Space-Optimal Multi-Writer Snapshot Objects Are Slow

Panagiota Fatourou
University of Ioannina
faturu@cs.uoi.gr

Faith Fich
University of Toronto
fich@cs.utoronto.ca

Eric Ruppert
York University
ruppert@cs.yorku.ca

Abstract

We consider the problem of wait-free implementation of a multi-writer snapshot object with $m \geq 2$ components shared by $n > m$ processes. It is known that this can be done using m multi-writer registers. We give a matching lower bound, slightly improving the previous space lower bound. The main focus of the paper, however, is on time complexity. The best known upper bound on the number of steps a process has to take to perform one operation of the snapshot is $O(n)$. When m is much smaller than n, an implementation whose time complexity is a function of m rather than n would be better. We show that this cannot be achieved for any space-optimal implementation: We prove that $\Omega(n)$ steps are required to perform a SCAN operation in the worst case, even if $m = 2$. This significantly improves previous $\Omega(\min(m, n))$ lower bounds. Our proof also yields insight into the structure of any space-optimal implementation, showing that processes simulating the snapshot operations must access the registers in a very constrained way.

1 Introduction

A snapshot object is a shared data structure that consists of a collection of components, each capable of storing a value. The object supports two kinds of operations: a process can UPDATE a component and atomically SCAN the entire collection to obtain the values of all of the components. It was introduced independently by Afek, Attiya, Dolev, Gafni, Merritt, and Shavit [1], Anderson [3], and Aspnes and Herlihy [9]. Two types of snapshot objects have commonly been considered. In a multi-writer snapshot object, any process can UPDATE any component, while in a single-writer snapshot, each process owns one component of the snapshot object, and only the owner of a component can UPDATE it.

A snapshot object allows processes to obtain a consistent view of an entire block of shared memory, even if concurrent UPDATES are being performed by several processes. Hence, 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 registers. For example, snapshot objects have been used to solve randomized consensus [7, 8] and approximate agreement [11] and to implement bounded concurrent timestamps [16] and other types of objects [9, 17]. Because snapshot objects are much easier for programmers to use, researchers have spent a great deal of effort on finding efficient implementations of snapshots from registers, which, unlike snapshot objects, are provided in real systems. This paper studies the complexity of wait-free implementations of multi-writer snapshot objects from multi-writer registers.

Anderson [4] showed how a multi-writer snapshot object with m components shared by n processes can be implemented from one single-writer snapshot object with n (larger) components. Combined with efficient single-writer snapshot implementations [12, 20, 21], this shows that a multi-writer snapshot can be implemented either from n bounded single-writer registers so that the time complexity of UPDATE is $O(n \log n)$ and the time complexity of SCAN is $O(n)$ or from an unbounded number of multi-writer registers with $O(n)$ time complexity for both UPDATE and SCAN.

A direct implementation of a multi-writer snapshot object with m components shared by n processes using m multi-writer registers and n single-writer registers was given by Afek, Attiya, Dolev, Gafni, Merritt, and Shavit [1]. It performs $O(n^2 + mn)$ steps for UPDATE and SCAN operations. Handshaking bits are used to keep the register size bounded. If sequence numbers are used instead, an implementation using m unbounded multi-writer registers and $O(mn)$ steps per operation can be obtained.

To implement a single-writer snapshot object with n components shared by n processes, n single-writer registers are necessary and sufficient: Each process must have some way to communicate an UPDATE of its component to other processes. Conversely, multiple single-writer registers writable by the same process can be combined into one (larger) single-writer register. Israeli and Shirazi [22] proved that the time complexity of an UPDATE operation is $\Omega(n)$ for any implementation of an n-process, n-component single-writer snapshot object from single-writer registers. Jayanti, Tan and Toueg [23] proved that any randomized nonblocking implementation (and hence any wait-free implementation) of a single-writer snapshot object for n processes requires at least $n-1$ (multi-writer) registers, even if the registers can have unbounded size. Furthermore, they showed that if the implementation is deterministic, then SCAN operations take at least $n-1$ steps in the worst case. An n-process multi-writer snapshot object with m components trivially implements a single-writer snapshot object for $\min(m, n)$ processes. This implies a lower bound of $\min(m, n) - 1$ on the space and time complexity of any implementation of the multi-writer object from unbounded multi-writer registers.

The weak test-and-set object, introduced by Anderson and Yang [6], provides a weak form of mutual exclusion: if two or more processes try to enter the critical section simultaneously, it might be that no process succeeds. Afek, Boxer and Touitou [2] proved that, for any constants k and d, there is a sufficiently large n such that it is impossible to (deterministically) implement a weak test-and-set object shared by n or more processes using d or fewer multi-writer registers (in addition to single-writer registers), where each operation by a process running by itself finishes within k steps. Using a simple reduction, they prove this implies that any adaptive, long-lived implementation of a collect object (a single-writer snapshot object in which `SCAN` is not required to be atomic) must use a nonconstant number of multi-writer registers (in addition to single-writer registers).

One small contribution of our paper is that, for $m < n$, we improve the lower bound on the number of multi-writer registers required to implement an m-component multi-writer snapshot object to m, matching the known upper bound. Our main result is a new lower bound on time complexity for space optimal implementations. Specifically, we prove that, for any implementation of a multi-writer snapshot object with $m \geq 2$ components shared by $n > m$ processes from m multi-writer registers, the number of steps taken by a process to perform a `SCAN` operation is at least $n-3$ in the worst case. In the common case where the number of components, m, is small compared to the number of processes, n, this lower bound is a significant improvement to the known lower bound of $\min(m, n) - 1$ [23].

