The Complexity of Updating Multi-writer Snapshot Objects

Hagit Attiya[⋆], Faith Ellen[⋆⋆], and Panagiota Fatourou[⋆⋆⋆]

Abstract. This paper proves $\Omega(m)$ lower bounds on the step complexity of `UPDATE` operations for partitioned implementations of m-component multi-writer snapshot objects from base objects of any type. These are implementations in which each base object is only modifed by processes performing `UPDATE` operations to one specific component. In particular, we show that any space-optimal implementation of a multi-writer snapshot object from historyless objects is partitioned. This work extends a similar lower bound by Israeli and Shirazi for implementations of m-component single-writer snapshot objects from single-writer registers.

1 Introduction

An important problem in shared memory distributed systems is to obtain a consistent view of the contents of the memory while updates to the memory are happening concurrently. This problem can be formalized as the implementation of a snapshot object that can be accessed concurrently by different processes. A *snapshot object* [1] consists of a set of $m > 1$ components, each capable of storing a value. Processes can perform two different types of operations: `UPDATE` any individual component to have a specific value or atomically `SCAN` to obtain the values of all the components. A *single-writer* snapshot object is a restricted version in which there are the same number of updaters as components and only process p_i can `UPDATE` the i'th component. In a *multi-writer* snapshot object, there is no restriction on which processes may `UPDATE` a component.

It is often much easier to design fault-tolerant algorithms for asynchronous systems and prove them correct if one can think of the shared memory as a snapshot object, rather than as a collection of individual objects. This is why researchers have spent a great deal of effort on finding efficient implementations of snapshot objects from *base* objects that are provided in real systems, like registers or swap objects.

Many implementations of snapshot objects are known, most of them from registers [1,2,3,4,9,13,14], and several from stronger base objects [16,17]. However,

[⋆] The Technion, Israel, supported by the Israel Science Foundation (grant number 953/06).

[⋆⋆] University of Toronto, Canada, supported by the Natural Sciences and Engineering Research Council of Canada, a Lady Davis Fellowship, and the Scalable Synchronization Research Group of SUN Microsystems, Inc.

[⋆⋆⋆] University of Ioannina, Greece, supported by the Greek Ministry of Education's E.P.E.A.E.K. II programme for studies in informatics.

S. Chaudhuri et al. (Eds.): ICDCN 2006, LNCS 4308, pp. 319–330, 2006.
© Springer-Verlag Berlin Heidelberg 2006

only a few lower bounds on the complexity of implementing snapshot objects are known.

Jayanti, Tan, and Toueg [18] proved that any implementation of an m-component single-writer snapshot object from historyless and resettable consensus objects requires at least $m-1$ objects and SCANS take at least $m-1$ steps. For multi-writer snapshot objects implemented from historyless objects, Fatourou, Fich and Ruppert [7] improved the space lower bound to m, which is optimal, since there are implementations from m registers. More importantly, they proved that in any space-optimal implementation of a binary snapshot object from historyless objects shared by $n > m+1$ processes, the worst case step complexity of SCAN is in $\Omega(mn)$, which is asymptotically tight. This proof relies on a number of structural lemmas which show that any space-optimal snapshot implementation from historyless objects has a special form.

Israeli and Shirazi [15] proved a tight $\Omega(m)$ lower bound on the step complexity of UPDATE, for implementations of m-component single-writer snapshot objects, under the assumption that only single-writer registers are used. No lower bounds were known for the step complexity of an UPDATE operation when other base objects, in particular, multi-writer registers, can be used.

In this paper, we prove that the worst case step complexity of UPDATE in any implementation of an n-process m-component multi-writer snapshot object from $m < n$ registers or $m < n-1$ historyless objects is in $\Omega(m)$. We begin by extending Fatourou, Fich, and Ruppert's structural lemmas [6,7] for space-optimal implementations to show that any such implementation is *partitioned.* Partitioned implementations can use any number of base objects, but each base object can only be modified by processes performing UPDATES to one specific component. Then we prove that, for any partitioned implementation of a multi-writer snapshot object implemented from base objects of *any* type, the worst case complexity of UPDATE is in $\Omega(m)$. We do this in two ways.

First, we observe that, if $n \geq 2m$, then any n-process partitioned implementation of an m-component multi-writer snapshot also gives an implementation of an m-component single-writer snapshot object, shared by m updaters and m scanners, from single-writer registers. Then the result follows from Israeli and Shirazi's lower bound [15]. Their proof and, hence, this result, requires that the number of possible values for each component is infinite.

