A Tight Time Lower Bound for Space-Optimal Implementations of Multi-Writer Snapshots

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

A snapshot object consists of a collection of $m > 1$ components, each capable of storing a value, shared by n processes in an asynchronous shared-memory distributed system. It supports two operations: a process can `UPDATE` any individual component or atomically `SCAN` the entire collection to obtain the values of all the components. It is possible to implement a snapshot object using m registers so that each operation takes $O(mn)$ time.

In a previous paper, we proved that m registers are necessary to implement a snapshot object with $m < n-1$ components. Here we prove that, for any such space-optimal implementation, $\Omega(mn)$ steps are required to perform a `SCAN` operation in the worst case, matching the upper bound. We also extend our space and time lower bounds to implementations that use single-writer registers in addition to the multi-writer registers. Specifically, we prove that at least m multi-writer registers are still needed, provided the `SCANS` do not read a large fraction of the single-writer registers. We also prove that any implementation that uses single-writer registers in addition to m multi-writer registers uses $\Omega(\sqrt{mn})$ steps in the worst case. Our proof yields insight into the structure of any implementation that uses only m multi-writer registers, showing that processes must access the multi-writer registers in a very constrained way.

Categories and Subject Descriptors

E.1 [**Data**]: Distributed Data Structures; F.2.3 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity—*Tradeoffs between Complexity Measures*

General Terms

Theory

Keywords

snapshot, lower bounds, space-optimal, shared-memory distributed computing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
STOC'03, June 9–11, 2003, San Diego, California, USA.
Copyright 2003 ACM 1-58113-674-9/03/0006 ...$5.00.

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 the components. 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 [18] and other types of objects [9, 19]. 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.

We consider the problem of wait-free implementation of a (multi-writer) snapshot object with $m > 2$ components shared by $n > m + 1$ processes. An implementation from multi-writer registers requires at least m registers, even if the size of each register is unbounded, and an $\Omega(n)$ lower bound is known on the worst case time complexity of a `SCAN` operation for any implementation using only m multi-writer registers [15]. Here, we significantly improve the lower bound to $\Omega(mn)$, matching a known upper bound [1].

Covering arguments, introduced by Burns and Lynch [14], have been used to prove a variety of lower bounds in distributed computing [2, 5, 10, 16, 17, 25, 27], mostly for space complexity. In a covering argument, a configuration is constructed in which some processes are poised to write to different registers. Information in those registers can be hidden by having these processes overwrite the values stored there. Our lower bound proof uses covering arguments in a variety of ways. We begin by defining a *fatal* configuration (where some processes are covering too many registers) and prove that correct space-optimal snapshot implementations cannot ever reach a fatal configuration. This allows us to derive a significant amount of structural information about space optimal implementations: each component must be simulated by one multi-writer register and this is the only multi-writer register to which processes performing `UPDATES` to this component can write. Furthermore, `SCANS` cannot write to multi-writer registers. Thus, we can restrict atten-

tion to algorithms with these properties. Next, we show how an adversary can 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.

The execution that is constructed is comprised of a number of rounds. In each round, if the troublesome SCAN does not read nearly every register, then an adversary can put essential information in a register that the troublesome SCAN does not read, causing it to behave incorrectly. However, if the troublesome SCAN does read nearly every register during the round, the adversary can use one process to hide this essential piece of information by overwriting it just before the troublesome SCAN reads it. This shows that space-optimal implementations of SCAN must proceed by repeatedly reading all or almost all of the m multi-writer registers. A sequence of such collects is, in fact, the way known snapshot implementations perform SCAN.

Our proof is sufficiently general to handle implementations that use one single-writer register per process, in addition to multi-writer registers. In this case, we show that, unless SCAN and UPDATE operations read a large fraction of the single-writer registers, at least m multi-writer registers are still needed. We also get an $\Omega(\sqrt{mn})$ time complexity lower bound on any implementation that uses m multi-writer registers and n single-writer registers. The key idea needed for this extension is to throw away some processes at each round, such as those whose single-writer registers are read by the troublesome SCAN. This renders reads of single-writer registers useless for gathering information.

We give a brief description of related work in Section 2 and the model in Section 3. Section 4 contains our results that constrain the behaviour of implementations of a snapshot object. Then, in Section 5, we use these results to construct an execution in which the troublesome SCAN takes many steps. We conclude with a brief discussion in Section 6.

2. RELATED WORK

Snapshot objects were 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.

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, 22, 23], this shows that a multi-writer snapshot can be implemented 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]. Its time complexity is $O(n^2 + mn)$ and its registers have bounded size. This is based on their $O(n^2)$ time implementation of a single-writer snapshot object from bounded size registers, which, in turn, is based on their $O(n^2)$ time implementation of a single-writer snapshot object from unbounded registers. To keep the register sizes bounded, they use handshaking bits instead of sequence numbers. We describe an $O(mn)$ time implementation of a multi-writer snapshot object from m unbounded registers, based on their first and third algorithms. Each process that performs an UPDATE first performs a SCAN and then writes the resulting vector of values, its id, and a sequence number together with the updated value to the corresponding multi-writer register. The sequence numbers are used to distinguish the different times a process UPDATES the same component with the same value. To perform a SCAN, a process p repeatedly reads all m registers until either two consecutive passes give the same result or p reads two writes by some other process q that have occurred since the first pass. Note that, in the latter case, the first write by q may include the result of a SCAN that is out of date, but the second write contains the result of a SCAN that began after p's SCAN began, so p can return this result. A SCAN will terminate after at most $2n + 1$ passes, so this algorithm has complexity $O(mn)$. Here we show that this implementation is optimal.

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 [24] 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 [25] proved that any randomized implementation of an n-component 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 $m < n$ components trivially implements a single-writer snapshot object for m processes. This implies a lower bound of $m - 1$ on the space and time complexity of any implementation of an m-component multi-writer snapshot object from unbounded multi-writer registers.

In [15], we improved the lower bound on the number of registers to m for $m < n$. More significantly, we obtained an $\Omega(n)$ time bound for implementations using m multi-writer registers. We build on these results to obtain the improved lower bounds proved here.

