6-1-2006

Wait-Free and Obstruction-Free Snapshot

Khanh Do Ba
Dartmouth College

Wait-Free and Obstruction-Free Snapshot

Khanh Do Ba
Advisor: Prasad Jayanti

June 2006

Senior Honors Thesis
Dartmouth Computer Science Technical Report TR2006-578

Abstract

The *snapshot problem* was first proposed over a decade ago [1, 2] and has since been well-studied in the distributed algorithms community [4, 5, 6, 7, 8, 9, 12, 13, 14, 17, 18]. The challenge is to design a data structure consisting of m components, shared by upto n concurrent processes, that supports two operations. The first, UPDATE(i, v), atomically writes v to the ith component. The second, SCAN$()$, returns an atomic snapshot of all m components. We consider two termination properties: *wait-freedom*, which requires a process to always terminate in a bounded number of its own steps, and the weaker *obstruction-freedom*, which requires such termination only for processes that eventually execute uninterrupted [10].

First, we present a simple, time and space optimal, obstruction-free solution to the *single-writer, multi-scanner* version of the snapshot problem (wherein concurrent UPDATEs never occur on the same component). Second, we assume hardware support for compare&swap (CAS) to give a time-optimal, wait-free solution to the *multi-writer, single-scanner* snapshot problem (wherein concurrent SCANs never occur). This algorithm uses only $O(mn)$ space and has optimal CAS, write and remote-reference complexities. Additionally, it can be augmented to implement a general snapshot object with the same time and space bounds, thus improving the space complexity of $O(mn^2)$ of the only previously known time-optimal solution [14].

Snapshot algorithm	Primitive used	UPDATE time	SCAN time	Space
This paper	CAS or LL/SC	$O(1)$	$O(m)$	$O(mn)$
Afek et al. [1]	read/write	$O(mn)$	$O(mn)$	$O(mn + n^2)$
Anderson [2]	read/write	$O(2^{mn})$	$O(2^{mn})$	$O(m^3 n^4 \log n)$
Haldar and Vidyasankar [9]	read/write	$O(mn)$	$O(mn)$	$O(mn^2)$
Jayanti [13]	CAS or LL/SC	$O(m)$	$O(m)$	$O(mn^2)$
Jayanti [14]	CAS or LL/SC	$O(1)$	$O(m)$	$O(mn^2)$

Table 1: Comparison of multi-writer snapshot algorithms.

Snapshot algorithm	Primitive used	UPDATE time	SCAN time	Space
Afek et al. [1]	read/write	$O(n^2)$	$O(n^2)$	$O(n^2)$
Anderson [2]	read/write	$O(2^n)$	$O(2^n)$	$O(n^3 \log n)$
Aspnes and Herlihy [4]	read/write	$O(n^2)$	$O(n^2)$	$O(n^3)$
Attiya et al. [5]	test&set	$O(n)$	$O(n)$	∞
Attiya and Rachman [6], *uses unbounded registers*	read/write	$O(n \log n)$	$O(n \log n)$	∞
Chandra and Dwork [7]	CAS or LL/SC	$O(n)$	$O(n)$	∞
Dwork et al. [8], *weak snapshot*	read/write	$O(n)$	$O(n)$	$O(n^2)$
Haldar and Vidyasankar [9]	read/write	$O(n^2)$	$O(n^2)$	$O(n^3)$
Israeli et al. [12]	read/write	$O(n)$	$O(n \log n)$	$O(n^2)$
Kirousis et al. [17], *single-scanner*	read/write	$O(1)$	$O(n)$	$O(n)$
Riany et al. [18]	CAS or LL/SC, and fetch&inc	$O(1)$	$O(n)$	$O(n^2)$

Table 2: Comparison of single-writer snapshot algorithms.

1 Introduction

The *snapshot problem* was first proposed independently by Anderson [2] and Afek et al. [1] in 1990, and has since enjoyed a considerable amount of attention and research in the distributed algorithms community [4, 5, 6, 7, 8, 9, 12, 13, 14, 17, 18]. Previous results are summarized in Figures 1 and 2, which are largely lifted from Tables 1 and 2 in [14].

The motivation lies in the need to obtain an instantaneous global view of a large portion of active memory, without disturbing the underlying computation. That is, a process may want to simultaneously read multiple words of shared memory in a system that only supports single-word operations, even as each word is being concurrently modified. Such need arises in a multitude of applications, including checkpointing a computation, backing up a disk or memory currently in use, debugging parallel programs, and obtaining a consistent reading of multiple sensors in a sensor network.

1.1 Background and definitions

Formally, the problem is to design a data structure called the *snapshot object* consisting of m components, shared by n processes, that supports two operations. The first, UPDATE(i, v), writes v to the ith component. The second, SCAN(), returns a snapshot of all m components. These operations must be implemented from hardware-supported primitives: most commonly, read/write, and in many multiprocess systems, various read-modify-write primitives such as compare&swap (CAS) and limited forms of load-linked/store-conditional/validate (LL/SC/VL).

As a sidebar, we quickly remind the reader of the specifications of these atomic operations, which we will make use of in some of our algorithms. If X is a shared variable, $CAS(X, \mathit{old}, \mathit{new})$ sets X to new if $X = \mathit{old}$, returning true, or returns false if $X \neq \mathit{old}$. $LL(X)$ returns the value of X. A subsequent $SC(X, \mathit{new})$ sets X to new if X has not been modified since the last $LL(X)$, returning true, and returns false otherwise. Lastly, $VL(X)$ returns true if X has not been modified since the last $LL(X)$, and false otherwise.

A restricted version of the snapshot problem, known as *single-writer* snapshot, assumes that concurrent UPDATEs never occur on the same component. Similarly, another version, known as *single-scanner* snapshot, assumes that concurrent SCANs never occur. The most general version where arbitrary concurrency is allowed is known as *multi-writer*, *multi-scanner* snapshot. In this paper we consider all combinations of these restrictions.

A conceivable "solution" to the snapshot problem is to simply read one component at a time to obtain the composite. However, even under the single-writer, single-scanner assumption, this would result in an inconsistent view (i.e., not a "snapshot") since each component may be concurrently written to. We therefore require implementations to be *atomic* (or *linearizable*), that is, an operation must appear to take effect instantaneously at some point during the interval of its execution. See [11] for a more detailed discussion of linearizability.

It is also natural to require some progress guarantees. Research in distributed algorithms has focused on the following three lock-free conditions: an algorithm is (1) *wait-free* if every operation completes in a bounded number of its own steps, (2) *non-blocking* if some operation always completes in a bounded number of system steps, and (3) *obstruction-free* if every operation always completes in a bounded number of steps provided no other process takes any steps. It is easy to see that any wait-free algorithm is non-blocking, and that any non-blocking algorithm is obstruction-free. Of course, wait-freedom is the ideal, but it often comes at the cost of high conceptual as well as computational complexity. On the other hand, recent research suggests that obstruction-freedom is often sufficent in many practical systems, and is much easier to achieve [10]. In this paper, we consider both wait-free and obstruction-free implementations of the snapshot object.