Secondly, we give a direct proof of the lower bound for partitioned implementations. This proof even applies to binary snapshot objects (i.e. when each component is only a single bit). Moreover, it does not rely on the existence of a large number of processes: it suffices that there are three processes, two which can perform UPDATES and one which can perform SCAN. This shows that it is the number of components in a snapshot object, not the number of processes, that is responsible for UPDATES taking a long time in partitioned implementations.

The best known multi-writer snapshot implementation is partitioned [1] and has $O(mn)$ step complexity for both UPDATE and SCAN. There are also a number of implementations which are not partitioned and have $o(m)$ step complexity for UPDATE. They show that the lower bounds do not hold if the restriction to partitioned implementations is removed. For example, there is a trivial implementation of a multi-writer snapshot object with $O(1)$ steps for UPDATE and SCAN. The implementation uses a single object that contains a *view* of the snapshot object: a SCAN simply reads this view; an UPDATE of component i atomically reads the view and modifies component i of the view.

Jayanti [16] uses an f-array to implement a multi-writer snapshot object with $O(\log m)$ steps for an UPDATE and one step for a SCAN. It uses $m - 1$ LL/SC objects, each containing a view, arranged in a binary tree. This implementation is not partitioned, since a process doing an UPDATE follows a path to the root from the leaf corresponding to the component, performing SC operations as it goes. The root contains the current values of all m components, so a SCAN can simply read the root.

Another implementation of a snapshot object has $O(1)$ steps for an UPDATE and an unbounded (but finite) number of steps for a SCAN. The implementation follows Kirousis, Spirakis, and Tsigas [19], and uses an $m \times \infty$ array of registers. Each row contains values written to a specific component of the snapshot object. The scanners also maintain a pointer, telling the updaters the column in which to put their value. An UPDATE to component i reads the pointer and writes its value in the corresponding place in row i of the array. A SCAN does fetch&inc on the pointer and then, for each component i, reads backwards in row i from the current value of the pointer (towards the beginning of the array) until it finds a non-empty entry. This implementation says that an $\Omega(m)$ lower bound does not exist if scanners can write to the same register or there is only one scanner, which can write to a single-writer register.

Fatourou and Kallimanis [8] recently presented two implementations of m-component multi-writer snapshot objects that support a single scanner and any number of updaters. One of these uses $m + 1$ registers and has $O(m^2)$ step complexity for both SCAN and UPDATE. Except for a single-writer register used by the scanner, this implementation is partitioned. The other implementation uses an unbounded number of registers, but has $O(m)$ step complexity for both SCAN and UPDATE. It is not partitioned. They also prove that any implementation from registers which supports a single-scanner must use at least m registers and the step complexity of SCAN is in $\Omega(m)$ if the implementation is space-optimal.

Riany, Shavit and Touitou [21] implement an m-component single-writer snapshot object, from $O(m^2)$ LL/SC objects. In their implementation, UPDATE has constant step complexity and SCAN has $O(m)$ step complexity. Their algorithm is based on a single-scanner snapshot object, which is a simplification of the algorithm by Kirousis, Spirakis, and Tsigas [19]. In their multi-scanner algorithm, scanners collaborate in collecting a view in order to reduce work and to guarantee that returned views are consistent. This algorithm is not partitioned: Although the updaters write only to a single object associated with their component, scanners write to multiple shared objects.

Jayanti [17] implements an m-component multi-writer snapshot object, with the same step complexity, from $O(mn^2)$ CAS objects. This algorithm also starts with a single-writer single-scanner snapshot object, and extends it in two steps. The algorithm is not partitioned.

2 The Model

We use a standard model of asynchronous shared-memory systems [5,20]. In this model, a set of n deterministic processes $P = \{p_1, \ldots, p_n\}$ communicate by accessing shared base objects. The system is totally asynchronous, so the order in which operations are performed by processes is assumed to be controlled by an adversarial scheduler. Algorithms must work correctly regardless of the schedule the adversary chooses.

A *configuration* is a complete description of the system at some point in time. It is comprised of the internal state of each process and the state of each shared object. A *step* of a process consists of a single operation accessing a shared object, the response to that operation, and some local computation that may cause the internal state of the process to change.

An *execution* starting from a configuration C is a sequence of steps, in which the steps performed by each process follow the algorithm for that process (starting from its state in C) and the responses to the operations performed on each object are in accordance with its specification (and the value stored in the object at configuration C). A configuration C is *reachable* if there is a finite execution that results in C starting from some initial configuration. An execution is *solo* if every step is performed by the same process.

