Efficient Synchronous Snapshots*

Alex Brodsky
University of Toronto
10 King's College Road
Toronto, Canada
abrodsky@cs.utoronto.ca

Faith Ellen Fich
University of Toronto
10 King's College Road
Toronto, Canada
fich@cs.utoronto.ca

ABSTRACT

A snapshot is an important object in distributed computing whose implementation in asynchronous systems has been studied extensively. It consists of a collection of $m > 1$ components, each storing a value, and supports two atomic operations: an UPDATE of a specified component's value and a SCAN of all components to determine their values at some point in time.

In this paper, we investigate implementations of a multiwriter snapshot object in a synchronous shared memory model. In this setting, we show that a snapshot object can be efficiently implemented and prove a tight tradeoff between the complexity of the SCAN and the UPDATE operations. First, we describe a wait-free implementation that performs UPDATE in $O(1)$ time and SCAN in $O(m)$ time, using only slightly more than twice the amount of space needed to simply store the m values. We also describe a variant that performs UPDATE in $O(1)$ time and SCAN in $O(n)$ time.

Second, we describe a wait-free implementation that performs UPDATE in $O(\log m)$ time and SCAN in $O(1)$ time, and a variant that performs UPDATE in $O(\log n)$ time and SCAN in $O(1)$ time.

Third, we show how to combine these implementations to realize two implementations that perform UPDATE in $\Theta(\log(m/c))$ time and SCAN in $\Theta(c)$ time, for $1 \leq c \leq m$, or perform UPDATE in $\Theta(\log(n/c))$ time and SCAN in $\Theta(c)$ time, for $1 \leq c \leq n$. This implies that Time[UPDATE] $\in O(\log(\min\{m,n\}/\text{Time[SCAN]}))$. We also prove that Time[UPDATE] $\in \Omega(\log(\min\{m,n\}/\text{Time[SCAN]}))$, which matches our upper bound.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*shared memory*;
C.1.2 [**Processor Architectures**]: Multiprocessors—*MIMD*;
D.4.1 [**Operating Systems**]: Process Management—*Concurrency*

General Terms

Algorithms, Design, Theory

*This work was supported by the Natural Sciences and Engineering Research Council of Canada and by Sun Microsystems.

PODC'04, July 25–28, 2004, St. Johns, Newfoundland, Canada.

Keywords

multiprocessor algorithms, shared memory objects

1. INTRODUCTION

A **snapshot** is an important and well-studied object in distributed computing. The object stores a collection of $m > 1$ components, $A[1], A[2], \ldots, A[m]$, each of which stores a value from some domain D. It supports two operations, an atomic UPDATE of the value of a specified component and an atomic SCAN of all components to determine their values at some point in time.

A snapshot object allows processes to obtain consistent backups of the contents of a read-write shared memory. Similarly, it can be used in a sensor network, to collect simultaneous readings from the sensors whose values may change frequently. A snapshot object can also be used to obtain checkpoints of the global state of a distributed computation, which can facilitate debugging of distributed programs. Another application is a time-stamp system, where a process that wants to obtain a time-stamp performs a SCAN to find all time-stamps currently in use, and then UPDATES a component with its new time-stamp, which has value one more than the maximum value it saw.

There are wait-free, and hence, fault tolerant implementations of atomic snapshot objects from registers in asynchronous systems [1, 2, 4, 6, 13, 14]. These have enabled the development of wait-free algorithms for many important tasks using only read-write shared memory [3, 5, 12]. Unfortunately, these implementations are expensive, using time that depends on the number of processes, n, that can access the snapshot object and using registers that are substantially larger than the size of the components, or even unbounded. In some cases, this expense is known to be inherent. For example, any wait-free m-component snapshot object implemented from m registers, for $m < n$, has worst case time complexity $\Omega(mn)$ for SCAN [9] and any wait-free single-writer snapshot object implemented from single-writer registers has worst case time complexity $\Omega(n)$ for UPDATE [15].

In contrast, in synchronous systems, there are simple, efficient implementations of wait-free snapshot objects. Neiger and Singh [16] give a wait-free snapshot implementation using only m registers, each of which is the size of an individual component, so that both UPDATE and SCAN take $m+1$ steps. Their implementation divides the computation time into m read-slots and 1 write-slot. An UPDATE operation waits until a write-slot occurs and then writes the new value into the corresponding register. The SCAN reads the i'th register during the i'th read-slot, pausing during the write-slot. UPDATES occurring in the same time step are linearized in increasing order by component index. A SCAN is linearized in the write-slot that occurs during the SCAN, or if there is no such write-slot, in the write-slot immediately preceding its first step. This write-slot

may also contain UPDATES to some components. These operations are linearized so that UPDATES to components that the SCAN has not yet read occur before the linearization point of the SCAN, which, in turn occurs before UPDATES to components that the SCAN has already read. Neiger and Singh [16] also present a variant of this implementation in which SCAN takes $\lceil \frac{3m}{2} \rceil + 1$ steps and UPDATE takes $\lceil \frac{m}{2} \rceil + 1$ steps. In these implementations, the shared registers store no information except the value of the most recent update. Under this assumption, Neiger and Singh [16] proved that the cost of an UPDATE operation plus a SCAN operation is bounded from below by $\lfloor \frac{3m}{2} \rfloor$, matching their upper bound to within a small constant factor.

They also mention a simple implementation of a single-writer snapshot object (where only the i'th process can update component $A[i]$) in which UPDATE takes constant time, SCAN takes $\Theta(n)$ time, and only n single-writer registers are used, but each register has unbounded size. The idea is that whenever a process UPDATES its component, it writes the value and time of its most recent UPDATE and the values and times of all UPDATES it performed during the last n steps.

We show that the time complexity of this simple algorithm can still be achieved, to within a small constant factor, using only a small constant factor more space than that required to store the result of a single SCAN. Thus, the assumption made by Neiger and Singh is too restrictive, unnecessarily limiting what an implementation can do. Specifically, we give a wait-free implementation that uses $3m$ registers ($2m$ of which are the size of individual components and m of which have only 2 bits), performs UPDATE in $O(1)$ time, and performs SCAN in $O(m)$ time. A small variant uses only m registers, but the same total number of bits and runs slightly faster. As well, we present a third variant that performs UPDATE in $O(1)$ time, and performs SCAN in $O(n)$ time. In the first two implementations (and in Neiger and Singh's bounded space implementations), a process does not write when performing a SCAN and a process only writes to register(s) associated with the component it is UPDATING. Thus, they can also be used for implementing a single-writer snapshot object from single-writer registers.

We also present a different wait-free implementation in which SCAN takes 1 step and UPDATE takes $O(\log m)$ steps, although larger registers are required. (A register must be large enough to hold m component values.) We also describe a straightforward variant in which SCAN takes 1 step and UPDATE takes $O(\log n)$ steps.

Finally, we show how to combine our implementations to obtain a tradeoff: Time[UPDATE] $\in O(\log(\min\{m,n\}/\text{Time[SCAN]}))$, and prove that Time[UPDATE] $\in \Omega(\log(\min\{m,n\}/\text{Time[SCAN]}))$, which matches the upper bound.