Our proof uses covering arguments, which were introduced by Burns and Lynch [14] and have subsequently been used to prove a variety of lower bounds [2, 5, 10, 15, 23, 25], mostly for space complexity. A number of key ideas are used in our proof. First, we identify a restricted class of snapshot algorithms and prove that lower bounds for this class are sufficient to give more general lower bounds. Then we construct an execution containing a troublesome `SCAN` operation. By carefully interleaving `UPDATE` operations of other processes between the steps of our troublesome `SCAN`, we are able to limit both the amount of information the process executing this `SCAN` can access and the length of time the information remains relevant. Certain `SCAN` operations are performed atomically to constrain the linearization of these `UPDATES`.

Most lower bounds that use a covering argument proceed by constructing a configuration in which all the shared objects are covered. Then a single process P runs until it enters the critical section (for mutual exclusion) or decides a value (for consensus). Next, the processes covering the objects accessed by P access the objects themselves to remove all traces of P's steps. Finally, another process runs, resulting in an invalid configuration (where the process also enters the critical section or decides a different value). In contrast, we start from a class of indistinguishable configurations in which each register can be covered by multiple processes. We use these processes to destroy the information in a register just before the troublesome `SCAN` reads it. This allows us to construct, step by step, a long execution during which the `SCAN` does not have enough information to return a valid snapshot.

Afek, Boxer, and Touitou [2] first prove their lower bound under two assumptions: the implementation uses no single-writer registers and all processes write to exactly the same sequence of multi-writer registers when trying to set the weak test-and-set object. Then they show how to modify their proofs to obtain the more general result. For example, to remove the second assumption, they employ a Ramsey-theoretic technique that throws away a large fraction of their processes. Like them, we prove our main result using a number of assumptions about the implementation. However, we first prove that every correct implementation must obey these constraints. This means our assumptions are removed without decreasing the lower bound obtained.

After a brief description of the model in Section 2, we prove a number of results in Section 3 that constrain the behaviour of a space-optimal implementation of a multi-writer snapshot object. Among these, we show that when two processes `UPDATE` different components, they must perform their first `write` operations to different registers. This implies our space lower bound. Then, in Section 4, we use these results to construct an execution in which a particular `SCAN` operation performs at least $n - 3$ `read` operations. In the final section, extensions to our work are discussed.

2 Model

This paper concerns the asynchronous shared memory model in which a set of n deterministic processes $\mathcal{P} = \{p_1, \ldots, p_n\}$ communicate by accessing shared objects. Each object has a type, which specifies the operations that can be performed on it and the results of these operations. Operations are performed atomically. 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. Formal definitions of this model can be found in [13, 24].

A *(multi-writer) register* is an object that stores a *value* from a specified set. The register is *unbounded* if this set is infinite. A register supports two atomic operations, `read` and `write`. A `read` operation returns the value of the register and leaves the value unchanged, while a `write`(v) operation changes the value stored in the register to v and returns `ack`. A *single-writer* register is restricted so that only one particular process can perform write operations.

A *(multi-writer) snapshot* object is an object with m components $A_1, \ldots, A_m$, each of which stores a *value* from a specified set. The snapshot object supports two atomic operations `SCAN` and `UPDATE`. The operation `UPDATE`(j, v) sets the value of component A_j to v. The `SCAN` operation returns a vector consisting of the values of the m components. A *single-writer snapshot* object is a snapshot object with n components, one for each process, with the restriction that only process p_i can update component A_i.

A *configuration* describes a system at some point in time. It is comprised of a state for each process and a value for each shared object. Two configurations C_1 and C_2 are *similar* to a process p, denoted $C_1 \sim_p C_2$, if process p has the same state in both C_1 and C_2. The configurations C_1 and C_2 are *indistinguishable* to process p, denoted $C_1 \approx_p C_2$, if C_1 and C_2 are similar to p and each shared object has the same value in both C_1 and C_2.

A *step* of a process consists of a single operation accessing a shared object, the response to that operation, and zero or more local operations that are performed after the access (and which may cause the state of the process to change).

We assume that each step is performed atomically; however, the result is the same if the local operations are performed (in the same order) at any time before the process next accesses a shared object. A step by process p is *applicable* to configuration C, if, starting in configuration C, process p can perform the step (according to its algorithm) and will get the indicated response (according to the specification of the shared object it accesses).

An *execution* is a sequence of steps, each indicating the process performing the step and the shared object which it accesses. An execution is *solo* if every step is performed by the same process. An execution α is *legal* starting from configuration C, if each step obeys the algorithm and the specifications of the objects. The configuration obtained from C by performing α is denoted $C\alpha$. More formally, α is a legal execution if and only if for every prefix α', the next step of α must be applicable to $C\alpha'$. A configuration C is *reachable* if there is a finite legal execution starting from an initial configuration and ending in C. The following facts are simple consequences of these definitions.

Observation 2.1 *If α is a finite legal execution starting from configuration C and a set of processes $\mathcal{P}$ take no steps in α, then $C \sim_p C\alpha$ for all $p \in \mathcal{P}$.*

Observation 2.2 *If α is a legal execution starting from configuration C_1 in which only processes in $\mathcal{P}$ take steps and C_2 is a configuration such that $C_1 \approx_p C_2$ for all $p \in \mathcal{P}$, then α is a legal execution starting from C_2 and $C_1\alpha \approx_p C_2\alpha$ for all $p \in \mathcal{P}$.*

A process *covers* a register R in a configuration if it will `write` to R when next allocated a step by the scheduler. For a set of registers $\mathcal{R}$, the set of processes $\mathcal{P}_\mathcal{R}$ *covers* $\mathcal{R}$ in configuration C if each process of $\mathcal{P}_\mathcal{R}$ covers a register of $\mathcal{R}$ in C and each register of $\mathcal{R}$ is covered by some process of $\mathcal{P}_\mathcal{R}$. A *block write* is a legal execution from C in which each process in $\mathcal{P}_\mathcal{R}$ takes exactly one step.

