dominates all views $V_1, \ldots, V_l$. That is, the union is a view V_u such that for every i, $V_u[i]$ equals $V_j[i]$ with maximal *seq* field. (All the sets of views that we use in the paper are trivially unanimous. Therefore we use unions of sets of views without explicitly stating that the sets are unanimous.)

[1] Combining the roles of scans and updates was implicitly done in previous works, where update operations not only write new values but also return views.

Classifier(K, I_i): returns(O_i) (Code for p_i)
0: write I_i to R_i
1: read $R_1, \ldots, R_n$
2: **if** $|\cup\{R_1, \ldots, R_n\}| > K$ **then**
3: read $R_1, \ldots, R_n$ and **return**$(O_i = \cup\{R_1, \ldots, R_n\})$
4: **else return**$(O_i = I_i)$

FIG. 1. *The classifier procedure.*

3.2. The *classifier* procedure. We start by introducing a procedure called *classifier*, with parameter K. Each processor p_i starts the procedure with an input view I_i, and upon termination, returns an output view O_i. The *classifier* procedure appears in Figure 1. The processors use a set of single-writer multireader registers $R_1, \ldots, R_n$.

In the *classifier* procedure each processor p_i starts with some local knowledge that is held in I_i. The goal of the *classifier* procedure is to update the processors' knowledge in some organized manner. Roughly speaking, the processors that use the procedure are classified into two groups such that the processors in the first group retain their original knowledge, while each processor in the second group increases its knowledge to dominate the knowledge of all the processors in the first group. Specifically, processors in the first group are called *slaves* and are defined as the processors that terminate the procedure in line 4. Processors in the second group are called *masters* and are defined as the processors that terminate the procedure in line 3. The classifying property of the procedure is the crux of the m-shot snapshot object. Notice that the *classifier* procedure provides very little guarantee on the number of masters and slaves. In particular, it is possible that all processors are classified as masters.

3.3. The implementation. To implement the scate operation for an m-shot snapshot object, we construct a full binary tree with $\log m$ levels and $m-1$ nodes.[2] The nodes of the tree are labeled by an in-order numbering on the tree, assigning labels in increasing order from the set $\{1, \ldots, m-1\}$. For each node v, we denote the label of v by $Label(v)$. The labels given by the in-order search can be presented in the following recursive manner: the root (in level 1) is labeled $\frac{m}{2}$; inductively, if a node v in level ℓ is labeled $Label(v)$, then the left child of v, denoted by $v.left$, is labeled $Label(v) - \frac{m}{2^{\ell+1}}$, and the right child of v, denoted by $v.right$, is labeled $Label(v) + \frac{m}{2^{\ell+1}}$. (See Figure 2.)

Since each processor may perform several scate operations, we do not identify an operation with the processor that executes it. In the rest of the section, we refer to operations as independent entities that "execute themselves."

Intuitively, each operation traverses the tree downwards starting from the root. Inside the tree, operations that arrive at some node execute the *classifier* procedure using the label of the node as the parameter K. The *classifier* procedure of each node separates the arriving operations so that less knowledgeable operations (slaves) proceed to the left and more knowledgeable operations (masters) proceed to the right. This process continues throughout the levels of the tree; an operation terminates when it arrives at a leaf of the tree.

The main idea in this construction is that operations are ordered in the leaves (from left to right) according to the amount of knowledge they have collected. As we prove in the following, when two operations are separated by some node, then the final

[2] We assume m is an integral power of 2. Otherwise, standard padding techniques can be applied.

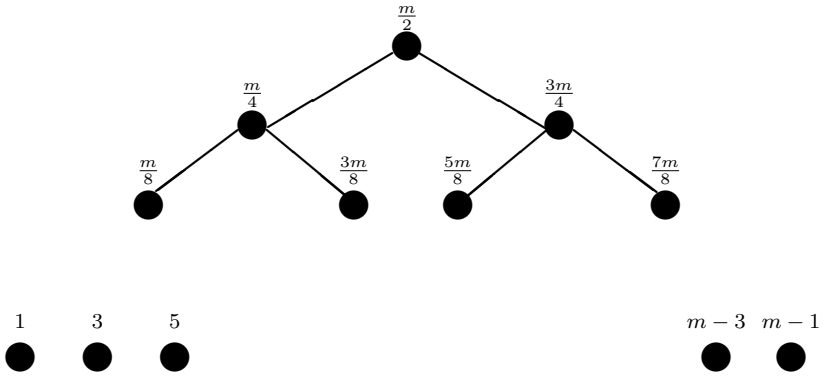

FIG. 2. *The classification tree.*

knowledge of the operation that proceeded to the right dominates the final knowledge of the operation that proceeded to the left. This guarantees that operations arriving at different leaves are comparable and are ordered from left to right. In addition, we prove that if two operations traverse exactly the same path in the tree, then they have exactly the same final knowledge. Such two operations undergo a "squeezing" process, where the difference in their knowledge is constantly reduced as they move toward the leaves of the tree. Finally, when two operations arrive at the same leaf the difference in their knowledge is squeezed to zero, and they are forced to have exactly the same knowledge.

To implement this intuitive idea, we associate with each node a separate area in the shared memory that contains a set of n single-writer multireader registers $R_1, \ldots, R_n$. These registers are initialized to empty views that contain $(\perp, 0, 0)$ in each entry and are used to execute the *classifier* procedure at that node. In addition, each processor p_i has a local variable $current_i$ that is initialized at the beginning of each operation by p_i. This local variable stores the accumulated knowledge of p_i during the execution of the operation. For ease of exposition, we add a $(\log m + 1)$th level to the tree, which now contains the leaves of the tree. These leaves have no labels and no associated registers, and they serve only as "final stations" for the operations that traverse the tree.

All *scate* operations, up to m, are executed on the tree constructed above. A *scate* operation *op* by p_i is executed as follows: first, *op* writes the value of the operation into S_i. Second, *op* reads the n segments $S_1, \ldots, S_n$, and sets $current_i$ to hold the view that contains the values read from the segments. Then *op* starts traversing the tree by entering the root. In general, when *op* enters some node v, it uses the value of $current_i$ as an input vector I_i, executes a $classifier(Label(v), current_i)$ procedure at v, and updates its $current_i$ variable to hold the value returned by the procedure. If p_i terminates the *classifier* procedure in v as a master, it enters $v.right$; otherwise it enters $v.left$. When *op* enters a leaf, it terminates and returns the value of $current_i$ as its final view. For clarity, we denote by $current_{i,\ell}$ the value of $current_i$ as *op* enters level ℓ. The precise code for a scate operation (by p_i) appears in Figure 3.

3.4. Correctness proof. We start by stating the properties of the *classifier* procedures that are executed in the various nodes of the tree. We first introduce some notation. For each node v, $Ops(v)$ denotes the set of operations that traverse through v. At each node v, each operation in $Ops(v)$ is classified either as a master or as a slave. The set of operations that are classified as masters at v is denoted by

scate(*value*) (Code for p_i)
1: $S_i := (value, incremented\ seq\ and\ counter)$
2: **for** $j := 1, \ldots, n$ $current_{i,1}[j] := S_j$
3: $v := root$
4: **for** $\ell := 1, \ldots, \log m$ **do**
5: $current_{i,\ell+1} :=$*Classifier*$(Label(v), current_{i,\ell})$
6: **if** master **then** *v:=v.right*
7: **if** slave **then** *v:=v.left*
8: **return**$(current_{i,\log m+1})$

FIG. 3. *The code for a scate operation.*

$M(v)$, and the set of operations that are classified as slaves at v is denoted by $S(v)$. In addition, we denote the input view of an operation op_i for the *classifier* procedure at level ℓ by $I_{i,\ell}$. (If p_j is the processor that executes op_i, then $I_{i,\ell}$ is the value assigned to $current_{j,\ell}$ during op_i.)

