Atomic Snapshots of Shared Memory

Yehuda Afek* Hagit Attiya† Danny Dolev‡

Eli Gafni§ Michael Merritt¶ Nir Shavit‖

Abstract

An *atomic snapshot memory* is a shared data structure allowing concurrent processes to store information in a collection of shared registers, all of which may be read in a single atomic *scan* operation. This paper presents three wait-free implementations of atomic snapshot memory. Two constructions implement wait-free single-writer atomic snapshot memory from wait-free atomic single-writer, n-reader registers. A third construction implements a wait-free n-writer atomic snapshot memory from n-writer, n-reader registers. The first implementation uses *unbounded* (integer) fields in these registers, while the other implementations use only *bounded* registers. All operations require $\Theta(n^2)$ reads and writes to the component shared registers in the worst case.

*Tel-Aviv University and AT&T Bell Laboratories.

†Laboratory for Computer Science, MIT. Supported by NSF grants CCR-8611442 and CCR-8915206, by ONR contract no N00014-85-K-0168, and by DARPA contracts no N00014-83-K-0125 and N00014-89-J-1988.

‡IBM Almaden Research Center and Hebrew University.

§Tel-Aviv University and University of California, Los Angeles, Supported by NSF Presidential Young Investigator Award under grant DCR84-51396 & matching funds from XEROX Co. under grant W881111.

¶AT&T Bell Laboratories.

‖IBM Almaden Research Center and Stanford University. Most of this work was performed while the author was at the Hebrew University, Jerusalem, visiting AT&T Bell Laboratories and the TDS group at MIT, supported by NSF contract no CCR-8611442, by ONR contract no N0014-85-K-0168, and by DARPA contracts no N00014-83-K-0125 and N00014-89-J-1988.

1 Introduction

Obtaining an instantaneous global picture of a system, from partial observations made over a period of time as the system state evolves, is a fundamental problem in distributed and concurrent computing. Indeed, much of the difficulty in proving correctness of concurrent programs is due to the need to argue based on "inconsistent" views of shared memory, obtained concurrently with other process's modifications. Verification of concurrent algorithms is thus complicated by the need for a "non-interference" step [Owi75, OG76]. By simplifying (or eliminating) the non-interference step, atomic snapshot memories can greatly simplify the design and verification of many concurrent algorithms. Examples include exclusion problems [K78, L86c, DGS88], construction of atomic multi-writer multi-reader registers [VA86, Blo87, PB87, S88, LTV89], concurrent time-stamp systems [DS89], randomized consensus [A88, AH89, ADS89, A90] and wait-free implementation of data structures [AH90].

This paper introduces a general formulation of *atomic snapshot memory*, shared memory partitioned into words written (*updated*) by individual processes, or instantaneously read (*scanned*) in its entirety. It presents three wait-free implementations of atomic snapshot memories, constructed from wait-free atomic registers. (In

[A89a, A89b, An90], Anderson independently introduces the same notion and presents bounded implementations. See Section 6 for a discussion.) The first implementation uses unbounded (integer) fields in these registers, and is particularly easy to understand. The second implementation uses bounded registers. Its correctness proof follows the ideas of the unbounded implementation. Both constructions implement a single-writer snapshot memory, in which each word may be updated by only one process, from single-writer, n-reader registers. The third algorithm implements a multi-writer snapshot memory [A89b] from wait-free atomic n-writer, n-reader registers, again echoing key ideas from the earlier constructions. Each *update* or *scan* operation requires $\Theta(n^2)$ reads and writes to the relevant embedded atomic registers, in the worst case.

A related data structure, *multiple assignment*, allows processes to atomically update nontrivial and intersecting subsets of the memory words, and to read one location at a time. However, multiple assignment has no wait-free implementation from read/write registers [H88]. The fact that wait-free atomic snapshot memories can be implemented from wait-free atomic registers stands in contrast to the impossibility results in [H88].

Section 2 of this paper defines single-writer and multi-writer atomic snapshot memories. Section 3 contains an implementation of single-writer snapshot memories from unbounded single-writer multi-reader registers, Section 4 presents an implementation of single-writer snapshot memories from bounded single-writer registers, and Section 5 presents an implementation of multi-writer snapshot memories from bounded multi-writer, multi-reader registers. Section 6 concludes with a discussion of the results, related work and directions for future research.

2 Atomic Snapshot Memories

Consider a shared memory divided into *words*, where each word holds a data value. In the single-writer case, there is one word for each process, which only it writes (in its entirety) and the others read. In the multi-writer case, any of the words may be read or written by any of the processes. An n-process atomic snapshot memory supports two types of operations, $scan_i$ and $update_i$, that are available to each process P_i. Executions of scans and updates can each be considered to have occurred as primitive atomic events between the beginning and end of the corresponding operation execution interval, so that the "serialization sequence" of such atomic events satisfies the natural semantics. That is, each scan operation returns a vector $\bar{v}$ of values such that each v_k is the argument of the last update to word k that is serialized before that scan. (This variant of serializability is called "linearizability" [HW87].) This intuition is made precise in the following subsection.