In this paper, we consider the problem of implementing a (multi-writer) snapshot object with m components using a system in which the only available objects are (multi-writer) registers. An *implementation* provides, for each process, an algorithm for performing `UPDATE` and an algorithm for performing `SCAN`. Before an execution of the latter algorithm terminates, it must produce a response, namely a vector of m values. A simulated operation is *pending* at some configuration of an execution if the process performing it has already executed the first step of this invocation of its algorithm, but has not yet finished performing the algorithm.

We consider only *linearizable, wait-free* implementations [18, 19]. Wait-freedom requires that every process has to execute only a finite number of steps to perform any simulated operation, even if other processes may crash or run at arbitrary speeds. Linearizability requires that, in any execution, every simulated operation appears to take effect at some instant, called the *linearization point* of the operation, during the period of time it is pending. Thus, there is a linear order of the simulated `UPDATE` and `SCAN` operations which would produce the same responses for each operation as in the execution. Furthermore, if the algorithm for one simulated operation starts executing after the algorithm for another simulated operation has finished, the latter operation comes later in the linear order.

The *time complexity* of an implementation is the maximum number of steps taken by a process to perform an `UPDATE` or `SCAN` operation.

3 Register Accesses are Constrained

Consider any implementation of a multi-writer snapshot object with $m \geq 2$ components shared by $n > m$ processes using m multi-writer registers. Our goal is to construct an execution in which one `SCAN` operation, called the *troublesome* `SCAN`, performs $\Omega(n)$ register accesses. Before constructing the execution, we prove in this section that, in any correct, space-optimal implementation of a snapshot object, processes access the registers in a very constrained way. We shall prove that no process may write to any register while performing a `SCAN` operation. We shall also show that every component A_j of the snapshot object has an associated register R_j such that R_j will be the first register written by any solo execution of an `UPDATE` to component A_j that begins from the initial configuration. The fact that snapshot operations access the registers in this constrained way will allow us, in Section 4, to arrange processes to cover all the registers that the troublesome `SCAN` will read. These processes will then be used to prevent the troublesome `SCAN` from getting the information it needs to terminate.

For any configuration C and for any set of processes $\mathcal{P}'$, the set of components with a pending `UPDATE` at C by a process in $\mathcal{P}'$ is denoted $CPU(C, \mathcal{P}')$. For each subset $\mathcal{A}$ of snapshot components, the set of processes that have a pending `UPDATE` at C to a component of $\mathcal{A}$ at C is denoted $PPU(C, \mathcal{A})$. For single components or processes, we abuse notation by writing $PPU(C, A_j)$ and $CPU(C, p)$ instead of $PPU(C, \{A_j\})$ and $CPU(C, \{p\})$.

An important notion in the proofs that follow is that of a hidden set of processes. Intuitively, a set of processes $\mathcal{P}'$ is hidden in configuration C if no process outside $\mathcal{P}'$ can deduce (either from its own local state or from the contents of registers) that any process in $\mathcal{P}'$ is in the middle of an `UPDATE`. Roughly speaking, there are three ways a process can be hidden. The process may not, in fact, be performing an `UPDATE`. It may be that the process has begun performing an `UPDATE`, but has not yet written enough evidence of the `UPDATE` into the registers, so that the evidence may be obliterated before processes outside $\mathcal{P}'$ see it. Finally, the process may have performed enough of its `UPDATE` so that processes outside $\mathcal{P}'$ cannot tell that the `UPDATE` is not complete. Knowing that a set $\mathcal{P}'$ is hidden in C allows us to draw conclusions about the behaviour of the implemented snapshot object. For example, if a process outside $\mathcal{P}'$, starting from C, performs `UPDATE`(i, v), for some component A_i and then performs a `SCAN`, it must return the value v for that component, regardless of any pending `UPDATE` operations to the component by processes of $\mathcal{P}'$. This is because we can pretend that any pending `UPDATE` operations of the second type mentioned above never happened, and that any of the third type are linearized before C. The following definition formalizes the idea of a hidden set of processes.

Definition 3.1 A set of processes $\mathcal{P}'$ is *hidden* in some configuration C if there exists a reachable configuration C' such that no process in $\mathcal{P}'$ has a pending `UPDATE` at C' and, for each process $p \notin \mathcal{P}'$, $C \approx_p C'$.

Note that if $\mathcal{P}''$ is a subset of a set of processes $\mathcal{P}'$ hidden in C, it does not necessarily follow that $\mathcal{P}''$ is hidden in C.

Covering arguments, introduced by Burns and Lynch [14] play an important role in our proofs. In such arguments it is often useful to show how one can reach, from a configuration where some registers are covered, another configuration where one more register is covered. The following lemma will allow us to do this if some processes are hidden.

Lemma 3.2 *Suppose $\mathcal{P}_\mathcal{R}$ covers $\mathcal{R}$ in configuration C and no process in $\mathcal{P}_\mathcal{R}$ has a pending* UPDATE *on component A_i in C. Let p be a process outside $\mathcal{P}_\mathcal{R}$. If $PPU(C, A_i) \setminus \{p\}$ is hidden in C, then in any solo execution by p that starts from C and includes a complete* UPDATE *to A_i, the* UPDATE *must contain a* `write` *to a register outside $\mathcal{R}$.*

Sketch of proof: First consider the simple case where no process, except possibly p, has a pending update on A_i at C. Suppose p runs by itself, starting from C, until it performs a complete `UPDATE` U on A_i. If U contains `write` operations only to registers in $\mathcal{R}$, all evidence that U occurred can be obliterated by next performing a block write to $\mathcal{R}$ by processes in $\mathcal{P}_\mathcal{R}$. This leads to a contradiction: a `SCAN` performed after the block write must return U's value for A_i, but the `SCAN` has no way of finding out about U.

