Atomic Snapshots of Shared Memory

YEHUDA AFEK
Tel-Aviv University, Tel-Aviv, Israel and AT&T Bell Laboratories, Murray Hill, New Jersey

HAGIT ATTIA
Technion, Haifa, Israel

DANNY DOLEV
Hebrew University, Jerusalem, Israel and IBM Almaden Research Center, San Jose, California

ELI GAFNI
Tel-Aviv University, Tel-Aviv, Israel and University of California at Los Angeles, Los Angeles, California

MICHAEL MERRITT
AT&T Bell Laboratories, Murray Hill, New Jersey

AND NIR SHAVIT
IBM Almaden Research Center, San Jose, California and Stanford University, Stanford, California

Abstract. This paper introduces a general formulation of atomic snapshot memory, a shared memory partitioned into words written (updated) by individual processes, or instantaneously read (scanned) in its entirety. This paper presents three wait-free implementations of atomic snapshot memory. The first implementation in this paper 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 from atomic $n$-writer, $n$-reader registers, again echoing key ideas from the earlier constructions. All operations require $\Theta(n^2)$ reads and writes to the component shared registers in the worst case.

Categories and Subject Descriptors: B.3.2 [Memory Structures]: Design Styles—shared memory; C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors)—multiple-instruction-stream, multiple-data-stream processors (MIMD); D.4.1 [Operating Systems]: Process Management—concurrency, multiprocessing/multiprogramming, synchronization

General Terms: Shared Memory, Algorithms, Concurrency

Additional Key Words and Phrases: Fault-tolerance, Atomic, Snapshot, Consistent State
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 [26, 27]. 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 [19, 14, 20], construction of atomic multi-writer multi-reader registers [31, 29, 30, 23], concurrent time-stamp systems [15], approximate agreement [11], randomized consensus [1, 7, 10, 6] and wait-free implementation of data structures [8].

This paper introduces a general formulation of atomic snapshot memory, a 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 atomic registers. Anderson independently introduces the same notion and presents bounded implementations [3, 4, 5]. Section 6 discusses relationships between the various implementations. The first implementation in this paper 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 ([4]) from 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 [17]. The fact that wait-free atomic snapshot memories can be implemented from atomic registers stands in contrast to the


Il. Attiya's and N. Shavit's research was partially supported by National Science Foundation grant CCR-8611442, by Office of Naval Research contract N00014-85-K-0168, and by DARPA contracts N00014-83-K-0125 and N00014-89-J-1988. E. Gafni's research was partially supported by National Science Foundation grant DCR84-51396 and XEROX Co. grant W881111. Part of this work was done while N. Shavit was at Hebrew University, Jerusalem, visiting AT&T Bell Laboratories and the Theory of Distributed Systems group at Massachusetts Institute of Technology, and while Il. Attiya was at the Laboratory for Computer Science at Massachusetts Institute of Technology.

Authors' present addresses: Y. Afek, Computer Science Department, Tel-Aviv University, Ramat-Aviv Israel 69978; Il. Attiya, Department of Computer Science, Technion, Haifa, Israel 32000; D. Dolev, Department of Computer Science, Hebrew University, Jerusalem Israel 91904; E. Gafni, 3752 Boehler Hall, Computer Science Department, U.C.L.A. Los Angeles, California 90024; M. Merritt, 600 Mountain Ave., Murray Hill, NJ 07974; N. Shavit, Laboratory for Computer Science, MIT NE43, 367 Technology Square Cambridge MA 02139.
impossibility results in [17]. The construction of atomic snapshot memories (and data objects that can be built using them) sheds some light on the borderline between what can and what can not be implemented from atomic registers.

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* and *update* by each process *P*<sub>i</sub>, *i* ∈ {1..*n*}. The *scan* operation has no arguments and returns a vector of *n* elements from an arbitrary set of data values. The *update* operation takes a data value as an argument and does not return a value. 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, the call by the process and the return by the memory, so that the "serialization sequence" of such atomic events satisfies the natural semantics. That is, each *scan* operation returns a vector *d* of data values such that each *d*<sub>k</sub> is the argument of the last *update* to word *k* that is serialized before that *scan*. (This variant of serializability is called "linearizability" [18].) This intuition is made precise in the following subsection.

Two further restrictions are imposed on implementations of atomic snapshot memories. The first restriction can be described as the *architectural restrictions* imposed on solutions (cf. [22, 17]), and requires that any snapshot implementation 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., [29, 23].

The second restriction imposed on snapshot memory implementations is that they satisfy the property of *wait-freedom* [21, 28]. That is, every snapshot operation by process *P*<sub>i</sub> will terminate, regardless of the behavior of other processes, assuming only that local steps of *P*<sub>i</sub> and operations on embedded shared objects terminate. The reader is referred to [21, 17, 2] for discussions and proposed definitions of wait-freedom. The *update* and *scan* operations implemented in this paper require at most *Θ*(*n*<sup>2</sup>) local operations and reads and writes to the component shared registers. They are thus wait-free under any of the proposed definitions.