Two further restrictions are imposed on implementations of atomic snapshot memories. First, following e.g. [L86b, H88], any snapshot implementation is required to be constructed with single-writer, multi-reader atomic registers as the only shared objects. The single-writer algorithms in Sections 3 and 4 satisfy this restriction directly, and the multi-writer algorithm in Section 5 satisfies this restriction when the embedded multi-writer registers are in turn implemented with one of the previously known constructions from single-writer registers, e.g., [PB87, LTV89].

The second restriction imposed on snapshot memory implementations is that they satisfy the property of *wait-freedom* [L86a, P83]. That is, every snapshot operation by process P_i will terminate in a bounded number of atomic steps of P_i, regardless of the behavior of other processes, assuming only that local steps of P_i and operations on embedded shared objects terminate in bounded time. (The reader is referred to [L86a, H88, AG88] for discussions and proposed definitions of wait-freedom.)

2.1 A Specification of Single-Writer Snapshot Memories

Following [LT87, H88], a single-writer atomic snapshot memory for n processes and a particular value set *Value* is an automaton with two types of input `Request` actions: $\texttt{UpdateRequest}_i(v)$ and $\texttt{ScanRequest}_i$, and two types of output `Return` actions: $\texttt{UpdateReturn}_i$ and $\texttt{ScanReturn}_i(v_1, ..., v_n)$, for any $i \in \{1..n\}$, and for all $v, v_1, ..., v_n \in \mathit{Value}$. These actions are called the *interface snapshot actions.*

The formal specification of single-writer snapshot memory is based on a particular automaton, SWS. In addition to the interface snapshot actions, SWS has two types of internal actions, $\texttt{Update}_i(v)$, and $\texttt{Scan}_i(v_1, ..., v_n)$, for any $i \in \{1..n\}$ and for all $v, v_1, ..., v_n \in \mathit{Value}$. The states of SWS contain an n-entry array *Mem* of type *Value* and n interface variables H_i. The interface variables may hold as value any of the interface snapshot actions, or a special value $\perp$.

Process P_i interacts with SWS by issuing a request (an $\texttt{UpdateRequest}_i(v)$ or $\texttt{ScanRequest}_i$ action). The result is to store the input action in the variable H_i, enabling the appropriate internal action ($\texttt{Update}_i(v)$ or $\texttt{Scan}_i(v_1, ..., v_n)$). The internal action in turn assigns an appropriate output action to H_i, and in the case of $\texttt{Update}_i(v)$, assigns v to Mem_i as well. The change to the interface value H_i enables the appropriate output ($\texttt{UpdateReturn}_i$ or $\texttt{ScanReturn}_i(v_1, ..., v_n)$ action). Initially, each $H_i = \perp$ and $\mathit{Mem}_i = v_{init} \in \mathit{Value}$.

The steps of SWS appear in Figure 1, with the convention that actions without preconditions are always enabled (e.g., input actions), and that state components not explicitly described in the effect of an action are presumed to retain their old value.

Note that, while requests and returns by different processes may be interleaved, these actions only alter the interface variables for the associated processes. The "real" work is done by the atomic internal actions, formalizing the intuition that operations of atomic memories can be assumed to have occurred at some instant between the invocation and response. Accordingly, an op-

$\texttt{UpdateRequest}_i(v)$
Effect: $H_i := \texttt{UpdateRequest}_i(v)$

$\texttt{Update}_i(v)$
Precondition: $H_i = \texttt{UpdateRequest}_i(v)$
Effect: $\mathit{Mem}[i] := v$
$H_i := \texttt{UpdateReturn}_i$

$\texttt{UpdateReturn}_i$
Precondition: $H_i = \texttt{UpdateReturn}_i$

$\texttt{ScanRequest}_i$
Effect: $H_i := \texttt{ScanRequest}_i$

$\texttt{Scan}_i(v_1, ..., v_n)$
Precondition: $H_i = \texttt{ScanRequest}_i$
$\mathit{Mem} = (v_1, ..., v_n)$
Effect: $H_i := \texttt{ScanReturn}_i(v_1, ..., v_n)$

$\texttt{ScanReturn}_i(v_1, ..., v_n)$
Precondition: $H_i = \texttt{ScanReturn}_i(v_1, ..., v_n)$
Effect: $H_i := \perp$

Figure 1: The SWS automaton.

eration of SWS in α is said to be *serialized* at the point of its associated `Update` or `Scan` operation.

The *well-formed* behaviors of SWS are those in which the environment never issues two $\texttt{Request}_i$ inputs without waiting for an intervening, matching $\texttt{Return}_i$ output. An automaton A *implements a single-writer atomic snapshot memory* provided A has the interface snapshot actions as its input and output actions, and provided every well-formed behavior of A is also a behavior of SWS. *

2.2 A Specification of Multi-Writer Snapshot Memories

Multi-writer snapshot memories are straightforward generalizations of single-writer snapshot memories, and can be specified analagously. Specifically, a multi-writer snapshot memory for n processes, a particular value set *Value* and m memory elements is an automaton with input actions: $\texttt{UpdateRequest}_i(k, v)$ and $\texttt{ScanRequest}_i$, and output actions: $\texttt{UpdateReturn}_i$ and $\texttt{ScanReturn}_i(v_1, ..., v_m)$, for all $i \in \{1..n\}$, $k \in \{1, \ldots, m\}$, and $v, v_1, ..., v_m \in \mathit{Value}$.