LEMMA 3.1. *Let v be some node at level ℓ. Let L and H be nonnegative integers such that $L \leq Label(v) \leq H$. If $L < |I_{i,\ell}| \leq H$ for every $op_i \in Ops(v)$, and $|\cup\{I_{i,\ell} : op_i \in Ops(v)\}| \leq H$, then*
(b1) *for every $op_i \in M(v)$, $Label(v) < |I_{i,\ell+1}| \leq H$,*
(b2) *for every $op_i \in S(v)$, $L < |I_{i,\ell+1}| \leq Label(v)$,*
(b3) $|\cup\{I_{i,\ell+1} : op_i \in M(v)\}| \leq H$,
(b4) $|\cup\{I_{i,\ell+1} : op_i \in S(v)\}| \leq Label(v)$, *and*
(b5) *for every $op_i \in M(v)$, $I_{i,\ell+1}$ dominates $\cup\{I_{j,\ell+1} : op_j \in S(v)\}$.*

Proof. Properties (b1)–(b3) are immediate from the code.

Property (b4) is proved by contradiction. Assume that $|\cup\{I_{i,\ell+1} : op_i \in S(v)\}| > Label(v)$. Since for each $op_i \in S(v)$ we have $I_{i,\ell+1} = I_{i,\ell}$, it follows that $|\cup\{I_{i,\ell} : op_i \in S(v)\}| > Label(v)$. Let op_j be the last operation in $S(v)$ that executes line 0 in the *classifier* procedure of v. When op_j executes line 1 of the procedure, all $I_{i,\ell}$ such that $op_i \in S(v)$ are already written in the registers of v. Since the value in any register of any node is overwritten only with values that dominate it, op_j collects a view with size greater than $Label(v)$. This contradicts the assumption that $op_j \in S(v)$.

To prove (b5), we show that when $op_i \in M(v)$ starts to execute line 3 in the *classifier* procedure of v, all $\{I_{j,\ell} : op_j \in S(v)\}$ are already written in the registers of v. Otherwise, if some $op_j \in S(v)$ has not yet written $I_{j,\ell}$, then when op_j executes line 1 of the procedure it reads registers' values that dominate the registers' values that op_i read in line 1. This contradicts the assumption that $op_j \in S(v)$. □

Using the properties of the *classifier* procedure as stated in the above lemma, we now prove that all the views returned in scate operations are comparable. To show that, we first prove that the views returned by operations that terminate in different leaves of the tree are comparable. The following two simple lemmas are implied by the code.

LEMMA 3.2. *Let op_i be an operation that returns V_i. Let v be a node such that $op_i \in S(v)$ and let ℓ be v's level. Then V_i is dominated by $\cup\{I_{j,\ell+1} : op_j \in S(v)\}$.*

LEMMA 3.3. *Let op_i be an operation that returns V_i. Let v be a node such that $op_i \in M(v)$ and let ℓ be v's level. Then V_i dominates $I_{j,\ell+1}$ for any $op_j \in S(v)$.*

The next lemma uses Lemmas 3.2 and 3.3 to prove that the views returned by operations that terminate in different leaves are comparable.

LEMMA 3.4. *Let op_i and op_j be two operations that terminate in leaves v_i and v_j, respectively, where $v_i \neq v_j$. Then the views returned by op_i and op_j are comparable.*

Proof. Let v be the node with maximal level (closest to the leaves) such that both op_i and op_j belong to $Ops(v)$, and let ℓ be its level. Since $v_i \neq v_j$, $\ell < \log m + 1$, that is, v is an inner node. Since v is not a leaf, one of op_i and op_j is a master in v, and the other is a slave at v. Assume, without loss of generality, that $op_i \in S(v)$ and $op_j \in M(v)$.

By Lemma 3.2, the view returned by op_i is dominated by $\cup\{I_{k,\ell+1} : op_k \in S(v)\}$. By Lemma 3.3, the view returned by op_j dominates each $I_{k,\ell+1}$, if $op_k \in S(v)$, and therefore dominates $\cup\{I_{k,\ell+1}, op_k \in S(v)\}$. Thus, the view returned by op_j dominates the view returned by op_i. □

To complete the comparability proof, we show that the output views of operations that terminate in the same leaf are comparable. The next lemma formally captures the intuitive idea of the "squeezed" difference in knowledge. The lemma bounds the size of the inputs $I_{i,\ell}$ and their union at some node v of some level ℓ as a function of $Label(v)$ and ℓ.

LEMMA 3.5. *Let v be an inner node of level ℓ. Then,*

(1) *for every $op_i \in Ops(v)$, $Label(v) - \frac{m}{2^\ell} < |I_{i,\ell}| \leq Label(v) + \frac{m}{2^\ell}$, and*
(2) $|\cup\{I_{i,\ell} : op_i \in Ops(v)\}| \leq Label(v) + \frac{m}{2^\ell}$.

Proof. The proof is by induction on ℓ. For the induction base $\ell = 1$, the lemma is straightforward since the total number of operations is at most m. For the induction step, assume the lemma holds for all nodes in level $\ell - 1$ and consider an arbitrary node v in level $\ell > 1$. Let v' be the parent of v and consider the *classifier* procedure with parameter $K = Label(v')$ that is executed by $Ops(v')$ in v'. By the induction hypothesis we have

(1) $Label(v') - \frac{m}{2^{\ell-1}} < |I_{i,\ell-1}| \leq Label(v') + \frac{m}{2^{\ell-1}}$, for any $op_i \in Ops(v')$, and
(2) $|\cup\{I_{i,\ell-1} : op_i \in Ops(v')\}| \leq Label(v') + \frac{m}{2^{\ell-1}}$.

If we denote $L = Label(v') - \frac{m}{2^{\ell-1}}$ and $H = Label(v') + \frac{m}{2^{\ell-1}}$, then these are exactly the conditions of Lemma 3.1. We have two cases.

Case 1. If $v = v'.right$, then $K = Label(v') = Label(v) - \frac{m}{2^\ell}$, and $Ops(v) = M(v')$. We have by (b1) and (b3) of Lemma 3.1 that

(1) for any $op_i \in Ops(v)$, $Label(v) - \frac{m}{2^\ell} < |I_{i,\ell}| \leq Label(v) + \frac{m}{2^\ell}$, and
(2) $|\cup\{I_{i,\ell} : op_i \in Ops(v)\}| \leq Label(v) + \frac{m}{2^\ell}$,

which are the required conditions for the operations in $Ops(v)$.

Case 2. If $v = v'.left$, then $K = Label(v') = Label(v) + \frac{m}{2^\ell}$, and $Ops(v) = S(v')$. In this case, the same equations are implied by (b2) and (b4) of Lemma 3.1. □

The next lemma proves that the views returned by two operations that terminate at the same leaf are equal, and in particular, comparable.