Now consider the general case. Let α_p be a solo execution by p from C that ends with an `UPDATE` U to A_i. Since $PPU(C, A_i) \setminus \{p\}$ is hidden in C, there is another configuration C' with no pending `UPDATE` operations on A_i (except possibly one by p) such that configuration $C' \approx_{p'} C$ for all $p' \notin PPU(C, A_i) \setminus \{p\}$. In particular, $C' \approx_p C$, so α_p is a legal execution starting from C'. In C', no process (except possibly p) has a pending `UPDATE` on A_i, and $\mathcal{P}_\mathcal{R}$ covers $\mathcal{R}$. This is now the simple case that we considered in the previous paragraph, so the same argument proves that U must contain a `write` to some register outside $\mathcal{R}$. ∎

Let C_0 be an initial configuration in which all components of the snapshot object have value 0 and no process has a pending `UPDATE` or `SCAN` operation. Let $\mathcal{R} = \mathcal{P}_\mathcal{R} = \emptyset$, and consider a solo execution of an `UPDATE` operation on some component A_i by some process p. Notice that at C_0 there are no pending `UPDATE` operations on A_i, so $PPU(C_0, A_i) \setminus \{p\} = \emptyset$ is hidden in C_0. Therefore, Lemma 3.2 implies that the `UPDATE` contains a `write` to some register.

Corollary 3.3 *A solo execution of an* UPDATE *operation starting from an initial configuration must* `write` *to some register.*

We now define the key concept of a fatal configuration. It is fatal in the sense that, if it is reachable, then the implementation must be incorrect. If a configuration C is l-fatal, then it contains a set of l processes $\mathcal{P}_\mathcal{R}$ covering a set of l registers $\mathcal{R}$, but there are less than l snapshot components to which processes in $\mathcal{P}_\mathcal{R}$ have pending `UPDATE` operations. Now suppose there is an execution from C in which complete `UPDATE` operations to all other components are performed, followed by a block write to the l registers by processes in $\mathcal{P}_\mathcal{R}$. Then a subsequent `SCAN` will not be able to determine the correct values of some of these other components, because an adversary can construct an execution where the $m - l$ registers outside $\mathcal{R}$ do not contain sufficient information to describe the values written into $m - l + 1$ components by the complete `UPDATE` operations. The following formal definition specifies more precisely the conditions under which this kind of bad situation can arise.

Definition 3.4 Consider any integer l, $1 \le l \le m+1$. We say that a configuration C is *l-fatal* if there exists a set $\mathcal{R}$ of l registers and a set $\mathcal{P}_\mathcal{R}$ of l processes such that:

1. $\mathcal{P}_\mathcal{R}$ covers $\mathcal{R}$ at C,
2. $|CPU(C, \mathcal{P}_\mathcal{R})| < l$, and
3. $\mathcal{P}'$ is hidden in C, for all sets of processes $\mathcal{P}'$ that have no pending `UPDATE` operations to components in $CPU(C, \mathcal{P}_\mathcal{R})$ (i.e. such that $CPU(C, \mathcal{P}_\mathcal{R}) \cap CPU(C, \mathcal{P}') = \emptyset$).

Lemma 3.5 *For any integer l, $1 \le l \le m$, there is no reachable l-fatal configuration.*

Proof: We use reverse induction on l. Fix any integer l such that $1 \le l \le m$, and assume that there are no reachable l'-fatal configurations for $l < l' \le m$.

Suppose there is a reachable l-fatal configuration C. Let $\mathcal{R}$ be a set of l registers and let $\mathcal{P}_\mathcal{R}$ be a set of l processes that witness the l-fatality of C. Since $|CPU(C, \mathcal{P}_\mathcal{R})| < l \le m$, there exists a component $A_i \notin CPU(C, \mathcal{P}_\mathcal{R})$. Since $|\mathcal{P}_\mathcal{R}| = l \le m < n$, there exists $p \notin \mathcal{P}_\mathcal{R}$. Then $PPU(C, A_i) \setminus \{p\}$ is hidden in C.

Starting from C, let p finish any pending operation it may have at C, and then perform a complete `UPDATE` operation U on A_i. By Lemma 3.2, U contains a `write` to a register R not in $\mathcal{R}$. If $l = m$, then we already have a contradiction, since $\mathcal{R}$ contains all m registers in the system.

The remainder of the proof covers the case where $l < m$. Let C_1 be the configuration just before U writes to R for the first time.

We prove that C_1 is $(l+1)$-fatal, contrary to the induction hypothesis. Let $\mathcal{R}_1 = \mathcal{R} \cup \{R\}$, and let $\mathcal{P}_1 = \mathcal{P}_\mathcal{R} \cup \{p\}$. In configuration C_1, the processes of $\mathcal{P}_1$ have pending `UPDATE` operations on fewer than $l+1$ components, and they cover the $l+1$ registers of $\mathcal{R}_1$. Consider now any set $\mathcal{P}'$ of processes such that $CPU(C_1, \mathcal{P}_1) \cap CPU(C_1, \mathcal{P}') = \emptyset$. It remains to prove that $\mathcal{P}'$ is hidden in C_1.

We must construct a reachable configuration C'_1 such that no process in $\mathcal{P}'$ has a pending `UPDATE` at C'_1 and $C_1 \approx_{p'} C'_1$, for each process $p' \notin \mathcal{P}'$. In C_1, the sets of processes $\mathcal{P}_1$ (which contains $\mathcal{P}_\mathcal{R}$) and $\mathcal{P}'$ are updating disjoint sets of components. Since none of the processes in $\mathcal{P}_\mathcal{R}$ or $\mathcal{P}'$ take any steps between C and C_1, it follows that $CPU(C, \mathcal{P}_\mathcal{R})$ and $CPU(C, \mathcal{P}')$ are also disjoint. From the definition of l-fatal, if follows that $\mathcal{P}'$ is hidden in C. Thus, there exists a configuration C' such that no process in $\mathcal{P}'$ has a pending `UPDATE` at C' and, for each process $p' \notin \mathcal{P}'$, $C' \approx_{p'} C$. In particular, $C' \approx_p C$, so p's solo execution from C to C_1 can also begin from C', taking the system to a configuration which is the required C'_1. It follows by Observation 2.2 that $C_1 \approx_p C'_1$.