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 oper-

ation by a process running by itself finishes within k steps. Using a simple reduction, they prove a space lower bound on long-lived implementations for which the time complexity depends on the number of processes that run concurrently, rather than the number of processes in the system. Specifically, they consider such an implementation of a *collect* object, which is a weak form of single-writer snapshot object where the `SCAN` is not required to be atomic. They prove that a non-constant number of multi-writer registers (in addition to single-writer registers) must be used.

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. Like us, by using only some of the processes, they are able to modify an execution in which `reads` of single-writer registers occur to obtain an execution in which `reads` of single-writer registers do not occur (or, at least, provide no useful information). 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 show that our assumptions hold for any space-optimal implementation of a multi-writer snapshot, so that the removal of these assumptions does not weaken the lower bound.

3. MODEL

We use a fairly standard model of asynchronous shared-memory systems in which a set of n deterministic processes $\mathcal{P} = \{p_1, \ldots, p_n\}$ communicate by accessing shared data structures called objects. Each operation is performed on an object atomically and returns a result to the process that invoked it. 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, 26].

A *multi-writer register* is an object that stores a value from a specified set. The register is *unbounded* if this set is infinite. Processes may access a register in two ways: they can `read` the value of the register or `write` a value into the register. 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 operations. 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 *configuration* describes a system at some point in time. It is comprised of a state for each process and the value stored in each object. 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. 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 of an algorithm is *legal* if the sequence of steps performed by each process follows the algorithm for that process, and, for each object, the responses to the operations performed on it are in accordance with the object's specification (and the value initially stored in the object). We use $\alpha \cdot \beta$ to denote the concatenation of two executions α and β. A configuration C is *reachable* if there is a finite, legal execution starting from an initial configuration and ending in C.

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* by $\mathcal{P}_\mathcal{R}$ 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 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. 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 $CPU(C, \mathcal{P}')$. A process is *inactive* at a configuration C if it does not have a pending `UPDATE` or `SCAN` operation.

We consider only *linearizable, wait-free* implementations [20, 21]. 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, 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.

4. CONSTRAINTS ON MULTI-WRITER REGISTER ACCESSES

Consider any implementation of an m-component multi-writer snapshot object for n processes from at most m multi-writer registers and n single-writer registers. We assume that $n \geq m + 2$. In this section, we show that processes are constrained in the ways that they can access the multi-writer registers. These constraints will allow us to construct an execution in Section 5 where one operation takes many steps.

Covering arguments 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 technical lemma will allow us to do this.

LEMMA 4.1. *Suppose a set of processes $\mathcal{P}_\mathcal{R}$ covers a set of registers $\mathcal{R}$ in configuration C and no process in $\mathcal{P}_\mathcal{R}$ has a pending* `UPDATE` *on some component A_i in C. Further suppose that all processes not in $\mathcal{P}_\mathcal{R}$ are inactive at C. Let C' be the configuration obtained from C when the processes in $\mathcal{P}_\mathcal{R}$ perform a block write. Consider the solo execution of a* `SCAN` *by a process $q \notin \mathcal{P}_\mathcal{R}$ starting from C' and let $\mathcal{B}$ be the set of processes whose single-writer registers q* `reads`*. Then, for all $p \notin \mathcal{P}_\mathcal{R} \cup \{q\} \cup \mathcal{B}$, the solo execution by p of an* `UPDATE` *to A_i starting from C must contain a* `write` *to a multi-writer register outside $\mathcal{R}$.*

PROOF. Suppose not. Let C'' be the configuration obtained when p performs a solo execution of an `UPDATE` to A_i starting from C and then the processes in $\mathcal{P}_\mathcal{R}$ perform a block write. Since p does not `write` to a multi-writer register outside $\mathcal{R}$, each shared object, except for p's single-writer register, has the same value in both C' and C''. Since q is in the same state in both C' and C'' and it does not `read` from p's single-writer register in the solo execution of its `SCAN` starting from C', it returns the same value when it performs this `SCAN` starting from C''. But any `SCAN` starting from C'' must return the new value of A_i and any `SCAN` starting from C' must return the old value. This is a contradiction. □

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. (The version of the definition we use here is simpler than our original definition in [15].)

DEFINITION 4.2. Consider any integer l, $1 \leq l \leq m$. We say that a configuration C is *l-fatal* if there exists a set $\mathcal{R}$ of l multi-writer registers and a set $\mathcal{P}_\mathcal{R}$ of l processes such that $\mathcal{P}_\mathcal{R}$ covers $\mathcal{R}$ at C and $|CPU(C, \mathcal{P}_\mathcal{R})| < l$.

Suppose there is an execution from an l-fatal configuration C in which complete `UPDATE` operations to all components outside $CPU(C, \mathcal{P}_\mathcal{R})$ are performed, followed by a block write by the processes in $\mathcal{P}_\mathcal{R}$ to the l multi-writer registers in $\mathcal{R}$. Then if a subsequent `SCAN` does not `read` very many single-writer registers, it will not be able to determine the correct values of some of these components. This is because an adversary can construct an execution where the $m - l$ multi-writer registers outside $\mathcal{R}$ do not contain sufficient information to describe the values written into $m - l + 1$ components by the complete `UPDATE` operations. This argument is formalized in the following lemma.

LEMMA 4.3. *Suppose that every solo execution of a* `SCAN` *performs* `reads` *from fewer than $n - m - 1$ different single-writer registers. Then, for any integer l, $1 \leq l \leq m$, there is no reachable l-fatal configuration.*

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

Suppose there is a reachable l-fatal configuration C_1. Let $\mathcal{R}$ be a set of l multi-writer registers and let $\mathcal{P}_\mathcal{R}$ be a set of processes that witness the l-fatality of C_1. Let C_2 be the configuration obtained from C_1 by running all processes not in $\mathcal{P}_\mathcal{R}$ until they are inactive. Since $|CPU(C_2, \mathcal{P}_\mathcal{R})| < l \leq m$, there exists a component $A_i \notin CPU(C_2, \mathcal{P}_\mathcal{R})$. Let C_2' be the configuration obtained from C_2 when the processes in $\mathcal{P}_\mathcal{R}$ perform a block write.