2. THE MODEL

We consider a synchronous distributed system in which a set of n deterministic processes, $p_1, p_2, \ldots, p_n$, communicate by reading and writing shared multiwriter registers. The operation $l \leftarrow \texttt{read}\ r$ reads the shared register r and stores the result in the local variable l. Similarly, the operation $\texttt{write}\ l \rightarrow r$ writes the value of the local variable l into the shared register r. A process step consists of a single read or write operation and a bounded number of local operations that do not access any shared registers. At each time step, each process can perform at most one step. The processes share a global clock, which is incremented modulo k at the end of each step. We assume that k can be fixed at run-time.

We assume linearizability, so all shared memory operations appear to take place atomically in some order between the time they actually begin and end. For synchronous computation, this means that reads and writes by processes that occur during the same time step give results that are consistent with these operations being performed sequentially in some order chosen by a scheduler. If all writes are scheduled before all reads at each time step, the semantics are the same as the ARBITRARY PRAM [10, 11]. In the ARBITRARY PRAM model, an adversary chooses an arbitrary process to successfully perform its write from among all processes attempting to simultaneously write to the same shared register. More generally, in our model, two different reads that occur during the same time step can return different values if a write is linearized between them.

The values stored in the snapshot object are from a domain D, each of which can be stored in one register. The complexity of a snapshot implementation is measured by the worst case number of steps a process takes to perform an UPDATE and the worst case number of steps a process takes to perform a SCAN.

3. A SNAPSHOT IMPLEMENTATION WITH CONSTANT UPDATE TIME

Here, we present an implementation of a snapshot object that performs UPDATES in at most 5 steps and performs SCANS using at most $6m-1$ steps. We begin by giving an overview of the ideas behind the implementation. Then we give the complete implementation and prove its correctness. In Section 3.5, we describe a similar implementation that performs UPDATES in a constant number of steps and performs SCANS in $O(n)$ steps. It uses bounded timestamps, which we define more formally and explain how to implement in Section 3.4.

3.1 Overview

The collection of components is represented by three arrays of m multiwriter registers, A_0, A_1, and T. Each register in A_0 and A_1 stores a value from the domain D. Each register in T stores only 2 bits and is used like a modulo 4 counter. Only UPDATE operations modify these shared registers.

The implementation divides the computation into two alternating phases, 0-**write phases** and 1-**write phases**, each consisting of $2m$ consecutive steps. In a 0-write phase, array A_0 may be modified, but array A_1 cannot be modified, while in a 1-write phase, the reverse is true. (See Figures 1a and b.) To ensure a consistent snapshot, arrays A_0 and A_1 will only be read during phases in which they will not be modified. The implementation uses $\Phi \in \{0,1\}$ to denote the parity of the current write-phase, which is 0 for a 0-write phase and 1 for a 1-write phase.

The SCAN operation takes two complete consecutive phases to be performed and will be linearized between these two phases. For convenience, we call these two phases Phase I and Phase II. (See Figure 1c.) Note that Phase I can be either a 0-write phase or a 1-write phase. Phase II will have the opposite parity. During each of Phase I and Phase II, the process performing the SCAN collects the values in one of A_0 or A_1. It collects the values in T during both phases. Then, for each component, it must determine whether the corresponding location of A_0 or A_1 contains the value of the most recent UPDATE to that component occurring prior to the linearization point. To do this, the tag stored in the corresponding location of T is used.

An UPDATE operation writes the updated value to the location corresponding to the updated component in one of A_0 or A_1, depending on the parity of the current phase. In addition, it may modify the corresponding tag.

Rule 3.1. UPDATE *increments the corresponding tag if and only if its parity does not match the parity of the current phase.*

For example, in Figure 1d, the UPDATE operations in the third

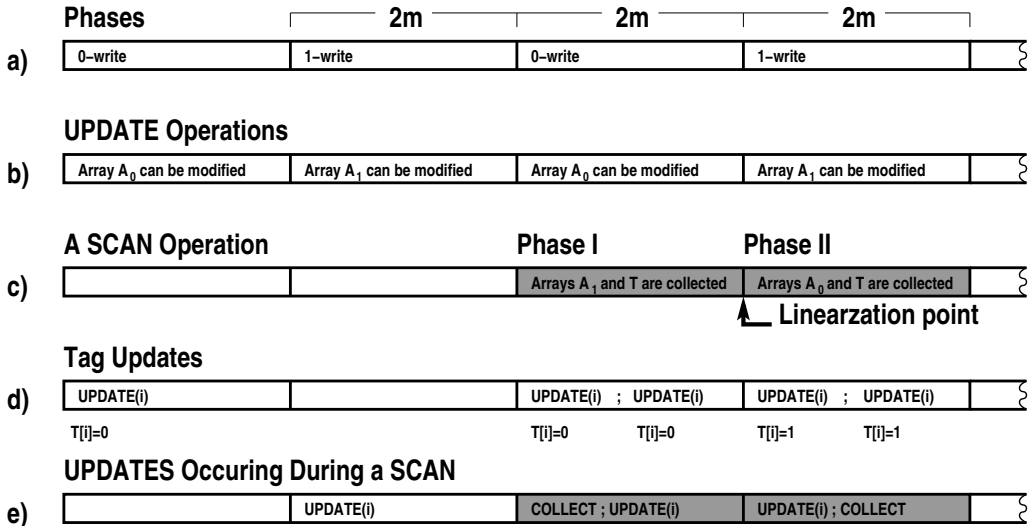

Figure 1: Implementation overview.

phase, which is a 0-write phase, do not increment register $T[i]$ because the parity matches. However, the first UPDATE in the forth phase increments $T[i]$ because the previous UPDATE occurred in a phase with the opposite parity.

Rule 3.1 ensures that the parity of a component's tag indicates whether its most recent UPDATE occurred in a 0-write phase or a 1-write phase, and hence, whether A_0 or A_1 stores the value of this most recent UPDATE.

A change in the value of a tag between Phase I and Phase II of a SCAN indicates that an UPDATE to the corresponding component has occurred since the last time the tag was read (i.e., in Phase I). If the parity of the new tag (the one read in Phase II) is the same as the parity of Phase I, then UPDATES of that component occurred during Phase I and the value read during Phase II should be returned. The other case is when the parity of the new tag is the same as the parity of Phase II. Then one or more UPDATES occurred during Phase II and the tag must have had the opposite parity at the linearization point. Again the value read during Phase II should be returned.

For example, consider the situation depicted in Figure 1e. There are UPDATE operations to component i occurring in three consecutive phases. A SCAN operation starts at the beginning of the second of these phases. Because its linearization point occurs between the second and third of these phases, it should return the value of the second UPDATE. In Phase I of this SCAN, the value and tag corresponding to component i are collected before the second UPDATE occurs, but, in Phase II, they are collected after the third UPDATE occurs. Note that these two tags will have the same parity. This would also happen if only the first of these three UPDATES occurred, in which case, the value of this first UPDATE should be returned. Thus, knowing the parity of the phase in which an UPDATE occurs does not suffice to determine whether its value should be returned. However, the fact that the tags read in Phase I and Phase II differ by 2 reveals that an UPDATE occurred during Phase I, and hence, its value should be returned.