An object is *historyless* [10] if all its non-trivial operations (i.e. operations which might change the value of the object) overwrite one another. The most familiar historyless object is a *register*, which is accessed either by a trivial *read* operation, which returns the value of the register without changing it, or by a non-trivial *write* operation with one parameter, which changes the register's value to the value of its parameter, overwriting its previous value. A *single-writer* register is restricted so that only one particular process can perform write operations; a *multi-writer* register can be written to by any process. Another type of historyless base object is a *swap* object, which supports *read* and *swap* operations. A process *covers* a historyless object in a configuration if the process will perform a non-trivial operation on that object, when it takes its next step. A set of processes $P' \subseteq P$ covers a set of objects $\mathcal{O}$ in a configuration if each process in P' covers an object in $\mathcal{O}$ and each object in $\mathcal{O}$ is covered by some process in P'.

A *(multi-writer) snapshot object* is an object with m components $1, \ldots, m$, each of which stores a value from a specified set. When this set is $\{0, 1\}$, it is called a *binary snapshot object.* A snapshot object supports two operations. The operation UPDATE(i, v) sets the value of component i to v. The SCAN operation returns a vector consisting of the values of the m components.

A *(wait-free) implementation* provides, for each process, an algorithm for performing UPDATE and an algorithm for performing SCAN. Every process executes only a finite number of steps to perform these operations, even if other processes run at arbitrary speeds or may crash. An UPDATE or SCAN operation by a process is *pending* after an execution prefix if the process has already executed the first step of this invocation of its algorithm, but has not yet finished performing the algorithm. A *serial-update execution* of a snapshot implementation is an execution in which UPDATE operations do not overlap; that is, in any configuration, there is at most one pending UPDATE operation.

We restrict attention to *linearizable implementations* [12]. This means that, in each execution, each simulated UPDATE or SCAN operation appears to take effect at some instant during the period of time it is pending. Thus, there is a linearization of the implemented operations which would produce the same responses for each operation as in the execution. Furthermore, if one simulated operation finishes before another simulated operation begins, the latter operation comes later in the linearization.

Consider any execution of a snapshot implementation that reaches a configuration C in which no process has a pending UPDATE to component i. Then there is a value v_i such that, for any SCAN starting at or after C and finishing before any subsequent UPDATE to component i begins, the value of component i returned by this SCAN is v_i. This is because all such SCAN operations must be linearized after all the UPDATES to component i that complete before C and before all the UPDATES to component i that start after C. We call v_i the *value of component i at configuration C.* If there are no processes with pending UPDATES to any component in configuration C, then the *value of the snapshot object at configuration C* is $(v_1, \ldots, v_m)$, where v_i is the value of component i at C.

3 Partitioned and Space-Optimal Snapshot Implementations

In this section, partitioned implementations of snapshot objects are defined and we prove that space-optimal implementations of multi-writer snapshot objects from historyless objects are partitioned.

Definition 1. *A snapshot implementation is* partitioned *if all of the objects can be partitioned into m disjoint sets $\mathcal{O}_1, \ldots, \mathcal{O}_m$ such that, in any serial-update execution from an initial configuration C_0, only processes performing* UPDATES *to component i can change the value of objects in $\mathcal{O}_i$.*

In particular, in any serial-update execution of a partitioned implementation, SCAN operations cannot change the value of any object. Any implementation of a single-writer snapshot object from single-writer registers in which SCAN operations do not write is partitioned.

Consider any n-process implementation of an m-component snapshot object from a set $\mathcal{H}$ of $m < n$ registers or $m < n - 1$ historyless objects. In the full version of the paper, we prove that this implementation is partitioned. The proof relies on structural lemmas from Fatourou, Fich, and Ruppert [6,7], which hold for such implementations.

Lemma 1. *Let C be a configuration with no pending* UPDATES. *Suppose that x is not the value of component i at C. Then the solo execution by any process, starting from C, in which it performs* UPDATE(i, x) *(after completing its pending* SCAN, *if necessary), must change the value of at least one shared object.*

Lemma 2. SCAN *operations do not perform non-trivial operations.*

For any configuration, C, without pending UPDATES, let $H(C, i, j, x)$ denote the first object that process p_j covers when performing a solo execution of UPDATE(i, x) starting from C (after completing its pending SCAN, if necessary). By Lemma 1, if x is not the value of component i at C, then this object exists.

Let C_0 denote the initial configuration in which the initial value of every component is 0. Let $H_i = H(C_0, i, i, 1)$ be the first object that process p_i covers when performing a solo execution of UPDATE$(i,1)$ starting from configuration C_0.