Straightforward modifications of the automaton SWS of Figure 1 are used to constrain implementations of multi-writer snapshot memories, just as SWS constrained single-writer snapshot memories. (The details are left to the reader.)

3 The Unbounded Single-Writer Algorithm

The algorithm is based on two observations:
Observation 1: Suppose every update leaves a unique, indelible mark whenever it writes to the memory. If two sequential reads of the entire memory return identical values, where one read started after the first completed, then the values returned constitute a snapshot [PB87].

This observation alone supports a simple unbounded algorithm, although one which is not wait-free. The kth update by processor P_i simply writes the update value v and a sequence number k to a shared register in a single atomic write. Scanners repeatedly collect the values of all n registers, until two such collect operations return identical values. By Observation 1, such a successful *double collect* is a snapshot.

Because updates may occur between every two successive collect operations, this algorithm is not wait-free. However, the scanner may attribute every unsuccessful double collect to a particular updating process, whose sequence number was observed to change. Thus:
Observation 2: If a scan sees another process move (complete an update) twice, that process executed a complete update operation within the interval of the scan.

Suppose every update performs a scan and writes the snapshot value atomically with the value and sequence number. Now a scanner who sees two updates by the same process can borrow the snapshot value written by the second update.

A straightforward implementation uses the following shared data structures. (See Figure 2.) Each process P_i has a single-writer, n-reader atomic register, r_i, that P_i writes and all processes read. The register has three fields, $\mathit{value}(r_i)$ (of type *Value*), $\mathit{seq}(r_i)$ (of type integer) and $\mathit{view}(r_i)$ (a vector of n entries of type *Value*). The *value* and *view* fields are initialized to v_{init} and the *seq* fields are initialized to 0.

The value of seq_i is stored (locally) across invocations of update_i. In addition, each scan operation has a local vector *moved*, in which it records, for each other process, whether it has performed an update operation that overlapped the scan operation. The *collect* operation by any process i reads each register r_j, $j \in \{1..n\}$, in an arbitrary order, returning a vector of records read, indexed by process id.

*Alternative approaches to specifying concurrent objects are via their serial specification [HW87] or as a set of *axioms* (cf. [L86a, M86]). Axiomatic specifications for snapshot memories appear in [A89a, A89b, ADS89].

```
procedure scan_i
begin
  0:  for j = 1 to n do moved_j := 0 od;
  1:  ā := collect;
      /* (value, seq, view) triples */
  2:  b̄ := collect;
  3:  if (∀j ∈ {1..n})
          (seq(a_j) = seq(b_j)) then
              /* Nobody moved. */
  4:    return (value(b_1), ..., value(b_n));
  5:  for j = 1 to n do
  6:      if seq(a_j) ≠ seq(b_j) then
                      /* P_j moved */
  7:        if moved_j = 1 then
          /* P_j moved once before! */
  8:          return (view(b_j));
  9:        else moved_j := moved_j + 1 ;
          od;
  10: goto line 1 ;
end scan_i;

procedure update_i (value)
begin
  1: s̄ := scan_i; /* Embedded scan. */
  2: r_i := (value, seq_i + 1, s̄) ;
end update_i;
```

Figure 2: The unbounded single-writer algorithm.

3.1 Correctness Proof

The proof strategy is to construct an explicit serialization. That is, given an infinite or finite run of the system, calls and returns from the *update*$_i$ procedures are identified with the UpdateRequest$_i$ and UpdateReturn$_i$ actions, and calls and returns from *scan*$_i$ procedures (unless called from within *updates*), are identified with the ScanRequest$_i$ and ScanReturn$_i$ actions. The scan and update operations themselves consist of sequences of more primitive operations that are either reads and writes of atomic registers, or manipulations of local data. The former are atomic by assumption; the latter are trivially atomic. Hence, an arbitrary run of an n-process system can be considered to be a (possibly infinite) sequence of interface snapshot actions, and atomic reads, writes or local data manipulations. Given this sequence, Scan$_i$ and Update$_i$ actions are added so that the resulting sequence, projected on the actions of SWS, is a schedule of that automaton. Hence, the algorithm is atomic.

Consider then any sequence $\alpha = \pi_1\pi_2...$, where each π_j is either an action of SWS, a read $read_i(r_j = v)$ by P_i of atomic register r_j returning v, a write $write_i(r_i = v)$ by P_i of v to r_i, or a local computation event. Denote by α_k the k-length prefix of α. Although the internal states of the atomic register implementations are not known, for any such finite prefix α_k of α it is natural to define the *state of the shared memory after* α_k, or $state(\alpha_k)$, to be the vector $(a_1, ..., a_n)$, where a_i is the value of the last write by process P_i in α_k, or the initial value if P_i has not yet written. If $state(\alpha_k) = (a_1, ..., a_n)$, then $snapshot(\alpha_k)$ denotes $(value(a_1), ..., value(a_n))$. The sequence $snapshot(\alpha_0), snapshot(\alpha_1), snapshot(\alpha_2)...$ serves as the basis for the serialization of α.

