Atomic Snapshots of Shared Memory

YEHUDA AFEK

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

HAGIT ATTIYA

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

A preliminary version of this paper appeared in *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing* (Quebec City, Quebec, Aug.). ACM, New York, 1990, pp. 1–14.

H. Attiya's and N. Shavit's research was partially supported by National Science Foundation grant CCR-86-11442, 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 DCR 84-51396 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 H. 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; H. Attiya, Department of Computer Science, Technion, Haifa, Israel 32000; D. Dolev, Department of Computer Science, Hebrew University, Jerusalem, Israel 91904; E. Gafni, 3732 Boelter 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.

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: Algorithms, Concurrency, Shared Memory

Additional Key Words and Phrases: Atomic, consistent state, fault-tolerance, snapshot

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 "noninterference" step [26, 27]. By simplifying (or eliminating) the noninterference step, atomic snapshot memories can greatly simplify the design and verification of many concurrent algorithms. Examples include exclusion problems [14, 19, 20], construction of atomic multi-writer multi-reader registers [23, 29–31], concurrent time-stamp systems [15], approximate agreement [11], randomized consensus [1, 6, 7, 10] 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 [2–4]. 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 [3] 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 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 multiwriter atomic snapshot memories. Section 3 contains an implementation of single-writer snapshot memories from unbounded single-writer multireader 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 entirely) and the others read. In the multiwriter 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 $update_i$ by each process P_i, $i \in \{1 \cdot\cdot n\}$. The $scan_i$ operation has no arguments and returns a vector of n elements from an arbitrary set of data values. The $upate_i$ operation takes a data value as an argument and does not return a value. Executions of *scan*s and *update*s 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 $\bar{d}$ of data values such that each d_k 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 restriction can be described as the *architectural restrictions* imposed on solutions (cf. [17, 21]), and requires that any snapshot implementation be constructed with single-writer, multireader 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, for example, [23] and [29].

The second restriction imposed on snapshot memory implementations is that they satisfy the property of *wait-freedom* [22, 28]. That is, every snapshot operation by process P_i will terminate, regardless of the behavior of other processes, assuming only that local steps of P_i and operations on embedded shared objects terminate. The reader is referred to [5], [17], and [22] for discussions and proposed definitions of wait-freedom. The *update* and *scan* operations implemented in this paper require at most $\Theta(n^2)$ 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. 22, 25]). Axiomatic specifications for snapshot memories appear in [2], [3], and [15].

2.1. SPECIFICATION OF SINGLE-WRITER SNAPSHOT MEMORIES. Following [17] and [24], a single-writer atomic snapshot memory for n processes and a particular data set *Data* is an automaton with two types of input Request actions: $\mathsf{UpdateRequest}_i(d)$ and $\mathsf{ScanRequest}_i$, and two types of output Return actions: $\mathsf{UpdateReturn}_i(d_1, \ldots, d_n)$, 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 Request, and the process cannot prevent the memory from issuing a Request. Automata interact by identifying common actions.) The Request and Return actions are called the *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 *canonical single-writer snapshot automaton*. That is, a correct implementations S of a single-writer snapshot memory is one that the processes cannot distinguish from the canonical automaton. If the processes interact with S, the resulting *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, $\mathsf{Update}_i(d)$, and $\mathsf{Scan}_i(d_1, \ldots, d_n)$, 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 *Mem* of the 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 $\perp$.

Process P_i interacts with the automaton by issuing a request (an $\mathsf{UpdataRequest}_i(d)$ or $\mathsf{ScanRequest}_i$ action). The result is to store the input action in the state variable H_i, enabling the appropriate internal action ($\mathsf{Update}_i(d)$ or $\mathsf{Scan}_i(d_1, \ldots, d_n)$). The internal action in turn assigns an appropriate output action to H_i, and in the case of $\mathsf{Update}_i(d)$, assigns d to Mem_i as well. The change to the interface value H_i enables the appropriate output ($\mathsf{UpdateReturn}_i$ or $\mathsf{ScanReturn}_i(d_1, \ldots, d_n)$ action). Initially, each $H_i = \perp$ and $Mem_i = d_{init} \in Data$.