Lemma 3. *$H_i \neq H_j$ for distinct $i, j \in \{1, \ldots, m\}$.*

Consider any serial-update execution α of the implementation starting from C_0. Let $C_0, C_1, \ldots$ denote the sequence of configurations without pending UPDATES that occur in α. We state two additional structural properties about this restricted execution. The proofs appear in the full version of the paper.

Lemma 4. *If x is not the value of component i at configuration C_k, then, for every process p_j, $H(C_k, i, j, x) = H_i$.*

Lemma 5. *In any serial-update execution starting from C_0, the only shared object to which non-trivial operations are performed during an* UPDATE *to component i is H_i.*

Together with Lemmas 2 and 3, Lemma 5 shows that space-optimal implementations of multi-writer snapshot objects from historyless objects are partitioned.

Corollary 1. *Any implementation of an m-component multi-writer snapshot object shared by $n \geq 3$ processes from $m < n$ registers or from $m < n - 1$ historyless objects is partitioned.*

4 A Simple Reduction

Israeli and Shirazi [15] proved the following lower bound for single-writer snapshot objects implemented from single-writer registers. Then a simple reduction extends this result to a lower bound for partitioned implementations of multi-writer snapshot objects from any base objects. With Corollary 1, this gives a lower bound on the step complexity of UPDATE operations for space-optimal implementations of multi-writer snapshot objects.

Theorem 1. *In any implementation of a single-writer snapshot object, over an infinite domain, shared by m updaters and m scanners, from single-writer registers, there is a serial-update execution in which some* UPDATE *operation takes $\Omega(m)$ steps.*

This lower bound can be extended to partitioned implementations of multi-writer snapshot objects from any base objects by observing that a partitioned implementation of a multi-writer snapshot object can be used to implement a single-writer snapshot object using only single-writer registers. Specifically, the objects that can be changed by processes performing UPDATES to component i are represented by different fields of the single-writer register of process p_i.

Theorem 2. *In any partitioned implementation of an m-component multi-writer snapshot object, over an infinite domain, shared by m updaters and m scanners, there is a serial-update execution in which some* UPDATE *operation takes $\Omega(m)$ steps.*

Note that the same reduction also works for implementations in which each process also has a single-writer register to which it can write at any time.

Combining Corollary 1 and Theorem 2 yields our first lower bound for space-optimal implementations.

Theorem 3. *Any implementation of an m-component multi-writer snapshot object, over an infinite domain, shared by $n \geq 2m$ processes, from m historyless objects, requires $\Omega(m)$ steps for an* UPDATE*, in the worst case.*

This theorem is true even if the implementation only works for serial-update executions. For this special case, there is a space-optimal implementation with $O(m)$ step complexity for both SCAN and UPDATE. The algorithm uses m registers, each containing a view plus a sequence number. A SCAN simply collects all m registers, picks one with the maximal sequence number, and returns its view. To perform UPDATE(i, x), a process collects all registers, picks one with the maximal sequence number, changes component i of its view to x, and writes this new view, together with a bigger sequence number, to the i'th register.

5 Partitioned Binary Snapshot Implementations

In this section, we give a direct proof of an $\Omega(m)$ lower bound on the number of steps to perform UPDATE in any partitioned implementation of an m-component snapshot object (from any base objects). Unlike the lower bounds in the preceding section, it does not require the set of possible values of the components to be infinite. In fact, this proof even applies to binary snapshot objects. Furthermore, the lower bounds in the preceding section required $2m$ processes: m scanners and m updaters. Here, the lower bound for partitioned implementations applies even when there are only two updaters and one scanner. We also get a lower bound for space-optimal implementations that applies when the *total* number of processes is greater than m (for implementations from registers) or $m + 1$ (for implementations from historyless objects).

The proof of our lower bound for partitioned implementations is similar in structure to the proof of Theorem 1 by Israeli and Shirazi [15]. Specifically, we prove that if all UPDATES take at most $m/6$ steps, then it is possible to construct an infinite serial-update execution consisting of a single SCAN operation with one complete UPDATE operation immediately before each step of the SCAN. This will contradict the assumption that the implementation is wait-free. To do this, we build successively longer executions of this form.