The update operations are serialized at the same point in the run as their embedded writes. A *scan*$_i$ operation has a successful double collect when the test in line 3 is passed; following the two collects $\bar{a}$:= *collect* in line 1 and $\bar{b}$:= *collect* in line 2, the sequence numbers in $\bar{a}$ and $\bar{b}$ are identical. Scans with successful double col-

lects are serialized between the end of the collect in line 1 and the beginning of the second collect in line 2. Lemma 3.1 proves that the values returned by such a scan constitute a snapshot during this interval.

Lemma 3.1 *Let $\alpha = \pi_1\pi_2...$ be a run of the unbounded algorithm in which a particular* scan$_i$ *operation has a successful double collect: $\bar{a}$:=* collect *in line 1 and $\bar{b}$:=* collect *in line 2. Let π_u and π_w be the last read of the first collect and the first read of the second collect, respectively. Then for every prefix α_v of α, $u \leq v \leq w$, $snapshot(\alpha_v) = (value(b_1), ..., value(b_n))$.*

Proof: By contradiction. That is, suppose that two successive reads by P_i of r_j in lines 1 and 2 return the same sequence number, and that an update by P_j is serialized between the two reads. Since the update is serialized with its embedded write, a write by P_j to r_j also occurs between the two reads. Furthermore, the sequence number in the second read must be strictly greater than the sequence number in the first read, a contradiction. The lemma follows. ■

The remaining scans return when they observe an updater move twice: they will be serialized in the same interval as the embedded scan. The next lemma guarantees that this interval is contained in the interval of the scan.

Lemma 3.2 *Let $\alpha = \pi_1\pi_2...$ be a run of the unbounded algorithm in which a particular* scan$_i$ *operation observes changes in process P_j's sequence number field during two different double collects. Then the value of r_j read during the last collect was written by a* scan$_j$ *operation that began after the first of the four collects.*

These two lemmas imply that all scans are correctly serialized somewhere in their intervals.

Lemma 3.3 *Let $\alpha = \pi_1\pi_2...$ be a run of the unbounded algorithm in which a particular* scan$_i$ *operation beginning in event π_u returns $(v_1, ..., v_n)$ in event π_w. Then $snapshot(\alpha_v) = (v_1, ..., v_n)$ for some v, $u \leq v \leq w$.*

By the pigeon-hole principle, in $n + 1$ double collects one must be successful or some updater must be observed moving twice. Hence scans are wait-free. This in turn implies that updates are wait-free.

Lemma 3.4 *Every scan or update operation by process P_i returns after $O(n^2)$ atomic steps of P_i, $\forall i \in \{1..n\}$.*

This discussion is summarized in the following theorem.

Theorem 3.5 *The unbounded algorithm implements a wait-free single-writer snapshot memory.*

4 The Bounded Single-Writer Algorithm

The sequence numbers in the unbounded algorithm enable scan operations to detect changes to the memory due to concurrent updates. To achieve the same effect with bounded registers, each scanner/updater pair of processes communicates via two atomic bits, each written by one and read by the other. Before performing a double collect, a scan operation sets its bit equal to the value read in the other bit. If after the double collect, the bits are observed by the scanner to be not equal, then the updater changed its bit (moved) after the scanner's first read of that bit.

Specifically, the bounded single-writer algorithm of Figure 3 replaces the unbounded sequence number field of r_i with n pairs of *handshake* bits [P83, L86b]. That is, for each process pair (P_i, P_j) the register r_i contains the bit field $p_{i,j}$. Additional atomic single-writer single-reader bits $q_{i,j}$ are written by P_i and read by P_j. The $q_{i,j}$ bits are written when P_i scans, (to the values read from the $p_{j,i}$ bits) and the $p_{i,j}$ bits are written when P_i updates, (to the negations of the values read from the $q_{j,i}$ bits). An additional *toggle bit*, $toggle(r_i)$, is changed during every update, to ensure that each write operation changes the register value.

```
procedure scan_i
begin
 0:  for j = 1 to n do moved_j := 0 od;
 0.5: for j = 1 to n do q_{i,j} := p_{j,i}(r_j) od;
                        /* Handshake. */
 1:  ā := collect;
/* (value, bit vector, bit, view) tuples */
 2:  b̄ := collect;
 3:  if (∀j ∈ {1..n}),
              (p_{j,i}(a_j) = p_{j,i}(b_j) = q_{i,j}
         and toggle(a_j) = toggle(b_j)) then
                  /* Nobody moved. */
 4:     return (value(b_1), ..., value(b_n));
 5:  else for j = 1 to n do
 6:        if p_{j,i}(a_j) ≠ q_{i,j} or p_{j,i}(b_j) ≠ q_{i,j}
              or toggle(a_j) ≠ toggle(b_j) then
                        /* P_j moved */
 7:           if moved_j = 1 then
          /* P_j moved once before! */
 8:              return (view(b_j));
 9:           else moved_j := moved_j + 1 ;
              od;
 10: goto line 0.5 ;
end scan_i;

procedure update_i (value)
begin
   0: for j = 1 to n do f_j := ¬q_{j,i} od;
      /* Collect handshake values. */
   1: s̄ := scan_i; /* Embedded scan. */
   2: r_i := (value, f̄, ¬toggle(r_i), s̄) ;
end update_i;
```

Figure 3: The bounded single-writer algorithm.