The next two subsections give automata-based formal specifications of snapshot memories. These specifications do not include the architectural restrictions described above. Including them
would be straightforward, though tedious—the interested reader is referred to [17]. Alternative approaches to specifying concurrent objects are via their serial specification [18] or as a set of axioms (cf. [21, 25]). Axiomatic specifications for snapshot memories appear in [3, 4, 10].

## 2.1 Specification of Single-Writer Snapshot Memories

Following [24, 17], a single-writer atomic snapshot memory for \( n \) processes and a particular data set \( Data \) is an automaton with two types of input \( \text{Request} \) actions: \( \text{UpdateRequest}_i(d) \) and \( \text{ScanRequest}_i \), and two types of output \( \text{Return} \) actions: \( \text{UpdateReturn}_i \) and \( \text{ScanReturn}_i(d_1, \ldots, d_n) \), for any \( i \in \{1, \ldots, n\} \), and for all \( d, d_1, \ldots, d_n \in Data \). (In brief, the actions are labels on state transitions, and input actions must be enabled from every state—the snapshot memory cannot prevent a process from issuing a \( \text{Request} \), and the process cannot prevent the memory from issuing a \( \text{Return} \). Automata interact by identifying common actions.) The \( \text{Request} \) and \( \text{Return} \) actions are called the \textit{interface snapshot actions}. Intuitively, the environment requests (calls) operations by issuing input actions, and the algorithm returns answers using output actions. Formally, the environment may be modeled as \( n \) processes, automata \( P_1, \ldots, P_n \), with the snapshot memory input and output actions as complementary output and input actions.

The formal specification of single-writer snapshot memory is based on a particular automaton, the \textit{canonical single-writer snapshot automaton}. That is, a correct implementation \( S \) of a single-writer snapshot memory is one which the processes cannot distinguish from the canonical automaton. If the processes interact with \( S \), the resulting \textit{behavior}, or sequence of interface actions, is one which could occur when interacting with the canonical automaton.

In addition to the interface snapshot actions, the canonical automaton has two types of internal actions, \( \text{Update}_i(d) \), and \( \text{Scan}_i(d_1, \ldots, d_n) \), for any \( i \in \{1, \ldots, n\} \) and for all \( d, d_1, \ldots, d_n \in Data \). The states of the canonical automaton contain an \( n \)-entry array \( \text{Mem} \) of type \( Data \) and \( n \) interface variables \( H_i \). The interface variables may hold as value any of the interface snapshot actions, or a special value \( \bot \).

<table>
<thead>
<tr>
<th>( \text{UpdateRequest}_i(d) )</th>
<th>( \text{Effect:} \quad H_i := \text{UpdateRequest}_i(d) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Update}_i(d) )</td>
<td>( \text{Precondition:} \quad H_i = \text{UpdateRequest}_i(d) )</td>
</tr>
<tr>
<td></td>
<td>( \text{Effect:} \quad \text{Mem}[i] := d )</td>
</tr>
<tr>
<td></td>
<td>( H_i := \text{UpdateReturn}_i )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \text{ScanRequest}_i )</th>
<th>( \text{Effect:} \quad H_i := \text{ScanRequest}_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Scan}_i(d_1, \ldots, d_n) )</td>
<td>( \text{Precondition:} \quad H_i = \text{ScanRequest}_i )</td>
</tr>
<tr>
<td></td>
<td>( \text{Effect:} \quad \text{Mem} = (d_1, \ldots, d_n) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \text{UpdateReturn}_i )</th>
<th>( \text{Precondition:} \quad H_i = \text{UpdateReturn}_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Effect:} \quad H_i := \bot )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \text{ScanReturn}_i(d_1, \ldots, d_n) )</th>
<th>( \text{Precondition:} \quad H_i = \text{ScanReturn}_i(d_1, \ldots, d_n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Effect:} \quad H_i := \bot )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: The canonical single-writer snapshot automaton.
Process $P_i$ interacts with the automaton by issuing a request (an $\text{UpdateRequest}_i(d)$ or $\text{ScanRequest}_i$ action). The result is to store the input action in the state variable $H_i$, enabling the appropriate internal action ($\text{Update}_i(d)$ or $\text{Scan}_i(d_1, ..., d_m)$). The internal action in turn assigns an appropriate output action to $H_i$, and in the case of $\text{Update}_i(d)$, assigns $d$ to $\text{Mem}_i$ as well. The change to the interface value $H_i$ enables the appropriate output ($\text{UpdateReturn}_i$ or $\text{ScanReturn}_i(d_1, ..., d_m)$) action). Initially, each $H_i = \perp$ and $\text{Mem}_i = d_{\text{init}} \in \text{Data}$.