A *flippable execution of length* k is a finite execution $U_0 s_1 U_1 \cdots s_k U_k$ performed by two updaters, p_0 and p_1, and one scanner, q, starting from an initial configuration C_0, such that:

- U_j is a solo execution of a complete UPDATE operation performed by p_h, where $h \equiv j \bmod 2$, in which it complements the value of one component, for $j = 0, \ldots, k$,
- consecutive UPDATE operations are to different components,
- s_j is a single step of a SCAN operation performed by q, for $j = 1, \ldots, k$, and
- for any $f > 1$ and for any sequence $j_1 < \cdots < j_f$, where $1 \leq j_\ell + 1 < j_{\ell+1} \leq k$ for $1 \leq \ell < f$, the execution $U_0 s_1 U_1 \cdots s_{j_\ell - 1} U_{j_\ell - 1} s_{j_\ell} U_{j_\ell} s_{j_\ell + 1} \cdots s_k U_k$ starting from C_0 is indistinguishable (to all processes) from the flipped execution $U_0 s_1 U_1 \cdots s_{j_\ell - 1} U_{j_\ell} U_{j_\ell - 1} s_{\ell_j} s_{j_\ell + 1} \cdots s_k U_k$ starting from C_0 in which UPDATE U_{j_ℓ} is performed before $U_{j_\ell - 1} s_{j_\ell}$ instead of after $U_{j_\ell - 1} s_{j_\ell}$, for $\ell = 1, \ldots, f$.

When we say that an instance of an UPDATE operation complements the value of component i, we mean that it is an instance of UPDATE$(i, \overline{v_i})$, where $v_i \in \{0, 1\}$ is the value of component i of the snapshot object just before the operation begins.

Next, we show that a flippable execution cannot contain a completed SCAN operation, by showing that there is no place a SCAN operation can be linearized. This will allow us to extend the flippable execution to a longer one.

Lemma 6. *No* SCAN *operation in a flippable execution has terminated.*

Proof. Let $E = U_0 s_1 U_1 \cdots s_k U_k$ be a flippable execution starting from configuration C_0. To obtain a contradiction, suppose that process q has completed a SCAN by the end of E. Let $v = (v_1, \ldots, v_m)$ be the result of this SCAN. There might be many configurations during this execution at which there are no pending UPDATES and the value of the snapshot object is v. (If the domain of the snapshot object were infinite, this difficulty could be avoided by requiring every UPDATE to use a different value.) Let $j_1 < \cdots < j_f$ be a list of all indices $j_\ell \in \{1, \ldots, k\}$ such that v is the value of the snapshot object between $U_{j_\ell - 1}$ and U_{j_ℓ}, for $\ell = 1, \ldots, f$. Note that, since U_{j_ℓ} complements the value of some component, v is not the value of the snapshot object between U_{j_ℓ} and $U_{j_\ell + 1}$. Hence $j_\ell + 1 < j_{\ell+1}$.

Consider the flipped execution $F = U_0 s_1 U_1 \cdots s_{j_\ell - 1} U_{j_\ell} U_{j_\ell - 1} s_{j_\ell} s_{j_\ell + 1} \cdots s_k U_k$ starting from C_0 in which UPDATE U_{j_ℓ} is performed before $U_{j_\ell - 1} s_{j_\ell}$ instead of after $U_{j_\ell - 1} s_{j_\ell}$, for $\ell = 1, \ldots, f$. The executions E and F are indistinguishable to process q, so q returns the same result for its SCAN in both executions. Thus, in F, the SCAN by q must be linearized at some point where the value of the snapshot object is v.

Since the UPDATES do not overlap, they are linearized in the order $U_0, U_1, \ldots,$ U_k in E and in the same order in F, except that the order of $U_{j_\ell - 1}$ and U_{ℓ_j} are flipped, for $\ell = 1, \ldots, f$. Consecutive `UPDATES` are to different components, so the value of the snapshot object is same after both have been performed, no matter which of the two is performed first. Hence, at all points in the linearization of F, except between U_{j_ℓ} and $U_{j_\ell - 1}$, for $\ell = 1, \ldots, f$, the value of the snapshot object is the same as its value at the corresponding point in the linearization of E. Recall that, in E, the value of the snapshot object is not v between U_{j-1} and U_j, for $j \neq j_1, \ldots, j_f$.

Now consider the situation between $U_{j_\ell - 1}$ and U_{j_ℓ} for any $\ell \in \{1, \ldots, f\}$. In E, the snapshot object has value v and U_{j_ℓ} is an instance of $\texttt{UPDATE}(i, \overline{v_i})$, where v_i is the value of component i of v. This implies that, between U_{j_ℓ} and $U_{j_\ell - 1}$ in F, the value of component i of the snapshot object is $\overline{v_i}$ and, hence, the value of the snapshot object is not v.

Since the `SCAN` by q begins after U_0 in F, it must be linearized after U_0. If $j_f < k$, then the `SCAN` by q finishes before U_k in both E and F and, hence, must be linearized before U_k. If $j_f = k$, then the value of the snapshot object is v between U_{k-1} and U_k in E. This implies that the value of the snapshot object is not v either after U_k in E or after U_{k-1} in F. In this case, the `SCAN` by q must be linearized before U_{k-1} in F.