Let $q \notin \mathcal{P}_\mathcal{R}$. (Such a process exists since $|\mathcal{P}_\mathcal{R}| = l \leq m < n$.) Consider the solo execution of a `SCAN` by q starting from C_2' and let $\mathcal{B}$ be the set of processes whose single-writer registers q `reads`. Since $|\mathcal{P}_\mathcal{R}| = l \leq m$ and $|\mathcal{B}| < n - m - 1$, there exists $p \notin \mathcal{P}_\mathcal{R} \cup \{q\} \cup \mathcal{B}$. Hence, by Lemma 4.1, p's solo execution of an `UPDATE` to A_i starting from C_2 must contain a `write` to a multi-writer register $R \notin \mathcal{R}$.

If $l = m$, then we have a contradiction, since $\mathcal{R}$ contains all m multi-writer registers in the system.

If $l < m$, let C_3 be the reachable configuration obtained by performing p's solo execution of an `UPDATE` to A_i starting from C_2 until just before p `writes` to R for the first time. Let $\mathcal{R}' = \mathcal{R} \cup \{R\}$, and let $\mathcal{P}_{\mathcal{R}'} = \mathcal{P}_\mathcal{R} \cup \{p\}$. Then $|\mathcal{R}'| = |\mathcal{P}_{\mathcal{R}'}| = l + 1$, $\mathcal{P}_{\mathcal{R}'}$ covers $\mathcal{R}'$ at C_3, and $CPU(C_3, \mathcal{P}_{\mathcal{R}'}) = CPU(C_2, \mathcal{P}_\mathcal{R}) \cup \{A_i\}$, so $|CPU(C_3, \mathcal{P}_{\mathcal{R}'})| < l + 1$. Thus C_3 is $(l+1)$-fatal, contrary to the induction hypothesis. □

Assume, for the remaining lemmas in this section, that every solo execution of a `SCAN` performs `reads` from fewer than $n - m - 1$ different single-writer registers. We now show that if processes do not access the multi-writer registers in a particular way, the system will get into a fatal configuration.

LEMMA 4.4. `SCAN` *operations never* `write` *to multi-writer registers.*

PROOF. Suppose there is an execution of a `SCAN` operation by a process p that contains a `write` to a multi-writer register R. Consider the configuration C that occurs just before the `write`. Since $\{p\}$ covers $\{R\}$ and $CPU(C, \{p\})$ is empty, this configuration is 1-fatal, contradicting Lemma 4.3. □

Consider the solo execution of a `SCAN` by some process t, starting from the initial configuration C_0 and let $\mathcal{D}$ be the set containing t and all processes whose single-writer registers t `reads`. Let $d = |\mathcal{D}| < n - m$. For each process $p_i \notin \mathcal{D}$ and each component A_j, consider the solo execution of `UPDATE`(j, p_i) by process p_i starting from the initial configuration C_0. Since all processes are inactive at C_0, Lemma 4.1 implies that this execution contains at least one `write` to a multi-writer register. Denote by $R(j, p_i)$ the first multi-writer register written by p_i in this execution and denote by $I(j, p_i)$ the longest prefix of this execution containing no `write` operations to multi-writer registers. (The sequence $I(j, p_i)$ may contain no steps at all.)

LEMMA 4.5. *Consider any component A_j. Suppose that $2s < n - d$ and all processes $p_i \notin \mathcal{D}$* `read` *from at most s different single-writer registers during $I(j, p_i)$. Then, for distinct processes $p_i, p_{i'} \notin \mathcal{D}$, $R(j, p_i) = R(j, p_{i'})$.*

PROOF. For each process $p_i \notin \mathcal{D}$, let $\mathcal{S}_i$ denote the set of processes whose single-writer registers are `read` by p_i during $I(j, p_i)$.

First suppose that $p_{i'} \notin \mathcal{S}_i$. Consider the execution $I(j, p_{i'}) \cdot I(j, p_i)$ starting from C_0 and let C be the resulting configuration. This execution is legal since $p_{i'}$ performs no `writes` to multi-writer registers during $I(j, p_{i'})$ and p_i does not `read` from $p_{i'}$'s single-writer register during $I(j, p_i)$. Note that $\{p_i, p_{i'}\}$ covers $\{R(j, p_i), R(j, p_{i'})\}$ and $CPU(C, \{p_i, p_{i'}\}) = \{A_j\}$. If $R(j, p_i) \neq R(j, p_{i'})$, then C is

2-fatal. But this contradicts Lemma 4.3. Hence $R(j,p_i) = R(j,p_{i'})$. Similarly, $p_i \notin \mathcal{S}_{i'}$ implies that $R(j,p_i) = R(j,p_{i'})$.

Now suppose that $p_{i'} \in \mathcal{S}_i$ and $p_i \in \mathcal{S}_{i'}$. Since $|\mathcal{S}_i|+|\mathcal{S}_{i'}| \leq 2s < n-d$, there exists a process $p_k \notin \mathcal{D}$ such that $p_k \notin \mathcal{S}_i \cup \mathcal{S}_{i'}$. Then, from the previous paragraph, $R(j,p_i) = R(j,p_k)$ and $R(j,p_{i'}) = R(j,p_k)$. Hence $R(j,p_i) = R(j,p_{i'})$. □

For the remaining lemmas in this section, assume $2s < n-d$ and all processes $p_i \notin \mathcal{D}$ `read` from at most s different single-writer registers during $I(j,p_i)$, for all j. Then, for each snapshot component A_j, we can define R_j to be the register $R(j,p_i)$ covered by each process $p_i \notin \mathcal{D}$ immediately after it executes $I(j,p_i)$ starting from C_0. That is, R_j is the first multi-writer register written by `UPDATES` to component A_j performed by processes not in $\mathcal{D}$.