In summary, Phase II of a SCAN determines, for each component i, whether register $A_0[i]$ or $A_1[i]$ holds the value of the most recent UPDATE that occurred prior to Phase II, and returns the value stored therein. During the collect in Phase II, the implementation first checks whether the parity of a tag matches the parity of the preceding phase, and then whether the tag differs from the corresponding tag collected in Phase I. If either of these two conditions are met, the register collected in Phase II contains the value of the most recent UPDATE. Otherwise, the value of the most recent UPDATE is in the corresponding register collected in Phase I.

3.2 The Implementation and Its Complexity

Detailed implementations of SCAN and UPDATE are given in Figures 2 and 3. Here, we discuss the worst case number of time steps used by a process to perform each of these two operations.

For correctness, the actions of an UPDATE operation must not span multiple phases. Thus, if an UPDATE begins at the last or second last step of a phase, it waits until the start of the next phase before proceeding. The first write modifies either array A_0 or array A_1, depending on the current write-phase. The write is followed

```
proc UPDATE(i, v)
  variable Φ, h;

  if (time mod 2m) > (2m − 3) then
    // wait until next phase begins
    wait until time ≡ 0 mod 2m;
  end if

  // determine the parity of the phase
  Φ ← if (time mod 4m) < 2m then 0 else 1;

  // Write new value
  write v → A_Φ[i];

  // Check and increment tag
  h ← read T[i];
  if h ≠ Φ mod 2 then
    h ← h + 1 mod 4;
    write h → T[i];
  end if
end proc
```

Figure 2: Constant-time UPDATE implementation.

```
proc SCAN
  variable Result[1...m];
  variable Φ, h, g[1...m];

  // wait until next phase begins
  wait until time ≡ 0 mod 2m;

  // determine the parity of the phase
  Φ ← if (time mod 4m) < 2m then 0 else 1;

  // Phase I: Φ-write phase
  // collect arrays A_Φ̄ and T
  for i = 1...m do
    Result[i] ← read A_Φ̄[i];
    g[i] ← read T[i];
  end for

  // Phase II: Φ̄-write phase
  for i = 1...m do
    h ← read T[i];
    if (h ≡ Φ mod 2) or (h ≠ g[i]) then
      Result[i] ← read A_Φ[i];
    end if
  end for

  return(Result);
end proc
```

Figure 3: Linear-time SCAN implementation.

by a read of register $T[i]$. The second write only occurs when the value of register $T[i]$ must be incremented. The increment occurs if and only if the parity of the current phase does not match the parity of register $T[i]$. Thus, in the worst case, an UPDATE takes five steps.

The SCAN operation first waits until a new phase begins. This takes at most $2m-1$ steps. At the start of the phase, the SCAN collects array A_1 or array A_0, depending on whether the current phase is a 0-write phase or a 1-write phase, and the tags in T. Thus, Phase I consists of exactly $2m$ read operations. Since the values in $A_{\overline{\Phi}}$ do not change during a phase of parity Φ, it does not actually matter in what order they are read.

In Phase II, array T is collected again. As each tag is read, the implementation checks the tag's parity and whether the tag differs from the previously collected one. It uses this information to decide whether to replace the corresponding value read from $A_{\overline{\Phi}}$ in Phase I with the corresponding value in A_Φ. In Phase II, at most $2m$ read operations are performed, and so, the total number of steps required for a SCAN is at most $6m-1$.

The time complexity of the operations can be reduced by almost a factor of 2 by using larger registers. Specifically, A_0, A_1, and T can be combined into a single array consisting of m registers, each of which has 3 fields, $a_0, a_1 \in D$ and $t \in \{0,1,2,3\}$. Each phase now consists of m steps instead of $2m$. Thus, a SCAN operation would take at most $3m-1$ steps, of which $2m$ steps would be reads. An UPDATE operation would perform one read step and one write step. It might also have to wait for one step, so that its actions do not span two different phases. Thus, it would take at most 3 steps. Note that during a 0-write phase, field a_1 of a register would not be changed and during a 1-write phase, field a_0 of a register would not be changed. In other words, these fields would be written back unmodified.

3.3 Proof of Correctness

The proof of correctness of the implementation given in Figures 2 and 3 depends on the following simple observation.

Observation 3.2. *If an* UPDATE *to component i occurs during a Φ-write phase, for $\Phi \in \{0,1\}$, then the value of the* UPDATE *is written to $A_\Phi[i]$ and the parity of $T[i]$ is or becomes Φ.*

Another observation is that the only writes to A_Φ are by UPDATES that occur during Φ-write phases and the only reads from A_Φ are by SCANS that occur during $\overline{\Phi}$-write phases. Thus, no register in A_Φ is both read and written in the same phase. In particular, this means that details of how reads and writes to A_Φ (or $A_{\overline{\Phi}}$) are linearized at each time step are not important.

In contrast, the tags in T can be read and written in the same phase. However, in each phase, each tag can be changed at most once.

Lemma 3.3. *The register $T[i]$ is incremented at most once per phase.*

PROOF. Note that every write to $T[i]$ is preceded by a read from $T[i]$ by the same process in the previous step of the phase.

Let t denote the tag in register $T[i]$ at the beginning of the phase. If the parity of t matches the parity of the phase, then every process that performs an UPDATE of component i during this phase will read t from $T[i]$, and hence, decide not to write to $T[i]$. Therefore, suppose that the parity of t does not match the parity of the phase.

Consider the first time step in the phase during which a new value is written to $T[i]$. Then all processes writing to $T[i]$ during this step will have read the value t in the preceding step, and hence, will write the same value, $t+1 \bmod 4$. Thus, processes reading from $T[i]$ during this step will read either t or $t+1 \bmod 4$. At the next step, some processes may also write $t+1 \bmod 4$ to $T[i]$. However, no process reading $t+1 \bmod 4$ from $T[i]$ will write to $T[i]$ during this phase, since the parity of $t+1 \bmod 4$ matches the parity of the phase. Thus, no additional writes to $T[i]$ occur at any later steps in the phase. □

The next lemma is key to the correctness of the implementation.

Lemma 3.4. *Suppose that Phase I of a* SCAN *occurs in a Φ-write phase, for $\Phi \in \{0,1\}$. Then, register $A_\Phi[i]$ contains the value of the most recent* UPDATE *prior to the start of Phase II of the* SCAN *if and only if $h \equiv \Phi \bmod 2$ or $h \neq g[i]$.*

PROOF. We consider four cases. First suppose that $h \equiv \Phi \bmod 2$. At the point during Phase II of the SCAN at which $T[i]$ is read, it has value h. (See Figure 4a.) Then, prior to this point, the most recent UPDATE must have occurred during a Φ-write phase and register $A_\Phi[i]$ must contain the value of this UPDATE. This follows from Observation 3.2. Since Phase II is a $\overline{\Phi}$-write phase, this UPDATE must have occurred prior to the start of Phase II. Hence register $A_\Phi[i]$ contains the value of the most recent UPDATE prior to the start of Phase II.