The value of the snapshot object is not v between U_0 and the last `UPDATE` performed in F. This contradicts the fact that the `SCAN` by q must be linearized at some point where the value of the snapshot object is v. □

Consider any partitioned implementation of an m-component binary snapshot object, shared by two updaters, p_0 and p_1, and a scanner q, in which serial `UPDATES` take at most $m/6$ steps. Suppose that `UPDATES` to component i only change the value of objects in $\mathcal{O}_i$. We show how to construct a flippable execution so that it can be repeatedly extended. The key idea is to choose the successive components to update so that the objects each `UPDATE` operation might change are not accessed during the previous or the next `UPDATE` operation nor during the step of the `SCAN` that precedes it. A counting argument shows that this is possible in each extension of the construction.

It is helpful to use a matrix to keep track of which objects a process accesses when it performs an `UPDATE`. For each $h \in \{0, 1\}$ and each configuration C without pending `UPDATES` in an update-serial execution, let B_C^h denote the $m \times m$ Boolean matrix where $B_C^h[i, j] = 1$ if and only if the solo execution of $\texttt{UPDATE}(i, \overline{v_i})$ by process p_h starting from configuration C accesses an object in $\mathcal{O}_j$, where v_i is the value of component i of the snapshot object in configuration C. In particular, $B_C^h[i, i] = 1$, since a process performing $\texttt{UPDATE}(i, \overline{v_i})$ must access an object in $\mathcal{O}_i$. (Otherwise, it changes the value of no objects and, hence, cannot affect the outcome of a `SCAN` that follows immediately afterwords.) We say that a column j of B_C^h is *light* if more than half of its entries are 0.

Lemma 7. *If process p_h takes at most $m/6$ steps to perform a solo* `UPDATE` *starting from configuration C, then there are at least $2m/3$ light columns in B_C^h.*

Proof. Process p_h performs at most $m/6$ steps in its solo execution of $\texttt{UPDATE}(i, \overline{b})$ starting from configuration C, where b is the value of component i of the snapshot object in configuration C. Hence, at most $m/6$ entries in row i of B^h_C are 1 and at most $m^2/6$ entries in B^h_C are 1. Let ℓ denote the number of light columns in B^h_C. Then, in each of the other $m - \ell$ columns, at least $m/2$ of the entries are 1. Thus at least $(m - \ell)m/2$ entries in B^h_C are 1. This implies that $m^2/6 \geq (m - \ell)m/2$ or, equivalently, $\ell \geq 2m/3$. □

We now prove the main technical lemma, which shows the existence of a flippable execution, but with one additional property that makes the construction proceed more easily.

Lemma 8. *If every updater takes at most $m/6$ steps to perform a solo* UPDATE*, then, for all $k \geq 0$, there is a flippable execution, $U_0 s_1 U_1 \cdots s_k U_k$, such that U_k is an* UPDATE *to some component i_k by a process $p_{\overline{h}}$ starting from configuration C_k, where $h \equiv (k+1) \bmod 2$ and column i_k of $B^h_{C_k}$ is light.*

Proof. By induction on k.

First consider the base case, $k = 0$. By Lemma 7, $B^1_{C_0}$ has at least $2m/3$ light columns. Let U_0 be a solo execution of $\texttt{UPDATE}(i_0, \overline{v_{i_0}})$ by process p_0 starting from configuration C_0, where i_0 is the index of a light column in $B^1_{C_0}$ and v_{i_0} is the value of component i_0 in C_0. Then the claim holds for $k = 0$.

For the induction step, suppose $E = U_0 s_1 U_1 \cdots s_k U_k$ is a flippable execution such that column i_k of $B^h_{C_k}$ is light, where U_k is a solo execution of $\texttt{UPDATE}(i_k, \overline{v_{i_k}})$ by process $p_{\overline{h}}$ starting from configuration C_k and v_{i_k} is the value of component i_k in C_k. By Lemma 6, process q has not completed its SCAN at the end of E. Let s_{k+1} denote the next step by process q and let C_{k+1} denote the configuration at the end of $E s_{k+1}$.

The component i_{k+1} for U_{k+1} to update will be chosen so that the resulting execution is flippable. But we must also choose it with some concern for the future, so that there is enough flexibility to choose the component to update in the following step of the induction.

To prepare for the future, we restrict i_{k+1} to be the index of a light column in $B^{\overline{h}}_{C_{k+1}}$. Let $I \subseteq \{1, \ldots, m\}$ be this set of indices. By Lemma 7, $|I| \geq 2m/3$.

