Noname manuscript No.
(will be inserted by the editor)

Lower and Upper Bounds for Single-Scanner Snapshot Implementations

Panagiota Fatourou · Nikolaos D. Kallimanis

Received: date / Accepted: date

Abstract We present a collection of upper and lower bounds on the complexity of asynchronous, wait-free, linearizable, single-scanner snapshot implementations from read-write registers. We argue that at least m registers are needed to implement a single-scanner snapshot with m components and we prove that, in space-optimal implementations, SCANS execute $\Omega(m^2)$ steps. We present an algorithm that runs in $O(m^2)$ steps and uses $m + 1$ registers.

We also present three implementations (namely, T-Opt, RT and RT-Opt) that beat the $\Omega(m^2)$ lower bound by using more registers. Specifically, T-Opt has step complexity $O(1)$ for UPDATE and $O(m)$ for SCAN. This step complexity is optimal, but the number of registers that T-Opt uses is unbounded. We then present interesting recycling techniques to bound the number and the size of registers used, resulting in RT and RT-Opt. Specifically, RT-Opt, which has optimal step complexity, uses $O(mn)$ bounded-size registers, where n is the total number of processes.

Our implementations are the *first* with step complexities that are (linear or quadratic) functions only of m (and not of n). Moreover, T-Opt and RT-Opt are the *first* implementations with optimal step complexity.

Keywords snapshots · single-scanner · multi-writer · atomic objects · wait-freedom · linearizability · asynchronous · distributed algorithms · shared memory computing · step optimal algorithms

A preliminary version of this work appeared in the Proceedings of the 25th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), 2006, pages 228–237 and the 26th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), 2007, pages 33–42.

Panagiota Fatourou
FORTH-ICS & University of Crete, Department of Computer Science
E-mail: faturu@ics.forth.gr
Present address:
Foundation for Research and Technology - Hellas (FORTH)
Institute of Computer Science (ICS)
N. Plastira 100
Vassilika Vouton, GR-70013 Heraklion
Crete Island, Greece

Nikolaos D. Kallimanis
FORTH-ICS
E-mail: nkallima@ics.forth.gr
Present address:
Foundation for Research and Technology - Hellas (FORTH)
Institute of Computer Science (ICS)
N. Plastira 100
Vassilika Vouton, GR-70013 Heraklion
Crete Island, Greece

1 Introduction

A fundamental problem in asynchronous, shared-memory systems is to obtain an instantaneous view of a block of shared memory while processes may be concurrently updating its cells. Snapshots are shared objects that provide such consistent views. Specifically, a *snapshot object* consists of an array of m components and supports two operations, `UPDATE` that changes the value of a component, and `SCAN`, which returns an instantaneous view of all components. Snapshots can be used to record the state of a system as it is changing, so they facilitate the solution of problems that have to perform an action whenever the global state of the system satisfies some condition [25]. Snapshots have been extensively used for the design and verification of distributed algorithms, e.g., for the construction of concurrent timestamps [17], approximate agreement [7], check-pointing and restarting [25], randomized consensus [4], and the design of complex distributed data structures [5].

A *multi-writer* snapshot allows each process to `UPDATE` any component. It can be implemented from read-write registers [3, 12–14, 21, 22]. A *single-writer snapshot* [1, 2, 8, 19, 20] is a restricted version, where each component has only one process that can `UPDATE` it. A snapshot *implementation* from read-write registers uses the registers to store the state of the snapshot components and provides an algorithm, for each process, to execute `SCAN` and `UPDATE`. A snapshot implementation is evaluated in terms of its *space complexity*, which is expressed in terms of the number (and the size) of registers it uses, and its *step complexity*, which is the maximum number of steps taken by a process in every execution to perform a `SCAN` or an `UPDATE`. The advantages of snapshots can be exploited only if it is possible to implement them efficiently.

Ideally, we would like to be able to design multi-writer snapshot implementations, which have step complexity that is independent of n, the total number of processes. (Usually, n is much larger than the number m of snapshot components.) However, it has been proved [14] that, in any implementation of multi-writer snapshot objects from a fixed number of read-write registers, the step complexity of `SCAN` grows without bound as n increases. Current snapshot implementations [1, 2, 8, 19, 20, 3, 12–14] from read-write registers have step complexity at least linear in n.