We use C'_1 to show that $\mathcal{P}'$ is hidden in C_1. Since only process p takes steps from C' to C'_1, it follows that $C' \sim_{p'} C'_1$, for all processes $p' \neq p$. Since no process in $\mathcal{P}'$ has a pending `UPDATE` at C', this implies that no process in $\mathcal{P}'$ has a pending `UPDATE` at C'_1. Furthermore, for each process $p' \notin \mathcal{P}' \cup \{p\}$, $C_1 \sim_{p'} C \sim_{p'} C' \sim_{p'} C'_1$. This and the fact that $C_1 \approx_p C'_1$ imply that for each process $p' \notin \mathcal{P}'$, $C_1 \approx_{p'} C'_1$. Thus, C_1 is $(l+1)$-fatal, and this completes the proof. ∎

Now we show that if processes do not access the registers in a particular way, the system will get into a fatal configuration. First, we show that if a `SCAN` operation ever contains a `write` to a register, there is a configuration where processes performing `UPDATE` operations to zero components cover one register. Such a configuration is 1-fatal.

Lemma 3.6 SCAN *operations never write to registers.*

Proof: Suppose a SCAN by some process p does perform a write operation on a register R in some execution. Starting from the configuration just before the write occurs, we run all other processes until they complete their operations, ending with a configuration C. We show that C will be 1-fatal, contradicting Lemma 3.5. Let $\mathcal{P}_{\mathcal{R}} = \{p\}$ and $\mathcal{R} = \{R\}$. The set $\mathcal{P}_{\mathcal{R}}$ covers $\mathcal{R}$ at C and $|CPU(C, \mathcal{P}_{\mathcal{R}})| = 0$. For any set of processes $\mathcal{P}'$, $\mathcal{P}'$ is hidden in C. (The reachable configuration that satisfies the conditions of Definition 3.1 is C itself, since none of the processes in $\mathcal{P}'$ have pending UPDATE operations in C.) ∎

We now turn attention to UPDATE operations. We consider executions where a collection of UPDATE operations all run until each one is about to perform its first write to a register. We show that we can associate one register to each component such that every UPDATE to the component covers that register.

Consider a solo execution by process p_i of the operation UPDATE(x_i, i), starting from the initial configuration C_0. By Corollary 3.3, this UPDATE contains at least one write to a register. Denote by $I(x_i, p_i)$ the longest prefix of this UPDATE containing no write operations. (The sequence $I(x_i, p_i)$ may contain no steps at all.) Let α_0 be the execution $I(x_1, p_1) \cdot \ldots \cdot I(x_{n-2}, p_{n-2})$. This is a legal execution because each $I(x_i, p_i)$ contains only read operations. Let Q_0 be the configuration at the end of α_0. Note that the configuration Q_0 depends on the values of $x_1, \ldots, x_{n-2}$.

Lemma 3.7 *For each component A_j, every process in $PPU(Q_0, A_j)$ covers the same register in Q_0.*

Proof: Assume not. Then there are processes p_i and $p_{i'}$ in $PPU(Q_0, A_j)$ covering different registers R and R' in Q_0. Let C be the configuration obtained by performing $I(x_i, p_i) \cdot I(x_{i'}, p_{i'})$ starting from C_0. (This execution is legal since it contains only the initial read steps of the two UPDATE operations.) We prove that C is 2-fatal, which contradicts Lemma 3.5. The set $\{p_i, p_{i'}\}$ covers $\{R, R'\}$ in C. Let $\mathcal{P}'$ be any set of processes that does not contain p_i or $p_{i'}$. Then $\mathcal{P}'$ is hidden in C, since none of the processes in $\mathcal{P}'$ have pending UPDATE operations in C. ∎

For each snapshot component A_l, let R_l denote the register covered in Q_0 by every process in $PPU(Q_0, A_l)$. The next lemma strengthens Lemma 3.7 to show that processes that perform UPDATE operations to different snapshot components must perform their first write operations to different registers.

Lemma 3.8 *$R_l \neq R_{l'}$ for distinct $l, l' \in \{1, \ldots, m\}$.*

Sketch of proof: Suppose not. Then there exist two processes p_i and $p_{i'}$ that are updating different components A_j and $A_{j'}$, but both cover register R in Q_0, i.e. $R_j = R_{j'} = R$. Ssince p_i performs no writes in $I(x_i, p_i)$, it follows that $\{p_i\}$ is hidden in Q_0.

Let $\mathcal{R} = \{R\}$ and let $\mathcal{P}_{\mathcal{R}} = \{p_{i'}\}$. Then $\mathcal{P}_{\mathcal{R}}$ covers $\mathcal{R}$ and no process in $\mathcal{P}_{\mathcal{R}}$ has a pending UPDATE on A_j in Q_0. Let process p_n (which has not yet taken any steps) run a complete UPDATE on A_j, starting from Q_0. Lemma 3.2 implies that p_n will write to a register $R' \neq R$ during its update to A_j. Let C be the configuration just before this write. At C, p_i and p_n are both updating A_j, but are covering R and R'. We show this configuration is 2-fatal, contradicting Lemma 3.5.