LEMMA 3.6. *Let op_i and op_j be two operations that terminate in the same leaf v. Then the views returned by op_i and op_j are equal.*

Proof. Let v' be the parent of v. Assume, without loss of generality, that $v = v'.right$; the proof if v is the left child of v' follows in the same manner. By Lemma 3.5, since v' is in level $\ell = \log m$,

(1) $Label(v') - 1 < |I_{k,\log m}| \leq Label(v') + 1$, for any $op_k \in Ops(v')$, and
(2) $|\cup\{I_{k,\log m} : op_k \in Ops(v')\}| \leq Label(v') + 1$.

The operations op_i and op_j execute the *classifier* procedure in v' with parameter $K = Label(v')$ and both terminate as masters and proceed to v. If we denote $L = Label(v') - 1$, and $H = Label(v') + 1$, then conditions (1) and (2) above are the required conditions for applying Lemma 3.1 to the *classifier* procedure that is executed in v'. Thus, by Lemma 3.1(b1), since op_i and op_j are in $M(v')$, we have

$|I_{i,\log m+1}| = |I_{j,\log m+1}| = \mathit{Label}(v') + 1$. In addition, by Lemma 3.1(b3), we have $|\cup \{I_{i,\log m+1}, I_{j,\log m+1}\}| = \mathit{Label}(v') + 1$. Therefore, $I_{i,\log m+1} = I_{j,\log m+1}$, which implies that the output views of op_i and op_j are equal. □

Lemmas 3.4 and 3.6 prove that the views returned by the scate operations are comparable. We now use these comparable scate operations to implement the linearizable scan and update operations of the snapshot object.

To execute an update(v) operation, a processor simply executes a scate(v) operation and ignores the view it returns. To execute a scan operation, a processor executes a scate(v) operation using the current value of its segment. Notice that although the same value is used, the *seq* and *counter* values are incremented. Thus, a scan operation by p_i changes the sequence number of S_i but does not change the value of S_i. Also notice that both scan and update operations return views. These views are later used for the linearization of the scan and update operations.

In order to define the linearization of operations on the snapshot object, we first order the scan operations and then insert the update operations. Consider any two scan operations sc_i and sc_j that return V_i and V_j, respectively. If $V_i \neq V_j$ and V_j dominates V_i, then sc_i is linearized before sc_j and vice versa if V_i dominates V_j. If $V_i = V_j$, then we order them first by the partial order $\rightarrow$, and if the operations are not ordered with respect to $\rightarrow$, then we break symmetry by the identities of the processors that execute the operations. This ordering of scans is well defined since a processor has only one operation outstanding at a time, and hence two operations by the same processor are always ordered by $\rightarrow$.

Next, we insert the update operations between the linearized scan operations. Consider an update operation that wrote a value (v, seq) to some segment S_i. The update operation is linearized after the last scan operation that returns a view that does not contain (v, seq) and before the first scan operation that contains (v, seq). Since scan operations are ordered by their views, each update operation fits exactly between two successive scan operations. We break symmetry between update operations that fit between the same two scan operations in the same manner as in the scan operations, that is, first by the partial order $\rightarrow$ and then by processors' identities. We now prove that this sequence is a linearization.

The next lemma follows immediately from the way update operations are linearized between scan operations.

LEMMA 3.7. *For any scan operation sc and for all segments S_i, the value returned by sc for S_i is the value written by the last update operation by processor p_i that is linearized before sc.*

Therefore, the linearization sequence we constructed preserves the semantics of the snapshot object. We now prove that it extends the partial order $\rightarrow$.

LEMMA 3.8. *For any two (scan/update) operations op_i and op_j, if $op_i \rightarrow op_j$ then op_i is linearized before op_j.*

Proof. There are four cases, according to operation types.

Case 1. Let sc_i and sc_j be two scan operations such that $sc_i \rightarrow sc_j$. By the code of the algorithm, the view returned by sc_i does not dominate the view returned by sc_j and hence the view returned by sc_j dominates the view returned by sc_i. Since scan operations are linearized by their views, this implies that sc_i is linearized before sc_j.

Case 2. Let sc_i be a scan operation and up_j be an update operation such that $sc_i \rightarrow up_j$. By the code of the algorithm, the view returned by sc_i does not contain the value written by up_j, and therefore, up_j is linearized after sc_i.

Case 3. Let up_i be an update operation and sc_j be a scan operation such that $up_i \rightarrow sc_j$. By the code of the algorithm, the view returned by sc_j contains the value written by up_i (or a value written by a later update operation by p_i,) and therefore, up_i is linearized before sc_j.

Case 4. Let up_i and up_j be two update operations such that $up_i \rightarrow up_j$. If up_i and up_j fit exactly between the same two scan operations, then due to the way symmetry is broken up_i is linearized before up_j, and the claim follows.

Otherwise, assume by way of contradiction that there exists a scan operation sc such that up_j is linearized before sc and sc is linearized before up_i. Thus, sc returns a view that contains the value written by up_j and does not contain the value written by up_i. Consider the scate operation that is executed to implement up_i. This scate operation returns a view that contains the value written by up_i but does not contain the value written by up_j. Therefore, this scate operation returns a view that is incomparable to the view returned by sc. This contradicts the comparability property of the views returned by the scate operations (Lemmas 3.4 and 3.6). □

Lemmas 3.7 and 3.8 prove that the scate operation of Figure 3 implements an m-shot snapshot object. The complexity analysis is obvious, and we have the following theorem.

THEOREM 3.9. *Each operation on the m-shot snapshot object implemented by the scate operation of Figure* 3 *requires $O(n \log m)$ operations on atomic single-writer multireader registers.*

Note that each processor has a view for each level of the classification tree. Denote by B the number of bits required to represent a view. Since the tree has $O(m)$ nodes, and for each node we have a view for each processor, it follows that the algorithm requires a total of $O(mnB)$ bits.

4. A general snapshot object.

4.1. An unbounded algorithm. A straightforward way to transform the m-shot snapshot object into an ∞-shot one is via the lattice agreement decision problem [4, 7, 11]. In this problem, processors start with inputs from a complete lattice and have to decide (in a nontrivial manner) on comparable outputs (in the lattice). It is fairly simple to use an n-shot snapshot object to solve lattice agreement and there is a general transformation that converts any lattice agreement algorithm into an implementation of an ∞-shot snapshot object [7]. The overhead of this transformation is $O(n)$ reads and writes per scan or update operation. Therefore, the m-shot snapshot object of the previous section can be converted into an ∞-shot snapshot object in which a scan or update operation requires $O(n \log n)$ operations.

Unfortunately, the general transformation of [7] extensively uses unbounded memory. That is, the transformation (possibly) replicates the memory area required for one lattice agreement algorithm, for each operation on the snapshot object. This is a consequence of the generality of the transformation, which does not assume anything about the lattice agreement algorithm. In tailoring the transformation to our m-shot snapshot object, the memory requirements can be reduced. That is, the number of registers can be bounded, and only their values increase by one with each new operation of the snapshot object. (The details, which are straightforward, will not be discussed here.) While these memory requirements are sufficient for any practical purpose, it is theoretically interesting to construct an ∞-shot snapshot object that requires only a bounded amount of shared memory.