In this paper, we show that the dependence of step complexity on n can be beaten, if we restrict attention to single-scanner snapshots [22, 24, 27]. A *single-scanner* snapshot is an interesting variant of a general snapshot object in which only one process, called the *scanner*, performs `SCAN` operations at any point in time.

Single-scanner snapshots have several applications and therefore studying their complexity is of interest. Many of the contemporary programming environments support garbage collection for reclaiming memory. In such environments, a process, known as a *garbage collector*, is periodically executed to reclaim the unused memory. Backup is another classical application of a single-scanner snapshot algorithm. In such systems, a single process is responsible for periodically taking snapshots of a system's critical data. As a last example, consider a debugging environment for parallel applications. In all these environments, it is important to take snapshots without interfering with the execution of the running application. The design of efficient wait-free single-scanner snapshot algorithms is therefore an interesting problem.

We study single-scanner, multi-writer snapshot implementations and present a collection of upper and lower bounds for their complexity. It turns out that single-scanner multi-writer snapshot implementations from read-write registers require at least m registers, even if the registers are of unbounded size. Jayanti, Tan, and Toueg [23] have presented a lower bound of $\Omega(n)$ on the step complexity of implementations of *perturbable* objects from read-write registers. They prove that *single-writer snapshots* are perturbable [23]. Their proof applies to the single-scanner case. A multi-writer snapshot trivially implements a single-writer snapshot for m processes. This implies a lower bound of $\Omega(m)$ on the step complexity of `SCAN` for single-scanner, multi-writer snapshots.

We present a lower bound of $\Omega(m^2)$ on the step complexity of `SCAN` for space-optimal single-scanner multi-writer snapshot implementations for $n > m$ processes. This lower bound holds even if each of the snapshot components can store only three different values. Additionally, we present a single-scanner multi-writer snapshot implementation, called Checkmarking, which has $O(m^2)$ step complexity and uses $m+1$ registers of unbounded size. Thus, Checkmarking uses just one more register than a space optimal implementation and its step complexity matches the lower bound we proved for such implementations to within a constant factor.

We also present the Time-efficient family of single scanner, multi-writer m-component snapshot implementations from read-write registers. It contains the first *step-optimal* implementations. These implementations have step complexity $O(m)$ for SCAN and $O(1)$ for UPDATE and use bounded-size registers. The first implementation, called T-Opt, is the simplest, but (in systems with no garbage collector) the number of registers it uses depends on the number of SCAN operations executed (that might be unbounded).

To improve space efficiency, we first present a relatively simple recycling technique that results in an implementation, called RT, that uses $O(mn)$ bounded-size registers, and has step complexity $O(1)$ for UPDATE and $O(n)$ for SCAN. Thus, RT sacrifices the step optimality of T-Opt for less space. We then introduce a more interesting recycling technique to get an implementation, called RT-Opt, that uses $O(mn)$ bounded-size read-write registers and achieves optimal step complexity, that is, step complexity $O(1)$ for UPDATE and $O(m)$ for SCAN. RT is a middle ground between T-Opt and RT-Opt; its design provides intuition for RT-Opt and simplifies its presentation. RT-Opt sacrifices space for better step complexity. We believe that it could be used to reduce the space complexity of other interesting distributed problems.

A practical snapshot implementation should ensure that the performance of UPDATE is within a small constant of that of a write. (It is usually not desirable to significantly increase the cost of updating shared memory.) The Time-efficient family ensures this property by having UPDATES perform a small number of accesses in shared memory.

We remark that T-Opt works even if processes do not have unique identifiers. Moreover, T-Opt and Checkmarking work even if the number of participating processes is unbounded. All our single-scanner implementations work even if several processes perform SCANS, although not simultaneously. T-Opt and RT do not require any changes. In order that RT-Opt works, some of the scanner's persistent (static) variables must be accessed by each process performing a SCAN, although these variables will never be accessed concurrently.