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

$\mathsf{UpdateRequest}_i(d)$		$\mathsf{ScanRequest}_i$	
Effect:	$H_i := \mathsf{UpdateRequest}_i(d)$	Effect:	$H_i := \mathsf{ScanRequest}_i$
$\mathsf{Update}_i(d)$		$\mathsf{Scan}_i(d_1, ..., d_n)$	
Precondition:	$H_i = \mathsf{UpdateRequest}_i(d)$	Precondition:	$H_i = \mathsf{ScanRequest}_i$
Effect:	$Mem[i] := d$		$Mem = (d_1, ..., d_n)$
	$H_i := \mathsf{UpdateReturn}_i$	Effect:	$H_i := \mathsf{ScanReturn}_i(d_1, ..., d_n)$
$\mathsf{UpdateReturn}_i$		$\mathsf{ScanReturn}_i(d_1, ..., d_n)$	
Precondition:	$H_i = \mathsf{UpdateReturn}_i$	Precondition:	$H_i = \mathsf{ScanReturn}_i(d_1, ..., d_n)$
Effect:	$H_i := \perp$	Effect:	$H_i := \perp$

FIG. 1. The canonical single-writer snapshot automaton.

canonical automaton in α is said to be *serialized* at the point of its associated Update or Scan operation.

The *well-formed* behaviors of the canonical automaton are those in which no pair of $\mathsf{Request}_i$ inputs occurs without an intervening 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 Requests, then S will not output redundant Returns. That is, if $\alpha\pi$ is a finite sequence of interface snapshot actions that is a behavior of S, with π a single output event and α 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 *Data* and m memory elements is an automaton with input actions: $\mathsf{UpdateRequest}_i(k, d)$, $\mathsf{ScanRequest}_i$, and output actions: $\mathsf{UpdateReturn}_i$, $\mathsf{ScanReturn}_i(d_1, \dots, d_m)$, for all $i \in \{1, \dots, n\}$, $k \in \{1, \dots, m\}$, and $d, d_1, \dots, d_m \in Data$. Call these the *multi-writer interface snapshot actions*. (Except for the addition of the address field k to the UpdateRequest actions, and 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 Update action has the additional address field k, and the 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,

$\mathsf{UpdateRequest}_i(k, d)$		$\mathsf{ScanRequest}_i$	
Effect:	$H_i := \mathsf{UpdateRequest}_i(k, d)$	Effect:	$H_i := \mathsf{ScanRequest}_i$
$\mathsf{Update}_i(k, d)$		$\mathsf{Scan}_i(d_1, \ldots, d_m)$	
Precondition:	$H_i = \mathsf{UpdateRequest}_i(k, d)$	Precondition:	$H_i = \mathsf{ScanRequest}_i$
Effect:	$Mem[k] := d$		$Mem = (d_1, \ldots, d_m)$
	$H_i := \mathsf{UpdateReturn}_i$	Effect:	$H_i := \mathsf{ScanReturn}_i(d_1, \ldots, d_m)$
$\mathsf{UpdateReturn}_i$		$\mathsf{ScanReturn}_i(d_1, \ldots, d_m)$	
Precondition:	$H_i = \mathsf{UpdateReturn}_i$	Precondition:	$H_i = \mathsf{ScanReturn}_i(d_1, \ldots, d_m)$
Effect:	$H_i := \perp$	Effect:	$H_i := \perp$

FIG. 2 The canonical multi-writer snapshot automaton.

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 and 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 [18, 22, 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 that is not wait-free. The kth *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

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

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

FIG. 3. The unbounded single-writer algorithm.

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 $\mathsf{UpdateRequest}_i$ and $\mathsf{UpdateReturn}_i$ actions, and calls and returns from $scan_i$ procedures (unless called from within *updates*), are identified with the $\mathsf{ScanRequest}_i$ and $\mathsf{ScanReturn}_i$ actions. Calls to $scan_i$ procedures from within *updates* are identified with actions $\mathsf{ScanRequest}_i^{int}$ and $\mathsf{ScanReturn}_i^{int}$ 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 and 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 implementations. 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.