Finally, in analyzing the efficiency of an algorithm, we consider a number of complexity measures. Most important of these, of course, are time and space. In addition, on a multiprocess system that supports concurrent read, write and/or compare&swap (CAS), it is often the case that a write or CAS operation takes considerably more time than a read operation. We would therefore like to minimize write and CAS complexities as well. Lastly, in many architectures, each processor is able to access a piece of shared memory without causing traffic on the global interconnection bus. Two such architectures that we consider here are the *distributed shared-memory* (DSM) and *cache-coherent* (CC) models [3]. On a DSM machine, each process simply has its own local block of shared memory. On a CC machine, each process has a local cache of all of shared memory, and some hardware protocol enforces consistency via the global bus whenever shared memory is modified. In both models, we would like to minimize the remote-reference complexity (i.e., the number of hub accesses) of each operation.

1.2 Our results

In Section 2, we present a conceptually simple, obstruction-free solution to the single-writer, multi-scanner snapshot problem, using only read/write primitives. We do so by first giving a wait-free, single-writer, single-scanner algorithm, then introducing a simple modification to augment it to the obstruction-free, multi-scanner algorithm. The latter in fact gives a stronger termination guarantee than just obstruction-freedom: UPDATEs are wait-free and SCANs can only be scuttled by other SCANs. In particular, it is wait-free in the single-scanner scenario.

The time complexity of the algorithm is $O(1)$ and $O(m)$ for UPDATEs and SCANs, respectively, and its space complexity is $O(m)$. Hence, it is trivially optimal in both time and space. A SCAN carries out $O(1)$ write operations, so it is write optimal as well. On a DSM machine, a SCAN by process p makes zero remote-references, and on a CC machine, it makes $O(k)$ remote-references, where k is the number of components touched by an UPDATE since the start of the last SCAN by p. An UPDATE of course has $O(1)$ remote-reference complexity in both models.

In Section 3, we assume support for LL/SC to give a wait-free algorithm for multi-writer, single-scanner snapshot. As for the previous algorithm, we introduce a wait-free, single-writer, single-scanner algorithm, and augment it to produce the multi-writer algorithm. The result has optimal time, $O(mn)$ space and constant CAS complexities, with write and remote-reference complexities as above. We also show how this algorithm can be modified to solve multi-writer, multi-scanner snapshot in the same time and space bounds, thus reducing the space requirement of Jayanti's previous time-optimal solution [14] from $O(mn^2)$ down to $O(mn)$.

Finally, in Section 4 we show a lowerbound that proves that on a DSM machine, any wait-free single-writer, single-scanner snapshot algorithm must have non-zero remote-reference complexity for UPDATEs. A consequence of this, which we will explain in more detail, is that all our single-scanner algorithms are indeed optimal in DSM remote-reference complexity.

Our algorithms are inspired by ideas from Jayanti's algorithms [14], but introduce novel techniques and constructions to achieve the mentioned improvements.

2 Toward obstruction-freedom

In this section, we first present a simple and efficient algorithm that implements a single-writer, single-scanner snapshot object that guarantees wait-freedom. We then give an easy modification to transform this into a (single-writer) multi-scanner snapshot object, but which guarantees the slightly weaker condition of obstruction-freedom. Both algorithms only require hardware support for read/write, and have optimal time, space and write complexities.

2.1 Wait-free, single-writer, single-scanner snapshot

The basic idea behind the algorithm is motivated by the single-writer, single-scanner algorithm in [14]. We keep a core array `A` on which UPDATEs and SCANs occur, and a second array `B` where UPDATEs concurrent with a SCAN forward their values in case they were missed in `A`. In [14], a SCAN writes $\perp$ to each component of `B` to ensure that if it later finds a non-$\perp$ value there, it must be current. To reduce the write-complexity of SCANs, we instead introduce a scheme of bounded sequence numbers. Specifically, a SCAN announces a unique sequence number, and UPDATEs that want to forward for it write that sequence number to the corresponding component of a third array `C` to let the SCAN know that a forwarding has occurred. The precise algorithm is given in Figure 1.

To discuss correctness of this algorithm, we adopt the line reference notation of [14]. That is, if OP is either a SCAN or an UPDATE, then $OP[\ell]$ denotes Line ℓ in OP's algorithm. If Line ℓ involves more than one atomic operation, than $OP[\ell_i]$ denotes the ith one. Here, we also introduce some natural terminology which will be useful later.

Definition 2.1. *Let U be an i-UPDATE and S be a SCAN. The period of S during which* `X` $=$ *true is called its* sip-section *(where sip stands for scan-in-progress). We say S* returns *U* (for component i) via `B` *if the test at $S[8_1]$ succeeds and U is the last* UPDATE *to write to* `B`$[i]$ *before S reads* `B`$[i]$ *($S[8_2]$), or* via `A` *if the test fails and U is the last* UPDATE *to write to* `A`$[i]$ *before S reads* `A`$[i]$ *($S[5_i]$). Finally, S* picks up *U* (for component i) *if S returns either U or a later* UPDATE.

Shared variables:

A[1..m]:	array of machine words, initialized to initial value of snapshot object
B[1..m]:	array of machine words, initialized arbitrarily
C[1..m]:	array of sequence numbers from $\{1, \ldots, m+1\}$, initialized arbitrarily
X:	boolean variable, initialized to $false$
S:	variable, with sequence number in $\{1, \ldots, m+1\}$, initialized arbitrarily

Update(i, v)

```
1  A[$i$] $\leftarrow v$
2  $x \leftarrow$ X
3  if $x$ then
4      $s \leftarrow$ S
5      B[$i$] $\leftarrow v$
6      C[$i$] $\leftarrow s$
```

Scan()

```
1  X $\leftarrow true$
2  for $i = 1$ to $m$ do $c[i] \leftarrow$ C[$i$]
3  choose $s \in \{1, \ldots, m+1\} : s \notin c$
4  S $\leftarrow s$
5  for $i = 1$ to $m$ do $a[i] \leftarrow$ A[$i$]
6  X $\leftarrow false$
7  for $i = 1$ to $m$ do
8      if C[$i$] $= s$ then $v[i] \leftarrow$ B[$i$]
9      else $v[i] \leftarrow a[i]$
10 return $v$
```

Figure 1: A wait-free, single-writer, single-scanner snapshot algorithm.

Armed with this notation, we can now define the linearization points of our operations.