4.1 Correctness Proof

For this algorithm, a *successful double collect* is a pair $\bar{a} :=$ *collect*; $\bar{b} :=$ *collect*; with all handshake bits $p_{j,i} = q_{i,j}$ and corresponding toggle bits in $\bar{a}$ and $\bar{b}$ identical. The main issue that has to be argued is that the handshake and toggle bits guarantee that a successful double collect produces a snapshot. This is proven in the following lemma.

Lemma 4.1 *Let $\alpha = \pi_1\pi_2...$ be a run of the bounded algorithm in which a particular* scan$_i$ *operation has a successful double collect: $\bar{a} :=$* collect *in line 1 and $\bar{b} :=$* collect *in line 2. Let π_u and π_w be the last read in line 1 and the first read of line 2, respectively. Then for every prefix α_v of α, $u \leq v \leq w$, $snapshot(\alpha_v) = (value(b_1), ..., value(b_n))$.*

Proof: As in the proof of Lemma 3.1, the proof is by contradiction. That is, suppose that two successive reads by P_i of r_j in a collect pair produce values of $p_{j,i}(r_j)$ that are equal to $q_{i,j}$'s most recently written value, and identical $toggle(r_j)$'s. Assume that a write by P_j to r_j is serialized between these two atomic read operations. Consider the last such write operation by P_j; being last, it must write the same handshake bit b and toggle bit t read by P_i. Since during an update P_j assigns to $p_{j,i}$ the negation of the value read in $q_{i,j}$, that read of $q_{i,j}$ must have preceded P_i's most recent write to $q_{i,j}$ of b. This implies the following sequence of events:

$read_j(q_{i,j} = \neg b)$,
/* update: handshake read */
$write_i(q_{i,j} = b)$,
/* scan: handshake write */
$read_i(p_{j,i}(r_j) = b, toggle(r_j) = t)$
/* scan: first collect */
$write_j(p_{j,i}(r_j) = b, toggle(r_j) = t)$
/* update: write */
$read_i(p_{j,i}(r_j) = b, toggle(r_j) = t)$.
/* second: second collect */

The first operation, the read by P_j, is a part of the same update as the later write by P_j, which by assumption is the last write by P_j serialized

between the two reads by P_i. It follows that no other write operation by P_j can be serialized between P_i's two reads. Then the two reads by P_i of r_j return values written by two successive writes by P_j, yet the toggle bits are identical, a contradiction. (The first of these writes by P_j does not appear in the sequence above: it is P_j's most recent previous write, and must precede the first event of the sequence, $read_j(q_{i,j} = \neg b)$.) Hence, no write operation by P_j can be serialized between P_i's two reads, and the claim follows. ■

The serialization, remaining lemmas and theorem from the unbounded algorithm translate directly to the bounded algorithm. (It is important that each update operation changes the *value*, *handshake* and *toggle* fields in a single atomic write operation.)

Lemma 4.2 *Let $\alpha = \pi_1\pi_2...$ be a run of the bounded algorithm in which a particular* scan_i *operation observes changes in process P_j's handshake or toggle bits during two different double collects. Then the value of r_j read during the last collect was written by a* scan_j *operation that began after the first of the four collects.*

Lemma 4.3 *Let $\alpha = \pi_1\pi_2...$ be a run of the bounded algorithm in which a particular* scan_i *operation beginning in event π_u returns $(v_1, ..., v_n)$ in event π_w. Then $snapshot(\alpha_v) = (v_1, ..., v_n)$ for some v, $u \leq v \leq w$.*

Lemma 4.4 *Every scan or update operation by process P_i returns after $O(n^2)$ atomic steps of P_i, $\forall i \in \{1..n\}$.*

Theorem 4.5 *The bounded algorithm implements a wait-free single-writer snapshot memory.*

5 The Bounded Multi-writer Algorithm

Because processes may now write to any memory location, the handshake bits and *view* fields are uncoupled from the *value* fields. The latter are stored in multi-writer, multi-reader registers r_k, where now the index k is a memory address not related to process indices. To ensure that each successive write to these registers has an observable effect, an *id* field and *toggle* bit field are also included: successive update operations by P_i to word k write i in the $id(r_k)$ field and alternate values in the *toggle* field. (The *id* field also allows a scan operation to attribute an observed change to a specific process.)

Because the handshake bits are not written atomically with the r_k registers, a scan may observe changes by the same update operation twice: once changing the handshake bits, and once changing the value of a memory word. Hence, a scan operation must observe process P_j move *three* times before the value in $view_j$ can be borrowed.

Hence, the algorithm of Figure 4 requires a multi-writer multi-reader register r_k for every memory address $k \in \{1, \ldots, m\}$, holding fields $value(r_k)$, $id(r_k)$ and $toggle(r_k)$ of type $Value$, $\{1..n\}$, and boolean. In addition, for every process P_i there are $2n$ single-writer multi-reader boolean registers $p_{i,j}$ and $q_{i,j}$, $\forall j \in \{1..n\}$, and a single-writer multi-reader register $view_i$, holding a vector of m entries each of type *Value*. The scan and update operations of a process i are described in Figure 4.