Consider then any sequence $\alpha = \pi_1 \pi_2 \cdots$, where each π_j is either an interface snapshot action, a local computation event, a Request and Return for a shared register, an internal action $\mathsf{Read}_i(r_j = v)$ by P_i of atomic register r_j returning v, or an internal write $\mathsf{Write}_i(r_i = v)$ 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 $state(\alpha_k)$, to be the vector $(v_1, \ldots, v_n)$, where v_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. (These are the values of the relevant state components of the embedded registers, as implemented by the canonical automata.) If $state(\alpha_k) = (v_1, \ldots, v_n)$, then $snapshot(\alpha_k)$ denotes $(v_1.data, \ldots, v_n.data)$. As indicated, the sequence $snapshot(\alpha_0)$, $snapshot(\alpha_1)$, $snapshot(\alpha_2), \ldots$ 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. (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 $a[1 \cdot\cdot n] := collect$ in line 2 and $b[1 \cdot\cdot n] := collect$ in line 3, the sequence numbers in $a[1 \cdot\cdot n]$ and $b[1 \cdot\cdot n]$ 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 $\alpha = \pi_1\pi_2 \cdots$ be a run of the unbounded algorithm in which a particular scan$_i$ operation has a successful double collect: $a = [1 \cdot\cdot n] :=$ collect in line 2 and $b[1 \cdot\cdot n] :=$ collect in line 3. 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, $u \le v \le w$, snapshot$(\alpha_v) = (b[1].data, \ldots, b[n].data)$.*

PROOF. Suppose a write by P_j to r_j is serialized between two successive reads by P_i or 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 $\alpha = \pi_1\pi_2 \cdots$ 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 or 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 *scan*s can be correctly serialized somewhere in their intervals.

LEMMA 3.3. *Let $\alpha = \pi_1\pi_2 \cdots$ be a run of the unbounded algorithm in which a particular scan$_i$ operation beginning in event π_u returns $(d_1, \ldots, d_n)$ in event π_w. Then snapshot$(\alpha_v) = (d_1, \ldots, d_n)$ for some v, $u \le v \le w$.*

PROOF. If the *scan$_i$* operation has a successful double collect, the result follows from Lemma 3.1. Assume instead the *scan$_i$* operation borrows a snapshot value read in r_j. By Lemma 3.2, the snapshot value read in r_j was obtained by the *scan$_j$* operation, embedded in an *update$_j$* 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$_j$* is contained between the first and last reads by P_i of r_j. Either the *scan$_j$* operation had a successful double collect, and the result again follows from Lemma 3.1, or there is another embedded $scan_k$, occurring entirely within the interval of the $scan_j$ operation, from which P_j 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 cononical 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 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 to 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 4 replaces the unbounded sequence numbers with two *handshake* bits per pair of processes [22, 28]. 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 $q_{j,i}$ 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 $q_{j,i}$ bits), and the $q_{j,i}$ bits are written when P_j scans (to the values read from the $p_{i,j}$ bits). An additional *toggle bit*, r_i*.toggle*, is change during every *update*, to ensure that each write operation changes the register value.