Let $\mathcal{P}'$ be a set of processes that do not have pending UPDATE operations on A_j in C. We must show $calP'$ is hidden in C. Since the processes of $\mathcal{P}'$ have not performed any write operations before Q_0, $\mathcal{P}'$ is hidden in Q_0. Let Q_0' be the configuration that satisfies Definition 3.1 for Q_0 and $\mathcal{P}'$. Process p_n cannot distinguish Q_0 from Q_0'. By Observation 2.2, p_n's execution that takes the system from Q_0 to C can also be run starting from Q_0', and no process except p_n can distinguish the resulting configuration, C', from C. There are no pending UPDATE operations by processes in $\mathcal{P}'$ at Q_0', so there are none at C'. ∎

Theorem 3.9 *Any n-process implementation of an m-component (multi-writer) snapshot object requires at least m registers, for $n > m$.*

Note that Theorem 3.9 is not true for $n < m$: As mentioned in the introduction, a multi-writer snapshot object shared by n processes can be implemented from n single-writer registers.

4 The Time Lower Bound

Our lower bound is obtained by using an adversary to construct an execution α in which a *troublesome* SCAN, denoted T, takes $\Omega(n)$ steps. This SCAN is performed by process $t = p_n$. In α, processes $p_1, \ldots, p_{n-2}$ each perform an UPDATE operation. Between every two of these UPDATE operations, process $q = p_{n-1}$ performs two UPDATE operations. All of these UPDATE operations use different values. At various points in the execution, SCAN operations are performed by the processes $p_1, \ldots, p_{n-3}$ to enforce the desired linearization of the UPDATE operations.

Just before t performs its i'th step (which must start with a read operation, by Lemma 3.6), the adversary has p_i execute the first write of its UPDATE to the register t is about to read. This ensures that t learns nothing about the UPDATE operations performed by q. The UPDATE operations performed by q are chosen so that, at any possible linearization point for T in α, at least one component of the snapshot object contains a value used by process q for an UPDATE. It follows that T cannot terminate during α.

Now we present a detailed description of the construction of α. The first part of α consists of the execution $\alpha_0 = I(x_1, p_1) \cdots I(x_{n-2}, p_{n-2})$, described in the previous section. Recall that $I(x_i, p_i)$ is a prefix of an UPDATE(x_i, i) operation by p_i. The execution α_0 takes the system from the initial configuration C_0 to Q_0, in which p_i covers register R_{x_i} for $1 \leq i \leq n-2$. Because processes do not perform any write operations during this portion of the execution, the adversary can delay choosing the components $A_{x_1}, \ldots, A_{x_{n-2}}$ these processes update.

Let $q_1', q_1, q_2', q_2, \ldots, q_{n-3}', q_{n-3}$, be distinct values, all chosen to be greater than $n-2$ so that they differ from the values for the UPDATE operations performed by $p_1, \ldots, p_{n-2}$. These will be used as the values for the UPDATE operations performed by process q.

The rest of α is defined inductively. Let $1 \leq i < n-2$ and suppose we have defined a prefix of α that takes the system to configuration Q_{i-1}. We show, in Lemma 4.3, that the troublesome SCAN has not yet terminated. By Lemma 3.6, its next operation is a read. Fix x_i to be the index of the register from which this read will occur. Let x_0 be some index different from x_1. The next fragment of α, denoted α_i, consists of the following six parts:

- $U(x_{i-1}, q'_i)$: the sequence of steps for the operation $\mathtt{UPDATE}(x_{i-1}, q'_i)$ performed by q,
- $U(x_i, q_i)$: the sequence of steps for the operation $\mathtt{UPDATE}(x_i, q_i)$ performed by q,
- w_i: the next step by p_i, which is a `write` to register R_{x_i},
- r_i: the i'th step by t, which is a `read` of register R_{x_i},
- $Z(x_i, p_i)$: the remainder of the operation $\mathtt{UPDATE}(x_i, i)$ by p_i, and
- S_i: the sequence of steps for the operation `SCAN` by p_i.

Let $U(x_i, p_i) = I(x_i, p_i) \cdot w_i \cdot Z(x_i, p_i)$ be the sequence of steps for the operation $\mathtt{UPDATE}(x_i, i)$ performed by p_i in α. Let Q_i be the configuration reached by running α_i, starting from Q_{i-1}. Then we have

$$\begin{aligned}
\alpha &= \alpha_0 \cdot \alpha_1 \cdot \ldots \cdot \alpha_{n-3} \\
&= I(x_1, p_1) \cdot I(x_2, p_2) \cdot \ldots \cdot I(x_{n-2}, p_{n-2}) \cdot \\
&\quad U(x_0, q'_1) \cdot U(x_1, q_1) \cdot w_1 \cdot r_1 \cdot Z(x_1, p_1) \cdot S_1 \cdot \\
&\quad U(x_1, q'_2) \cdot U(x_2, q_2) \cdot w_2 \cdot r_2 \cdot Z(x_2, p_2) \cdot S_2 \cdot \\
&\quad U(x_2, q'_3) \cdot U(x_3, q_3) \cdot w_3 \cdot r_3 \cdot Z(x_3, p_3) \cdot S_3 \cdot \\
&\quad \vdots \\
&\quad U(x_{n-4}, q'_{n-3}) \cdot U(x_{n-3}, q_{n-3}) \cdot w_{n-3} \cdot r_{n-3} \cdot \\
&\qquad Z(x_{n-3}, p_{n-3}) \cdot S_{n-3}.
\end{aligned}$$

Because the `SCAN` operations S_i and T begin after the first two `UPDATE` operations by q are complete, the result of every `SCAN` operation must contain at least two non-zero values. The execution α was carefully chosen so that we can reason about the ordering of various snapshot operations in the linearization. Some facts about the linearization follow immediately from the definition of α. If one operation is completed before another begins, the former operation must come before the latter in the linearization, so the linearization order for the `UPDATE` operations by q and for the `SCAN` operations performed by $p_1, \ldots, p_{n-3}$ is

$$\begin{array}{l}
U(x_0, q'_1), U(x_1, q_1), S_1, \\
U(x_1, q'_2), U(x_2, q_2), S_2, \\
U(x_2, q'_3), U(x_3, q_3), S_3, \\
\vdots \\
U(x_{n-4}, q'_{n-3}), U(x_{n-3}, q_{n-3}), S_{n-3}.
\end{array} \qquad (1)$$

Next, we determine where the linearization point of $U(x_i, p_i)$ fits into this sequence. Our goal is to prove that it must come between $U(x_i, q_i)$ and S_i. We do this by showing that the response returned by S_i includes the value i written into the snapshot object by p_i. To accomplish this, we first show that p_i's `write` w_i obliterates all evidence of the `UPDATE` operation $U(x_i, q_i)$ to the same component.