5.1 Correctness Proof

The serialization is defined as in the previous algorithms, with updates serialized with the (atomic) writes to the value registers. For this algorithm, a *successful double collect* occurs when the test in line 3 is passed. This test depends on steps 0.5 through 2.5, recording the handshake bits and the shared registers r_j twice: Step 0.5 implicitly collects the values of each $p_{j,i}$, by storing $p_{j,i}$ in $q_{i,j}$. The next three lines explicitly record the values of the r_k registers and the handshake bits in $\bar{a}$, $\bar{b}$ and $\bar{h}$, respectively. The test is passed if the handshake bits and *id*, *toggle* fields of the registers contain identical values in each pair of respective reads. Again, the main issue that has to be argued is that a successful double collect produces a snapshot.

```
procedure scan_i
begin
    0:   for j = 1 to n do moved_j := 0 od;
    0.5: for j = 1 to n do q_{i,j} := p_{j,i} od;
                        /* Handshake. */
    1:   ā := collect(r_k : k ∈ {1,...,m}) ;
           /* (value, id, bit) triples */
    2:   b̄ := collect(r_k : k ∈ {1,...,m}) ;
    2.5: h̄ := collect(p_{j,i} : j ∈ {1..n}) ;
                    /* handshake bits */
    3:   if (∀j ∈ {1..n}) (q_{i,j} = h_j)
           and (∀k ∈ {1,...,m})
             (id(a_k) = id(b_k))
           and (∀k ∈ {1,...,m})
             (toggle(a_k) = toggle(b_k)) then
                /* Nobody moved. */
    4:       return (value(b_1),...,value(b_m));
    5:   else for j = 1 to n do
    6:     if ( (q_{i,j} ≠ h_j)
            or ( (∃k, id(b_k) = j)
              (id(a_k) ≠ id(b_k)
              or toggle(a_k) ≠ toggle(b_k))))
           then          /* P_j moved */
    7:       if moved_j = 2 then
           /* P_j moved twice before! */
    8:         return (view_j);
    9:       else moved_j := moved_j + 1 ;
           od;
    10:  goto line 1 ;
    end scan_i;

procedure update_i (k,value)
/* Process P_i writes value to word k */
begin
    0: for j = 1 to n do p_{i,j} := ¬q_{j,i} od;
                        /* Handshake. */
    1: view_i := scan_i; /*Embedded scan:*/
 /* view_i is a single-writer register */
    1.5: t_k := ¬t_k;
/* local variable t̄ saved between calls */
    2: r_k := (value, i, t_k) ;
     /* r_k is a multi-writer register */
end update_i;
```

Figure 4: The bounded multi-writer algorithm.

Lemma 5.1 *Let $\alpha = \pi_1\pi_2...$ be a run of the bounded multi-writer algorithm in which a particular* scan$_i$ *operation has a successful double collect, including $\bar{a}$:=* collect *in line 1 and $\bar{b}$:=* collect *in line 2. Let π_u and π_w be the last read of line 1 and the first read of line 2, respectively. Then for every prefix α_v of α, $u \leq v \leq w$, $snapshot(\alpha_v) = (value(b_1),...,value(b_m))$.*

Proof: As in the proofs of Lemmas 4.1 and 3.1, the proof is by contradiction. Suppose then that two successive reads by P_i of r_k both produce the values $id(r_k) = j$ and $toggle(r_k) = t$, and the two reads of $p_{j,i}$ also produce the same value, c. Assume that an update to word k and hence a write to r_k is serialized between the two atomic reads of r_k in lines 1 and 2. Consider the last such write operation: because the second read by P_i returned $id(r_k) = j$, this last write is by P_j. Since the first read by P_i also returned $id(r_k) = j$ and the same *toggle* value t, there must be another intervening write by P_j to r_k, with *toggle* value $\neg t$, serialized between the two reads by P_i. It follows that the last write by P_j is part of an update that began after P_i's first read of r_k. Within that update, $p_{j,i}$ is set to $\neg q_{i,j}$. Henceforth, the value of $p_{j,i}$ cannot change until $q_{i,j}$ does, so the last read by P_i of $p_{j,i}$ recorded in h_j must see it equal to $\neg q_{i,j}$, a contradiction. Hence, no writes can be serialized between the two reads of r_k.

The full sequence of atomic events constructed in this argument is as follows:

$read_i(p_{j,i} = c)$,
/* P_i's first handshake collect */
$write_i(q_{i,j} = c)$,
/* P_i's handshake write */
$read_i(id(r_k) = j, toggle(r_k) = t)$
/* P_i's first r_k collect */
$write_j(id(r_k) = j, toggle(r_k) = \neg t))$
/* P_j's toggle bit write */
$read_j(q_{i,j} = c)$
/* P_j's handshake read for second write */
$write_j(p_{j,i} = \neg c)$
/* P_j's handshake write for second write */
$write_j(id(r_k) = j, toggle(r_k) = t))$
/* P_j's assumed write */
$read_i(id(r_k) = j, toggle(r_k) = t))$
/* P_i's second r_k collect */
$read_i(p_{j,i} = c)$,
/* P_i's second handshake collect */

It follows that a scanner with a successful double collect can conclude that no writes are serialized between the last read in line 2 and the first read in line 3. Hence, the values read are a snapshot, and the lemma follows. ■

The previous lemma says that the scans with successful double collects can be serialized correctly. It remains to argue that the scans which return borrowed values use values from scans that run entirely within their interval. As discussed, the crucial embedded scan lemma must make concession to the non-atomicity of writes to the handshake and value registers.