Now suppose that $h \not\equiv \Phi \bmod 2$ and $h = g[i]$. At the point during Phase I of the SCAN at which $T[i]$ is read, it has value $g[i]$. Then, prior to this point, the most recent UPDATE must have occurred during a $\overline{\Phi}$-write phase and register $A_{\overline{\Phi}}[i]$ must contain the value of this UPDATE. Again, this follows from Observation 3.2. Since Phase I is a Φ-write phase, this UPDATE must have occurred prior to the start of Phase I. (See Figure 4b.) Furthermore, no UPDATE to component i could have occurred during Phase I of the SCAN after the point at which $T[i]$ is read. Otherwise, register $T[i]$ would be incremented from $g[i]$ to $g[i]+1 \bmod 4$ in Phase I. By Lemma 3.3, $T[i]$ is incremented at most once in each of Phase I and Phase II. This implies

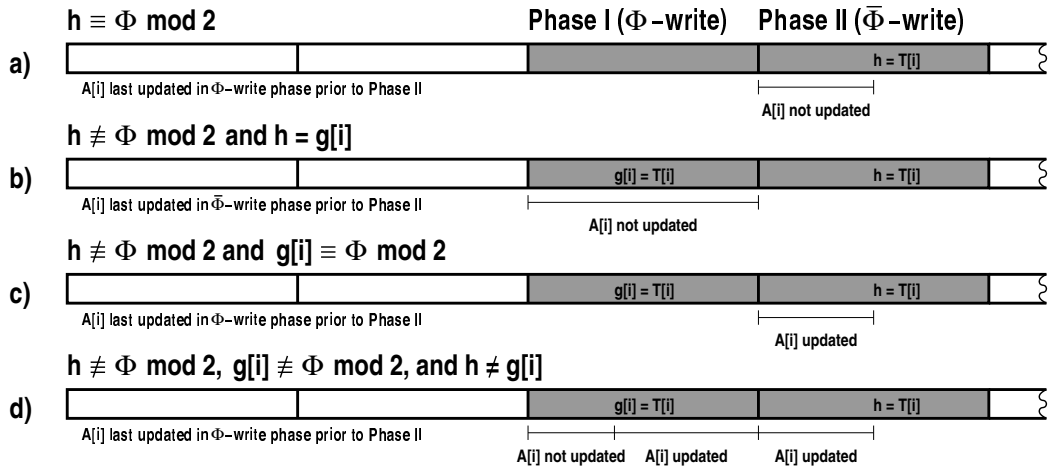

Figure 4: Determining the value of the most recent UPDATE**.**

that when $T[i]$ is read by the SCAN during Phase II, it is not equal to $g[i]$. This contradicts the assumption that $h = g[i]$. Thus, in this case, register $A_{\overline{\Phi}}[i]$ contains the value of the most recent UPDATE prior to the start of Phase II.

Now suppose that $h \not\equiv \Phi \bmod 2$ and $g[i] \equiv \Phi \bmod 2$. At the point during Phase I of the SCAN at which $T[i]$ is read, it has value $g[i]$. Then, after this point until Phase I ends, no UPDATE changes $T[i]$, because its parity, Φ, matches the parity of the phase. This is illustrated in Figure 4c. Thus, $T[i] \equiv \Phi \bmod 2$ immediately prior to Phase II. By Observation 3.2, the most recent UPDATE prior to the start of Phase II must have occurred during a Φ-write phase and its value is contained in register $A_\Phi[i]$.

Finally, suppose that $h \not\equiv \Phi \bmod 2$, $g[i] \not\equiv \Phi \bmod 2$, and $h \neq g[i]$. Then $h = g[i] + 2 \bmod 4$. By Lemma 3.3, $T[i]$ may only be incremented once per phase. Thus, the value of $T[i]$ must have been incremented twice between the first and second time that it was read by the SCAN: once in Phase I (after the first read of register $T[i]$) and once in Phase II (before the second read of register $T[i]$). This is illustrated in Figure 4d. Therefore, the most recent UPDATE prior to Phase II occurred in Phase I. Since Phase I is a Φ-write phase register $A_\Phi[i]$ contains its value. □

Next, we explain where to linearize each UPDATE and SCAN operation. An UPDATE operation can be linearized at the point of its first write (to either A_0 or A_1, depending on the phase in which it occurs). Note that the first `if` statement ensures that this write does not occur at the end of a phase. A SCAN operation waits until the beginning of a phase and performs reads during two consecutive phases. It is linearized between these two phases. Since multiple SCANS linearized at the same point should return the same result, the order in which they are linearized does not matter.

Finally, we claim that the result returned by each SCAN operation is consistent with this linearization. To see this, consider a SCAN and suppose its Phase I occurs during a Φ-write phase. From the algorithm, $Result[i] = A_\Phi[i]$, if $h \equiv \Phi \bmod 2$ or $h \neq g[i]$, and $Result[i] = A_{\overline{\Phi}}[i]$, otherwise. Then Lemma 3.4 implies that $Result[i]$ contains the value of the most recent UPDATE to component i that occurred prior to the start of Phase II. Hence, the result returned by the SCAN is consistent with what would be obtained by performing this operation atomically between Phase I and Phase II.

One important feature of our implementation is that SCAN does not perform any writes. Thus, if the multiwriter registers are replaced by single-writer registers, we immediately get an implementation of a single-writer snapshot object.

3.4 A Bounded Time-Stamp Object

In this section, we give specifications for a bounded time-stamp object and explain how to implement such an object shared by n processes using only registers. In the next subsection, we describe our snapshot implementation using this object.

An **r-time-stamp object** is used to atomically acquire, release, and compare time-stamps. The object is parameterized by r, the maximum number of stamps that can be in use at any time. The object's three operations are: `GetHistory`, `AcquireStamp`, and `ReleaseStamp`. A process uses `AcquireStamp` to acquire a stamp and `ReleaseStamp` to release a set of stamps. A stamp that has been acquired by a process but has not subsequently been released by that process is said to be **in use** by that process. Operation `AcquireStamp` returns a new stamp that was not in use by any process at the previous step. If multiple `AcquireStamp` operations return at the same step, they must all return the same stamp. A `GetHistory` operation returns a sequence of stamps, in the order that they were last returned by `AcquireStamp` operations. This sequence must include all stamps that are in use by any process. It may also include stamps that have already been released by all processes that had acquired them.