A method to bound the memory requirements of the general transformation appears in [13]. Here we show a direct approach for combining the ideas of the m-shot

snapshot object with synchronization mechanisms to obtain a bounded implementation of a general snapshot object.

4.2. Bounded ∞-shot snapshot object. As mentioned before, the transformation of [7] employs an infinite number of copies of a lattice agreement algorithm so that each processor executes at most one operation on each copy. The algorithm presented here uses similar ideas but with a single copy of the m-shot snapshot object of the previous section.

Recall that in the construction of the m-shot snapshot object, each segment S_i has two additional fields, *seq* and *counter*. The *counter* field indicates how many operations were performed by p_i, while *seq* distinguishes, for any two values of p_i, which is more up to date. For the bounded implementation, we maintain this information using only bounded memory. Intuitively, the *seq* field is maintained using bounded sequential timestamps; the details are discussed in section 4.4. The more difficult task is to maintain the *counter* field, used for the classification process, using bounded memory.

In the general algorithm, we use the same tree of height $\log m+1$ which is traversed by the operations, as in the m-shot object. In order to allow one tree to support an unbounded number of operations, instead of only m, the operations are divided into *virtual* rounds, each containing exactly m operations.

By appropriate control mechanisms, we separate operations from different rounds so that they are not interleaved. In this way, the behavior of operations of the same round correspond to executing m operations on a separate m-shot snapshot object.

4.2.1. The bounded counter mechanism. In the m-shot object, the *counter* field associated with each segment specifies how many times the segment was updated; summing the *counter* fields over all segments yields the total number of operations that were performed on the snapshot object. In the general implementation, the *counter* field associated with a segment specifies the number of times the segment was updated modulo m; in this way, summing the *counter* yields the total number of operations that were performed on the object modulo m. (Although this sum is actually in the range $1, \ldots, nm$, we only refer to its value modulo m.)

We use the following terminology. The *counter* fields are called the *local counters*. The sum of the local counters modulo m is the *global counter*. The values of the local counters, as well as the global counter, are in the range $0, \ldots, m-1$.

For the sake of the proof, it is convenient to consider the unbounded values of these counters as well. That is, with each local counter we associate a virtual counter with the real (unbounded) value of that counter. Summing the virtual counters defines the real value of the global counter. The real values of the counters are not used within the code but only for the analysis.

4.2.2. The handshake mechanism. In the algorithm, we need to know the chronological order of operations by different processors. Specifically, for any two processors, p_i and p_j, we wish to know how many operations p_i started since a certain point in p_j's last operation (and vice versa). Clearly, we cannot maintain the exact number of operations since it is inherently unbounded. Therefore, we only want to know if the number of operations that p_i started is either 0, 1, 2,..., $k-1$, or strictly more than $k-1$ (for some constant k). This is done with a handshake mechanism that was introduced in [6].

For every two processors, p_i and p_j, there are two handshake variables $H_{i,j}$ and $H_{j,i}$. $H_{i,j}$ is written by p_i and read by p_j, while $H_{j,i}$ is written by p_j and read by p_i. An intuitive way to describe the functionality of the handshake variables is to consider

Handshake$_i(j)$
0: $temp = H_{j,i}$
1: **if** $Dist(H_{i,j}, temp) = 0$ **then return**$(H_{i,j} + 1)$
2: **if** $Dist(H_{i,j}, temp) \leq k$ **then return**$(temp)$
3: **if** $Dist(H_{i,j}, temp) > 2k$ **then return**$(H_{i,j} + 1)$

FIG. 4. *The handshake*$_i(j)$ *procedure.*

Takeover$_i(j)$ Invoked with every read from p_j's variable
1: **if** $Dist(H_{i,j}, H_{j,i}) = k$ **then goto** Takeover by p_j code

FIG. 5. *The takeover*$_i(j)$ *procedure.*

a directed cycle with vertices numbered $0, \ldots, 3k$, where the direction is defined from t to $(t + 1)$ mod $(3k + 1)$. The variables $H_{i,j}$ and $H_{j,i}$ represent the positions of p_i and p_j on this cycle. To handshake with p_i, p_j checks the values of $H_{i,j}$ and $H_{j,i}$ and updates its own position on the cycle accordingly.

More precisely, the function *handshake*$_i(j)$ is called by p_i in order to update $H_{i,j}$ (Figure 4). Using the procedures *handshake*$_i(j)$ and *handshake*$_j(i)$ by p_i and p_j, respectively, maintains the invariant that the directed distance from $H_{i,j}$ to $H_{j,i}$ on the cycle, denoted $Dist(H_{i,j}, H_{j,i})$, is either in the range $[0, \ldots, k]$ or in the range $[2k, \ldots, 3k]$. This invariant is used to determine who is the more advanced of the two processors. If the distance from $H_{i,j}$ to $H_{j,i}$ is at most k (but not zero), then p_j is more advanced, and if the distance is between $2k$ and $3k$ then p_i is more advanced. (If the distance is zero then p_i and p_j are equally advanced.)

4.2.3. The failure detection mechanisms. In the implementation we present, a scate operation may temporarily *fail* in one of two ways. The first kind of failure occurs if some processor, say p_j, performs several operations while p_i traverses the classification tree. This kind of failure is called a *takeover* failure; when it occurs, we say that p_i was *overtaken* by p_j. The second kind of failure is a *wraparound* of the global counter, which occurs when the value of the global counter goes from m to 0 while p_i traverses the classification tree. We now describe the failures in more detail and explain the failure detection mechanisms we employ.

Takeover failures are detected by a mechanism that is constantly being operated (see Figure 5). Whenever a processor p_i reads a register of some other processor, say p_j, it checks the value of $H_{j,i}$ with respect to $H_{i,j}$. If $Dist(H_{i,j}, H_{j,i}) = k$, that is, p_j executed k or more handshakes since p_i executed its last handshake, then a takeover failure by p_j is detected. In this case, p_i jumps directly to a place in the code that handles this situation.

Wraparound failures are detected by a different mechanism. Before p_i traverses the tree, it collects the values of the local counters and computes a value for the global counter. Later, p_i checks for a wraparound by using the procedure *check-wraparound*. The procedure receives the global counter's value that p_i computed earlier and reads the local counters again to obtain a new value for the global counter. If this value is smaller than the previous one, then a wraparound has occurred, and p_i jumps directly to a place in the code that handles this situation. Note that a wraparound may occur, but the global counter's value obtained by the procedure is greater than the earlier value of the global counter and the wraparound failure is not detected. We will show

check-wraparound$_i$(*counter*)
1: *temp* := $\sum_i S_i.counter \bmod m$
2: **if** *temp* $\leq$ *counter* **then goto** Wraparound code

FIG. 6. *The check-wraparound*$_i$ *procedure.*

that when a wraparound failure occurs but is not detected, a takeover failure must be detected by the handshake mechanism.

4.2.4. Data structures. For simplicity, we assume that the shared memory consists of only n single-writer multireader registers, $R_1, \ldots, R_n$. All the information written by processor p_i is stored in its register R_i, which contains the following fields:

S_i. p_i's segment, with three fields: *value*, (unbounded) *seq*, and (modulo m) *counter*.