The steps of the canonical single-writer snapshot automaton 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 operation of the canonical automaton in $\alpha$ is said to be serialized at the point of its associated $\text{Update}$ or $\text{Scan}$ operation.

The well-formed behaviors of the canonical automaton are those in which no pair of $\text{Request}_i$ inputs occurs without an intervening $\text{Return}_i$ output. Intuitively, this means that each process has only one pending operation at any time. An automaton $S$ preserves well-formedness, provided it is never the first to violate well-formedness—if no process has input two concurrent $\text{Request}$’s, then $S$ will not output redundant $\text{Return}$’s. That is, if $\alpha \pi$ is a finite sequence of interface snapshot actions that is a behavior of $S$, with $\pi$ a single output event and $\alpha$ is well-formed, then $\alpha \pi$ is well-formed.

**Definition 1** An automaton $S$ implements a single-writer atomic snapshot memory (for the appropriate number of processes and data set) if and only if $S$ has the interface snapshot actions as its input and output actions, $S$ preserves well-formedness, and provided every well-formed behavior of $S$ is also a behavior of the canonical single-writer snapshot automaton.

### 2.2 Specification of Multi-Writer Snapshot Memories

Multi-writer snapshot memories are straightforward generalizations of single-writer snapshot memories, and can be specified analogously. Specifically, a multi-writer snapshot memory for $n$ processes, a particular data set $\text{Data}$ and $m$ memory elements is an automaton with input actions: $\text{UpdateRequest}_i(k,d)$, $\text{ScanRequest}_i$, and output actions: $\text{UpdateReturn}_i$, $\text{ScanReturn}_i(d_1, ..., d_m)$, for all $i \in \{1, ..., n\}$, $k \in \{1, ..., m\}$, and $d, d_1, ..., d_m \in \text{Data}$. Call these the multi-writer interface snapshot actions. (Except for the addition of the address field $k$ to the $\text{UpdateRequest}$ actions, and $\text{ScanReturn}$ containing $m$ rather than $n$ values, these are the same as the single-writer interface snapshot actions.) The canonical multi-writer snapshot automaton in Figure 2 is obtained via straightforward modifications of the canonical single-writer snapshot automaton. (The internal $\text{Update}$ action has the additional address field $k$, and the $\text{Scan}$ action specifies $m$ rather than $n$ values.) Well-formedness is defined just as for single-writer memories.
**Definition 2** An automaton $S$ implements a multi-writer atomic snapshot memory (for the appropriate number of processes and data set) if and only if $S$ has the multi-writer interface snapshot actions as its input and output actions, $S$ preserves well-formedness, and provided every well-formed behavior of $S$ is also a behavior of the canonical multi-writer snapshot automaton.

### 2.3 Reasoning about Read/Write Registers

A complete formal specification must describe the details of the lower-level interface, in which processes are permitted to reference local variables and to interact via reads and writes to atomic read/write registers. The specifications of snapshot memories based on canonical automata are examples of a general technique for specifying shared atomic objects. Read/write registers are instances of shared atomic primitives that are almost trivial to specify in this way, in which every operation on these shared primitives is modeled as a Request action input to the register, an internal Read or Write, and a Return action output by the register.

An automaton that satisfies such a specification (that is an implementation of the appropriate canonical automaton) is indistinguishable from the canonical automaton. Thus, it is a valid proof technique to ignore any specific implementation details of the read/write registers, and to assume that these operations occur as atomic actions sometime within the corresponding operation interval, just as happens in the canonical automaton [21, 18, 24].

The sections that follow present the algorithms in familiar pseudo-code style. Translating them into preconditions and effects on appropriately named internal and external actions is a straightforward but tedious exercise.
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. Then 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 [29].

This observation alone supports a simple unbounded algorithm, although one which is not wait-free. The $k$th update by processor $P_i$ simply writes the update value $d$ 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 3.) 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, $r_i.data$ (of type Data), $r_i.seq$ (of type integer) and $r_i.view$ (an array of $n$ Data values). The $data$ field and $n$ entries in the $view$ fields are initialized to $d_{init}$ and the $seq$ fields are initialized to 0.

Each scan operation has a local array $moved$, in which it records, for each other process, whether that process has been observed to change the memory during the course of the scan. The collect operation by any process $i$ reads each register $r_j$, $j \in \{1, \ldots, n\}$, in an arbitrary order (or in parallel), returning an array of records read, indexed by process id.