To implement a wait-free, linearizable r-time-stamp object, we use n shared single-writer registers $U_1, \ldots, U_n$, and two shared multiwriter registers L and B. The set of stamps that are currently in use by process j are recorded in U_j. This set is also stored locally by process j in a persistent variable $inuse_j$. The multiwriter register L has two fields: *sequence*, which is a sequence of at most $r + 2n - 1$ distinct stamps, and *counter*, which is a value from $\{1, \ldots, n\}$. The stamps in $L.sequence$ are stored in the order they were last returned by `AcquireStamp`. Initially $L.sequence$ is empty and $L.counter = 1$. Detailed implementations of `GetHistory`, `ReleaseStamp`, and `AcquireStamp` are given in Figures 5, 6, and 7.

When a process performs `GetHistory`, it reads L and returns the contents of its *sequence* field. To perform `ReleaseStamp`(T), a process removes the stamps in T from its set of in use stamps recorded in $inuse_j$ and its single-writer register U_j. The set of stamps currently in use is $\cup_j U_j$ which, by definition, has size at most r.

Acquiring stamps is slightly more complicated. A process se-

```
proc GetHistory()
  variable l;

  l ← read L;
  return(l.sequence);
end proc
```

Figure 5: Implementation of `GetHistory()`.

```
proc ReleaseStamp_j(T)
  variable inuse_j;

  // remove stamps in T from U_j
  inuse_j ← inuse_j \ T;
  write inuse_j → U_j;
end proc
```

Figure 6: Implementation of `ReleaseStamp()`.

lects a new stamp from among those not in $L.sequence$, appends it to $L.sequence$ and adds it to its set of in use stamps. Processes all select a new stamp in the same way, by choosing the smallest possible stamp that is not already in $L.sequence$. This ensures that all processes that return from `AcquireStamp` at the same step will return the same stamp.

Before selecting a new stamp, a process participates in a background cleanup procedure to remove stamps from $L.sequence$ that are no longer in use by any process. The idea is to cycle through the single-writer registers $U_1, \ldots, U_n$, in order, and record which stamps in $L.sequence$ are in use. At the end of the cycle, all the stamps that were not in use during the entire cycle are removed from $L.sequence$. Stamps released during the cycle will also be removed from $L.sequence$, provided they were not recorded before they were released. Each time a process performs `AcquireStamp`, it records information from one single-writer register. The field $L.counter$ specifies which single-writer register to examine.

The stamps in use during the cycle are recorded in the multi-writer register B. It contains a bit vector, with one bit corresponding to each stamp in $L.sequence$ at the beginning of the cycle. Stamps that are appended to $L.sequence$ during the cycle are not considered for removal. The bit vector B is initialized with the in use information from U_1. Some of its bits are set to 1 as the in use information from subsequent single-writer registers is recorded. At the end of the cycle, $L.sequence$ is compressed by removing all stamps whose corresponding bit in B is still 0.

Note that B never contains more than r bits with value 1. This is because there are at most r stamps in use at the start of the cycle and any new stamps acquired during the cycle are not recorded in B. Exactly n new stamps are acquired during the cycle, so, at the end of the cycle after it has been compressed, $L.sequence$ will contain at most $r+n$ stamps. This implies that $|B| \leq r+n$. Since $n-1$ new stamps are appended to $L.sequence$ during the cycle before it is compressed, $|L.sequence| \leq r+2n-1$. Thus at most $r+2n-1$ different stamps are ever used. This implies that $(r+2n-1)\lceil \log(r+2n-1) \rceil$ bits suffice to store $L.sequence$ and $r\log(r+2n-1)\rceil$ bits suffice in each single-writer register. In total, our implementation of an r-time-stamp uses $O(rn\log(r+n))$ bits of shared space.

To ensure that processes at different places in the cleanup procedure do not interfere with one another, processes synchronize by waiting for a most 5 steps so that they start at times that are multiples of 6. Thus the `AcquireStamp` takes at most 11 steps. Both `GetHistory` and `ReleaseStamp` take one step each.

Finally, note that the multiwriter registers L and B are only written to during `AcquireStamp`. Our algorithm ensures that processes writing to the same register at the same step write the same information. Thus it does not matter in what order these writes are linearized.

```
proc AcquireStamp_j()
  variable l, u, b, inuse_j, s, seq;

  // Synchronize and read registers
  wait until time ≡ 0 mod 6;
  l ← read L;
  u ← read U_{l.counter};

  // initialize or read bit vector
  if l.counter = 1 then
    b ← 0^{|l.sequence|};
  else
    b ← read B;
  end if

  // flag all stamps that are in use
  for i = 1...|b|
    if l.sequence_i ∈ u then b_i ← 1;

  // save B or compress l.sequence
  if l.counter ≠ n then
    write b → B;
  else
    seq ← ();
    for i = 1...|b|
      if b_i = 1 then seq ← (seq; l.sequence_i);
    seq ← (seq; l.sequence_{|b|+1} ··· l.sequence_{|l.sequence|});
    l.sequence ← seq;
  end if

  // select stamp, append to sequence, increment
  // counter, and save sequence & counter
  Select smallest s ∉ seq;
  l.sequence ← (seq; s);
  l.counter ← (l.counter mod n) + 1;
  write l → L;

  // add stamp to U_j and return stamp
  inuse_j ← inuse_j ∪ {s};
  write inuse_j → U_j;
  return(s);
end proc
```

Figure 7: Implementation of `AcquireStamp()`.

3.5 An Implementation for $n < m$

Now we describe an implementation of an m-component snapshot object that performs `UPDATE` in a constant number of steps and `SCAN` in $O(n)$ steps. Like the implementation in Section 3.2, this implementation uses a two-pass `SCAN`, but uses bounded time-stamps, instead of tags, to order the `UPDATES`.

Our implementation uses two arrays, A_0 and A_1. Each array consists of n single-writer registers, where registers $A_0[j]$ and $A_1[j]$ can only be written by process j. Each register stores an m-element array that contains the UPDATES performed by process j. Each element of these arrays holds a pair (v,s), consisting of a value $v \in D$ and a time-stamp s whose size is logarithmic in m. Thus, the registers used are $\Theta(m\log(m|D|))$ bits in size and $\Theta(m^2\log(m|D|))$ bits are used in total.

As in Section 3, time is divided into two phases, a 0-write phase and a 1-write phase. An UPDATE of snapshot component i by process j during a Φ-write phase writes the new value and a new time-stamp (it obtains by performing `AcquireStamp`) into the i'th array element of register $A_\Phi[j]$. Then process j performs `ReleaseStamp` to release the previous time-stamp it had written there (which it had to remember). This takes at most 13 steps. As before, an UPDATE must not span multiple phases. Thus, if the UPDATE begins less than 13 steps before the end of a phase, it waits until the beginning of the next phase. Hence, an UPDATE takes at most 25 steps. If the length of a write-phase is divisible by 6, then this can be reduced to 13 steps.

A SCAN consists of two phases. In each phase, the SCAN begins by performing `GetHistory` and then collects the contents of an array of n registers. This takes $n+1$ steps, since `GetHistory` takes 1 step. In Phase I, the process reads $A_{\overline{\Phi}}$ and, in Phase II, it reads A_Φ, where Φ denotes the parity of Phase I. The linearization point of the SCAN is between the two phases.