$Tree_i$. p_i's registers in the classification tree of the m-shot object (one register per node). Each register holds the same three fields as above.

$H_{i,1}, \ldots, H_{i,n}$. The handshake variables of p_i with respect to all of the other processors. For simplicity, we assume p_i holds handshake variables also with respect to itself.

$Last[1,2]$. $Last[1]$ holds the view returned by the last scate operation by p_i. $Last[2]$ holds the view returned by the penultimate scate operation by p_i.

In the code and throughout the correctness proof, we refer to the various fields of the registers $R_1, \ldots, R_n$ separately. Any *read* operation from some field of a register implies that the whole register is read. Any *write* operation to some field means writing some new value to that specific field and rewriting the current values to the other fields.

4.2.5. Code description. The code appears in Figure 7.

In the code, p_i starts by recording the sequence number of its last operation and then incrementing its local sequence number and counter variables. Then, p_i performs the handshake procedure for each processor and then calculates the global counter. At this point, p_i writes the value of the operation into its segment S_i. Notice that it is possible that this line is not executed at all, since p_i may detect a takeover failure while collecting the values of the local counters (in line 4). In this case, p_i jumps directly to line 17 to handle the takeover failure and writes the value of the operation into S_i there. (Failure handling is explained later.) p_i proceeds by performing a wraparound check. If a wraparound is detected, p_i jumps to line 23. If no wraparound is detected, p_i collects a local view of the segments and starts to traverse the classification tree. This part of the operation is performed almost exactly as in the m-shot snapshot object, except that the calculations regarding the sizes of views, performed in the *classifier* procedures, are done modulo m. If p_i traverses the tree without detecting any takeover failure, it obtains some temporary result. Then, p_i performs another wraparound detection procedure. If during this procedure no failure is detected, p_i returns the temporary result as the result of the operation (and updates $R_i.Last[1,2]$). Otherwise, p_i jumps to handle the detected failure.

Both kinds of failures, takeover and wraparound, are handled in a similar manner. When p_i detects that it was overtaken by p_j, it tries to copy p_j's last returned view. However, p_i is allowed to do so only if the last view returned by p_j contains p_i's current operation value. If not, p_i starts the operation all over again. When p_i detects a wraparound failure, it tries to find a sufficiently recent view that was returned by some operation and copies it. More precisely, p_i tries to find a penultimate view of

```
scate(value)      (Code for p_i)
0:  first-seq := sequence-number

    Start:
1:  sequence-number := sequence-number+1
2:  my-counter := (my-counter + 1) mod m
3:  for j = 1 to n do H_{i,j} := Handshake_i(j)
4:  g-counter := Σ_i S_i.counter mod m
5:  S_i := (value,sequence-number,my-counter)
6:  check-wraparound(g-counter)
7:  in_i := read S_1, ..., S_n
8:  v := root, current_{i,1} := in_i
9:  for ℓ = 1... log m do
10:     current_{i,ℓ+1} := Classifier(Label(v), current_{i,ℓ})
11:     if master then v := v.right
12:     if slave then v := v.left
13: temp-result := current_{i,log m+1}
14: check-wraparound(g-counter)
15: R_i.Last[1,2] := ⟨temp-result, R_i.Last[1]⟩
16: return temp-result

    Takeover by p_j code:
17: S_i := (value,sequence-number,my-counter)
18: temp-result := R_j.Last[1]
19: if temp-result[i].seq > first-seq then
20:     R_i.last[1,2] := ⟨temp-result, R_i.Last[1]⟩
21:     return temp-result
22: else goto Start

    Wraparound code:
23: if ∃R_j.Last[2][i].seq > first-seq then
24:     R_i.Last[1,2] := ⟨R_j.Last[1], R_i.Last[1]⟩
25:     return R_j.Last[1]
26: else goto Start
```

FIG. 7. *The scate operation.*

some processor that contains p_i's current operation value. If p_i finds such a processor, it copies its last view; otherwise, p_i starts the operation all over again.

As a consequence of the failure handling technique, a scate operation may consist of several *attempts*. (Each time a processor arrives at the label Start is the beginning of a new attempt.) For every scate operation, only its last attempt is successful and returns a view. The successful attempts can either return a view through the failure handling procedures or not. Therefore, we partition attempts into three types: *unsuccessful* attempts, which do not return a view; *indirect* attempts, which return a copied view in line 21 or 25; and *direct* attempts, which return a view in line 16.

Note that different attempts of the same operation have different sequence numbers. Therefore, the unsuccessful attempts may be thought of as independent operations that are "cut off" before completion. On the other hand, the same *first-seq* is used by all attempts of the same operation. The value of *first-seq* is used in order to

decide whether to copy another processor's view in the failure procedures. That is, the conditions in lines 19 and 23 are satisfied if the found view contains the sequence number of any of the attempts of the current operation.

4.3. Correctness proof. We first show that views returned by scate operations are comparable. Since only successful attempts return views, it suffices to prove comparability for them.

Define an ordering on attempts according to the order they update the segments. (This order has nothing to do with the linearization of scans and updates which will be presented later.) Specifically, for each attempt we consider the first time that it writes to S_i, either in line 5 or in line 17. This write is called the *actual update* of the attempt. Since writes are atomic, the ordering of actual updates defines an ordering among the attempts.

Based on the ordering of the attempts, we divide them into "virtual rounds" of size m. The first round contains the first m attempts, and in general, the ith round contains attempts $(i-1)m+1, \ldots, im$.

Recall that k is the constant for the handshake mechanism, and m is a constant that determines the height of the classification tree. These constants were left unspecified, and we now fix $k = 8$ and $m = (k+2)n = 10n$.

The following lemma implies that in order to prove the comparability of views returned by successful attempts, we can consider only the direct attempts.

LEMMA 4.1. *A view returned by an indirect attempt is also returned by some direct attempt.*

Proof. Toward a contradiction, let at_i be an indirect attempt that copies a view from some $R_j.Last[1]$ such that this view is not a direct view. Consider all the attempts that return the same view as at_i, and from these attempts let at_k be the attempt whose write before returning its view (in lines 20 or 24) is the first in the execution. The view returned by at_k must be direct; otherwise, there was some other attempt that returned this view and wrote it before at_k did, which is a contradiction. □

We next show that the views returned by direct attempts can be organized by the virtual rounds.

LEMMA 4.2. *Let at_i be a direct attempt in round r_i, and let at_j be a (direct or indirect) attempt in round $r_j > r_i$. Then at_i starts to execute its wraparound test in line* 14 *before at_j executes its actual update step.*

Proof. We slightly abuse notation and denote the processors that execute at_i and at_j by p_i and p_j, respectively. Note that p_i and p_j may be the same processor, while at_i and at_j are not the same attempt. This should not cause any confusion.

Consider the execution of line 4 in at_i, and let c be the value of *g-counter*. Since at_i is in round r_i, the value of the global counter is still less than $(r_i+1)m$ when p_i completes line 4. Now p_i executes its actual update step. Since at_i is direct, p_i continues without detecting any failure and arrives in line 14.