Definition 2.2 (Linearization Points). *Let S be a* Scan *and U be an* Update. *Define the linearization point of S to be $LP(S) = S[6]$, and that of U to be as follows.*

(1) If U straddles $LP(S')$ for some Scan *S' that does not pick it up, then $LP(U)$ is immediately after $LP(S')$.*

(2) Otherwise, $LP(U) = U[1]$.

The following three lemmas form the core of our proof of correctness.

Lemma 2.3. *Let U be an i-*Update *and S be a* Scan. *If $U[1] < S$, then S picks up U.*

Proof. Let $U' < U$ be any earlier i-Update. We need to show that S does not return U'. But S certainly does not return U' via A since U overwrites A[i] before S even starts. On the other hand, if S returns U' via B, then some $U'' > U'$ must write the matching sequence number s to C[i] after $S[2_i]$ and before $S[8_1]$. But then U'' overwrites B[i] before $S[8_2]$ as well, so S cannot possibly return U' via B. □

Lemma 2.4. *Let U and S be as above. If $U < LP(S)$, then S picks up U.*

Proof. Again, let $U' < U$ be any earlier i-Update. Suppose S returns U' via A. Then $U'[1] < S[5_i] < U[1]$, otherwise U would overwrite U' in A[i] before S reads it. Therefore, U lies entirely during the sip-section of S and hence must forward. Any subsequent Update during S cannot write a different sequence number to C[i], so S must in fact return via B.

Alternatively, suppose S returns U' via B. Then U must not forward, otherwise it would overwrite U' in B[i]; that is, $U[2] < S[1]$. But then by Lemma 2.3, S picks up U, so in particular does not return U'. □

Lemma 2.5. *Let U and S be as above. If $U > LP(S)$, then S does not pick up U.*

Proof. It suffices to show that S does not return U, since any later UPDATE would also satisfy the hypothesis of the lemma. Now, S cannot return U via A, since U has not even started yet when S reads A$[i]$. On the other hand, U always sees X $= false$ during the lifetime of S, so it cannot forward before S terminates. Hence S cannot return U via B either. □

An immediate corollary is

Lemma 2.6. *If an* UPDATE U *is picked up by a* SCAN S*, then* U *is also picked up by every subsequent* SCAN*.*

Proof. If S picks up U, then by Lemma 2.5, U starts before $S[6]$, hence before the start of any subsequent SCAN. By Lemma 2.3, any such SCAN would pick up U. □

Finally, to prove our algorithm correct, we need to show (1) that linearization points are validly defined and (2) that SCANs behave as they would if every operation were shrunk to their respective linearization points (note that UPDATEs do not return a value so always behave correctly). These conditions are formalized in the following two lemmas.

Lemma 2.7 (LP Validity). *If* OP *is either a* SCAN *or an* UPDATE*, then* $LP(OP)$ *is well-defined and lies within the execution interval of* OP*.*

Proof. The lemma clearly holds for SCANs. For an UPDATE U, it similarly holds if U falls under the second case. Suppose U falls under the first case and let S' be as in Definition 2.2. Then by Lemma 2.3, $S'[1] < U[1]$, so that $U[1]$ must occur between $S'[1]$ and $S'[6]$. Since SCANs are disjoint, no other SCAN can satisfy this condition, so $LP(U)$ is well-defined. Finally, $LP(U)$ lies within the interval of U since U is assumed to straddle $LP(S')$. □

Lemma 2.8 (Correctness of SCAN**).** *If* U *is an* UPDATE *and* S *is a* SCAN *such that* S *returns* U *for component* i*, then (1)* $LP(U) < LP(S)$ *and (2) no other* i*-*UPDATE U' *satisfies* $LP(U) < LP(U') < LP(S)$*.*

Proof. Assume S returns U for component i. If U straddles $LP(S')$ for some SCAN S' which does not pick it up, then S' must precede S by Lemma 2.6, so that $LP(U)$, which just follows $LP(S')$, precedes $LP(S)$. If no such S' exists, then since S returns U, $LP(U) = U[1] < LP(S)$ by Lemma 2.5. This proves (1).

To prove (2), suppose to the contrary that such a U' exists. Since S does not pick up U', by Lemma 2.4, $LP(S) < U'[last]$. But $U'[1] < LP(U') < LP(S)$, so U' in fact straddles $LP(S)$. By Definition 2.2, $LP(U')$ must then be immediately after $LP(S)$ — a contradiction. □

We sum up the algorithm in the following theorem. Note that the zero remote reference complexity of a SCAN on a DSM machine is achieved by simply making every shared variable local to the scanner.

Theorem 2.9. *The algorithm in Figure 1 implements an atomic single-writer, single-scanner snapshot object that guarantees wait-freedom.* UPDATE*s and* SCAN*s complete in* $O(1)$ *and* $O(m)$ *time, respectively, where a* SCAN *performs only three write operations on shared memory. The space complexity is* $O(m)$*. A* SCAN *by process* p *has zero and* $O(k)$ *remote-reference complexity in the DSM and CC models, respectively, where* k *is the number of components modified by an* UPDATE *since the start of the last* SCAN *by* p*.*

Shared variables:

$\mathtt{A}[1..m]$:	array of snapshot component values, initialized to initial value of snapshot object
$\mathtt{B}[1..m]$:	array of snapshot component values, initialized arbitrarily
$\mathtt{C}[1..m]$:	array of sequence numbers from $\{1, \ldots, m+1\}$, initialized arbitrarily
$\mathtt{X}$:	variable, with fields $\mathtt{X}.proc$ (a PID) and $\mathtt{X}.sip \in \{true, false\}$, initialized to $[*, false]$
$\mathtt{S}$:	variable, with fields $\mathtt{S}.proc$ (a PID) and $\mathtt{S}.seq \in \{1, \ldots, m+1\}$, initialized arbitrarily

Update(i, v)

```
1  $\mathtt{A}[i] \leftarrow v$
2  $x \leftarrow \mathtt{X}$
3  if $x.sip$ then
4      $s \leftarrow \mathtt{S}$
5      if $s.proc = x.proc$ then
6          $\mathtt{B}[i] \leftarrow v$
7          $\mathtt{C}[i] \leftarrow s.seq$
```

Scan(p)

```
1  $\mathtt{X} \leftarrow [p, true]$
2  for $i = 1$ to $m$ do $c[i] \leftarrow \mathtt{C}[i]$
3  choose $s \in \{1, \ldots, m+1\} : s \notin c$
4  $\mathtt{S} \leftarrow [p, s]$
5  for $i = 1$ to $m$ do $a[i] \leftarrow \mathtt{A}[i]$
6  if $\mathtt{X}.proc = p$ then
7      $\mathtt{X} \leftarrow [p, false]$
8      for $i = 1$ to $m$ do
9          if $\mathtt{C}[i] = s$ then $v[i] \leftarrow \mathtt{B}[i]$
10         else $v[i] \leftarrow a[i]$
11     if $\mathtt{S}.proc = p$ then return $v$
12 return $\perp$
```

Figure 2: An obstruction-free, single-writer, multi-scanner snapshot algorithm.