LEMMA 4.6. *Let α be an execution starting from C_0 such that some process $p \notin \mathcal{D}$ takes no steps in α and p's single-writer register is never read during α. Then, for each j, $1 \leq j \leq m$,* `UPDATE` *operations to component A_j in α never* `write` *to any multi-writer register other than R_j.*

PROOF. Suppose there is a process p_i that performs a `write` operation to a multi-writer register R other than R_j during the execution of an `UPDATE` to component A_j. Let α' denote the prefix of α up to the point that p_i is ready to perform this write to register R. Consider the execution $I(j,p) \cdot \alpha'$ and let C' be the resulting configuration. This execution is legal since p performs no `write` operations to multi-writer registers during $I(j,p)$ and no process ever reads p's single-writer register in α. Note that $\{p_i, p\}$ covers $\{R, R_j\}$ in C' and $CPU(C', \{p_i, p\}) = \{A_j\}$, so C' is 2-fatal, contradicting Lemma 4.3. □

The next result shows that processes which perform `UPDATE` operations to different snapshot components must `write` to different registers. Otherwise, evidence of an `UPDATE` operation to one component can be hidden by an `UPDATE` to another component and cause a `SCAN` to return an incorrect value.

LEMMA 4.7. *Suppose that every solo execution of a* `SCAN` `reads` *from fewer than $n/2$ different single-writer registers. Suppose that $2s' \leq n-2d-2$ and every process $p_i \notin \mathcal{D}$* `reads` *from at most s' different single-writer registers during its solo execution of* `UPDATE`(j,p_i) *starting from the initial configuration C_0. Then $R_{j'} \neq R_j$ for $j' \neq j$.*

PROOF. To obtain a contradiction, suppose $R_{j'} = R_j$ for some $j' \neq j$. For $p_i \notin \mathcal{D}$, let $\mathcal{S}'_i$ denote the set of at most s' processes whose single-writer registers are `read` by p_i during its solo execution of `UPDATE`(j,p_i) starting from the initial configuration C_0. By a simple averaging argument, there exists a process $p_{i'} \notin \mathcal{D}$ that occurs in $\mathcal{S}'_i$ for at most s' different processes $p_i \notin \mathcal{D}$. Let $\mathcal{K}$ denote the set of processes $p_i \notin \mathcal{D} \cup \{p_{i'}\}$ such that $p_{i'} \notin \mathcal{S}'_i$. Then $|\mathcal{K}| \geq n-d-1-s' \geq n/2$, since $s' \leq n/2-d-1$.

Let C be the configuration that results when $I(j',p_{i'})$ is performed starting from C_0 and let C' be the configuration obtained from C when $p_{i'}$ writes to R_j. Let $\mathcal{D}'$ denote the set of processes whose single-writer registers are `read` by t during its solo execution of a `SCAN` starting from C' and let $p_i \in \mathcal{K} - \mathcal{D}'$. Such a process exists, since $|\mathcal{D}'| < n/2$.

Since $t \in \mathcal{D}$, $p_{i'} \notin \mathcal{D}$ and $p_i \notin \mathcal{D} \cup \{p_{i'}\}$, the processes $p_i, p_{i'}$ and t are distinct. Process $p_{i'}$ covers R_j in C, all processes except $p_{i'}$ are inactive at C, $A_j \neq A_{j'}$, and $p_i \notin \{p_{i'}\} \cup \{t\} \cup \mathcal{D}'$. Lemma 4.1 implies that the solo execution U of `UPDATE`(j,p_i) by p_i starting from C must contain a `write` to a multi-writer register other than R_j.

Now $p_i \in \mathcal{K}$ implies that the solo execution U_0 of `UPDATE`(j,p_i) by p_i starting from C_0 does not `read` $p_{i'}$'s single-writer register. Furthermore $p_{i'}$ performs no `writes` to multi-writer registers during $I(j',p_{i'})$. Thus p_i performs exactly the same operations in U_0 and U. In particular, in U_0, process p_i `writes` to a multi-writer register other than R_j. This contradicts Lemma 4.6 with α consisting of the update U_0; $p_{i'}$ takes no steps during U_0, and its single-writer register is never read. □

This lemma implies the following lower bound on space.

THEOREM 4.8. *Any n-process implementation of an m-component (multi-writer) snapshot object from multi-writer registers and single-writer registers, where $1 < m \leq 3n/4$, requires at least m multi-writer registers, if fewer than $n/4-1$ different single-writer registers are* `read` *during each solo* `SCAN` *and* `UPDATE` *operation.*

PROOF. Consider any n-process implementation of an m-component (multi-writer) snapshot object, where $1 < m \leq 3n/4$ and assume that fewer than $n/4-1$ different single-writer registers are `read` during each solo `SCAN` and `UPDATE` operation.

We now check that all the conditions needed for the preceding lemma hold. Any solo execution of a `SCAN` operation reads from fewer than $n/4-1 = n-3n/4-1 \leq n-m-1$ different single-writer registers. Also, note that $n/4-1 \leq n/2$.

Consider the solo execution of `SCAN` by some process t, starting from the initial configuration C_0 and let $\mathcal{D}$ be the set containing t and those processes from whose single-writer registers t `reads`. Let $d = |\mathcal{D}|$. Then $d < n/4$. Each process $p_i \notin \mathcal{D}$ `reads` from fewer than $n/4-1 = n/2-n/4-1 < n/2-d-1 \leq (n-d)/2$ different single-write registers during its solo execution of `UPDATE`(j,p_i) starting from C_0.

Then Lemma 4.7 implies that this implementation uses at least m multi-writer registers. □

5. THE TIME LOWER BOUNDS

In this section we shall construct an execution in which one process t must take many steps to simulate a `SCAN` operation. The proof is quite lengthy, so we first give an outline of the key ideas.

Our construction is inductive, creating executions $\alpha_0, \alpha_1, \ldots$ where t takes more and more steps during its troublesome `SCAN`. The key part of the induction step is to show that t must take at least $m-2$ more steps after α_i before halting. We assume that it can halt early to derive a contradiction. In α_i, we have $2i$ processes $p_1, \ldots, p_i, p'_1, \ldots, p'_i$, called *visible* processes, that each begin to perform one `UPDATE`. The visible processes perform steps at the beginning of α_i until they are each covering a multi-writer register. The process t then begins its troublesome `SCAN`. Periodically, one of the visible processes performs a single `write` step on a multi-writer register immediately before t `reads` that register.