4.1. CORRECTNESS PROOF.

For this algorithm, a *successful double collect* is a pair $a[1 \cdot\cdot n] := collect$; $b[1 \cdot\cdot n] := collect$; with all handshake bits $p_{j,i} = q_{i,j}$ and corresponding toggle bits in $a[1 \cdot\cdot n]$ and $b[1 \cdot\cdot n]$ identical. The following lemma proves that the handshake and toggle bits guarantee that a successful double collect produces a snapshot.

```
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} := r_j.p_{j,i} od;                    /* Handshake. */
      2:       a[1..n] := collect;              /* (data, bit vector, toggle, view) tuples. */
      3:       b[1..n] := collect;              /* (data, bit vector, toggle, view) tuples. */
      4:       if (∀j ∈ {1,...,n}), (a[j].p_{j,i} = b[j].p_{j,i} = q_{i,j}
                   and a[j].toggle = b[j].toggle) then                   /* Nobody moved. */
      5:         return (b[1].data, ..., b[n].data);
      6:       else for j = 1 to n do
      7:           if a[j].p_{j,i} ≠ q_{i,j} or b[j].p_{j,i} ≠ q_{i,j}                /* P_j moved. */
                      or a[j].toggle ≠ b[j].toggle then
      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_i;

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

FIG. 4. The bounded single-writer algorithm.

LEMMA 4.1. *Let $\alpha = \pi_1 \pi_2 \cdots$ be a run of the bounded algorithm in which a particular $scan_i$ operation has a successful double collect: $a[1 \cdots n] :=$ collect in line 2 and $b[1 \cdots n] :=$ collect in line 3. Let π_u and π_w be the last read in line 2 and the first read of line 3, respectively. Then, for every prefix α_v of α, $u \le v \le w$, $snapshot(\alpha_v) = (b[1].data, \ldots, b[n].data)$.*

PROOF. We argue below that if two successive collects by P_i show no change in the handshake bit $p_{j,i}$, than at most one write to r_j can be serialized between the two reads of r_j by P_i. However, if such a write occurs, it will be observed to have changed the bit read in $r_j.toggle$. 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 $r_j.toggle$, 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 $r_j.toggle$ 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 $read(q_{i,j})$ must have preceded P_i's most recent write to $q_{i,j}$ of c. This implies two things, first that the $read(q_{i,j})$ 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 $write_i(q_{i,j} = c)$. 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.)

$P_i(scan)$	$P_j(update)$	
$read_i(p_{j,i} = c)$	$read_j(q_{i,j} = \neg c)$	/* Handshake read. */
$write_i(q_{i,j} = c)$		/* Handshake. */
$read_i(r_j.p_{j,i} = c, r_j.toggle = t_1)$		/* First collect. */
	$write_j(r_j.p_{j,i} = c, r_j.toggle = t_2)$	/* Write. */
$read_i(r_j.p_{j,i} = c, r_j.toggle = t_2)$		/* Second collect. */

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 $read_j(q_{i,j} = \neg c)$.) $\square$

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 $\alpha = \pi_1\pi_2 \cdots$ 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 an update$_j$ operation that began after the first of these four collects started.*

LEMMA 4.3. *Let $\alpha = \pi_1\pi_2 \cdots$ be a run of the bounded algorithm in which a particular scan$_i$ operation beginning in event π_u returns$(d_1, \ldots, d_n)$ in event π_w. Then snapshot$(\alpha_v) = (d_1, \ldots, d_n)$ 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, $\forall i \in \{1, \ldots, n\}$.*

LEMMA 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 *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 $r_k.id$ 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 words. Hence, a

```
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 ∈ {1,...,m}) ;         /* (data, id, toggle) triples. */
      3:       b[1..m] := collect(r_k : k ∈ {1,...,m}) ;         /* (data, id, toggle) triples. */
      3.5:     h[1..n] := collect(p_{j,i} : j ∈ {1,...,n}) ;                /* Handshake bits. */
      4:       if (∀j∈ {1,...,n}) (q_{i,j} = h[j])
                     and (∀k ∈ {1,...,m}) (a[k].id = b[k].id)             /* Nobody moved. */
                     and (∀k ∈ {1,...,m}) (a[k].toggle = b[k].toggle) then
      5:         return (b[1].data, ..., b[m].data);
      6:       else for j = 1 to n do
      7:             if ( (q_{i,j} ≠ h[j]) or ( (∃k, b[k].id = j)               /* P_j moved. */
                          (a[k].id ≠ b[k].id or a[k].toggle ≠ b[k].toggle) )) then
      8:                if moved[j] = 2 then                      /* P_j moved twice before! */
      9:                    return (view_j);
      10:               else moved[j] := moved[j] + 1 ;
                     od;
           od;
   end scan_i;