2.2 Obstruction-free, single-writer, multi-scanner snapshot

We now improve the previous algorithm to a single-writer, multi-scanner algorithm that guarantees obstruction-freedom. More precisely, it guarantees that Updates are wait-free and that Scans can only be scuttled by other Scans. Because Scans only write to $\mathtt{X}$ and $\mathtt{S}$, any interference they cause each other must be through these two variables. This fact greatly simplifies the task of interference-detection: we simply augment $\mathtt{X}$ and $\mathtt{S}$ with a PID field, which a Scan uses to announce its activity. The new algorithm is given in Figure 2. Note that Scans return $\perp$ when it detects interference by another Scan, so to give the termination guarantee of obstruction-freedom, we simply repeat Scans until a non-$\perp$ value is returned.

Correctness of this algorithm depends only on Updates and those Scans which return non-$\perp$ values, so we need only define linearization points for these operations. To this end, we generalize Definition 2.2 as follows.

Definition 2.10 (Linearization Points). *Let S be a* Scan *which returns a non-$\perp$ value, and U be an* Update*. Define the linearization point $LP(S)$ of S to be the first point after $S[6]$ at which $\mathtt{X}$ does not contain $[p, true]$, where p is the process executing S. Define the linearization point $LP(U)$ of U as follows.*

(1) If U straddles $LP(S')$ for some Scan *S' that does not pick it up, then $LP(U)$ is immediately after $LP(S')$.*

(2) Otherwise, $LP(U) = U[1]$.

The correctness conditions are as before, but applied only to the restricted class of operations for which we defined linearization points. Adapting terminology from Definition 2.1 in the natural way, we will prove our three main structural lemmas in this context, where Scans in the statements of the lemmas refer to ones returning valid snapshots.

We begin by observing that in any (valid) Scan S, the tests at $S[6]$ and $S[11]$ must succeed. It follows that `X` remains untouched from $S[1]$ to $S[6]$ and `S` remains untouched from $S[4]$ to $S[11]$. With these guarantees in mind, it is easy to see that the proofs of Lemmas 2.3, 2.4 and 2.5 for the single-scanner algorithm can be directly lifted to apply to the new algorithm. The proofs of LP Validity (Lemma 2.7) and Correctness of Scan (Lemma 2.8), as before, are consequences of the three structural lemmas, giving us the main result of this section in the following theorem.

Theorem 2.11. *The algorithm in Figure 2 implements an atomic single-writer, multi-scanner snapshot object that guarantees that* Update*s are wait-free and a* Scan *returns a valid snapshot if no other* Scan*s execute concurrently with it.* Update*s and* Scan*s complete in $O(1)$ and $O(m)$ time, respectively, where a* Scan *performs only three write operations to shared memory. The space complexity is $O(m)$. A* Scan *by process p has zero and $O(k)$ remote-reference complexity in the DSM and CC models, respectively, where k is the number of components modified by an* Update *since the start of the last* Scan *by p.*

3 Toward wait-freedom

In this section, we build toward a multi-writer, single-scanner snapshot object that guarantees wait-freedom. We derive some ideas from [14], but introduce a number of novel "tricks" to give a substantially different and improved algorithm. Specifically, our algorithm has the same complexities as the obstruction-free algorithm in Theorem 2.11, except for space, which is now $O(mn)$. Moreover, it can be extended to solve multi-scanner snapshot in the same time and space, thus improving the space bound on [14].

3.1 Modified wait-free, single-writer, single-scanner snapshot

We present the first step in the construction: the single-writer, single-scanner algorithm. It works in a similar fashion to the single-writer, single-scanner algorithm of the previous section, but to facilitate future transformations, we roll what had been two separate variables `X` and `S` into just one, namely, `X` with fields *sip* and *seq*, respectively. However, doing this involves changing the algorithm nontrivially, so we need some modifications to maintain correctness. In particular, we augment each component of `C` with a single *vld* bit indicating whether the sequence number stored there should be considered a completed forward, and Updates attempting to forward first write its sequence number to `C`$[i]$ with $vld = 0$, then repeat with $vld = 1$.

Intuitively, the reason for this double writing to `C`$[i]$ is to ensure that if an Update is poised to write $[1, s]$ to `C`$[i]$, thereby indicating a completed forwarding for any Scan using s as its sequence number, it must already have announced that sequence number in `C`$[i]$ as $[0, s]$ so that later Scans can avoid s. We therefore ensure that a stale Update does not incorrectly inform a Scan that a recent forwarding has just occurred. The precise algorithm is given in Figure 3.

We can define linearization points almost identically to Definition 2.2 as follows.

Definition 3.1 (Linearization Points). *Let S be a* Scan *and U be an* Update*. Define the linearization point of S to be $LP(S) = S[5]$, and that of U to be as follows.*

(1) If U straddles $LP(S')$ for some Scan *S' that does not pick it up, then $LP(U)$ is immediately after $LP(S')$.*

(2) Otherwise, $LP(U) = U[1]$.

Recall the three structural lemmas of the previous section, which we will use here as well to form the core of our proof. Note that the terminology in the lemmas is the natural adaptation of

Shared variables:

$\mathtt{A}[1..m]$:	array of machine words, initialized to initial value of snapshot object
$\mathtt{B}[1..m]$:	array of machine words, initialized arbitrarily
$\mathtt{C}[1..m]$:	array, with fields $\mathtt{C}[i].vld \in \{0,1\}$ and $\mathtt{C}[i].seq \in \{1,\ldots,m+1\}$, initialized arbitrarily
$\mathtt{X}$:	variable, with fields $\mathtt{X}.sip \in \{true, false\}$ and $\mathtt{X}.seq \in \{1,\ldots,m+1\}$, initialized to $[false,*]$

Update(i,v)

```
1 A[$i$] ← $v$
2 $x$ ← X
3 if $x.sip$ then
4     B[$i$] ← $v$
5     if C[$i$] $\neq [1, x.seq]$ then
6         C[$i$] ← $[0, x.seq]$
7         if X $= x$ then C[$i$] ← $[1, x.seq]$
```

Scan()

```
1 for $i = 1$ to $m$ do $c[i]$ ← C[$i$]
2 choose $s \in \{1,\ldots,m+1\} : [*, s] \notin c$
3 X ← $[true, s]$
4 for $i = 1$ to $m$ do $a[i]$ ← A[$i$]
5 X ← $[false, s]$
6 for $i = 1$ to $m$ do
7     if C[$i$] $= [1, s]$ then $v[i]$ ← B[$i$]
8     else $v[i] \leftarrow a[i]$
9 return $v$
```

Figure 3: A modified single-writer, single-scanner snapshot algorithm.