Assume, by way of contradiction, that p_j executes its actual update step in at_j before p_i starts line 14. Therefore, the value of the global counter is greater than $(r_i+1)m$ when p_i starts line 14, since at_j is in round $r_j > r_i$. Since p_i does not detect a wraparound in line 14, the value it reads is $c' \geq c$. This can happen only if the local counters were incremented at least $m = (k+2)n$ times since p_i started to execute line 4. In particular, at least one processor p_l incremented its counter at least $(k+2)$ times since p_i has started to execute line 4. Thus, p_l performs $handshake_l(i)$ at least $(k+1)$ times since p_i started to execute line 4, which implies that p_l performs

handshake$_l(i)$ at least $(k+1)$ times since p_i performed *handshake*$_i(l)$ in at_i. By the properties of the handshake mechanism, p_i will detect a takeover failure by p_l while executing line 14, which is a contradiction. □

This implies the following corollary.

COROLLARY 4.3. *Let at_i be a direct attempt in round r_i. The view returned by at_i does not contain any values written by attempts in rounds strictly greater than r_i.*

By the definition of rounds, when p_i reads $S_1, \ldots, S_n$ in line 7 it observes all the values from previous rounds. Furthermore, it is immediate from the code that any direct attempt returns a view which contains at least the values it reads in line 7. Therefore, we have the following corollary.

COROLLARY 4.4. *Let at_i be a direct attempt of round r_i. The view returned by at_i contains all the values written in rounds strictly smaller than r_i.*

The above corollaries indicate that a direct attempt in round r observes all the values of rounds smaller than r, plus some subset of the values of round r, and nothing from rounds greater than r. Thus, for any two direct attempts in different rounds, it is clear that the view returned by the later attempt dominates the view returned by the earlier one. Consequently, in order to prove comparability of all the direct views, we need only prove comparability of attempts in the same round. This is done in the next lemma.

LEMMA 4.5. *Let at_i and at_j be two direct attempts of round r. The views returned by at_i and at_j are comparable.*

Proof. By Lemma 4.2, until both at_i and at_j arrive at line 14, no value of round greater than r is written in the segments and certainly not in the registers of the tree. In addition, when either at_i or at_j reads the segments before starting to traverse the tree (at line 7), all $(r-1)m$ values of rounds $1, \ldots, r-1$ are already written in the segments. Thus, the contribution of these values to the calculations that are performed in the *classifier* procedures that are executed throughout at_i and at_j is cancelled out.

This implies that the process of traversing the tree by at_i and at_j has exactly the same properties of the m-shot object construction. The comparability of the views returned by at_i and at_j is implied by the same arguments as in the m-shot object (in the proofs of Lemmas 3.4 and 3.6). □

Combining the above lemma with Lemma 4.1 implies the following corollary.

COROLLARY 4.6. *The views returned by any two scate operations are comparable.*

Comparable scate operations are used to implement scans and updates exactly in the same way as in the m-shot object. That is, to execute an update(v) operation, a processor executes scate(v) operation and ignores the value it returns; to execute a scan operation, a processor executes a scate(v) operation with the current value of its segment.

We now linearize the scan and update operations. First we identify each (update or scan) operation with the unique pair (v, seq) that is written by the first attempt of the operation. Scans and updates are linearized as in the m-shot object. That is, the scans are linearized according to the (comparable) views they return, and the updates are linearized between the scans according to the values they write. Clearly, by the way updates are linearized between scans, we have the following lemma.

LEMMA 4.7. *For every scan sc and for every S_i, the value returned by sc for S_i is the value written by the last update by p_i that is linearized before sc.*

Therefore, the sequence preserves the semantics of the snapshot object. To show it is a linearization, it remains to prove that the above sequence is consistent with the real time order of operations, $\rightarrow$.

The proof is similar to the corresponding proof for the m-shot object, but it is more complicated since in the m-shot object all the returned views were direct, while here the proof must consider both direct and indirect views. We start by introducing some terminology.

We say that an operation *op* (scan or update) returns a direct view if the successful attempt of *op* is direct, and similarly for indirect view. In addition, we sometimes classify *op* itself as direct or indirect.

The *origin* of an operation *op* is the attempt that directly returned the view returned by *op*. Formally, the origin of an operation *op* is defined inductively as follows. If *op* is direct, then the origin of *op* is the last attempt executed in *op*. Otherwise, if *op* is indirect and copies the view returned by op', then the origin of *op* is the origin of op'. In a similar manner, we define the *depth* of an operation *op*, which specifies the distance of *op* from its origin. If *op* is direct, then its depth is zero. Otherwise, if *op* is indirect and copies the view returned by op', then the depth of *op* equals the depth of op' plus one.

An *interval* is a subsequence of consecutive primitive operations in the execution. The *interval of an operation* is the interval starting with the execution of the first statement of the operation and ending with the execution of the last statement of the operation (not including the **Return** statement). The *interval of an attempt* is defined similarly.

An interval is *unsafe* if some processor starts and terminates two consecutive unsuccessful attempts in this interval. Otherwise, the interval is *safe*.

To show that the sequence defined above is consistent with $\rightarrow$, it suffices to prove that any indirect operation starts before its origin. This implies that the view copied from the origin is sufficiently up to date, and thus, the indirect operation is linearized within its interval. The intuitive proof argues that if an operation fails (due to either takeover or wraparound), then during the time interval of the operation many other operations were performed. At least some of these operations are completely contained in the interval, and therefore, the view copied by the indirect operation must be sufficiently up to date.

Unfortunately, the above intuition is not accurate since the failure mechanisms guarantee only that during the interval of an indirect operation there are many *attempts*. However, it is possible that not many of the attempts are successful, and therefore, not many operations are completed during this interval. This means that there are no up to date views to be copied. To overcome this problem we must show that an operation does not contain many attempts. This will imply that if there are many attempts in some interval, then there are many operations as well. To prove that an operation does not contain many attempts, we have to show that after a small number of unsuccessful attempts, an operation will find its value in some already existing view (or penultimate view). In turn, this relies on the fact that when a failure is detected, there are sufficiently up to date views that were obtained by other operations. On the face of it, this argument seems circular.

Put another way, the difficulty arises because the proof of partial correctness (processors return values that are up to date) relies on the assumption that operations terminate, and vice versa. We sidestep this circularity by first proving partial correctness if the operation's interval is safe, that is, all operations inside it terminate after (at most) two attempts. Using this fact, we then prove that any interval is safe, i.e., all operations terminate after (at most) two attempts. This implies that the claim holds for any operation.

LEMMA 4.8. *If op's interval is safe, then op's origin starts after op starts.*

Proof. The proof is by induction on d, the depth of op. The base case, $d = 0$, follows since the last attempt of op is the origin of op. For the induction step, let op be an operation with depth $d > 0$, and assume the lemma holds for any operation of depth $d - 1$ whose interval is safe. Since $d > 0$, op is indirect, and it copies the view of some operation op' of processor p' with depth $d - 1$. Let at and at' denote the successful attempts of op and op', respectively. There are two cases.

Case 1. op copies the view of op' due to a takeover failure. Since takeover failures are detected by the handshake procedure, p' has executed its handshake procedure at least $k \geq 6$ times while at was executed. Therefore, p' starts and completes at least four consecutive attempts during at's interval. Since at's interval is safe, at least two of these attempts are successful. Therefore, p' completes at least two operations while at is executed. The attempt at copies the view returned by op', which is the last preceding view returned by p'. The above implies that op' starts after at starts. By the induction hypothesis, the origin of op' starts after op' starts. Since this is also the origin of op, it follows that the origin of op starts after op starts.