Next, we ensure that it is possible to interchange $U_k s_{k+1}$ and U_{k+1}. To do this, we first restrict our choices for i_{k+1} so that process p_h will not access any object during U_{k+1} that might have changed value during U_k. Let $I' = \{i \in I \mid B^h_{C_k}[i, i_k] = 0\}$ consist of all indices $i \in I$ such that process p_h does not access any object in $\mathcal{O}_{i_k}$ when performing a solo execution of $\texttt{UPDATE}(i, \overline{v_i})$ starting from configuration C_k, where v_i is the value of component i in C_k. This ensures that U_{k+1} behaves exactly the same when performed starting from C_k as from C_{k+1}. By the induction hypothesis, column i_k of $B^h_{C_k}$ is light, so there are less than $m/2$ indices $i \in I$ such that $B_{C_k}[i, i_k] = 1$. Hence, $|I'| > |I| - m/2 \geq m/6$.

We further restrict our choices for i_{k+1} so that the other updater, $p_{\overline{h}}$, does not access any object during U_k whose value might be changed during U_{k+1}. Let $I'' = \{i \in I' \mid B^{\overline{h}}_{C_k}[i_k, i] = 0\}$ consist of all indices $i \in I'$ such that process $p_{\overline{h}}$ does not access any object in $\mathcal{O}_i$ during U_k, the solo execution of $\mathtt{UPDATE}(i_k, \overline{v_{i_k}})$ starting from configuration C_k. Then U_k behaves exactly the same when performed starting from C_{k+1} as from C_k. Row i_k of $B^{\overline{h}}_{C_k}$ contains at most $m/6$ entries with value 1. Since $B^{\overline{h}}_{C_k}[i_k, i_k] = 1 = B^{h}_{C_k}[i_k, i_k]$, at least one i with $B^{\overline{h}}_{C_k}[i_k, i] = 0$ is not in I'. Thus $|I''| \geq |I'| - m/6 + 1 > 1$. Let $i_{k+1} \in I''$ be such that q does not access an object in $\mathcal{O}_{i_{k+1}}$ during its single step s_{k+1}. Note that $i_{k+1} \neq i_k$, since $B^{\overline{h}}_{C_k}[i_k, i_k] = 1$. Let U_{k+1} be a solo execution of $\mathtt{UPDATE}(i_{k+1}, v_{i_{k+1}})$ starting from C_{k+1}, where $v_{i_{k+1}}$ is the value of component i_{k+1} of the snapshot object in configuration C_{k+1}.

It remains to prove that the execution $E' = U_0 s_1 U_1 \cdots s_{k+1} U_{k+1}$ starting from C_0 is flippable. Consider any sequence $j_1 < \cdots < j_f$ where $1 \leq j_\ell + 1 < j_{\ell+1} \leq k$ for $1 \leq \ell < f$. By the induction hypothesis, the execution $E = U_0 s_1 U_1 \cdots s_{j_\ell - 1} U_{j_\ell - 1} s_{j_\ell} U_{j_\ell} s_{j_\ell + 1} \cdots s_k U_k$ starting from C_0 is indistinguishable (to all processes) from the flipped execution $F = U_0 s_1 U_1 \cdots s_{j_\ell - 1} U_{j_\ell} s_{j_\ell} U_{j_\ell - 1} s_{j_\ell + 1} \cdots s_k U_k$ starting from C_0. In particular, the configurations at the end of these two executions are the same. Hence, executions $E' = E s_{k+1} U_{k+1}$ and $F s_{k+1} U_{k+1}$ starting from C_0 are indistinguishable to all processes.

If $j_f + 1 < k + 1$, we must also consider the sequence $j_1 < \cdots < j_f < j_{f+1}$, where $j_{f+1} = k + 1$. In this case, the flipped execution F' differs from $F s_{k+1} U_{k+1}$ in that U_{k+1} precedes $U_k s_{k+1}$ instead of following it.

F' is indistinguishable from $F s_{k+1} U_{k+1}$ to p_h because $i_{k+1} \in I'$ implies that p_h does not access any objects in $\mathcal{O}_{i_k}$ during U_{k+1} starting from configuration C_k. Since the implementation is partitioned, these are the only objects whose values can change during $U_k s_{k+1}$. F' is indistinguishable from $F s_{k+1} U_{k+1}$ to $p_{\overline{h}}$ because $i_{k+1} \in I''$ implies that $p_{\overline{h}}$ does not access any objects in $\mathcal{O}_{i_{k+1}}$ during U_k and these are the only objects whose values can change during U_{k+1}. F' is indistinguishable from $F s_{k+1} U_{k+1}$ to q because q does not access an object in $\mathcal{O}_{i_{k+1}}$ during its single step s_{k+1}. F' is indistinguishable from $F s_{k+1} U_{k+1}$ to all other processes, since they take no steps during $U_k s_{k+1} U_{k+1}$. By transitivity, F' is indistinguishable from E' to all processes and the claim holds for $k + 1$. □

Our lower bounds follow from Lemmas 6 and 8 and Corollary 1. These lower bounds apply even if the implementation only works for serial-update executions.