Definition 2.1 to the current algorithm, which has a very similar basic structure. The proofs of Lemmas 2.3 and 2.5 from the previous section lift directly to apply here, but Lemma 2.4 needs a new proof, since the old one depended on the fact that the first operation by a Scan sets X to *true*. We give his new proof below.

Proof of Lemma 2.4. Consider any prior i-Update U'. As before, we need to show that S does not return U'. Suppose S returns U' via A. Then $U'[1] < S[4_i] < U[1]$, otherwise U would overwrite U' in $\mathtt{A}[i]$ before S reads it. But U completes before $S[5]$, so it lies entirely within the sip-section of S and must therefore forward. Moreover, any subsequent U'' attempting to forward during S would fail the test at $U''[5]$, so would not modify $\mathtt{C}[i]$. This ensures that S returns via B, contradicting our hypothesis.

Now, suppose S returns U' via B. Then U must complete without writing to $\mathtt{B}[i]$, otherwise it would overwrite U'; that is, $U[2] < S[3]$. Let U'' be the Update which last sets $\mathtt{C}[i]$ to $[1,s]$ before $S[7_1]$. Note that $U'' \leq U'$, otherwise it would overwrite U' in $\mathtt{B}[i]$, so in particular, $U'' < S[3]$. It follows that U'' "validates" X ($U''[7_1]$) during the sip-section of an earlier Scan, at which point $\mathtt{C}[i]$ contains $[0,s]$. Since there are no concurrent i-Updates, this remains so until U'' writes $[1,s]$ there, implying that at the moment S reads $\mathtt{C}[i]$ ($S[1_i]$), $\mathtt{C}[i]$ holds either $[0,s]$ (if U'' has not yet written there the second time) or $[1,s]$ (if it has). In either case, this contradicts the choice of the sequence number s by S. □

Once again, LP Validity and Correctness of Scan follows from these three lemmas, resulting in the following theorem.

Theorem 3.2. *The algorithm in Figure 3 implements an atomic single-writer, single-scanner snapshot object that guarantees wait-freedom.* Update*s and* Scan*s complete in $O(1)$ and $O(m)$ time, respectively, where a* Scan *performs only three write operations on shared memory. The space complexity is $O(m)$. A* Scan *by process p has zero and $O(k)$ remote-reference complexity in the DSM and CC models, respectively, where k is the number of components modified by an* Update *since the start of the last* Scan *by p.*

Shared variables:

$\mathtt{A}[1..m]$:	array of machine words, initialized to initial value of snapshot object
$\mathtt{B}[1..m]$:	array of machine words, initialized arbitrarily
$\mathtt{C}[1..m]$:	array, with fields $\mathtt{C}[i].vld \in \{0,1\}$ and $\mathtt{C}[i].seq \in \{1,\ldots,m+1\}$, initialized arbitrarily
$\mathtt{X}$:	variable, with fields $\mathtt{X}.sip \in \{true, false\}$ and $\mathtt{X}.seq \in \{1,\ldots,m+1\}$, initialized to $[false,*]$

Update(i,v)

```
1  $\mathtt{A}[i] \leftarrow v$
2  $x \leftarrow LL(\mathtt{X})$
3  if $x.sip$ then
4      ForwardB$(i)$
4'     ForwardB$(i)$
5      AnnounceC$(i, x.seq)$
5'     AnnounceC$(i, x.seq)$
6      ForwardC$(i, x.seq)$
```

ForwardB(i)

```
b1  $LL(\mathtt{B}[i])$
b2  $a \leftarrow \mathtt{A}[i]$
b3  if $VL(\mathtt{X})$ then
b4      $SC(\mathtt{B}[i], a)$
```

AnnounceC(i,s)

```
a1  $c \leftarrow LL(\mathtt{C}[i])$
a2  if $VL(\mathtt{X}) \wedge c.seq \neq s$ then
a3      $SC(\mathtt{C}[i], [0,s])$
```

ForwardC(i,s)

```
c1  $c \leftarrow LL(\mathtt{C}[i])$
c2  if $VL(\mathtt{X}) \wedge c = [0,s]$ then
c3      $SC(\mathtt{C}[i], [1,s])$
```

Scan()

```
1  for $i = 1$ to $m$ do $c[i] \leftarrow \mathtt{C}[i]$
2  choose $s \in \{1,\ldots,m+1\} : [*,s] \notin c$
3  $\mathtt{X} \leftarrow [true, s]$
4  for $i = 1$ to $m$ do $a[i] \leftarrow \mathtt{A}[i]$
5  $\mathtt{X} \leftarrow [false, s]$
6  for $i = 1$ to $m$ do
7      if $\mathtt{C}[i] = [1,s]$ then $v[i] \leftarrow \mathtt{B}[i]$
8      else $v[i] \leftarrow a[i]$
9  return $v$
```

Figure 4: A multi-writer, single-scanner snapshot algorithm.

3.2 Wait-free, multi-writer, single-scanner snapshot

The transformation of the single-writer algorithm above to a multi-writer one requires a substantial amount of modification to the Update procedure. We need to resort to LL/SC/VL operations on the B and C arrays to help coordinate forwarding by concurrent Updates. The validation of X that used to be carried out by a simple read must now also be implemented via an LL/SC/VL variable. An Update itself tries to carry out the same steps as it did before, but forwarding must now be highly coordinated. In particular, an i-Update carries out two ForwardBs to ensure that a current value is written to $\mathtt{B}[i]$, whether by itself or a concurrent Update; then it carries out two AnnounceCs and one ForwardB to push $\mathtt{C}[i]$ through the values $[0,s]$ and then $[1,s]$, much like it would have attempted to do in the single-writer algorithm. The multiplicity (or lack thereof) of these operations, which may seem bizarre at first glance, will become clear in the proof. The algorithm is given in Figure 4.

We can generalize most of the terminology in Definition 2.1 to apply here, except for the concept of *picking up*, which requires an ordering on Updates. For this purpose, we will order Updates according to when they write to A, i.e., their first action.

Using this ordering, we define linearization points as follows.

Definition 3.3 (Linearization Points). *Let S be a* Scan *and U be an* Update*. Define the linearization point of S to be $LP(S) = S[5]$, and that of U to be as follows.*

(1) If U straddles $LP(S')$ for some Scan *S' that does not pick it up, then $LP(U)$ is immediately after $LP(S')$ and $LP(U')$ for any earlier i-*Update *U' that satisfies the same relationship with S'.*

(2) Otherwise, $LP(U) = U[1]$.