As it performs a SCAN, a process locally stores the value and the time-stamp of the most recent UPDATE it has seen for each component. To do so, it compares each time-stamp it reads with the time-stamp it has stored for the same component. If this time-stamp is more recent according to the result of the `GetHistory` operation performed at the beginning of the current phase, the process records this time-stamp and its corresponding value in place of the ones it has stored. By the end of Phase II, the process has the value of the most recent UPDATE performed at each component prior to the beginning of Phase II. It returns this array of values. Its correctness depends on the fact that A_Φ is not modified during Phase II.

To ensure there is no interference between SCANS and UPDATES, a SCAN must first wait until the beginning of a phase. Thus, a SCAN takes at most $3n+2$ steps.

4. A SNAPSHOT IMPLEMENTATION WITH CONSTANT SCAN TIME

Intriguingly, a SCAN operation need not take $\Omega(m)$ steps. In this section, we describe an m-component multiwriter snapshot object whose SCAN complexity is exactly 1 step, and whose UPDATE complexity is $2\lceil\log m\rceil+4$ steps. The implementation uses $2m-1$ registers. We begin with an overview of the implementation and informally argue its correctness. The detailed implementation is given in Figures 9 and 10.

The implementation consists of $2m-1$ multiwriter registers, denoted by $A[1]\ldots A[2m-1]$, that are arranged in a balanced (heap-shaped) strictly binary tree, where register $A[j]$ is the parent of the two sibling registers $A[2j]$ and $A[2j+1]$; see Figure 8. The m leaf registers, $A[m],\ldots,A[2m-1]$ each stores one component of the snapshot object. Every other node stores one component for each leaf in the subtree rooted at that node. Thus, the root register, $A[1]$, stores the entire vector of m components.

The SCAN consists of a single read operation that reads the root register and returns its contents. Because it occurs at a single step, this step must be its linearization point.

An UPDATE to the i'th component consists of at most $\lceil\log m\rceil+1$ writes and at most $\lceil\log m\rceil+1$ reads. It begins by mapping component i to register j such that component 1 is stored at the leftmost leaf, component 2 is stored at the second leftmost leaf, and so on. The UPDATE then proceeds by writing the new value to the register at this leaf, reading it back, and waiting for one step. This procedure ensures that all concurrent updates to component i proceed with the same value. The UPDATE then enters a loop; in each iteration it reads the value stored at its sibling, proceeds to its parent, and writes the vector of values it has collected to the register there. It continues in this manner up the tree, until it writes the entire vector of m components to the register $A[1]$ at the root; see Figure 8. The linearization point of the UPDATE occurs at the step during which the write to $A[1]$ occurs.

Since all UPDATES wait until $time \equiv 0 \bmod 2$ before accessing shared memory and wait an additional step after the first read, the total number of steps used by an UPDATE is at most $2\lceil\log m\rceil+4$. These brief delays ensure that all UPDATEs perform writes to shared registers during even time steps and perform reads from shared registers during odd time steps. Hence, these reads and writes do not interfere with one another.

It is also the case that all processes simultaneously writing to the same shared register write the same value. This is because after the initial write, the process reads the same register, which results in agreement between all concurrent UPDATES of the same component. Furthermore, immediately after a process writes to the register at a node, it reads from the register at its sibling. Hence, all processes that simultaneously arrive at a node will have collected the same sequence of values, whether they came from its left child or its right child.

This implementation uses $2m-1$ registers. The maximum size of any register is $m\log|D|$ bits, needed to store the entire result. However, if not all registers need to be the same size, the total space requirement decreases from $\Theta(m^2\log|D|)$ bits to $O(m\log m\log|D|)$ bits.

4.1 An Implementation for $n < m$

There is a similar implementation that performs a SCAN in one step and an UPDATE in $\Theta(\log n)$ steps. This is an improvement when the number of processes, n, is significantly smaller than the number of components, m.

The implementation uses a balanced, strictly binary tree of multi-writer registers with n leaves, one associated with each process. The root of the tree contains an array of m components, which is read and returned by processes performing SCANS.

The other registers each contain a set of components that are to be updated together with their new values. A process that wants to perform an UPDATE first writes to its associated leaf. It writes

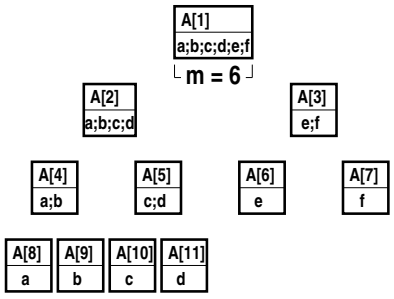

Figure 8: The registers are organized into a balanced strictly binary tree.

```
proc SCAN
  variable Result[1...m];
  Result ← read A[1];
  return(Result);
end proc
```

Figure 9: Implementation of the constant-time SCAN.

```
proc UPDATE(i, v)
  variable r, s;

  wait until time ≡ 0 mod 2;

  // store component i at the i'th leftmost leaf
  j ← m + (i + 2^⌈log m⌉ − 1 mod m);

  write v → A[j];      // begin update
  r ← read A[j];       // sync concurrent writes
  wait for 1 step;     // skip write step

  while j ≠ 1 do
    if j ≡ 1 mod 2 then
      s ← read A[j − 1];   // read left sibling
      r ← (s;r);           // and prepend to r
    else
      s ← read A[j + 1];   // read right sibling
      r ← (r;s);           // and append to r
    end if

    j ← ⌊j/2⌋;
    write r → A[j];      // update parent
  end while
end proc
```

Figure 10: Implementation of the logarithmic-time UPDATE.

both the component it wants to UPDATE and its new value. Then the process progresses up the tree, at each internal node combining the information from its two children. Specifically, in a phase of 3 steps, a process at a node reads the information in its sibling's register, erases the information in the node's register and goes to their parent. There it writes the union of the component sets from these two registers, together with one new value for each. (When there is a different new value for the same component at the two children, the value in the left child is chosen.)

When a process reaches the root, instead of writing a set of components and new values, it reads the array at the root, locally updates its components as specified by the information from its two children, and writes the updated array back to register at the root.

A process waits for at most 2 steps until the beginning of a phase, before writing to its associated leaf. This ensures that all processes proceed up the tree without interfering with one another. Thus, a process whose associated leaf is at depth d takes at most $3d+4$ steps. Since the tree is balanced, $d \in \Theta(\log n)$.

5. A TIGHT TRADE-OFF BETWEEN SCAN TIME AND UPDATE TIME

Let S denote the worst case time complexity of SCAN and U denote the worst case time complexity of UPDATE in a particular implementation. We show how to combine the constant-time UPDATE implementations described in Section 3 and the constant-time SCAN implementations described in Section 4 to obtain implementations such that $U \in O(\log(\min\{m,n\}/S))$. Then, we use an information theoretic argument to prove that this tradeoff is tight.