Case 2. op copies the view of op' due to a wraparound failure. Let op'' be the operation of p' that precedes op'. By the condition for copying the view of op', the view returned by op'' contains the value written by op. Therefore, op'' does not terminate before op starts. In particular, op' starts after op starts. By the induction hypothesis, the origin of op' starts after op' starts. Since this is also the origin of op, it follows that the origin of op starts after op starts. □

We now prove that all intervals are safe, by showing that every operation terminates after at most two attempts.

LEMMA 4.9. *Every operation contains at most two attempts.*

Proof. Assume, by way of contradiction, that there is an operation op_i by processor p_i that contains two consecutive unsuccessful attempts, at_1, at_2. Assume that the interval from the start of at_1 to the completion of at_2 is minimal, that is, all intervals contained in it are safe. (Such a minimal interval exists because the execution is a sequence.) We prove that at_2 must be successful. There are two cases.

Case 1. at_2 fails due to a takeover failure by processor p_j. In this case, p_j executes its handshake procedure at least $k \geq 6$ times during at_2's interval. This implies that in this interval p_j starts and completes at least four attempts. Since any interval strictly contained in at_2's interval is safe, at least two of these attempts are successful. Let op_j be the last operation completed by p_j in at_2's interval. It follows that op_j starts after at_2 starts, and therefore after the actual update of op_i to S_i (since at_2 is not the first attempt of op_i). Since op_j's interval is safe, Lemma 4.8 implies that op_j's origin starts after op_j starts, and therefore after the value of op_i is written in S_i. This implies that the view returned by op_j contains the value written by op_i. Therefore, when p_i discovers a takeover by p_j in at_2, it can copy the view of op_j, and hence at_2 is successful, which is a contradiction.

Case 2. at_2 fails due to a wraparound failure. Consider the interval from the start of at_1 to the completion of at_2. If at_1 is unsuccessful due to a takeover failure, then clearly there is a processor p_j that executes its handshake procedure at least $k \geq 8$ times during this interval. Otherwise, if both at_1 and at_2 fail due to a wraparound failure, then again it is guaranteed that during their interval there is a processor p_j that executes its handshake procedure at least $k \geq 8$ times. This implies that in this interval p_j starts and completes at least six attempts. Since this interval is safe, at least three of these attempts are successful. This implies that p_j starts and completes at least two operations in this interval. As before, since this interval is safe, Lemma 4.8 implies that the last two operations of p_j in this interval return views that contain

the value written by op_i. Therefore, when p_i discovers a wraparound failure in at_2, it can copy the last view returned by p_j, and hence at_2 is successful, which is a contradiction. □

Thus, all operation intervals are safe, and therefore Lemma 4.8 can be applied to any operation to obtain the following corollary.

COROLLARY 4.10. *For any operation op, the origin of op starts after op starts.*

This implies that indirect operations copy views which are up to date. Since direct operations clearly observe the value they write, and since indirect operations copy other processors' view only if it includes their value, we have the following lemma.

LEMMA 4.11. *Any scan or update operation returns a view that contains its own value.*

The following lemma proves that the linearization sequence preserves the real time order of the operations.

LEMMA 4.12. *For any two (scan/update) operations op_i and op_j, if $op_i \rightarrow op_j$ then op_i is linearized before op_j.*

Proof. There are four cases, according to operation types.

Case 1. Let sc_i and sc_j be two scan operations such that $sc_i \rightarrow sc_j$. By Lemma 4.11, sc_j returns a view that contains the value it writes. Furthermore, sc_i does not return a view that contains the value of sc_j. Since the views returned by sc_i and sc_j are comparable (Corollary 4.6), it must be that the view returned by sc_j dominates the view returned by sc_i. Therefore, sc_i is linearized before sc_j.

Case 2. Let sc_i be a scan operation and up_j be an update operation such that $sc_i \rightarrow up_j$. By the code of the algorithm, the view returned by sc_i does not contain the value written by up_j, and therefore up_j is linearized after sc_i.

Case 3. Let up_i be an update operation and sc_j be a scan operation such that $up_i \rightarrow sc_j$. By Corollary 4.10, the origin of sc_j starts after sc_j does, and therefore after up_i's actual update. Since the origin is a direct attempt, it reads up_i's value. Therefore, sc_j returns a view that contains the value written by up_i, and hence sc_j is linearized after up_i.

Case 4. Let up_i and up_j be two update operations such that $up_i \rightarrow up_j$. If up_i and up_j fit exactly between the same two scan operations, then due to the way symmetry is broken, up_i is linearized before up_j, and the lemma follows.

Otherwise, if up_j is linearized before up_i, then there exists a scan operation sc such that up_j is linearized before sc and sc is linearized before up_i. Thus, sc returns a view that contains the value written by up_j and does not contain the value written by up_i. Consider the scate operation that is executed to implement up_i. This scate operation returns a view that contains the value written by up_i (Lemma 4.11) but does not contain the value written by up_j (since $up_i \rightarrow up_j$). Therefore, this scate operation returns a view that is incomparable to the view returned by sc. This contradicts the comparability property of the views returned by scate operations (Corollary 4.6). □

Lemmas 4.7 and 4.12 imply that the sequence of scans and updates defined above is indeed a linearization. By Lemma 4.9, each scate operation contains at most two attempts. Each attempt requires $O(n \log m) = O(n \log n)$ operations on atomic single-writer multireader registers, which implies the following lemma.

LEMMA 4.13. *Any scan or update operation terminates after at most $O(n \log n)$ operations on atomic single-writer multireader registers.*

Note that, in addition to a single copy of the m-shot classification tree, each processor maintains n handshake variables (each with $O(k)$ possible values) and two views. Since k is a constant and $m = O(n)$, the algorithm requires a total of $O(n^2 B)$

bits, where as before, B is the number of bits required for representing a view. Note that B is still unbounded, since the algorithm still uses unbounded sequence numbers.

4.4. Bounding the sequence numbers. So far, we presented the ∞-shot snapshot object using unbounded sequence numbers to allow every processor to distinguish, for any set of values of another processor, which one is the most up to date. When sequence numbers are unbounded this goal is easily achieved by choosing the value with the maximal sequence number. To avoid unbounded values we use *bounded sequential timestamps*, a concept introduced in [20]. In our case, each processor generates its own set of timestamps (timestamps of different processors are not compared). Therefore, we can use ideas of [14] to implement these timestamps. Below, we briefly describe these ideas; the reader is referred to [14, 13] for further details.

The main idea is to allow a processor to know which of its sequence numbers might be in use in the system. If this can be done, then a processor can simply choose a new sequence number to be some value that is not in use; to let other processors know what is the ordering among its sequence numbers, the processor holds an ordered list of all its currently used sequence numbers. If the number of sequence numbers that might be in use is bounded, then the processor can draw its sequence numbers from a bounded set of values (thus effectively recycling them).