Lemma 4.1 *For $1 \le i < n-2$, $U(x_i, q_i)$ performs* `write` *operations only to register R_{x_i}.*

Sketch of proof: Suppose that the `UPDATE` operation $U(x_i, q_i)$ performs `write` operations to some other register $R \neq R_{x_i}$. Let C be the configuration just before the first such `write` occurs. We show that C is 2-fatal, contradicting Lemma 3.5. For the purpose of this proof, the adversary chooses $x_{n-2} = x_i$. Then in C, p_{n-2} and q are both performing `UPDATE` operations on A_{x_i}, but are covering two registers, R_{x_i} and R. Let $\mathcal{P}'$ be any set of processes that do not have pending `UPDATE` operations on A_{x_i} at C. We must show $\mathcal{P}'$ is hidden in C. Let α' be the prefix of α that leads to configuration C. If all steps by processes in $\mathcal{P}'$ that have pending `UPDATE` operations at C are removed from α', the execution leads to a different configuration C'. The removed steps are all `read` operations, so the execution is still legal, and no process outside $\mathcal{P}'$ can distinguish C from C'. ■

Lemma 4.2 *$U(x_i, p_i)$ must be linearized after $U(x_i, q_i)$ and before S_i.*

Sketch of proof: The operation $U(x_i, p_i)$ terminates before S_i begins and must therefore be linearized earlier. We shall show that the `SCAN` S_i returns i as the value of A_{x_i}. It follows that $U(x_i, p_i)$ must be linearized after $U(x_i, q_i)$.

Let α' be the prefix of α up to the end of S_i. Let α'' be the execution obtained by removing from α' the `UPDATE` operation $U(x_i, q_i)$ and all steps by processes $p_{i+1}, \ldots, p_{n-2}$. Hence,

$$\begin{aligned}
\alpha' &= I(x_1, p_1) \cdot \ldots \cdot I(x_{n-2}, p_{n-2}) \cdot \alpha_1 \cdot \ldots \cdot \alpha_{i-1} \cdot \\
&\quad U(x_{i-1}, q'_i) \cdot U(x_i, q_i) \cdot w_i \cdot r_i \cdot Z(x_i, p_i) \cdot S_i, \text{ and} \\
\alpha'' &= I(x_1, p_1) \cdot \ldots \cdot I(x_i, p_i) \cdot \alpha_1 \cdot \ldots \cdot \alpha_{i-1} \cdot \\
&\quad U(x_{i-1}, q'_i) \cdot w_i \cdot r_i \cdot Z(x_i, p_i) \cdot S_i
\end{aligned}$$

By Lemma 4.1, $U(x_i, q_i)$ contains `write` operations only to R_{x_i}. Once $U(x_i, q_i)$ is complete, register R_{x_i} is immediately overwritten by w_i in α'. Furthermore, processes $p_{i+1}, \ldots, p_{n-2}$ perform only `read` operations in α'. It follows from Observation 2.2 that α'' is a legal execution.

In α, the only `UPDATE` operations that can possibly be linearized between $U(x_i, q_i)$ and S_i are $U(x_i, p_i), \ldots,$ $U(x_{n-2}, p_{n-2})$; all other `UPDATE` operations terminate before $U(x_i, q_i)$ begins or begin after S_i terminates. So the value S_i returns for component A_{x_i} must be an element of the set $\{q_i, i, i+1, \ldots, n-2\}$. However, S_i returns the same value in α'', since α' and α'' are indistinguishable to p_i. In α'', `UPDATE` operations with values $q_i, i+1, \ldots, n-2$ do not exist, so S_i cannot possibly return any of these values. Thus S_i must return the value i. This implies that $U(x_i, p_i)$ is linearized after $U(x_i, q_i)$ in α. ■

Combining this lemma with the partial linearization in (1), we can construct the linearization of all operations in α except the troublesome `SCAN` and $U(x_{n-2}, p_{n-2})$:

$$\begin{array}{l}
U(x_0, q'_1), U(x_1, q_1), U(x_1, p_1), S_1, \\
U(x_1, q'_2), U(x_2, q_2), U(x_2, p_2), S_2, \\
U(x_2, q'_3), U(x_3, q_3), U(x_3, p_3), S_3, \\
\vdots \\
U(x_{n-4}, q'_{n-3}), U(x_{n-3}, q_{n-3}), U(x_{n-3}, p_{n-3}), S_{n-3}.
\end{array} \qquad (2)$$

Observe that the snapshot object never contains values from the set $\{1, 2, \ldots, n-3\}$ in more than one component at the same time: the value entered into the snapshot object by $U(x_j, p_j)$ is changed by the `UPDATE` operation $U(x_j, q'_{j+1})$ before $U(x_{j+1}, p_{j+1})$ occurs. This is the sole purpose of the `UPDATE` operations $U(x_j, q'_{j+1})$ in our construction. It follows that T must return at least one value written by q. We now show this is impossible unless T takes at least $n-3$ steps.