The ordering also makes our three structural lemmas from the previous section well-defined, and we will again use them to prove our algorithm correct. However, we will need to give completely new proofs for all three lemmas, as the algorithm has changed significantly. We begin by proving a technical lemma that makes use of the seemingly strange multiplicities of the forwarding steps in an UPDATE.

Lemma 3.4. *If U is an i-UPDATE, S is a SCAN, and $U[2..c2]$ occurs entirely during the sip-section of S, then the following are true.*

(1) During the FORWARDBs, some $\mathtt{A}[i]$-value at least as recent as the start of the FORWARDBs is written to $\mathtt{B}[i]$.

(2) During the ANNOUNCECs, $\mathtt{C}[i]$ must at some point contain either $[0, s]$ or $[1, s]$, where s is the sequence number used by U. Moreover, this must have been written after $S[1_i]$.

(3) During the FORWARDC, $\mathtt{C}[i]$ must at some point contain $[1, s]$. Moreover, this must have been written after $S[1_i]$.

Proof of (1). If at least one of the FORWARDBs succeed, then we are done, so suppose they both fail. The validations of $\mathtt{X}$ ($U[b3]$) certainly succeed, so both failures must be due to interrupted *LL*/*SC* pairs. This implies that during the first FORWARDB, $\mathtt{B}[i]$ is written to, so that the FORWARDB F which causes the second failure must *LL* after this writing, and hence after the start of $U[4]$. But then it also reads $\mathtt{A}[i]$ after the start of $U[4]$, so the value it writes to $\mathtt{B}[i]$ satisfies the recentness condition in the lemma. □

Proof of (2). Similarly, if at least one of the ANNOUNCECs succeed, then we are done, so suppose they both fail. We know $\mathtt{X}$ validates both times, so each failure is due to either a failed $c.seq \neq s$ test or an interrupted *LL*/*SC* pair. If the test fails in either ANNOUNCEC, then clearly $\mathtt{C}[i]$ must contain either $[0, s]$ or $[1, s]$ during the corresponding *LL*. Moreover, $\mathtt{C}[i]$ contains a different sequence number at $S[1_i]$, so this value must be written there after $S[1_i]$, satisfying the lemma. Finally, if both failures are caused by interrupted *LL*/*SC* pairs, then as above some successful ANNOUNCEC or FORWARDC must occur entirely during the execution of the failed ANNOUNCECs, which also satisfies the lemma. □

Proof of (3). If the single FORWARDC succeeds, then we are done. If it fails, this can be owing to either a failed $c = [0, s]$ test or an interrupted *LL*/*SC* pair. By part (2), we know that at some point before the FORWARDC starts (but during S), either $[0, s]$ or $[1, s]$ is written to $\mathtt{C}[i]$. Observe that once $\mathtt{C}[i]$ contains $[0, s]$, the only thing that can be subsequently written to it during S is $[1, s]$. Likewise, once it contains $[1, s]$, nothing can be written to it again during S. The first observation tells us that if the *LL*/*SC* pair is interrupted, this can only be by a writing of $[1, s]$, and the two observations together tell us that a failed $c = [0, s]$ test guarantees that the FORWARDC must in fact see $[1, s]$ in $\mathtt{C}[i]$ and that this value remains through its execution. In either case, the lemma is satisfied. □

We can now prove our three main lemmas as follows.

Proof of Lemma 2.3. Let U' be an earlier i-UPDATE. Then U overwrites U' in $\mathtt{A}[i]$ before S reads $\mathtt{A}[i]$, so the only way for S to return U' is via $\mathtt{B}[i]$. Suppose this is the case, and let U'' be the UPDATE that last writes to $\mathtt{C}[i]$ before S reads it to find $[1, s]$, where s is the matching sequence number ($S[7_1]$). Note that for U'' to successfully FORWARDC, it must successfully validate $\mathtt{X}$ at $U''[c2]$. This forces the whole segment $U''[2..c2]$ to occur within a single sip-section, satisfying the hypothesis of Lemma 3.4. We then need to consider two cases: (1) where this sip-section belongs to S and (2) where it belongs to an earlier SCAN S'.

In (1), Lemma 3.4(1) tells us that during the sip-section of S, an $\texttt{A}[i]$-value at least as recent as that of U (in particular, not that of U') is written to $\texttt{B}[i]$. But any subsequent value written to $\texttt{B}[i]$ can only be more recent, so when S reads $\texttt{B}[i]$ (after its sip-section) it must see a strictly newer value than that of U', contradicting the assumption that S returns U' via $\texttt{B}$.

In (2), Lemma 3.4(2) tells us that at some point after $S'[1_i]$ but during S', either $[0, s]$ or $[1, s]$ must be written to $\texttt{C}[i]$. Furthermore, during the successful FORWARDC by U'', $\texttt{C}[i]$ cannot be touched except by U'' itself, and any ANNOUNCEC or FORWARDC that occurs in between these two write operations to $\texttt{C}[i]$ lies entirely during S' and so must write the same sequence number s as well. It follows that when S reads $\texttt{C}[i]$ to select its sequence number ($S[1_i]$), it sees $[*, s]$, contradicting our assumption that it chooses s. □

Proof of Lemma 2.4. If U starts before S does, then by Lemma 2.3 U is picked up, so let us assume that $U[1] > S[1_1]$. Let U' be any prior i-UPDATE.

Suppose S returns U' via $\texttt{A}$. Then $U'[1] < S[4_i] < U[1]$, otherwise U' would be overwritten in $\texttt{A}[i]$ before S reads it. But U completes before $S[5]$, so U must in fact occur entirely between $S[4_i]$ and $S[5]$; in particular, during the sip-section of S, satisfying the hypothesis of Lemma 3.4. Then by Lemma 3.4(3), $[1, s]$ is written to $\texttt{C}[i]$ some time during S before $S[5]$. Any subsequent ANNOUNCEC or FORWARDC that occurs during S would not write to $\texttt{C}[i]$ again (it would fail the $VL(\texttt{X})$, $c.seq \neq s$ or $c = [0, s]$ test), so this value must remain in $\texttt{C}[i]$ until S reads it in $S[7_1]$. This contradicts the assumption that S returns via $\texttt{A}$ for component i.