The difficult part of the above idea is keeping track of the sequence numbers that are in use in the system. The natural idea that comes to mind is that all of the sequence numbers of a processor that are written somewhere in the shared memory at some point are the ones that are in use. However, there might be situations where some processor, p_i, reads a certain sequence number, x, and then x is overwritten and "disappears" from the shared memory. Later on, p_i might rewrite x in the shared memory. The *traceable use* abstraction of [14] solves this problem of "hidden" values by forcing a processor that reads a sequence number from the shared memory to leave evidence that this sequence number was read. This results in a slightly more complicated mechanism for reading and writing values from the shared memory.

To allow values to be recycled the processor invokes a "garbage collection" of sequence numbers, whose execution is spread over the duration of several operations (see further details in [14, 13]).

The number of (low-level read and write) operations required for generating bounded sequence numbers is linear, and therefore the $O(n \log n)$ complexity of the snapshot algorithm is not affected.

In the implementation of the traceable use abstraction, the number of sequence numbers that each processor uses is bounded by $O(n^2)$ times the total number of sequence numbers of that processor that may be in the system concurrently (cf. [13]). In our case, each processor can have at most $O(n^2)$ of its own sequence numbers in the system concurrently. Thus, the total number of sequence numbers that are used by each processor is $O(n^4)$, and the size of the sequence numbers is therefore $O(\log n)$. In addition, each processor must hold an ordered list of all its sequence numbers that are currently in use. The list requires $O(n^4 \log n)$ bits per processor.

As was calculated before, the algorithm requires $O(n^2 B)$ bits, where B is the number of bits required to represent a view. To calculate B, recall that a view contains n entries, each with three fields: the actual value of the entry, the *counter* field ($O(\log n)$ bits), and the *seq* field (now bounded to require $O(\log n)$ bits). Therefore,

the number of bits required to represent a view is $O(n \log n)$, plus n times the number of bits required to represent an actual values of the snapshot object, which we denote by $|V|$. Thus, the total space complexity is $O(n^3(\log n + |V|) + n^5 \log n)$ bits.

5. Discussion. We introduced an implementation of a bounded atomic snapshot object in which each update or scan operation requires $O(n \log n)$ operations on atomic single-writer multireader registers. (As was previously mentioned, one of the operations can be made linear by the results of [23].) Obviously, at least $\Omega(n)$ operations are required for implementing the scan operation for an atomic snapshot object, and by [22] this is also the lower bound for implementing the update operation. Needless to say, it will be very interesting to close the $O(\log n)$ gap between our implementation and this lower bound.

REFERENCES

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

[2] J. H. Anderson, *Composite registers*, Distrib. Comput., 6 (1993), pp. 141–154.

[3] J. Aspnes, *Time- and space-efficient randomized consensus*, in Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing, ACM, New York, 1990, pp. 325–331.

[4] J. Aspnes and M. P. Herlihy, *Wait-free data structures in the asynchronous PRAM model*, in Proceedings of the 2nd Annual Symposium on Parallel Algorithms and Architectures, ACM, New York, 1990, pp. 340–349.

[5] H. Attiya, A. Bar-Noy, and D. Dolev, *Sharing memory robustly in message-passing systems*, in Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing, ACM, New York, 1990, pp. 363–376.

[6] H. Attiya, D. Dolev, and N. Shavit, *Bounded polynomial randomized consensus*, in Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing, 1989, ACM, New York, pp. 281–293.

[7] H. Attiya, M. Herlihy, and O. Rachman, *Atomic snapshots using lattice agreement*, Distrib. Comput., 8 (1995), pp. 121–132.

[8] H. Attiya, N. A. Lynch, and N. Shavit, *Are wait-free algorithms fast?*, J. ACM, 41 (1994), pp. 725–763.

[9] T. Chandra and C. Dwork, *Using Consensus to Solve Atomic Snapshots*, 1992, manuscript.

[10] D. Dolev and N. Shavit, *Bounded concurrent time-stamping*, SIAM J. Comput., 26 (1997), pp. 418–455.

[11] C. Dwork, private communication.

[12] C. Dwork, M. P. Herlihy, S. A. Plotkin, and O. Waarts, *Time-lapse snapshots*, in Proceedings of the Israel Symposium on the Theory of Computing and Systems, Haifa, Israel, 1992, Lecture Notes in Comput. Sci. 601, D. Dolev, Z. Galil, and M. Rodeh, eds., Springer-Verlag, Berlin, pp. 154–170.

[13] C. Dwork, M. P. Herlihy, and O. Waarts, *Bounded round numbers*, in Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing, ACM, New York, 1993, pp. 53–64.

[14] C. Dwork and O. Waarts, *Simple and efficient concurrent timestamping or bounded concurrent timestamping are comprehensible*, in Proceedings of the 24th ACM Symposium on Theory of Computing, ACM, New York, 1992, pp. 655–666.

[15] R. Gawlick, N. Lynch, and N. Shavit, *Concurrent timestamping made simple*, in Proceedings of the Israel Symposium on the Theory of Computing and Systems, Haifa, Israel, 1992, Lecture Notes in Comput. Sci. 601, D. Dolev, Z. Galil, and M. Rodeh, eds., Springer-Verlag, Berlin, pp. 171–183.

[16] M. P. Herlihy, *Wait-free synchronization*, ACM Trans. Programming Lang. Systems, 13 (1991), pp. 124–149.

[17] M. P. Herlihy, *Randomized wait-free objects*, in Proceedings of the 10th ACM Symposium on Principles of Distributed Computing, ACM, New York, 1991, pp. 11–21.

[18] M. P. Herlihy and J. M. Wing, *Linearizability: A correctness condition for concurrent objects*, ACM Trans. Programming Lang. Systems, 12 (1990), pp. 463–492.

[19] M. Inoue, W. Chen, T. Masuzawa, and N. Tokura, *Linear-time snapshot using multi-writer multi-reader registers*, in Proceedings of the 8th International Workshop on Distributed Algorithms, Terschelling, The Netherlands, 1994, Lecture Notes in Comput. Sci. 857, G. Tel and P. Vitanyi, eds., Springer-Verlag, Berlin, 1994, pp. 130–140.

[20] A. Israeli and M. Li, *Bounded time stamps*, Distrib. Comput., 6 (1987), pp. 205–209.

[21] A. Israeli and A. Shirazi, *Efficient Snapshot Protocol Using* 2*-Lattice Agreement*, 1992, manuscript.

[22] A. Israeli and A. Shirazi, *The time complexity of updating snapshot memories*, in 2nd Annual European Symposium on Algorithms, Lecture Notes in Comput. Sci. 855, Springer-Verlag, New York, 1994, pp. 171–182.

[23] A. Israeli, A. Shaham, and A. Shirazi, *Linear-time snapshot protocols for unbalanced systems*, in Proceedings of the 7th International Workshop on Distributed Algorithms, A. Schiper, ed., Lausanne, Switzerland, 1993, Lecture Notes in Comput. Sci. 725, Springer-Verlag, Berlin, 1993, pp. 26–38.

[24] L. M. Kirousis, P. Spirakis, and Ph. Tsigas, *Reading many variables in one atomic operation: Solutions with linear or sublinear complexity*, in Proceedings of the 5th International Workshop on Distributed Algorithms, Delphi, Greece, 1991, Lecture Notes in Comput. Sci. 579, S. Toueg, P. Spirakis, and L. Kirousis, eds., Springer-Verlag, Berlin, 1991, pp. 229–241.