Lemma 4.3 *The troublesome* SCAN T *cannot terminate before it performs* $n-3$ *steps.*

Sketch of proof: Suppose T terminates within $i < n-3$ steps. Clearly $i > 0$; otherwise SCAN operations would never access shared memory. Let $\alpha' = I(x_1, p_1) \cdot I(x_2, p_2) \cdot \ldots \cdot I(x_i, p_i) \cdot w_1 \cdot r_1 \cdot w_2 \cdot r_2 \cdot \ldots w_i \cdot r_i$. Notice that this is a legal execution which the process t executing T cannot distinguish from α. Recall that T must return non-zero values for at least two components in α because T begins after $U(x_0, q_1')$ and $U(x_1, q_1)$ are complete. But the only non-zero values that the troublesome SCAN can return in α' (and hence in α) for any component are $1, \ldots, i$; any other values imply the result is invalid in α', because UPDATE operations with no other values exist in α'. So at least two components of T's response contain values from the set $\{1, \ldots, i\}$. This contradiction shows the lemma holds. ∎

Lemma 4.3 immediately gives a lower bound on the time complexity of T.

Theorem 4.4 *Any* n*-process implementation of an* m*-component (multi-writer) snapshot object from* m *(multi-writer) registers requires* $\Omega(n)$ *steps for a* SCAN *operation.*

5 Conclusion

As in [10, 15, 23], the lower bound given here can easily be generalized to apply to the problem of implementing a snapshot object from any historyless objects. A *historyless object* [15] is one where the value of an object depends only on the last operation applied to it.

Our construction of incorrect executions starting from any fatal configuration used very few properties of snapshot objects. Thus, the concept of a fatal configuration may be more generally applicable for establishing time complexity lower bounds of space-limited algorithms. For example, fatal configurations might be useful in studying the time complexity of implementing an arbitrary set of m objects using m registers, provided that it is possible to read the objects without changing them.

For the time complexity of SCAN operations in space-optimal implementations, there is still a large gap between our lower bound of $\Omega(n)$ and the upper bound of $O(mn)$ obtained by modifying the algorithm of Afek et al. [1]. Obtaining a bigger lower bound might involve considering executions in which some processes perform many operations. This is the case for some covering arguments [14, 23], but not others [2, 5, 15]. In our proof, only one process, q, performs many operations. The rest of the processes perform at most one UPDATE and one SCAN each.

We would also like to extend our lower bound to implementations of m-component multi-writer snapshot objects that use more than m multi-writer registers, perhaps obtaining a tradeoff between time and space. As a first step, we are working on a lower bound for the case where single-writer registers are available in addition to the m multi-writer registers.

Acknowledgements

This research was supported by the Natural Sciences and Engineering Research Council of Canada and Communications and Information Technology Canada. A major part of this work was done while Panagiota Fatourou was affiliated with the Department of Computer Science at the University of Toronto.

References

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4), pages 873–890, September 1993.

[2] Yehuda Afek, Pazi Boxer, and Dan Touitou. Bounds on the shared memory requirements for long-lived and adaptive objects. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 81–89, 2000.

[3] James H. Anderson. Composite registers. *Distributed Computing*, 6(3), pages 141–154, April 1993.

[4] James H. Anderson. Multi-writer composite registers. *Distributed Computing*, 7(4), pages 175–195, 1994.

[5] James H. Anderson and Yong-Jik Kim. An improved lower bound for the time complexity of mutual exclusion. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 90–99, 2001.

[6] James H. Anderson and Jae-Heon Yang. Time/contention trade-offs for multiprocessor synchronization. *Information and Computation*, 124(1), pages 68–84, January 1996.

[7] James Aspnes. Time- and space-efficient randomized consensus. *Journal of Algorithms*, 14(3), pages 414–431, May 1993.

[8] James Aspnes and Maurice Herlihy. Fast, randomized consensus using shared memory. *Journal of Algorithms*, 11(2), pages 441–461, September 1990.

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

[10] Hagit Attiya, Alla Gorbach, and Shlomo Moran. Computing in totally anonymous asynchronous shared memory systems. In *Distributed Computing, 12th International Symposium*, volume 1499 of *LNCS*, pages 49–61, 1998. Full version available from `www.cs.technion.ac.il/~hagit`.

[11] Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? *Journal of the ACM*, 41(4), pages 725–763, July 1994.

[12] Hagit Attiya and Ophir Rachman. Atomic snapshots in $O(n \log n)$ operations. *SIAM Journal on Computing*, 27(2), pages 319–340, April 1998.

[13] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* McGraw-Hill, 1998.

[14] James Burns and Nancy Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2), pages 171–184, December 1993.

[15] Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *Journal of the ACM*, 45(5), pages 843–862, September 1998.

[16] Rainer Gawlick, Nancy Lynch, and Nir 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.

[17] Maurice Herlihy. Randomized wait-free objects. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, pages 11–21, 1991.

[18] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1), pages 124–149, January 1991.

[19] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), pages 463–492, July 1990.

[20] Michiko Inoue, Wei Chen, Toshimitsu Masuzawa, and Nobuki Tokura. Linear time snapshots using multi-writer multi-reader registers. In *Distributed Algorithms, 8th International Workshop*, volume 857 of *LNCS*, pages 130–140, 1994.

[21] A. Israeli, A. Shaham, and A. Shirazi. Linear-time snapshot implementations in unbalanced systems. *Mathematical Systems Theory*, 28(5), pages 469–486, September/October 1995.

[22] Amos Israeli and Asaf Shirazi. The time complexity of updating snapshot memories. *Information Processing Letters*, 65(1), pages 33–40, January 1998.

[23] Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2), pages 438–456, 2000.

[24] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[25] Shlomo Moran, Gadi Taubenfeld, and Irit Yadin. Concurrent counting. *Journal of Computer and System Sciences*, 53(1), pages 61–78, August 1996.