3.1 Correctness Proof

The proof strategy is to construct an explicit serialization—to construct, from every run of the unbounded algorithm, a run of the canonical snapshot automaton that has the same behavior. That is, given an infinite or finite well-formed run of the unbounded algorithm, 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. Calls to $scan_i$ procedures from within updates are identified with actions $ScanRequest_i^{int}$ and $ScanReturn_i^{int}$ that are internal to the snapshot implementation automaton, but are otherwise treated identically to their external counterparts.
The scan and update operations themselves consist of sequences of more primitive operations that are either manipulations of local data or reads and writes of atomic registers. The former are trivially atomic, and can be modeled as single actions. The latter are atomic by assumption—that is, the atomic registers used by the algorithm are assumed to be implementations of the canonical read/write register automaton. Hence, it suffices to consider runs in which these registers are actually implemented by the specific canonical automata [24].

Hence, an arbitrary run of the unbounded algorithm can be considered to be a (possibly infinite) sequence of interface snapshot actions, local data manipulations, and interface or internal actions of the shared registers. (These are Request actions input to the registers, internal Read or Write actions, and Return actions output by the registers.) Given this sequence, we explicitly identify serialization points for the snapshot operations within each operation interval. That is, we first insert internal Update and Scan actions within the run of the implementation. This is done so that the resulting sequence of interface and internal snapshot actions (ignoring the local data and shared register actions) is a run of the canonical snapshot automaton.

```
procedure scan;
begin
  0: for j = 1 to n do moved[j] := 0 od;
  1: while true do
  2:   a[1..n] := collect; /* (data, seq, view) triples. */
  3:   b[1..n] := collect; /* (data, seq, view) triples. */
  4:   if (∀ j ∈ {1, ..., n}) (a[j].seq = b[j].seq) then
  5:     return (b[1], data,..., b[n], data); /* Nobody moved. */
  6:   else for j = 1 to n do
  7:     if a[j].seq ≠ b[j].seq then /* P_j moved. */
  8:       if moved[j] = 1 then /* P_j moved once before! */
  9:         return (b[j], view);
 10:     else moved[j] := moved[j] + 1;
   od;
  od;
end scan;

procedure update_i (data)
begin
  1: s[1..n] := scan; /* Embedded scan */
  2: r_i := (data, r_i, seq+1, s[1..n])
end update_i;
```

Figure 3: The unbounded single-writer algorithm.

Consider then any sequence α = π_1 π_2,..., where each π_j is either an interface snapshot action, a local computation event, a Request or Return for a shared register, an internal action Read_i(r_j = v) by P_i of atomic register r_j returning v, or an internal write Write_i(r_i = v) by P_i of v to r_i. Denote by α_k the k-length prefix of α. 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(α_k), to be the vector (v_1,..., v_n), where v_i is the value
of the last write by process \( P_i \) in \( \alpha_k \), or the initial value if \( P_i \) has not yet written. (These are the values of the relevant state components of the embedded registers, as implemented by the canonical automata.) If \( \text{state}(\alpha_k) = (v_1, ..., v_n) \), then \( \text{snapshot}(\alpha_k) \) denotes \((v_1, \text{data}, ..., v_n, \text{data})\). As indicated, the sequence \( \text{snapshot}(\alpha_0), \text{snapshot}(\alpha_1), \text{snapshot}(\alpha_2) \) ... serves as the basis for the serialization of \( \alpha \).

The update operations are serialized at the same point in the run as their embedded writes. (That is, Update actions are inserted into the sequence at this point. No Update action is inserted for an incomplete update that has not yet written its register.) A scan\(_i\) operation has a successful double collect when the test in line 4 is passed. That is, following the two collects \( a[1..n] := \text{collect} \) in line 2 and \( b[1..n] := \text{collect} \) in line 3, the sequence numbers in \( a[1..n] \) and \( b[1..n] \) are identical. Those scans with successful double collects are serialized between the end of the first collect in line 2 and the beginning of the second collect in line 3. (Specifically, a Scan action is inserted between the last Return action from the \( n \) shared registers read in the first collect, and the first Request action to the \( n \) shared registers read in the second collect.) Lemma 3.1 proves that the values returned by such a scan constitute a snapshot during this interval.

**Lemma 3.1** Let \( \alpha = \pi_1 \pi_2 \ldots \) be a run of the unbounded algorithm in which a particular scan\(_i\) operation has a successful double collect: \( a[1..n] := \text{collect} \) in line 2 and \( b[1..n] := \text{collect} \) in line 3. Let \( \pi_u \) and \( \pi_v \) be the last Read of the first collect and the first Read of the second collect, respectively. Then for every prefix \( \alpha_v \) of \( \alpha \), \( u \leq v \leq w \), \( \text{snapshot}(\alpha_v) = (b[1], \text{data}, ..., b[n], \text{data}) \).