Table 1 summarizes known single-scanner snapshot implementations from registers.

The rest of the paper is organized as follows. In Section 2, we discuss related work. Our model is presented in Section 3. In Section 4, we present the $\Omega(m^2)$ lower bound on the step complexity of space-optimal, single-scanner multi-writer snapshots. Checkmarking is presented in Section 5, and T-Opt, RT and RT-Opt are presented in Sections 6, 7, and 8, respectively. A discussion and some open problems are provided in Section 9.

2 Related Work

Fatourou, Fich, and Ruppert have proved in [11, 13] that multi-scanner multi-writer m-component snapshot implementations from read-write registers require at least m registers. Moreover, they have presented a lower bound of $\Omega(mn)$ on the step complexity of SCANS for such implementations that are space optimal. Covering arguments [9] were used to prove these lower bounds; first, a number of structural properties for these implementations were presented and then, these properties were used to construct an execution where a *troublesome* SCAN takes many steps. We employ similar arguments in order to prove our lower bounds. It is not difficult to observe that the structural properties proved in [13] also hold for the case of single-scanner implementations. This leads to the observation that the space lower bound of $\Omega(m)$ proved in [13] also holds for the single-scanner case. However, to prove the lower bound on the step complexity, we had to cope with several complications. To prove the lower bound of $\Omega(mn)$, it is essential that, in the execution constructed in [13], a number of SCANS take place concurrently with the troublesome SCAN in order to prove that the troublesome SCAN needs to take more and more steps. In the single-scanner case, it is not possible to have more than one concurrent SCAN active at each point in time, so it is only the troublesome SCAN, which is allowed to be active during the execution. This makes our construction more difficult and delicate and differentiates it from that presented in [13].

Attiya, Ellen and Fatourou [6] proved a lower bound of $\Omega(m)$ on the step complexity of UPDATE for partitioned implementations of multi-scanner, multi-writer snapshots from base objects of any type. An implementation is *partitioned* if each base object can only be modified by processes performing UPDATES to one specific component. T-Opt is a partitioned implementation of single-scanner multi-writer snapshots and has step complexity $O(1)$ for UPDATE. So, the lower bound in [6] can be beaten if we restrict attention to the single-scanner case.

The first single-scanner, multi-writer snapshot implementations from read-write registers were presented by Kirousis, Spirakis and Tsigas [24]. Their first implementation uses an unbounded number of

Implementation	SW/MW	Regs Type	Regs Number	Regs Size	SCAN	UPDATE
Checkmarking, this paper	MW	MW r/w	$m+1$	unbounded	$O(m^2)$	$O(m^2)$
T-Opt, this paper	MW	MW r/w	∞	unbounded	$O(m)$	$O(1)$
RT, this paper	MW	MW r/w	$O(mn)$	$O(\log n)$	$O(n)$	$O(1)$
RT-Opt, this paper	MW	MW r/w	$O(mn)$	$O(\log n)$	$O(m)$	$O(1)$
Kirousis *et al.* [24]	MW	MW r/w	$O(mn)$	$O(mn \log n)$	$O(mn)$	$O(1)$
Riany *et al.* [27]	SW	SW r/w	$n+1$	unbounded	$O(n)$	$O(1)$
Jayanti [22]	SW	SW r/w	$O(n)$	$O(1)$	$O(n)$	$O(1)$
Jayanti [22]	MW	LL/SC & r/w	$O(m)$	$O(1)$	$O(m)$	$O(1)$

Table 1 Summary of known single-scanner snapshot implementations.

registers and has unbounded step complexity for SCAN. A register recycling technique, which leads to an implementation that uses $O(mn)$ bounded-size registers and has step complexity $O(mn)$ for SCAN and $O(1)$ for UPDATE, is also presented in [24]. As in the recycled implementation in [24], RT-Opt uses a two dimensional array of registers with $O(n)$ rows of m registers each. However, the recycling technique employed by RT-Opt is much simpler than that proposed in [24], since RT-Opt recycles rows of this array and not a single element of an appropriate row for each component, as done in the implementation in [24]. Remarkably, our implementations significantly improve upon the implementations in [24] in terms of their step complexity. This is accomplished by employing different techniques to achieve fast termination.