5.1 The Upper Bound

Theorem 5.1. *For $1 \le c \le \min\{m,n\}$, there exists an m-component synchronous multiwriter snapshot object implementation such that the complexity of a* SCAN *operation is in $\Theta(c)$ and the complexity of an* UPDATE *operation is in $\Theta(\log(\min\{m,n\}/c))$.*

PROOF. For the purposes of presentation, assume that the components are numbered $0 \ldots m-1$ and that the processes are numbered $0 \ldots n-1$.

First, consider the case when $m \le n$. Let $1 \le c \le m$, let $f = \lceil m/c \rceil$, and define a **chunk** to be a vector from D^f (or D^{f-1}).

The object we construct is based on a central c-component snapshot object with constant UPDATE time, which is used to store c chunks. Each chunk is represented using an f or $f-1$ component snapshot object with constant SCAN time. This is illustrated in Figure 11.

The m components are assigned in a roughly equal manner to the chunks. Specifically, the i'th component is stored in vector element $\lfloor i/c \rfloor$ of chunk $(i \bmod c)$. An UPDATE of component i first updates element $\lfloor i/c \rfloor$ of chunk $(i \bmod c)$. This is accomplished in the same manner as in Section 4 and takes $2\log f + O(1)$ steps. The only modification is that the last write (to the root register of the chunk) is replaced by an UPDATE to component $(i \bmod c)$ of the central snapshot object. The linearization point of the UPDATE to the central snapshot object is used. The complexity of the UPDATE operation is in $2\log f + O(1) \subseteq \Theta(\log(m/c))$.

The SCAN, which is invoked on the central snapshot object, reads the c chunks, takes $\Theta(c)$ steps. The chunks are then merged to yield the result. The SCAN is linearized between its two complete phases.

Second, consider the case when $n < m$. Let $1 \le c \le n$, let $g = \lceil n/c \rceil$, define a **group** to be a set of g (or $g-1$) processes, and a **group snapshot** as a partial m-component snapshot comprising the UPDATES of the g processes.

The implementation is centered on a c-process, m-component snapshot object with a constant-time UPDATE, which uses multiwriter registers in place of single-writer registers and is used to store c group snapshots; one for each group of g (or $g-1$) processes. Each group snapshot is maintained by a g-process, m-component snapshot object with a constant-time SCAN. This implementation has a similar structure to the one described above, but comprises implementations described in Subsections 3.5 and 4.1.

The n processes are assigned in a roughly equal manner to the groups. Specifically, the j'th process belongs to group $(j \bmod c)$. An UPDATE by process j of component i first updates the group snapshot of group $(j \bmod c)$; this is accomplished in the same manner as in Subsection 4.1 and takes $\Theta(\log g)$ steps. The last write (to the root register of the group snapshot) is replaced by an UPDATE on the c-process snapshot object, which identifies processes by their group ID rather then their process ID. Consequently, the UPDATE is invoked with a set of component updates, but only needs to acquire one time-stamp. The linearization point of the UPDATE to the central snapshot object is used. The complexity of the UPDATE operation is in $\Theta(\log g) \subseteq \Theta(\log(n/c))$.

The SCAN, which is invoked on the c-process, m-component snapshot object takes $\Theta(c)$ steps to merge the c group snapshots into one snapshot. The SCAN is linearized between its two complete phases. □

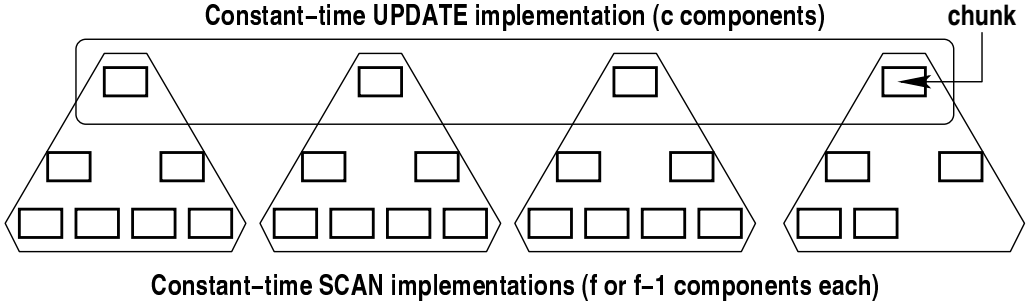

Figure 11: A 15 component hybrid snapshot object.

5.2 The Lower Bound

In this section we prove the matching lowerbound, namely that Time[UPDATE] $\in \Omega(\log(\min\{m,n\}/\text{Time[SCAN]}))$. To prove this lower bound, we use a technique of Beame [7] for proving lower bounds in concurrent-write PRAM models. The idea is to consider a set of executions that each begin with a set of concurrent UPDATES. For each process and memory cell, we derive an upper bound on how many different executions from this set it can distinguish between at a given time step. We use this to derive a lower bound on the complexity of a SCAN operation that begins after the UPDATES are complete. In all of the executions we consider, only process p_i UPDATES component i, so this lower bound also applies to the implementation of a single-writer snapshot object.

Theorem 5.2. *In any implementation of an m-component synchronous snapshot object shared by n processes that communicate through shared registers, the worst case time complexities S and U of* SCAN *and* UPDATE *must satisfy the inequality* $2^\mu \leq (2\mu(2\mu+1))^{2^{U-2}(S+1)}$, *where* $\mu = \min\{m,n\}$.

PROOF. Let $I \subseteq \{1,\ldots,\mu\}$. Consider an execution α_I where, starting at time 1, process p_i UPDATES component i to have value 1, for each $i \in I$. This execution starts from an initial configuration in which each component of the snapshot object has value 0. In this execution, all reads performed at a given step are linearized before all writes and all writes are linearized in order of process identity.

For each process p, we say that α_I and $\alpha_{I'}$ are indistinguishable to p at time t if p is in the same state at the end of step t of both executions. Let $P(p,t)$ denote the partition of all subsets of $\{1,\ldots,\mu\}$ induced by the equivalence relation $\sim$, where $I \sim I'$ if and only if α_I and $\alpha_{I'}$ are indistinguishable to p at time t. Let P_t be the least upper bound of the number of classes in the partition $P(p,t)$ taken over all processes p. Note that $P_0 = 2$.

Similarly, for each shared register j, we say that α_I and $\alpha_{I'}$ are indistinguishable to j at time t if j has the same contents at the end of step t of both executions. Let $C(j,t)$ denote the partition of all subsets of $\{1,\ldots,\mu\}$ induced by the equivalence relation $\sim$, where $I \sim I'$ if and only if α_I and $\alpha_{I'}$ are indistinguishable to j at time t. Let C_t be the least upper bound of the number of classes in the partition $C(j,t)$ taken over all registers j. Then $C_0 = 1$.

The register a process p reads from during step $t+1$ depends only on its state at the end of step t. Thus, each class of $P(p,t)$ is partitioned into at most C_t classes, depending on the contents of the register process p reads. Hence, $P_{t+1} \leq P_t C_t$.