**Proof:** Suppose a write by \( P_j \) to \( r_j \) is serialized between two successive reads by \( P_i \) of \( r_j \) in lines 2 and 3. Since the sequence number in \( r_j \) is incremented with each write, the sequence number returned by the second read will be strictly greater than that returned by the first. It follows that if the sequence numbers are not observed to change, no write by \( P_j \) is serialized between the successive reads. This implies the result. ■

Alternatively, a scan may return when it observes an updater move twice: it will be serialized just after the serialization point of the embedded scan. The next lemma guarantees that the embedded scan is entirely contained in the interval of the enclosing scan.

**Lemma 3.2** Let \( \alpha = \pi_1 \pi_2 \ldots \) 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 an update\(_j\) operation that began after the first of these four collects started.

**Proof:** If two successive reads by \( P_i \) of \( r_j \) in lines 2 and 3 return different sequence numbers, then at least one write by \( P_j \) to \( r_j \) is serialized between the two reads. If a second pair of successive reads by \( P_i \) of \( r_j \) in lines 2 and 3 return different sequence numbers, then at least one other write by \( P_j \) to \( r_j \) is serialized between this pair of reads. Process \( P_j \) writes to \( r_j \) only as the final step of each update\(_j\) operation. Hence, one update\(_j\) operation ended sometime after the first read by \( P_i \), and the write step of another occurs between the last pair of reads by \( P_i \). Since update\(_j\) operations run serially (only one UpdateRequest\(_j\) is outstanding at a time), the lemma follows. ■
These two lemmas imply that all scans can be correctly serialized somewhere in their intervals.

**Lemma 3.3** Let \( \alpha = \pi_1 \pi_2 \ldots \) be a run of the unbounded algorithm in which a particular scan operation beginning in event \( \pi_u \) returns \( (d_1, \ldots, d_n) \) in event \( \pi_v \). Then \( \text{snapshot}(\alpha_v) = (d_1, \ldots, d_n) \) for some \( v, u \leq v \leq w \).

**Proof**: If the scan \( \alpha \) operation has a successful double collect, the result follows from Lemma 3.1. Assume instead the scan \( \alpha \) operation borrows a snapshot value read in \( r_j \). By Lemma 3.2, the snapshot value read in \( r_j \) was obtained by a scan \( \beta \) operation, embedded in an update \( \gamma \) operation, which in turn started after the first read by \( P_i \) of \( r_j \) and wrote before the last read by \( P_i \) of \( r_j \). Hence the interval of the embedded scan \( \beta \) is contained between the first and last reads by \( P_i \) of \( r_j \). Either the scan \( \beta \) operation had a successful double collect, and the result again follows from Lemma 3.1, or there is another embedded scan \( \alpha \), occurring entirely within the interval of the scan \( \beta \) operation, from which \( P_i \) borrowed. This argument can be applied inductively, noting that there can be at most \( n \) concurrent operations in the system. Hence, eventually the embedded scan must have succeeded via a successful double collect, and the result follows by Lemma 3.1 and transitivity of containment of the embedded scan intervals.

By Lemma 3.3, during the interval of every complete scan operation there is at least one state in which the data values returned were simultaneously held in all the registers. Each completed scan is serialized at this point. (That is, an internal Scan action is inserted into the sequence after one such state.) The update operations were serialized with their embedded writes and all completed scans have now been serialized. An easy induction suffices to show that the resulting sequence of interface snapshot actions and internal Update and Scan actions is a run of the canonical automaton.

This leaves only the wait-free requirement. 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 \( P_i \), \( \forall i \in \{1, \ldots, n\} \).

This discussion is summarized in the following theorem.

**Theorem 1** 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
procedure \texttt{scan};
begin
0: for \( j = 1 \) to \( n \) do \texttt{moved[\( j \)]} := 0 od;
1: while \texttt{true} do
1.5: for \( j = 1 \) to \( n \) do \texttt{\( q_{i,j} := r_j.p_{i,j} \) od;} /* Handshake. */
2: \texttt{\( a[1..n] := collect; \)} /* (data, bit vector, toggle, view) tuples. */
3: \texttt{\( b[1..n] := collect; \)} /* (data, bit vector, toggle, view) tuples. */
4: if \((\forall j \in \{1, \ldots, n\}), \( (a[j].p_{i,j} = p_{i,j}, b[j] = q_{i,j}\) and \( a[j].\text{toggle} = b[j].\text{toggle} \)) then /* Nobody moved. */
5: \texttt{return \( (b[1].data, \ldots, b[n].data) \);} 
6: else for \( j = 1 \) to \( n \) do 
7: if \( a[j].p_{i,j} \neq q_{i,j} \) or \( b[j].p_{i,j} \neq q_{i,j} \) /* \( P_j \) moved. */
8: or \( a[j].\text{toggle} \neq b[j].\text{toggle} \) then
9: \texttt{\( r_i := (data, f[1..n], \neg r_i, \text{toggle}, s[1..n]); \)} /* \( P_j \) moved once before! */
10: \texttt{\textbf{else} \( \texttt{moved[\( j \)] := \texttt{moved[\( j \)]} + 1 \); \)}
\texttt{\textbf{od}; \)}
\texttt{\textbf{od}; \)}
end \texttt{scan};