In constructing α_i, we ensure that there are some processes, called *hidden* processes, which do not take steps in α_i and whose single-writer registers are not read during α_i. Furthermore, it is possible to insert into α_i `UPDATES` by some

hidden processes in such a way that all evidence of those UPDATES is obliterated by the writes of visible processes before t can read any of it. We then establish the order in which all UPDATES must be linearized in the augmented execution, and it follows from this linearization that t must return at least one value written by a hidden process. But this is impossible since t has no way of getting any information about the UPDATES performed by the hidden processes.

We now present the construction in more detail. Suppose every execution of a SCAN or UPDATE operation reads at most κ different single-writer registers. Assume that κ is chosen to be small enough to ensure that the conditions needed for Lemmas 4.3 to 4.6 are satisfied. (We shall show later how to choose the value of κ for the two particular situations we are interested in.) Assume that the snapshot object initially contains 0 in every component. In the following construction, various UPDATES will be performed. Assume the values used as the argument of all UPDATES are distinct, non-zero values.

For $0 \leq i \leq \left\lfloor \frac{n-5}{11\kappa+2} \right\rfloor$ we shall inductively construct a set of processes Y_i, and an execution $\alpha_i = I(x'_i, p'_i) \cdot I(x_i, p_i) \cdots I(x'_1, p'_1) \cdot I(x_1, p_1) \cdot L_1 \cdot w_1 \cdot r_1 \cdot w'_1 \cdot L_2 \cdot w_2 \cdot r_2 \cdot w'_2 \cdots L_i \cdot w_i \cdot r_i \cdot w'_i$ starting from the initial configuration C_0, where for $1 \leq j \leq i$, w_j is a write by p_j to register R_{x_j}, w'_j is a write by p'_j to register $R_{x'_j}$, r_j is a read by t of register R_{x_j}, and L_j is a sequence of steps performed by t. These will be constructed to satisfy the following claims:

(1) for $1 \leq j \leq i$, L_j contains reads from at least $m-3$ different multi-writer registers,

(2) for $1 \leq j \leq i$, t does not read R_{x_j} or $R_{x'_j}$ during L_j,

(3) $Y_i \subseteq Y_0$ and $|Y_i| \geq n - (3i+1)\kappa - 2i - 1$,

(4) for all $p \in Y_i$, p takes no steps in α_i and p's single-writer register is never read in α_i,

(5) for $1 \leq j \leq i$, $x'_j \neq x_j$,

(6) for $1 < j \leq i$, $x'_j \neq x'_{j-1}$,

(7) all of t's operations in α_i are part of a single SCAN operation, and

(8) if t continues to run by itself from the final configuration Q_i of α_i, it will read at least $(m-2)$ distinct multi-writer registers before completing its SCAN.

Base Case ($i = 0$): Let α_0 be the null execution in which no process takes any steps. Let $Y_0 = \{p$: the solo SCAN by t, starting from C_0, never reads p's single-writer register$\} - \{t\}$. Claims (1), (2), (4), (5), (6) and (7) are trivial. Claim (3) follows from the hypothesis that a SCAN operation will read at most κ different single-writer registers.

Let α be the execution in which t performs a solo SCAN, starting from C_0. We now show that something stronger than claim (8) holds for the base case: t must read every multi-writer register in α. Suppose there is some multi-writer register R_j that is not read. Let $p \in Y_0$. Let $U(j,p)$ be the execution of an UPDATE performed on A_j by p, starting from C_0. Lemma 4.6 implies that $U(j,p)$ contains write operations only to register R_j and p's single-writer register. Since t does not read register R_j or p's single-writer register in α, it follows that $\alpha' = U(j,p) \cdot \alpha$ is a legal execution from C_0 and α, α' are indistinguishable to t. Hence, t returns the same vector of values in both executions. In α', the SCAN operation by t starts after the UPDATE by p has been completed. Therefore, t must return p's value in α'. Clearly, t cannot return p's value in α, since the UPDATE by p does not exist in α. This is a contradiction.

Induction Step: Suppose $1 \leq i \leq \left\lfloor \frac{n-5}{11\kappa+2} \right\rfloor$ and assume that we have defined α_{i-1} and Y_{i-1} so that the claims are satisfied. We now construct α_i and Y_i and show that they, too, satisfy the claims.

By part (8) of the induction hypothesis, in a solo execution by t from the final configuration Q_{i-1} of α_{i-1}, process t will eventually read $m-2$ distinct multi-writer registers. Let L_i be the sequence of t's steps, starting from Q_{i-1} until t is poised to read the $(m-2)$nd distinct multi-writer register for the first time. Let x_i be the index of the multi-writer register that t is poised to read after executing L_i. Let $x'_i \notin \{x'_{i-1}, x_i\}$ be the index of some multi-writer register that is not read in L_i. Since there are three multi-writer registers that are not read in L_i, there exists such an x'_i.

Using part (3) of the induction hypothesis, and the fact that $i < \frac{n-4}{11\kappa+2}$, we have $|Y_{i-1}| \geq n - (3(i-1)+1)\kappa - 2(i-1) - 1 = n - (11\kappa+2)i + 8\kappa i + 2\kappa + 1 > n - (11\kappa+2)\frac{n-4}{11\kappa+2} + 8\kappa i + 2\kappa + 1 = 8\kappa i + 2\kappa + 5 \geq (5i+2)\kappa + 3\kappa + 5$.

Let p_i be a process in Y_{i-1} whose single-writer register is not read during L_i. Let p'_i be a different process in Y_{i-1} whose single-writer register is not read during $I(x_i, p_i)$ or L_i. These processes exist because at most κ single-writer registers are read during each of $I(x_i, p_i)$ and L_i, and $|Y_{i-1}| \geq 2\kappa + 3$. Let Y_i be the set of processes in $Y_{i-1} - \{p_i, p'_i\}$ whose single-writer registers are not read by any of the steps in $I(x_i, p_i), I(x'_i, p'_i)$ and L_i. The execution α_i is then defined to be $I(x'_i, p'_i) \cdot I(x_i, p_i) \cdot \alpha_{i-1} \cdot L_i \cdot w_i \cdot r_i \cdot w'_i$, where r_i is a step by t that reads register R_{x_i}.