Lemma 5.2 *Let $\alpha = \pi_1\pi_2...$ be a run of the bounded multi-writer algorithm in which a particular* scan$_i$ *operation detects changes in process P_j's handshake bit or writes by P_j to value registers during three different double collects. Then the value of $view_j$ read by P_i after the last collect was returned by a* scan$_j$ *operation that began after the first of the six collects.*

Proof: The proof of this lemma rests on the sequence of relevant atomic write steps that P_j makes in successive updates:

write to $p_{j,i}$
write to $view_j$
write to r_{k_1}
write to $p_{j,i}$
write to $view_j$
write to r_{k_2}
.
.
.

Observing any three changes, in the $p_{j,i}$ or value registers, means that an intervening scan must have taken place and have been recorded in $view_j$. Either this scan or a more recent scan by P_j will be read by P_i. ■

These two lemmas imply:

Lemma 5.3 *Let $\alpha = \pi_1\pi_2...$ be a run of the bounded multi-writer algorithm in which a particular* scan$_i$ *operation beginning in event π_u returns $(v_1, ..., v_n)$ in event π_w. Then $snapshot(\alpha_v) = (v_1, ..., v_n)$ for some v, $u \leq v \leq w$.*

As before, the pigeon-hole principle implies that in $2n + 1$ double collects one must be successful or some updater must be observed moving three times. Hence scans are wait-free. This in turn implies that updates are wait-free.

Theorem 5.4 *The bounded multi-writer algorithm implements a wait-free multi-writer snapshot memory.*

6 Discussion and Directions for Further Research

The *distributed snapshot* of Chandy and Lamport [CL85] provides a simple solution to the similar problem for message-passing systems. The distributed snapshot algorithm has proven a useful tool in solving other distributed problems (see, e.g., [G86, BT84]), and it is likely snapshot memories will play a similar role in concurrent programming.

Interestingly, distributed snapshots are not true instantaneous images of the global state, such as scans of snapshot memories produce. However, distributed snapshots are indistinguishable, within the system itself, from true instantaneous images. By applying the emulators of [ABD] to the constructions presented in this paper, implementations of atomic snapshot memory are obtained in message-passing systems. Snapshots obtained this way are true instantaneous images of the global state. In addition, these implementations are resilient to process and link failures, as long as a majority of the system remains connected.

Anderson [A89a, An90] has obtained, independently, bounded implementations of single-writer atomic snapshots. Memory operations in Anderson's implementation of the single-writer snapshot memory perform $\Theta(2^n)$ reads and writes to atomic single-writer multi-reader registers, in the worst case.

Anderson originally posed the multi-writer snapshot problem, and uses single-writer atomic snapshots to construct multi-writer atomic snapshots [A89b, An90]. Together with the bounded single-writer algorithm of this paper, this provided the first polynomial construction of a shared memory object that can be instantaneously checkpointed. The multi-writer algorithm of this paper gives an alternative implementation, building instead on multi-writer atomic registers. The efficiency of these constructions may be compared by considering two compound constructions, tracing back to operations on single-writer atomic registers. Anderson's multi-writer algorithm, based on the bounded single-writer algorithm of this paper, requires $\Theta(n^4)$ single-writer operations per update or scan operation in the worst case. Our multi-writer algorithm, based on multi-writer registers, in turn implemented from single-writer registers, requires $\Theta(n^3)$ single-writer operations per update or scan operation in the worst case (using the most efficient known construction of multi-writer registers from single-writer, due to Li, Tromp and Vitanyi [LTV89]). It is interesting to speculate whether other, more efficient solutions can be found.

Indeed, an interesting open question is the inherent complexity of implementing atomic snapshots, in terms of both time and space. In all known bounded algorithms the scanners write to the updaters–is this necessary? The scans do a large number of reads–is this also necessary?

Another question is to find other applications for atomic snapshots, in addition to the ones described.

The most challenging avenue of research seems to be the relation between the power of unbounded and bounded wait-free algorithms. Can any primitive that is not syntactically unbounded[†] be implemented using bounded shared memory? Specifically, is there a uniform transformation of any unbounded wait-free solution for some problem into a bounded wait-free solution? Even a precise definition of this class of problems is not obvious.

Finally, snapshot memories, though apparently more powerful than registers, nevertheless have bounded wait-free implementations from those simple primitives. Herlihy showed that many interesting primitives do not have wait-free implementations from registers [H88]. Is it possible to "close the gap" further, and construct yet more powerful primitives from registers? More ambitiously, is it possible to construct a hierarchy of objects implementable from atomic registers, providing a theoretical basis for the intuition that snapshot memories are more powerful single-writer registers?

Acknowledgements: The authors would like

[†]Clearly, procedures that return integer or other unbounded values will not have bounded implementations.

to thank Maurice Herlihy and Nancy Lynch for helpful discussions.

References

[A88] K. Abrahamson, "On Achieving Consensus Using a Shared Memory," *Proc. 7th ACM Symp. on Principles of Distributed Computing*, 1988, pp. 291–302.

[A89a] J. H. Anderson, "Composite Registers," TR-89-25, Department of Computer Science, The University of Texas at Austin, September 1989.

[A89b] J. H. Anderson, "Multiple-Writer Composite Registers," TR-89-26, Department of Computer Science, The University of Texas at Austin, September 1989.