procedure \texttt{update}(\texttt{data})
begin
0: for \( j = 1 \) to \( n \) do \texttt{\( f_j := \neg q_{i,j} \) od;} /* Collect handshake values. */
1: \texttt{\( s[1..n] := \texttt{scan}; \)} /* Embedded \texttt{scan}. */
2: \texttt{\( r_i := (data, f[1..n], \neg r_i, \text{toggle}, s[1..n]); \)}
end \texttt{update};

Figure 4: The bounded single-writer algorithm.

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 4 replaces the unbounded sequence numbers with two \textit{handshake} bits per pair of processes [28, 22]. That is, for each process pair \((P_i, P_j)\) the register \( r_i \) contains the bit field \( p_{i,j} \), and additional atomic single-writer single-reader one-bit registers \( q_{i,j} \) are written by \( P_j \) and read by \( P_i \). The \( p_{i,j} \) bits are written when \( P_i \) updates (to the negations of the values read from the \( q_{i,j} \) bits), and the \( q_{i,j} \) bits are written when \( P_j \) scans (to the values read from the \( p_{i,j} \) bits). An additional \textit{toggle} bit, \( r_i.\text{toggle} \), is changed during every \texttt{update}, to ensure that each write operation changes the register value.

4.1 Correctness Proof

For this algorithm, a \textit{successful double collect} is a pair \texttt{\( a[1..n] := collect; b[1..n] := collect; \)} with all handshake bits \( p_{i,j} = q_{i,j} \) and corresponding toggle bits in \( a[1..n] \) and \( b[1..n] \) identical. The
This implies two things, first that the value read in operation considered above, and secondly that any earlier write occurs, it will be observed to have changed the bit read in $r_j,\text{toggle}$. The result follows.

Suppose then that the two successive reads by $P_i$ of $r_j$ both return the value $c$ for $r_j,p_{j,i}$, that $c$ is the value most recently written to $q_{i,j}$, and that these same reads return the values $t_1$ and $t_2$ in $r_j,\text{toggle}$, respectively. Further assume that an update to word $j$, and hence a write to $r_j$ by $P_j$, is serialized between the two atomic reads of $r_j$ in lines 2 and 3. Consider the last such write operation: being last, it must write the handshake value $c$ and toggle value $t_2$ to $r_j,p_{j,i}$ and $r_j,\text{toggle}$ read by the second read of $r_j$ 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 $\text{read}(q_{i,j})$ must have preceded $P_i$'s most recent write to $q_{i,j}$ of $c$. This implies two things, first that the $\text{read}(q_{i,j})$ operation by $P_j$ is part of the same, final update operation considered above, and secondly that any earlier update by $P_j$ must have been finished before the $\text{write}_i(q_{i,j} = c)$. The partial order of events in this discussion is: (The two initial events by $P_i$ and $P_j$ may occur in either order, and are shown on the same line.)

\[
\begin{array}{ll}
P_i \text{ (scan)} & \quad P_j \text{ (update)} \\
\text{read}_i(p_{j,i} = c) & \quad \text{read}_j(q_{i,j} = \overline{c}) \quad /\!\!\!\!/ \text{ Handshake read.} \\
\text{write}_i(q_{i,j} = c) & \quad /\!\!\!\!/ \text{ Handshake.} \\
\text{read}_i(r_j,p_{j,i} = c, r_j,\text{toggle} = t_1) & \quad \text{write}_j(r_j,p_{j,i} = c, r_j,\text{toggle} = t_2) \quad /\!\!\!\!/ \text{ First collect.} \\
\text{read}_i(r_j,p_{j,i} = c, r_j,\text{toggle} = t_2) & \quad /\!\!\!\!/ \text{ Second collect.}
\end{array}
\]

It follows that no other write operation by $P_j$ can be serialized between $P_i$'s final two reads of $r_j$. Then these two reads by $P_i$ of $r_j$ return values written by two successive writes by $P_j$, so the toggle bit values returned must be different, $t_1 \neq t_2$. (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 operation by $P_j$, the $\text{read}_j(q_{i,j} = \overline{c})$.)

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 data, handshake and toggle fields in a single atomic write operation.)

**Lemma 4.2** Let $\alpha = \pi_1 \pi_2 \ldots$ be a run of the bounded algorithm in which a particular scan, 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 an update \( j \) operation that began after the first of these four collects started.