Alternatively, suppose that S returns U' via $\texttt{B}$. Let U'' be as in the previous proof, and consider two cases: (1) where $U[1] < S[3]$ and (2) where $U[1] > S[3]$. In (1), applying Lemma 3.4(1) to U'' tells us that an $\texttt{A}[i]$-value at least recent as that of U is written to $\texttt{B}[i]$ before $S[7_2]$ (note that the value in $\texttt{B}[i]$ never gets "older" since successful FORWARDBs are disjoint). In (2), applying the lemma to U leads to the same conclusion, so in either case, we have a contradiction of the assumption that S returns U' via $\texttt{B}$. □

Proof of Lemma 2.5. As in the previous section, it suffices to show that S does not return U, since every subsequent UPDATE would also satisfy the hypothesis of the lemma. S certainly does not return U via $\texttt{A}[i]$ because by the time S reads $\texttt{A}[i]$ U has not even started yet. On the other hand, for S to return U via $\texttt{B}$, some FORWARDB F must read $\texttt{A}[i]$ ($F[b2]$) after $U[1]$ and subsequently validate $\texttt{X}$ ($F[b3]$) during the sip-section of S — clearly not possible since the sip-section precedes $U[1]$. □

Correctness of the algorithm, as usual, is a consequence of these three lemmas. The space complexity of implementing the LL/SC/VL arrays $\texttt{B}$ and $\texttt{C}$, as well as the variable $\texttt{X}$, is $O(mn)$ by the algorithm of [15], and the time complexity of each such operation is $O(1)$. Moreover, only SC requires CAS in this implementation, so CAS complexity is equivalent to SC complexity in our algorithm. We sum up the result in the following theorem.

Theorem 3.5. *The algorithm in Figure 4 implements an atomic multi-writer, single-scanner snapshot object that guarantees wait-freedom.* UPDATE*s and* SCAN*s complete in $O(1)$ and $O(m)$ time, respectively, where a* SCAN *performs $O(1)$ CAS and write operations. The space complexity is $O(mn)$. A* SCAN *by process p has zero and $O(k)$ remote-reference complexity in the DSM and CC models, respectively, where k is the number of components modified by an* UPDATE *since the start of the last* SCAN *by p.*

We remark here without proof that the construction of Jayanti [14] can be used to extend this algorithm to support multiple scanners, with time and space complexities remaining the same. Note that in [14], this final augmentation increased space by a factor of n. This is due to SCANs having to write to $\Theta(m)$ locations, thus having much more conflict amongst themselves. The trick used to resolve these conflicts there, having copies of entire shared arrays for each process, is the cause of the space increase. In our case, however, SCANs only write to a single shared variable, so this trick is unnecessary. The resulting algorithm is given in Figure 5, and its guarantees are summarized in the following theorem.

Theorem 3.6. *The algorithm in Figure 5 implements an atomic multi-writer, multi-scanner snapshot object that guarantees wait-freedom.* UPDATE*s and* SCAN*s complete in $O(1)$ and $O(m)$ time, respectively, where a* SCAN *has $O(1)$ CAS complexity. The space complexity is $O(mn)$.*

4 Remote-reference complexity lowerbound

In this section, we prove that our algorithms of Theorems 2.9, 3.2 and 3.5 are optimal in their remote-reference complexity under the DMS model. In all these cases, we achieve $O(1)$ and zero remote-reference complexities for UPDATEs and SCANs, respectively, by making all of shared memory local to the scanner. The following theorem states that the $O(1)$ complexity for UPDATEs cannot be reduced to zero.

Theorem 4.1. *In any wait-free snapshot algorithm,* UPDATE*s cannot have zero remote-reference complexity on a DSM machine.*

Proof. Assume that each process is equipped with an unbounded block of local memory, and is capable of atomic read-modify-write on its own or any other process' block. Let $\mathcal{P}$ be any protocol that, given this power, implements a two-component, single-writer, single-scanner snapshot object, wherein UPDATEs have zero remote reference complexity. We will prove that $\mathcal{P}$ must fail either linearizability or wait-freedom.

We will present two infinite executions, α and β, involving three processes: p_1 repeatedly increments the first component, p_2 repeatedly increments the second component, and q executes a single SCAN. Observe that each UPDATE by p_1 or p_2 completes in a single atomic action, and q's SCAN consists of a sequence of alternating actions on the local memory blocks of p_1, denoted M_1, and p_2, denoted M_2 (q's own local memory is not shared by any other process, so can be considered internal). Thus, we will denote an increment by p_1 or p_2 as U_1 or U_2, respectively, and an action by q on M_1 or M_2 as S_1 or S_2, respectively.

We define α and β inductively by their respective ith prefixes as follows.

$$\alpha_i = \begin{cases} \epsilon & \text{if } i = 0 \\ \alpha_{i-1} \cdot S_1 \cdot U_1 \cdot U_2 \cdot S_2 & \text{if } i \geq 1 \end{cases}$$

$$\beta_i = \begin{cases} U_2 & \text{if } i = 0 \\ \beta_{i-1} \cdot S_1 \cdot S_2 \cdot U_2 \cdot U_1 & \text{if } i \geq 1. \end{cases}$$

It is easy to see that at every point in α after α_0, at least as many increments to the first component has occurred as to the second, while at every point in β after β_0, strictly fewer increments to the first component has occurred than to the second. Moreover, q initiates its SCAN after α_0 and β_0 in the two respective executions, so any linearizable snapshot returned by q during α must differ from one returned during β. We will prove in the following lemma that q cannot distinguish between α and β, making this impossible. Thus, q's SCAN can never return a linearizable snapshot in a wait-free manner, proving the theorem. □

Lemma 4.2. *For every $i \geq 0$, during α_i and β_i, q has identical histories.*

Proof. We use induction on i, with the following (stronger) inductive hypothesis.

(1_i) q has identical histories in α_i and β_i,

(2_i) p_1 has identical histories in α_i and β_i, and

(3_i) p_2 has identical histories in $\alpha_i \cdot S_1 \cdot U_1 \cdot U_2$ and β_i.

Before proceeding further, we make the key observation that a process' (atomic) action on a given memory block is completely determined by that process' previous history and that block's current state. We will use this fact extensively and without explicit mention in the rest of the proof.

For $i = 0$, (1_0) and (2_0) hold trivially, since neither q nor p_1 takes any steps in α_0 and β_0. To see that (3_0) holds as well, observe that the U_2 in $\alpha_0 \cdot S_1 \cdot U_1 \cdot U_2$ and that in β_0 are the first action by any process on M_2, as well as the first by p_2, so they must in fact be identical actions.

For our inductive case, consider $i \geq 1$ and assume (1_{i-1}), (2_{i-1}) and (3_{i-1}). If $i = 1$, then α_{i-1} and β_{i-1} contain no actions on M_1. If $i > 1$, then by (2_{i-1}), the last U_1 in α_{i-1} and β_{i-1} (which are also the last actions on M_1) are identical. In either case, the state of M_1 is identical at the end of α_{i-1} and β_{i-1}. It follows, via (1_{i-1}), that the subsequent S_1 in α_i and β_i must also be identical. Similarly, by (3_{i-1}), the state of M_2 is identical immediately before the corresponding S_2s, so these are identical as well. This proves (1_i).