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_{j,i} od;                              /* Handshake. */
      1: view_i := scan_i;                    /* Embedded scan: view_i is a single-writer register. */
      1.5: tog[k] := ¬tog[k];                   /* Local variable tog[1..n] saved between calls. */
      2: r_k := (data, i, tog[k]) ;                             /* r_k is a multi-writer register. */
    end update_i;
```

FIG. 5. The bounded multi-writer algorithm.

scan operation must observe process P_j move *three* times before the value in $view_j$ 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 $r_k.data$, $r_k.id$ and $r_k.toggle$ 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 $view_i$, 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 *update*s 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 $p_{j,i}$, 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 $a[1 \cdot\cdot m]$, $b[1 \cdot\cdot m]$, and $h[1 \cdot\cdot n]$, 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.

LEMMA 5.1. *Let $\alpha = \pi_1 \pi_2 \cdots$ be a run of the bounded multi-writer algorithm in which a particular $scan_i$ operation has a successful double collect, including $a[1 \cdot\cdot m] :=$ collect in line 2 and $b[1 \cdot\cdot m] :=$ collect in line 3. Let π_u and π_w be the last read of line 2 and the first read of line 3, respectively. Then, for every prefix α_v of α, $u \le v \le w$, $snapshot(\alpha_v) = (b[1].data, \ldots, b[m].data)$.*

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 $r_k.toggle$. 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 $r_k.toggle$, respectively, and the two associated reads of $p_{j,i}$ return the same value, c. Further assume that an *update* to work 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_i of $p_{j,i}$ returns c, 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 $read(q_{i,j})$ must have read $\neg c$, and so must have proceeded P_i's most recent write to $q_{i,j}$ of c. This implies two things, first that the $read(q_{i,j})$ 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 $write_i(q_{i,j} = c)$. The partial order of events in this discussion is:

$P_i(scan)$	$P_j(update)$	
$read_i(p_{j,i} = c)$	$read_j(q_{i,j} = \neg c)$	/* Handshake reads. */
$write_i(q_{i,j} = c)$	$write_j(p_{j,i} = c)$	/* Handshake writes. */
$read_i(r_k.id = j, r_k.toggle = t_1)$		/* First collect of r_k in line 2. */
	$write_j(r_k.id = j, r_k.toggle = t_2)$	/* Write. */
$read_i(r_k.id = j, r_k.toggle = t_2)$		/* Second collect of r_k in line 3. */
$read_i(P_{j,i} = c)$		/* Second handshake collect. */

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, $t_1 \ne t_2$. (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 $read_j(q_{i,j} = \neg c)$.) $\square$

The previous lemma says that the *scans* with successful double collects can be serialized correctly. It remains to argue that the *scans* that 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 nonatomicity of writes to the handshake and data registers.

LEMMA 5.2. *Let $\alpha = \pi_1 \pi_2 \cdots$ 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 $view_j$ 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 *update*s:

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}
$\vdots$

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 $view_j$, 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 $\alpha = \pi_1 \pi_2 \cdots$ be a run of the bounded multi-writer algorithm in which a particular $scan_i$ operation beginning in event π_u returns $(d_1, \ldots, d_m)$ in event π_w. Then $snapshot(\alpha_v) = (d_1, \ldots, d_m)$ for some v, $u \le v \le w$.*

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, *scan*s are wait-free. This in turn implies that *update*s 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 [13] provides a simple solution to the similar problem for message-passing system. The distributed snapshot algorithm has proven a useful tool in solving other distributed problems (see, e.g., [12] and [16]), 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 that 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 [2, 4] 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 [3, 4].

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 register, 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 multiwriter registers from single-writer, due to Li et al. [23]). It is interesting to speculate whether other, more efficient solutions can be found.[1]

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 *scan*s 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[2] 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 problem 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.

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

REFERENCES

1. ABRAHAMSON, K. On achieving consensus using a shared memory. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ont., Canada, Aug. 15–17). ACM, New York, 1988, pp. 291–302.

[1] 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.
[2] Clearly, procedures that return integer or other unbounded values will not have bounded implementations.