Theorem 4. *In any partitioned implementation of an m-component multi-writer binary snapshot object, shared by at least two updaters and one scanner, there is a serial-update execution in which some* UPDATE *operation takes more than $m/6$ steps.*

Theorem 5. *Any implementation of an m-component multi-writer binary snapshot object, shared by $n \geq 3$ processes, from $m < n$ registers or $m < n - 1$ historyless objects, requires $\Omega(m)$ steps for an* UPDATE*, in the worst case.*

References

1. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit, *Atomic snapshots of shared memory*, JACM, vol. 40, no. 4, September 1993, pages 873–890.
2. James H. Anderson, *Multi-writer composite registers*, Distributed Computing, vol. 7, no. 4, May 1994, pages 175–195.
3. Hagit Attiya and Arie Fouren, *Adaptive and efficient algorithms for lattice agreement and renaming*, SICOMP, vol. 31, no. 2, October 2001, pages 642–664.
4. Hagit Attiya and Ophir Rachman, *Atomic snapshots in $O(n \log n)$ operations*, SICOMP, vol. 27, no. 2, April 1998, pages 319–340.
5. Hagit Attiya and Jennifer Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2nd edition*, Wiley-Interscience, 2004.
6. Panagiota Fatourou, Faith Ellen Fich, and Eric Ruppert. *A Tight Time Lower Bound for Space-Optimal Implementations of Multi-Writer Snapshots*, STOC 2003, pages 259–268.
7. Panagiota Fatourou, Faith Ellen Fich, and Eric Ruppert. *A Tight Time Lower Bound for Space-Optimal Implementations of Multi-Writer Snapshots*, manuscript, 2006.
8. Panagiota Fatourou and Nikolaos Kallimanis, *Single-Scanner Multi-Writer Snapshot Implementations are Fast*, PODC 2006, to appear.
9. Faith Ellen Fich, *How Hard is it to Take a Snapshot?*, SOFSEM 2005, LNCS, vol. 3381, pages 27–35.
10. Faith Fich, Maurice Herlihy, and Nir Shavit, *On the Space Complexity of Randomized Synchronization*, JACM, vol. 45, no. 5, September 1998, pages 843-862.
11. Faith Ellen Fich and Eric Ruppert, *Hundreds of Impossibility Results for Distributed Computing*, Distributed Computing, vol. 16, no. 2–3, 2003, pages 121–163.
12. Maurice P. Herlihy and Jeannette M. Wing, *Linearizability: A correctness condition for concurrent objects*, TOPLAS, vol. 12, no. 3, July 1990, pages 463–492.
13. Michiko Inoue, Wei Chen, Toshimitsu Masuzawa, and Nobuki Tokura, *Linear time snapshots using multi-writer multi-reader registers*, WDAG 1994, LNCS, vol. 857, pages 130–140.
14. A. Israeli, A. Shaham, and A. Shirazi, *Linear-time snapshot implementations in unbalanced systems*, Mathematical Systems Theory, vol. 28, no. 5, September/October 1995, pages 469–486.
15. Amos Israeli and Assaf Shirazi, *The time complexity of updating snapshot memories*, Information Processing Letters, vol. 65, no. 1, 1998, pages 33–40.
16. Prasad Jayanti, *f-Arrays: Implementation and Applications*, PODC 2002, pages 270–279.
17. Prasad Jayanti, *An Optimal Multi-Writer Snapshot Algorithm*, STOC 2005, pages 723–732.
18. Prasad Jayanti, King Tan, and Sam Toueg, *Time and space lower bounds for nonblocking implementations*, SICOMP, vol. 30, no. 2, June 2000, pages 438–456.
19. L.M. Kirousis, P. Spirakis, and P. Tsigas, *Reading Many Variables in One Atomic Operation: Solutions with Linear or Sublinear Complexity*, IEEE Trans. on Parallel and Distributed Systems, vol. 5, no. 7, July 1994, pages 688–696.
20. Nancy Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996.
21. Yaron Riany, Nir Shavit, and Dan Touitou, *Towards a Practical Snapshot Algorithm*, Theoretical Computer Science, vol. 269, 2001, pages 163–201.