We first show that α_i is a legal execution. Processes in Y_{i-1} take no steps in α_{i-1} and their single-writer registers are not read in α_{i-1}, by part (4) of the induction hypothesis. Process $p_i \in Y_{i-1}$ was chosen so that its single-writer register is not read during L_i. Thus p_i takes no steps in $\alpha_{i-1} \cdot L_i$ and its single-writer register is not read during $\alpha_{i-1} \cdot L_i$. Since p_i does not write to any multi-writer register in $I(x_i, p_i)$, it follows that $I(x_i, p_i) \cdot \alpha_{i-1} \cdot L_i$ is a legal execution. Similarly, p'_i was chosen so that p'_i neither takes steps nor has its single-writer register read during the execution $I(x_i, p_i) \cdot \alpha_{i-1} \cdot L_i$, so the execution $I(x'_i, p'_i) \cdot I(x_i, p_i) \cdot \alpha_{i-1} \cdot L_i$ is also legal. The final three steps of α_i, $w_i \cdot r_i \cdot w'_i$, are simply the steps that the three processes p_i, t and p'_i are then poised to do next.

We now show that all eight claims are satisfied.

Claim (1) follows immediately from the definition of L_i and claim (2) follows immediately from the definitions of x_i and x'_i.

By the definition of Y_i and part (3) of the induction hypothesis, we have $Y_i \subseteq Y_{i-1} \subseteq Y_0$. Since $I(x'_i, p'_i), I(x_i, p_i)$ and L_i each read at most κ different single-writer registers, we have $|Y_i| \geq |Y_{i-1}| - 3\kappa - 2$, which is at least $n - (3(i-1)+1)\kappa - 2(i-1) - 1 - 3\kappa - 2 = n - (3i+1)\kappa - 2i - 1$ by part (3) of the induction hypothesis. So claim (3) is satisfied.

No process that takes steps in α_{i-1} is in Y_i, since $Y_i \subseteq Y_{i-1}$. The only processes that take steps in α_i but do not take steps in α_{i-1} are p_i and p'_i, but they are not in Y_i. No process whose single-writer register is read in α_{i-1} is in Y_i, since $Y_i \subseteq Y_{i-1}$. All processes whose single-writer registers are read in α_i but not in α_{i-1} are removed when we form Y_i from Y_{i-1}. So claim (4) holds.

Claims (5) and (6) follow directly from the definition of x'_i, and claim (7) is trivial.

Finally, we prove claim (8). To derive a contradiction, assume that in a solo execution of t from Q_i, t completes the troublesome `SCAN` after reading less than $(m-2)$ distinct multi-writer registers. Denote by L the sequence of t's steps when it starts from Q_i and runs to completion. Let $\alpha = \alpha_i \cdot L$. In Figure 1, α consists of all steps except those in the last three columns.

Our goal is to construct an execution δ that t cannot distinguish from α and show that t must return different results in the two executions. We will construct δ by adding operations to α in several stages.

We first check that Y_i contains enough processes for the construction that follows. By the definition of Y_i, $|Y_i| \geq |Y_{i-1}| - 3\kappa - 2$. We have shown that $|Y_{i-1}| > (5i+2)\kappa + 3\kappa + 5$, so $|Y_i| > (5i+2)\kappa + 3$. As we add operations to α, we shall have to apply Lemma 4.6 to the resulting execution. To do so, we need to guarantee that there is some process $p \notin \mathcal{D}$ that takes no steps in the execution. Consider any execution obtained from α by adding at most $5i+1$ `SCAN` and `UPDATE` operations performed by at most three different processes from Y_i. There are at most $(5i+2)\kappa$ processes in Y_i whose single-writer registers are read in the execution (including those read in L). Thus, there exists a process $p \in Y_i$ that takes no steps and whose single-writer register is never read during the execution, as required by Lemma 4.6.

Let q' be an element of Y_i whose single-writer register is not read during L. There exists such a q', since $|Y_i| > \kappa$ and at most κ single-writer registers are read during L. Let $x' \neq x'_i$ be one of the multi-writer registers not read during L. (Notice that such a register exists since there are more than two multi-writer registers not read during L by our assumption.) We create a new execution β by inserting $i+1$ `UPDATE` operations and $2i$ `SCAN` operations by q' into α as follows. An `UPDATE` $U(x'_1, q'_1)$ to component x'_1 is inserted before L_1. For $1 < j \leq i$, an `UPDATE` $U(x'_j, q'_j)$ is inserted before w'_{j-1}. An `UPDATE` $U(x', q'_{i+1})$ to component x' is inserted before w'_i. For $1 \leq j \leq i$, `SCAN` operations S_j and S'_j are inserted after r_j and w'_j, respectively. In Figure 1, β consists of all steps except those in the last two columns.

We check that β is a legal execution. Clearly, α cannot be distinguished from β by any of the processes $p_1, \ldots, p_i, p'_1, \ldots, p'_i$, since each of them takes only one step in β after q' starts running, and that step is a `write` operation. It remains to verify that none of the values written into registers by q' in β are ever read by any of t's operations. By the definition of q', t never `reads` the single-writer register belonging to q'. By Lemma 4.4, the `SCAN` operations performed by q' do not `write` to any multi-writer registers. Let $1 \leq j \leq i$. By Lemma 4.6, the only multi-writer register written by $U(x'_j, q'_j)$ is $R_{x'_j}$. That register is not read by t during L_j, by claim (2), or in the step r_j, by claim (5). The register $R_{x'_j}$ is overwritten by w'_j, so t never `reads` any value written there by $U(x'_j, q'_j)$. Again by Lemma 4.6, the only multi-writer register written by the `UPDATE` operation $U(x', q'_{i+1})$ is $R_{x'}$, but x' was chosen so that this register is not read by t during L. Thus, β is a legal execution.