The contents of register j at the end of step $t+1$ may depend on its contents at the end of step t or on the state of the last process that wrote to it during step $t+1$. There are μ different processes and each can write at most P_t different values (depending on the class of the partition it is in). Thus, $C_{t+1} \leq \mu P_t + C_t$.

It follows by induction that $P_t, C_t \leq (2\mu(2\mu+1))^{2^{t-2}}$ for $t \geq 2$.

Now fix a process p and let σ_I denote a solo SCAN performed by a process starting at time $U+1$ at the configuration resulting from α_I. Let $P'(p,t)$ denote the partition of all subsets of $\{1,\ldots,\mu\}$ induced by the equivalence relation $\sim$, where $I \sim I'$ if and only if $\alpha_I\sigma_I$ and $\alpha_{I'}\sigma_{I'}$ are indistinguishable to p at time $U+t$. Let P'_t be the number of classes in the partition $P'(p,t)$.

The register that process p reads from during step $U+t+1$ depends only on its state at the end of step $U+t$. Moreover, if p reads from a register that it has written to since step U, it gets no useful information. Thus, $P'_0 \leq P_U$ and $P'_{t+1} \leq P'_t C_U$, so $P'_t \leq P_U (C_U)^t$.

The SCAN completes by time $U+S$. Note that $P'_S = 2^\mu$. Otherwise, there are two executions $\alpha_I\sigma_I$ and $\alpha_{I'}\sigma_{I'}$ which are indistinguishable to p. This is impossible, since the result returned by p's SCAN must be different in these two executions.

Hence, $2^\mu \leq P_U (C_U)^S \leq (2\mu(2\mu+1))^{2^{U-2}(S+1)}$. □

Corollary 5.3.

$$\text{Time[UPDATE]} \in \Omega(\log(\min\{m,n\}/\text{Time[SCAN]}))$$

PROOF. Since $2^\mu \leq (2\mu(2\mu+1))^{2^{U-2}(S+1)} < (2\mu(2\mu+1))^{2^U S}$, it follows that $\log\mu \leq U + \log S + \log\log(2\mu(2\mu+1))$. Thus, $U \in \Omega(\log(\mu/S))$. □

6. CONCLUSION

In this paper we studied the complexity of implementing an m-component snapshot object from registers in a synchronous system with n processes.

We gave a wait-free implementation that performs UPDATE in $O(1)$ time and SCAN in $O(m)$ time. The implementation has small space overhead, requiring $2m$ registers that are the size of individual components and m registers that have only 2 bits. A simple variant uses only m registers, but the same total number of bits, and runs slightly faster. Just as in Neiger and Singh's [16] implementations, a process performs no writes during a SCAN, and only modifies the registers associated with the component during an UPDATE. Thus, our implementation can also be used to obtain a single-writer snapshot object from single-writer registers. As well, we briefly described a similar wait-free implementation that performs UPDATE in $O(1)$ time and SCAN in $O(n)$ time.

We also presented a different wait-free implementation in which SCAN takes 1 step and UPDATE takes $O(\log m)$ steps, as well as a variant of this implementation that takes 1 step to perform a SCAN and $O(\log n)$ steps to perform an UPDATE. For these implementations, we required large registers that can hold the values of all m components. Although SCANS perform no writes, UPDATES to different components will write to some of the same registers. Thus,

single-writer registers do not suffice for these implementations. In fact, using only single-writer registers, a simple adversary argument shows that SCANS take $\Omega(n)$ steps.

The implementations with constant UPDATE time and the implementations with constant SCAN time were combined to obtain two implementations: one that performs SCAN in $O(c)$ steps and UPDATE in $O(\log(m/c))$ steps, for $1 \leq c \leq m$, and another that performs SCAN in $O(c)$ steps and UPDATE in $O(\log(n/c))$ steps, for $1 \leq c \leq n$. We proved that $\text{Time}[\texttt{UPDATE}] \in \Omega(\log(\min\{m,n\}/\text{Time}[\texttt{SCAN}]))$, which matches our upper bounds. Thus, snapshots in the synchronous setting are significantly more efficient than their counterparts in the asynchronous setting.

We have recently started to investigate snapshot object implementations for semisynchronous systems [8]. We consider systems where each primitive operation may be delayed by at most Δ timesteps. The implementations described in this paper can be adapted to this semisynchronous environment by using a barrier-like mechanism, which allows processes to synchronize with each other, but prevents any process from blocking indefinitely. When a process enters the barrier, it repeatedly reads a shared multiwriter register for up to $T\Delta$ iterations, where T is the time complexity of the synchronous algorithm, or a part thereof. A process may exit the barrier if the register is modified by another process or if the maximum number of iterations have been performed. In the latter case, the process also modifies the register before exiting the barrier, releasing the remaining waiting process. This mechanism increases the upper bounds on the time complexity of the implementations by a factor of Δ^2. This differs by a factor of Δ from the corresponding synchronous lower bounds. One challenge is to reduce the gap between the upper and lower bounds on the complexity of snapshots in the semisynchronous setting.

7. REFERENCES

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, Sept. 1993.

[2] J. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, 1993.

[3] J. Aspnes and M. Herlihy. Fast, randomized consensus using shared memory. *J. Algorithms*, 11(2):441–461, Sept. 1990.

[4] J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Proc. 2nd ACM Symposium on Parallel Algorithms and Architectures*, pages 340–349, 1990.

[5] H. Attiya, N. Lynch, and N. Shavit. Are wait-free algorithms fast? *J. ACM*, 41(4):725–763, July 1994.

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

[7] P. Beame. Limits on the power of concurrent-write parallel machines. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 169–176, 1986.

[8] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Apr. 1988.

[9] P. Fatourou, F. Fich, and E. Ruppert. A tight time lower bound for space-optimal implementations of multi-writer snapshots. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 259–268, 2003.

[10] F. Fich. The complexity of computation on the parallel random access machine. In J. Reif, editor, *Synthesis of Parallel Algorithms*, pages 843–899. Morgan-Kaufmann, 1993.

[11] F. Fich, P. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation. *sicomp*, 17(3):606–627, June 1988.

[12] R. Gawlick, N. Lynch, and N. Shavit. Concurrent timestamping made simple. In *Proceedings of the Israel Symposium on the Theory of Computing and Systems*, volume 601 of *LNCS*, pages 171–183, 1992.

[13] M. Inoue, W. Chen, T. Masuzawa, and N. Tokura. Linear time snapshots using multi-writer multi-reader registers. In *Distributed Algorithms, 8th International Workshop*, volume 857 of *LNCS*, pages 130–140, 1994.

[14] A. Israeli, A. Shaham, and A. Shirazi. Linear-time snapshot implementations in unbalanced systems. *Mathematical Systems Theory*, 28(5):469–486, Sept./Oct. 1995.

[15] A. Israeli and A. Shirazi. The time complexity of updating snapshot memories. *Information Processing Letters*, 65(1):33–40, Jan. 1998.

[16] G. Neiger and R. Singh. Space-efficient atomic snapshots in synchronous systems. Technical Report GIT-CC-93-46, Georgia Institute of Technology. College of Computing, 1993.