2. ANDERSON, J. H. Composite registers. Tech. Rep. TR-89-25. Dept. Comput. Sci., Univ. Texas at Austin, Austin, Tex., Sept. 1989.
3. ANDERSON, J. H. Multiple-writer composite registers. Tech. Rep. TR-89-26. Dept. Comput. Sci., Univ. Texas at Austin, Austin, Tex., Sept. 1989.
4. ANDERSON, J. H. Composite registers. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing* (Quebec City, Que., Canada, Aug. 22–24). ACM, New York, 1990, pp. 15–29.
5. ANDERSON, J. H., AND GOUDA, M. G. The virtue of patience: Concurrent programming with and without waiting. Tech. Rep. TR.90.23. Dept. Comput. Sci., Univ. Texas at Austin, Austin, Tex., July 1990.
6. ASPNES, J. Time- and space-efficient randomized consensus. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing* (Quebec City, Que., Canada, Aug. 22–24). ACM, New York, 1990, pp. 325–331.
7. ASPNES, J., AND HERLIHY, M. P. Fast randomized consensus using shared memory. *J. Algorithms* (Sept. 1990), pp. 441–461.
8. ASPNES, J., AND HERLIHY, M. P. Wait-free data structures in the asychronous PRAM model. In *Proceedings of the 2nd Annual Symposium on Parallel Algorithms and Architectures* (July). ACM, New York, 1990, pp. 340–349.
9. ATTIYA, H., BAR-NOY, A., AND DOLEV, D. Sharing memory rebustly in message-passing systems. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing* (Quebec City, Que., Canada, Aug. 22–24). ACM, New York, 1990, pp. 363–375.
10. ATTIYA, H., DOLEV, D., AND SHAVIT, N. Bounded polynomial randomized consensus. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing* (Edmonton, Alb., Canada, Aug. 14–16). ACM New York, 1989, pp. 281–293.
11. ATTIYA, H., LYNCH, N. A., AND SHAVIT, N. Are wait-free algorithms fast? In *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science* (Oct.). IEEE New York, 1990, pp. 55–64.
12. BRACHA, G., AND TOUEG, S. Distributed deadlock detection. *Distr. Computing 2* (1987), 127–138.
13. CHANDY, K. M., AND LAMPORT, L. Distributed snapsots: Determining global states of distributed systems. *ACM Trans. Comput. Syst. 3*, 1 (Jan. 1985) 63–75.
14. DOLEV, D., GAFNI, E., AND SHAVIT, N. Toward a non-atomic era: *l*-exclusion as a test case. In *Proceedings of the 20th Annual ACM Symposium on the Theory of Computing* (Chicago, Ill., May 2–4). ACM, New York 1988, pp. 78–92.
15. DOLEV, D., AND SHAVIT, N. Bounded concurrent time-stamp systems are constructible! In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing* (Seattle, Wash., May 15–17). ACM, New York, 1989, pp. 454–465.
16. GAFNI, E. Perspective on distributed network protocols: A case for building blocks. In *Proceedings of MILCOM-86* (Monterey, Calif., Oct.). IEEE, New York, 1986, pp. 1.1.1–1.1.5.
17. HERLIHY, M. P. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst. 13*, 1 (Jan. 1991), 124–149.
18. HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst. 12*, 3 (July 1990), 463–492.
19. KATSEFF, H. P. A new solution to the critical section problem. In *Proceedings of the 10th Annual ACM Symposium on the Theory of Computing* (San Diego, Calif., May 1–3). ACM, New York, 1978, pp. 86–88.
20. LAMPORT, L. The mutual exclusion problem. Part II: Statement and solutions. *J. ACM 33*, 2 (Feb. 1986), 327–348.
21. LAMPORT, L. On interprocess communication. Part I: Basic formalism. *Dist. Comput. 1*, 1 (1986), 77–85.
22. LAMPORT, L. On interprocess communication. Part II: Algorithms, *Dist. Comput. 1*, 1 (1986), 86–101.
23. LI, M., TROMP, J., AND VITANYI, P. M. B. How to share concurrent wait-free variables. In *Proceedings of ICALP '89*. (Stresa, Italy, July 11–15). Lecture Notes in Computer Science, vol. 372. Springer Verlag, New York, 1990, pp. 488–505. (Expanded version: Report CS-R8916, CWI, Amsterdam, The Netherlands, Apr. 1989).
24. LYNCH, N. A., AND TUTTLE, M. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of 6th Annual ACM Symposium on Principles of Distributed Computing* (Aug.) ACM, New York, 1987, pp. 137–151. (Expanded version available as Tech. Rep.

MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass. (Apr. 1987)).

25. MISRA, J. Axioms for memory access in asynchronous hardware systems. *ACM Trans. Prog. Lang. Syst. 8*, 1 (Jan. 1986), 142–153.
26. OWICKI, S. Axiomatic Proof Techniques for Parallel Programs. Ph.D. dissertation. Cornell Univ., Aug. 1975.
27. OWICKI, S., AND GRIES, D. An axiomatic proof technique for parallel programs. *Act Inf 6*, 1 (Jan. 1976), 319–340.
28. PETERSON, G. L. Concurrent reading while writing. *ACM Trans. Prog. Lang. Syst. 5*, 1 (Jan. 1983), 46–55.
29. PETERSON, G. L., AND BURNS, J. E. Concurrent reading while writing II: The multi-writer case. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science* (Oct.). IEEE, New York, 1987, pp. 383–392
30. SCHAFFER, R. On the correctness of atomic multi-writer registers. Tech. Rep. MIT/LCS/TM-364. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., June 1988.
31. VITANYI, P. M. B., AND AWERBUCH, B. Atomic shared register access by asynchronous hardware. In *Proceedings of 27th Annual Symposium on Foundations of Computer Science* (Oct.). IEEE, New York, 1986, pp. 233–243.

RECEIVED MAY 1990; REVISED JULY 1991; ACCEPTED JANUARY 1992