We just argued that the last S_1 in α_i and in β_i are identical, so the state of M_1 after these actions is also identical. Since the next action on M_1 is the ith U_1 in both executions (and all previous U_1s already match by (2_{i-1})), they must be identical, proving (2_i).

In similar fashion, (1_i) implies that M_2 is in the same state at the end of α_i and $\beta_{i-1} \cdot S_1 \cdot S_2$. But since no other process acts on M_2 before p_2 performs its next U_2, this together with (3_{i-1}) implies that these corresponding U_2s are identical, proving (3_i) and the lemma. □

Remark. It is easy to see that we can allow our increments to wrap around modular 4 and still maintain disjointness between the two sets of legal snapshots. As a result, $\mathcal{P}$ must fail even if it only has to implement 2-bit components.

5 Conclusion

We considered versions of the snapshot problem with and without the assumptions of single-writer and single-scanner, under the termination requirements of wait-freedom and obstruction-freedom. First, we gave an obstruction-free, time and space optimal algorithm for single-writer, multi-scanner snapshot that requires only read/write and has optimal write complexity. Second, we gave a wait-free, time-optimal algorithm for multi-writer, single-scanner snapshot that uses $O(mn)$ space and requires CAS. It has optimal CAS and write complexities, and optimal remote-reference complexity in the DMS model. Moreover, it can be extended to a time-optimal, $O(mn)$-space algorithm for multi-writer, multi-scanner snapshot, thus improving the space bound on the only known time-optimal algorithm [14], which uses $O(mn^2)$ space. Finally, we prove a lowerbound which implies the above claimed optimality of remote-reference complexity in the DMS model.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt and N. Shavit. Atomic snapshots of shared memory. *Proc. ACM PODC*, 1–14, 1990.

[2] J. H. Anderson. Composite registers. *Proc. ACM PODC*, 15–29, 1990.

[3] J. H. Anderson. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, 16: 75–110, 2003.

[4] J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous PRAM model. *Proc. ACM SPAA*, 340–349, 1990.

[5] H. Attiya, M. Herlihy and O. Rachman. Atomic snapshots using lattice agreement. *Proc. IWDA*, 35–53, 1993.

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

[7] T. D. Chandra and C. Dwork. Using consensus to solve atomic snapshots. Unpublished manuscript, 1993.

[8] C. Dwork, M. Herlihy, S. Plotkin and O. Waarts. Time-lapse snapshots. *Proc. ISTCS*, 154–170, 1992.

[9] S. Haldar and K. Vidyasankar. Elegant constructions of atomic snapshot variables. Technical Report No. 9204, Dept. of Computer Science, Memorial Univ. of Newfoundland, St. John's, NF, Canada, 1992.

[10] M. Herlihy, V. Luchangco and M. Moir. Obstruction-free synchronization: double-ended queues as an example. *Proc. IEEE ICDCS*, 522–529, 2003.

[11] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Programming Languages and Systems*, 12(3): 463–492, 1990.

[12] A. Israeli, A. Shaham and A. Shirazi. Linear-time snapshot implementations in unbalanced systems. *Mathematical Systems Theory*, 28: 469–486, 1995.

[13] P. Jayanti. f-arrays: implementation and applications. *Proc. ACM PODC*, 270–279, 2002.

[14] P. Jayanti. An optimal multi-writer snapshot algorithm. *Proc. ACM STOC*, 723–732, 2005.

[15] P. Jayanti and S. Petrovic. Efficient and practical constructions of LL/SC variables. *Proc. ACM PODC*, 285–294, 2003.

[16] P. Jayanti and S. Petrovic. Efficient wait-free implementation of multiword LL/SC variables. *Proc. IEEE ICDCS*, 59–68, 2005.

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

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

Shared variables:

$\mathtt{A}[1..m]$:	array of machine words, initialized to initial value of snapshot object
$\mathtt{A}_p[1..m]$:	(for each process p) array of machine words, initialized arbitrarily
$\mathtt{B}[1..m]$:	array of snapshot component values, initialized arbitrarily
$\mathtt{C}[1..m]$:	array, with fields $\mathtt{C}[i].vld \in \{0,1\}$ and $\mathtt{C}[i].seq \in \{1,\ldots,m+1\}$, initialized arbitrarily
$\mathtt{X}$:	variable, with fields $\mathtt{X}.toggle \in \{0,1\}$, $\mathtt{X}.phase \in \{1,2,3\}$, $\mathtt{X}.seq \in \{1,\ldots,m+1\}$ and $\mathtt{X}.proc$ (a PID), initialized to $[0,1,*,*]$
$\mathtt{SS}$:	variable, with fields $\mathtt{SS}.toggle \in \{0,1\}$ and $\mathtt{SS}.ss$ (m-word array), initialized to $[1,*]$

```
UPDATE(i, v)
1   A[i] ← v
2   x ← LL(X)
3   if x.phase = 2 then
4       FORWARDB(i)
4'      FORWARDB(i)
5       ANNOUNCEC(i, x.seq)
5'      ANNOUNCEC(i, x.seq)
6       FORWARDC(i, x.seq)

FORWARDB(i)
b1  LL(B[i])
b2  a ← A[i]
b3  if VL(X) then
b4      SC(B[i], a)

ANNOUNCEC(i, s)
a1  c ← LL(C[i])
a2  if VL(X) ∧ c.seq ≠ s then
a3      SC(C[i], [0, s])

FORWARDC(i, s)
c1  c ← LL(C[i])
c2  if VL(X) ∧ c = [0, s] then
c3      SC(C[i], [1, s])

SCAN(p)
1   PUSHLS(p)
2   PUSHLS(p)
3   [toggle, v] ← LL(SS)
4   return v

PUSHLS(p)
    /* Phase 1: choose sequence number */
1   x ← LL(X)
2   if x.phase = 1 then
3       for i = 1 to m do c[i] ← C[i]
4       choose s ∈ {1, ..., m+1} : [*, s] ∉ c
5       x.phase ← 2; x.seq ← s
6       SC(X, x)

    /* Phase 2: read A */
7   x ← LL(X)
8   if x.phase = 2 then
9       for i = 1 to m do A_p[i] ← A[i]
10      x.phase ← 3; x.proc ← p
11      SC(X, x)

    /* Phase 3: read C, B, compile snapshot */
12  x ← LL(X)
13  if x.phase = 3 then
14      p ← x.proc
15      for i = 1 to m do
16          if C[i] = [1, s] then v[i] ← B[i]
17          else v[i] ← A_p[i]
18      [toggle, ss] ← LL(SS)
19      if toggle ≠ x.toggle ∧ VL(X) then
20          SC(SS, [$\overline{toggle}$, v])
21      x.phase ← 1; x.toggle ← $\overline{x.toggle}$
22      SC(X, x)
```

Figure 5: A multi-writer, multi-scanner snapshot algorithm.