**Lemma 4.3** Let \( \alpha = \pi_1 \pi_2 \ldots \) be a run of the bounded algorithm in which a particular scan \( j \) operation beginning in event \( \pi_u \) returns \( (d_1, \ldots, d_n) \) in event \( \pi_v \). Then \( \text{snapshot}(\alpha) = (d_1, \ldots, d_n) \) 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 \( P_i, \forall i \in \{1, \ldots, n\} \).

**Theorem 2** 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 data 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 \( r_k, \text{id} \) 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 \( \text{view}_j \) can be borrowed.

Hence, the algorithm of Figure 5 requires a multi-writer multi-reader register \( r_k \) for every memory address \( k \in \{1, \ldots, m\} \), holding fields \( r_k, \text{data}, r_k, \text{id} \) and \( r_k, \text{toggle} \) of type Data, \( \{1, \ldots, n\} \), and boolean. In addition, for every process \( P_i \) there are \( 2n \) single-writer multi-reader boolean registers \( p_{i,j} \) and \( q_{i,j}, \forall j \in \{1, \ldots, n\} \), and a single-writer multi-reader register \( \text{view}_i \), holding a vector of \( m \) Data values. The scan and update operations of a process \( i \) are described in Figure 5.

### 5.1 Correctness Proof

The serialization is defined as in the previous algorithms, with updates serialized with the (atomic) writes to the data registers. For this algorithm, a successful double collect occurs when the test in line 4 is passed. This test depends on steps 1.5 through 3.5, recording the handshake bits and the shared registers \( r_k \) twice: Step 1.5 implicitly collects the values of each \( p_{i,j} \), by storing \( p_{i,j} \) in \( q_{i,j} \). The next three lines explicitly record the values of the \( r_k \) registers and the handshake bits in \( a[1..m], b[1..m] \) and \( h[1..n] \), respectively. The test is passed if the handshake bits and id, toggle
procedure scan_i;
  begin
    0: for j = 1 to n do moved[j] := 0 od;
    1: while true do
      1.5: for j = 1 to n do q_{i,j} := p_{j,i} od; /* Handshake. */
      2: a[1..m] := collect(r_k : k \in \{1, \ldots, m\}) ; /* (data, id, toggle) triples. */
      3: b[1..m] := collect(r_k : k \in \{1, \ldots, m\}) ; /* (data, id, toggle) triples. */
      3.5: h[1..n] := collect(p_{j,i} : j \in \{1, \ldots, n\}) ; /* Handshake bits. */
      4: if (\forall j \in \{1, \ldots, n\})(q_{i,j} = h[j])
        and (\forall k \in \{1, \ldots, m\})(a[k].id = b[k].id)
        and (\forall k \in \{1, \ldots, m\})(a[k].toggle = b[k].toggle) /* Nobody moved. */
        then return (b[1].data, ..., b[m].data); /* Handshake bits. */
      5: else for j = 1 to n do
          6: if (q_{i,j} \neq h[j]) or (\exists k, b[k].id = j)
              /* P_j moved. */
              (a[k].id \neq b[k].id or a[k].toggle \neq b[k].toggle)
              then
            7: if moved[j] = 2 then /* P_j moved twice before! */
                8: return (view_i);
              9: else moved[j] := moved[j] + 1;
            od;
        od;
      10: end scan_i;
  end;

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

Figure 5: The bounded multi-writer algorithm.
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.

**Lemma 5.1** Let \( \alpha = \pi_1 \pi_2 \ldots \) be a run of the bounded multi-writer algorithm in which a particular scan \( \pi \) operation has a successful double collect, including \( a[1..m] := \text{collect in line 2} \) and \( b[1..m] := \text{collect in line 3} \). Let \( \pi_u \) and \( \pi_w \) be the last read of line 2 and the first read of line 3, respectively. Then for every prefix \( \alpha_u \) of \( \alpha \), \( u \leq v \leq w \), snapshot(\( \alpha_u \)) = (b[1].data, …, b[m].data).

**Proof:** As in the proof of Lemma 4.1, we argue below that if two successive collects by \( P_i \) return \( a[k].id = b[k].id = j \) and show no change in the handshake bit \( p_{j;i} \), then at most one write to \( r_k \), by \( P_j \), can be serialized between the two reads of \( r_k \) by \( P_i \). However, if such a write occurs, it will be observed to have changed the bit read in \( r_k\text{.toggle} \). The result follows.