[An90] J. H. Anderson, "Composite Registers," *Proc. 9th ACM Symp. on Principles of Distributed Computing*, 1990, this proceedings.

[AG88] J. H. Anderson, and M. G. Gouda, "The Virtue of Patience: Concurrent Programming With and Without Waiting," unpublished manuscript, Dept. of Computer Science, Austin, Texas, January 1988.

[AH89] J. Aspnes, and M. P. Herlihy, "Fast Randomized Consensus using Shared Memory," *Journal of Algorithms*, September 1990, to appear.

[A90] J. Aspnes, "Time- and Space-Efficient Randomized Consensus," *Proc. 9th ACM Symp. on Principles of Distributed Computing*, 1990, this proceedings.

[AH90] J. Aspnes and M. P. Herlihy "Wait-Free Data Structures in the Asynchronous PRAM Model," *Proc. of the 2nd Annual Symposium on Parallel Algorithms and Architectures*, July 1990, Crete, Greece, to appear.

[ABD] H. Attiya, A. Bar-Noy and D. Dolev, "Sharing Memory Robustly in Message-Passing Systems," *Proc. 9th ACM Symp. on Principles of Distributed Computing*, 1990, this proceedings.

[ADS89] H. Attiya, D. Dolev, and N. Shavit, "Bounded Polynomial Randomized Consensus," *Proc. 8th ACM Symp. on Principles of Distributed Computing*, 1989, pp. 281–293.

[BT84] G. Bracha, and S. Toueg, "A Distributed Algorithm for Generalized Deadlock Detection" *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, 1984, pp. 285–301.

[Blo87] B. Bloom, "Constructing Two-Writer Atomic Registers," *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 1987, pp. 249–259.

[CL85] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computing Systems*, pages 63–75, February 1985.

[DGS88] D. Dolev, E. Gafni, and N. Shavit, "Toward a Non-Atomic Era: ℓ-Exclusion as a Test Case," *Proc. 20th Annual ACM Symp. on the Theory of Computing*, 1988, pp. 78–92.

[DS89] D. Dolev, and N. Shavit, "Bounded Concurrent Time-Stamp Systems Are Constructible!" *Proc. 21th Annual ACM Symp. on Theory of Computing*, 1989, pp. 454–465.

[G86] E. Gafni, "Perspective on Distributed Network Protocols: A Case for Building Blocks," *MILCOM '86,* October 1986, Monterey, California.

[H88] M. P. Herlihy, "Wait Free Implementations of Concurrent Objects," *Proc. 7th ACM Symp. on Principles of Distributed Computing*, 1988, pp. 276–290.

[HW87] M. P. Herlihy and J. M. Wing, "Axioms for Concurrent Objects," *14th ACM Symp. on Principles of Programming Languages*, January 1987, pp. 13–25.

[K78] H. P. Katseff, "A New Solution to the Critical Section Problem," *Proc. 10th Annual ACM Symp. on the Theory of Computing*, 1978, pp. 86–88.

[L86a] L. Lamport, "On Interprocess Communication. Part I: Basic Formalism," *Distributed Computing 1, 2* 1986, 77–85.

[L86b] L. Lamport, "On Interprocess Communication. Part II: Algorithms," *Distributed Computing 1, 2* 1986, pp. 86–101.

[L86c] L. Lamport, "The Mutual Exclusion Problem. Part II: Statement and Solutions," *J. ACM 33, 2* 1986, pp. 327–348.

[LT87] N. Lynch and M. Tuttle. "Hierarchical Correctness Proofs for Distributed Algorithms." *Proc. 6th ACM Symp. on Principles of Distributed Computation*, 1987, pp. 137–151. Expanded version: Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA., April 1987.

[LTV89] M. Li, J. Tromp and P. M.B. Vitanyi, "How to Share Concurrent Wait-Free Variables," *ICALP 1989.* Expanded version: Report CS-R8916, CWI, Amsterdam, April 1989.

[M86] J. Misra, "Axioms for Memory Access in Asynchronous Hardware Systems," *ACM Transactions on Programming Languages and Systems,* Vol. 8, No. 1 (January 1986), pp. 142–153.

[OG76] S. Owicki and D. Gries, "An Axiomatic Proof Technique for Parallel Programs," *Acta Informatica,* 6(1):319–340, 1976.

[Owi75] S. Owicki, *Axiomatic Proof Techniques for Parallel Programs.* PhD thesis, Cornell University, August 1975.

[P83] G. L. Peterson, "Concurrent Reading While Writing," *ACM Transactions on Programming Languages and Systems,* Vol. 5, No. 1 (January 1983), pp. 46–55.

[PB87] G. L. Peterson, and J. E. Burns, "Concurrent Reading While Writing II : The Multi-Writer Case," *Proc. 28th Annual IEEE Symp. on Foundations of Computer Science,* 1987, pp. 383–392.

[S88] R. Schaffer, "On the Correctness of Atomic Multi-Writer Registers," MIT/LCS/TM-364, June 1988.

[VA86] P. M. B. Vitanyi and B. Awerbuch, "Atomic Shared Register Access by Asynchronous Hardware," *Proc. 27th IEEE Annual Symp. on Foundations of Computer Science*, pp. 233–243, 1986.