Let q be a process in $Y_i - \{q'\}$ whose single-writer register is not read during β. Such a process exists because the single-writer registers of at most $(3i+1)\kappa$ processes in Y_i are read by q' in β, and at most κ single-writer registers of processes in Y_i are read by t in β (during L), but $|Y_i| > (3i+2)\kappa + 1$. We construct another execution γ by inserting i `UPDATE` operations by q into β: for $1 \leq j \leq i$, we insert an `UPDATE` $U(x_i, q_i)$ just before L_i. The execution γ consists of all steps shown in Figure 1 except the last column.

We now check that γ is a legal execution. In β, no process `reads` q's single-writer register. Let $1 \leq j \leq i$. By Lemma 4.6, the only multi-writer register written by the newly inserted `UPDATE` operation $U(x_j, q_j)$ is R_{x_j}. This register is overwritten by w_j before any other process can `read` it, since L_j does not `read` R_{x_j}, by claim (2). Thus, none of the processes that take steps in β will notice the insertion of the `UPDATE` operations by q.

Now, let q'' be a process in $Y_i - \{q, q'\}$ whose single-writer register is not read during γ. Such a process exists because the registers of at most $(4i+2)\kappa$ processes in Y_i are read during γ, and $|Y_i| > (4i+2)\kappa + 2$. We create a new execution δ by inserting i `UPDATE` operations by q'' into γ: for $1 \leq j \leq i$, we insert an `UPDATE` $U(x'_i, q''_i)$ just before w'_i. The execution δ consists of all steps shown in Figure 1.

The argument that δ is a legal execution is similar to the argument for γ: for $1 \leq j \leq i$, the `UPDATE` operation $U(x'_j, q''_j)$ `writes` only to the single-writer register of q'' (which is never read in γ) and the multi-writer register $R_{x'_j}$, which is immediately overwritten by w'_j.

The execution δ has been constructed so that we can prove some very precise facts about the order in which its operations are linearized.

In δ, process t never sees any evidence of the existence of q, q' and q'', since t's sequence of steps is identical to its sequence in α, where the processes q, q' and q'' take no steps at all. For this reason, we call processes q, q' and q'' *hidden* processes, and the processes $p_1, \ldots, p_i, p'_1, \ldots, p'_i$ *visible* processes. The operations of the hidden processes must be linearized in the order in which they are performed, since none of them overlap with one another. Thus they are linearized in the following order:

$$
\begin{array}{l}
U(x'_1, q'_1), \\
U(x_1, q_1), S_1, U(x'_2, q'_2), U(x'_1, q''_1), S'_1, \\
U(x_2, q_2), S_2, U(x'_3, q'_3), U(x'_2, q''_2), S'_2, \\
\vdots \\
U(x_{i-1}, q_{i-1}), S_{i-1}, U(x'_i, q'_i), U(x'_{i-1}, q''_{i-1}), S'_{i-1}, \\
U(x_i, q_i), S_i, U(x', q'_{i+1}), U(x'_i, q''_i), S'_i.
\end{array}
\tag{1}
$$

Now we determine how the `UPDATE` operations of the visible processes fit into the linearization order. Let $1 \leq j \leq i$. In δ, the value returned for component A_{x_j} by `SCAN` S_j must either be q_j or the value written by some `UPDATE` operation that is linearized between $U(x_j, q_j)$ and S_j. However, S_j cannot return the value q_j because S_j must return the same output vector in the two executions β and δ (which are indistinguishable to q'), and $U(x_j, q_j)$ does not even exist in β. Thus, at least one of the `UPDATE` operations to A_{x_j} by a visible process is linearized between $U(x_j, q_j)$ and S_j. An identical argument shows that at least one of the `UPDATE` operations to $A_{x'_j}$ by a visible process must be linearized between $U(x'_j, q''_j)$ and S'_j. Since there are $2i$ `UPDATE` operations by visible processes, and we have shown that at least one must be linearized in each of $2i$ different positions in the linearization ordering, it follows that *exactly* one of them is linearized in each position. Thus we have the following linearization for all the operations of δ except t's `SCAN` (where each $U(x_j, v_j)$ and $U(x'_j, v'_j)$ represents one of the `UPDATES`

p_i	$\cdots$	p_1	p'_i	$\cdots$	p'_1	t	q'	q	q''
			$I(x'_i, p'_i)$						
$I(x_i, p_i)$				$\ddots$					
	$\ddots$				$I(x'_1, p'_1)$				
		$I(x_1, p_1)$					$U(x'_1, q'_1)$		
						L_1		$U(x_1, q_1)$	
		w_1				r_1	S_1		
							$U(x'_2, q'_2)$		$U(x'_1, q''_1)$
					w'_1		S'_1		
						L_2		$U(x_2, q_2)$	
	w_2					r_2	S_2		
							$U(x'_3, q'_3)$		$U(x'_2, q''_2)$
				w'_2			S'_2		
					$\vdots$				
						L_{i-1}		$U(x_{i-1}, q_{i-1})$	
	w_{i-1}					r_{i-1}	S_{i-1}		
							$U(x'_i, q'_i)$		$U(x'_{i-1}, q''_{i-1})$
				w'_{i-1}			S'_{i-1}		
						L_i		$U(x_i, q_i)$	
w_i						r_i	S_i		
							$U(x', q'_{i+1})$		$U(x'_i, q''_i)$
			w'_i				S'_i		
						L			

Figure 1: The execution δ. Each column represents one process whose name appears at the top of the column. Time is on the vertical axis, running from top to bottom.

by a visible process to component x_j and x'_j, respectively):

$U(x'_1, q'_1)$,
$U(x_1, q_1), U(x_1, v_1), S_1, U(x'_2, q'_2), U(x'_1, q''_1), U(x'_1, v'_1), S'_1,$
$U(x_2, q_2), U(x_2, v_2), S_2, U(x'_3, q'_3), U(x'_2, q''_2), U(x'_2, v'_2), S'_2,$
$\vdots$
$U(x_{i-1}, q_{i-1}), U(x_{i-1}, v_{i-1}), S_{i-1}, U(x'_i, q'_i), U(x'_{i-1}, q''_{i-1}),$
$U(x'_{i-1}, v'_{i-1}), S'_{i-1},$
$U(x_i, q_i), U(x_i, v_i), S_i, U(x', q''_i), U(x'_i, v'_i), S'_i.$

We now use this linearization to show that, at any time after the first `UPDATE`, the snapshot object contains at least one value written by a hidden process. It suffices to show that there is at least one such hidden value in the snapshot object after each of the `UPDATE` operations by visible processes.