Suppose then that the two successive reads by \( P_i \) of \( r_k \) return the values \( t_1 \) and \( t_2 \) in \( r_k\text{.toggle} \), respectively, and the two associated reads of \( p_{j;i} \) return the same value, \( \epsilon \). Further 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 2 and 3. Consider the last such write operation: being last, it must be a write by \( P_j \) writing the id value \( j \), and toggle bit \( t_2 \) read by the second read of \( r_k \) by \( P_i \). The final read by \( P_j \) of \( p_{j;i} \) returns \( \epsilon \), the result of an earlier write by \( P_j \) during an update. Since during an update \( P_j \) assigns to \( p_{j;i} \) the negation of the value read in \( q_{i,j} \), that \( \text{read}(q_{i,j}) \) must have read \( \neg \epsilon \), and so must have preceded \( P_i \)'s most recent write to \( q_{i,j} \) of \( \epsilon \). This implies two things, first that the \( \text{read}(q_{i,j}) \) operation by \( P_j \) is part of the same, final update operation considered above, and secondly that any earlier update by \( P_j \) must have been finished before the \( \text{write}_r(q_{i,j} = \epsilon) \). The partial order of events in this discussion is:

\[
\begin{array}{llll}
P_i (\text{scan}) & P_j (\text{update}) & \text{/* Handshake reads. */} \\
\text{read}_i(p_{j;i} = \epsilon) & \text{read}_j(q_{i;j} = \neg \epsilon) & \text{/* Handshake writes. */} \\
\text{write}_i(q_{i;j} = \epsilon) & \text{write}_j(p_{j;i} = \epsilon) & \text{/* First collect of } r_k \text{ in line 2. */} \\
\text{read}_i(r_k.id = j, r_k.toggle = t_1) & \text{write}_j(r_k.id = j, r_k.toggle = t_2) & \text{/* Write. */} \\
\text{read}_i(r_k.id = j, r_k.toggle = t_2) & \text{/* Second collect of } r_k \text{ in line 3. */} \\
\text{read}_i(p_{j;i} = \epsilon) & \text{/* Second handshake collect. */} \\
\end{array}
\]

It follows that no other write operation by \( P_j \) can be serialized between \( P_i \)'s final two reads of \( r_k \). Then these two reads by \( P_i \) of \( r_k \) return values written by two successive writes by \( P_j \), so the toggle bit values returned must be different, \( t_1 \neq t_2 \). (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 operation by \( P_j \), the \( \text{read}_j(q_{i;j} = \neg \epsilon) \).

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 data registers.
Lemma 5.2 Let $\alpha = \pi_1 \pi_2 \ldots$ 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 data registers during three different double collects. Then the value of $\text{view}_j$ read after the last collect was written by an update$_j$ operation that began after the first of these six collects started.

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 $\text{view}_j$
- write to $r_{k_1}$
- write to $p_{j,i}$
- write to $\text{view}_j$
- write to $r_{k_2}$
- 
- 
- 

Observing any three changes, in the $p_{j,i}$ or data registers, means that an intervening scan must have taken place and have been recorded in $\text{view}_j$. 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 $\alpha = \pi_1 \pi_2 \ldots$ be a run of the bounded multi-writer algorithm in which a particular scan$_i$ operation beginning in event $\pi_u$ returns $(d_1, \ldots, d_m)$ in event $\pi_w$. Then $\text{snapshot}(\alpha_v) = (d_1, \ldots, d_m)$ 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 3 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 [13] 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., [16, 12]), and it is likely snapshot memories will play a similar role in concurrent programming.
Interestingly, distributed snapshots are not true images of the global state: instead, a distributed snapshot returns one of a set of global states, each of which occurs in a system execution which is indistinguishable to the processes from the actual execution. This means that concurrent distributed snapshots may return conflicting images—two or more snapshots may not both be consistent with the process's other observations. Scans of snapshot memories are, by definition, simultaneously serializable with the update operations. By applying the emulators of [9] to the constructions presented in this paper, implementations of atomic snapshot memory are obtained in message-passing systems. Snapshots obtained this way are true 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 [3, 5] 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 [4, 5]. 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 $\Theta(n^2)$ 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 [23]). It is interesting to speculate whether other, more efficient solutions can be found.\footnote{Note that this measure of complexity ignores the size of the shared registers that are read and written in a single operation. The registers in these algorithms contain at most $\Theta(n)$ data fields.}

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 already known.

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\footnote{Clearly, procedures that return integer or other unbounded values will not have bounded implementations.} 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 seemingly more powerful than registers, nevertheless have bounded wait-free implementations from those simple primitives. In a paper that constructed a
computability hierarchy of atomic primitives, Herlihy showed that many interesting primitives do not have wait-free implementations from registers [17]. Is it possible to “close the gap” further, and construct yet more powerful primitives from registers? More ambitiously, is it possible to construct a complexity hierarchy of objects implementable from atomic registers, with natural notions of reduction and robust cost measures? Such a theory might provide a theoretical basis for the intuition that snapshot memories are more powerful than single-writer registers.

Acknowledgements: The authors thank Maurice Herlihy and Nancy Lynch for helpful discussions, and Galit Shemesh for comments on an earlier version of the paper.
References