After $U(x_1, v_1)$, there is still a hidden value q'_1 in component $A_{x'_1}$, since $x'_1 \neq x_1$. Let $2 \leq j \leq i$. Since $x'_j \neq x_j$ and $x'_j \neq x'_{j-1}$, the hidden value q'_j is still in $A_{x'_j}$ after $U(x_j, v_j)$.

Let $1 \leq j \leq i-1$. Just after $U(x'_j, v'_j)$, there is still a hidden value q'_{j+1} in component $A_{x'_{j+1}}$, since $x'_{j+1} \neq x'_j$. Furthermore, just after the final `UPDATE` $U(x'_i, v'_i)$, there is a hidden value q''_i in component $A_{x'}$, since x' was chosen to be different from x'_i.

Thus, no matter where t's `SCAN` in δ is linearized, it must return a value of a hidden process in at least one component of its output vector. The process t must return the same output vector in the execution α, since t takes the same sequence of steps in α and δ. However, t cannot return a hidden value in the execution α, since none of the `UPDATE` operations by hidden processes exist in that execution. This contradiction completes the proof of claim (8) and the proof that our construction is correct.

We can now use the construction to prove the time lower bounds.

THEOREM 5.1. *If $2 < m < n-1$ and $n > 6$, any n-process wait-free implementation of an m-component multi-writer snapshot object using m multi-writer registers has time complexity $\Omega(mn)$.*

PROOF. Since there are no single-writer registers available, the conditions needed to apply Lemmas 4.3 to 4.6 are obviously satisfied. This means that the construction works with $\kappa = 0$, and it can be carried out for $0 \leq i \leq \lfloor \frac{n-5}{2} \rfloor$. In the constructed execution, process t performs at least $\lfloor \frac{n-5}{2} \rfloor \cdot (m-2) \in \Omega(mn)$ `read` steps. □

THEOREM 5.2. *If $2 < m \leq 3n/4$ and $n > 6$, any n-process wait-free implementation of an m-component multi-writer snapshot object using m multi-writer registers and any number of single-writer registers has time complexity $\Omega(\sqrt{mn})$.*

PROOF. Assume there exists an implementation for which the worst-case time complexity is at most $\sqrt{mn}/4 - 1$. We now check that the conditions needed to apply Lemmas 4.3 to 4.6 are satisfied. Since $m \leq 3n/4 < n$, every execution of a `SCAN` or `UPDATE` operation `reads` at most $\kappa \leq \sqrt{mn}/4 - 1 < n/4 - 1 \leq n - m - 1$. Also, $d = |\mathcal{D}| < n/4$, so $\frac{n-d}{2} > 3n/8 > n/4 - 1$, which implies that every process p_i `reads` from fewer than $\frac{n-d}{2}$ single-writer registers during $I(j, p_i)$. This means we can carry out the construction described above for $0 \leq i \leq \left\lfloor \frac{n-5}{11(\sqrt{mn}/4-1)+2} \right\rfloor$. In the constructed execution, process t does at least $\left\lfloor \frac{n-5}{11(\sqrt{mn}/4-1)+2} \right\rfloor \cdot (m-2)$ `read` steps on multi-writer registers. Thus, the complexity of the implementation is at least $\min(\sqrt{mn}/4 - 1, \left\lfloor \frac{n-5}{11(\sqrt{mn}/4-1)+2} \right\rfloor \cdot (m-2)) \in \Omega(\sqrt{mn})$. □

6. DISCUSSION

For $m = n - 2$, our time complexity lower bound for any space-optimal implementation of an m-component snapshot object shared by n processes is $\Omega(n^2)$. This is particularly interesting since there is an implementation for $m = n$, with $O(n \log n)$ time complexity using n single-writer registers (which are less powerful than multi-writer registers) [4, 12]. Thus, there is an essential difference in the kind of algorithms that can be obtained for space-optimal snapshot implementations when the number of components is at least the number of processes and when it is only slightly smaller.

Ultimately, we would like to obtain a time-space tradeoff for snapshot implementations. One approach is to extend our lower bound on time to implementations using $m + k$ multi-writer registers for small values of k. Another approach is to improve our lower bound using single-writer registers and m multi-writer registers, for example, to $\Omega((\log n)\sqrt{mn})$, so that it matches the known $O(n \log n)$ upper bound when $m = n$.

Our lower bounds are strong in that they assume no bound on the size of the registers used in the implementation. The matching $O(mn)$ time algorithm uses only m multi-writer registers, but they are also of unbounded size. It is an interesting open question whether there exists an algorithm that can achieve these bounds using bounded size registers. Alternatively, is there a lower bound on the size of the registers needed for a space-optimal implementation, as a function of m, n, and the number of different values that can be stored in components?

An object is historyless [16] if, for each of its operations, either the value of the object is never changed by the operation (for example, `read`) or the value of the object after the operation is applied does not depend on its value beforehand (for example `write`, `test&set`, and `swap`). Our lower bounds apply to implementations of multi-writer snapshot objects from historyless objects, demonstrating that any extra power these historyless objects provide does not reduce the number of shared objects required, nor the worst-case time complexity of space-optimal implementations. It would be interesting to extend our lower bounds to other types of objects.

Acknowledgements

This research was supported by the Natural Sciences and Engineering Research Council of Canada, Communications and Information Technology Ontario, and the Greek Ministry of Education's E.P.E.A.E.K. II programme for supporting studies in informatics. Part of this work was done while Panagiota Fatourou was visiting the University of Toronto and York University.

7. 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] Panagiota Fatourou, Faith Fich, and Eric Ruppert. Space-optimal multi-writer snapshot objects are slow. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, pages 13–20, 2002.

[16] 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.

[17] Faith Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*. To appear.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

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

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