For single-writer snapshots, a simplified version of the implementation in [24] that uses $n+1$ single-writer registers of unbounded size and has step complexity $O(n)$ for SCAN and $O(1)$ for UPDATE is presented in [27]. Jayanti [22] has presented a simple, single-scanner, single-writer snapshot implementation from $O(n)$ bounded-size single-writer registers that has step complexity $O(n)$ for SCAN and $O(1)$ for UPDATE.

Recall that Table 1 summarizes known single-scanner snapshot implementations from registers. It is remarkable that the step complexity of all previously presented single-scanner snapshot implementations from read-write registers is a function of n. In [22], Jayanti has also presented a single-scanner snapshot implementation, which has step complexity $O(m)$ for SCAN and $O(1)$ for UPDATE. However, that implementation uses stronger base objects such as LL/SC registers.

Our implementations are the first asynchronous snapshot implementations, which have step complexity that is linear (or quadratic) in the number of snapshot components. Snapshot implementations that are partially synchronous are provided in [24], but the correctness of these implementations is heavily based on the timing assumptions. Moreover, these assumptions simplify their design significantly. A snapshot algorithm for a system in which the number of processes may be infinite is presented in [16].

3 Model

We consider a system in which a set P of n processes run concurrently and asynchronously. The processes communicate by accessing shared objects and may fail by crashing. If a process crashes, it takes no more steps.

A *read-write register* R is a base object that stores a value from a set and supports the atomic *primitives* write(R, v) that changes the value of R to v and returns an acknowledgment ack, and read(R), which returns the value of R without any change. All processes may perform write to a *multi-writer* register, whereas only one process may perform write to a *single-writer* register.

A (multi-writer) *snapshot* object A consists of m components $A_1, \ldots, A_m$, each capable of storing a value at a time; processes can perform two kinds of operations on the object: UPDATE(i, v), which updates component A_i with value v and returns ack, and SCAN, which returns a vector of m values, one for each component of the snapshot object.

We study implementations of multi-writer snapshot objects from read-write registers. An *implementation* uses the registers to simulate the state of the snapshot components and provides an algorithm, for each process, to implement each simulated operation (i.e., SCAN and UPDATE).

A *configuration* C is a vector consisting of the states of the processes and the states of the registers used by the implementation. A configuration describes the system at some point in time. In an *initial* configuration, all processes are in initial states and all registers contain initial values. We say that a process takes a *step* whenever it performs a single access (`read` or `write`) to some register. A step might as well contain the execution of local computation by the process that takes the step; this computation may cause a change to the state of the process. Each step is executed atomically.

An *execution* is an alternating sequence of configurations and steps starting with a configuration. An execution α is *legal*, starting from a configuration C, if the sequence of steps performed by each process follows the algorithm for that process (starting from its state in C) and for each register, the responses to the operations performed on the register are in accordance with its specification and the value stored in the register at configuration C. The *schedule* $\pi(\alpha)$ of an execution α is the subsequence of α consisting of the steps of α. A schedule π is *legal* from some configuration C if there is a legal execution α starting from C for which $\pi = \pi(\alpha)$.

A configuration C is *reachable* if there is a legal execution α starting from an initial configuration that results in C. A process is *poised* to perform a primitive on a register in a configuration C if it performs that primitive on the register when it is next allocated a step. A process *covers* a register R in a configuration C if it is poised to perform a `write` primitive to R at C. A set of processes P' *covers* a set of registers R' if $|P'| = |R'|$, each process in P' covers a register in R' and each register in R' is covered by a process in P'. Two executions α and α' are *indistinguishable* to some set of processes P', denoted by $\alpha \approx_{P'} \alpha'$, if the sequence of steps performed by each process p in P' and the responses p received are the same in α and α'. In a *solo* execution, all steps are performed by the same process.

Let *op* be some `SCAN` or `UPDATE` operation in α. The *execution interval* of *op* is the subsequence of α that starts with the configuration that precedes the invocation (i.e., the first step) of *op* and ends when *op* responds. If the response of *op* precedes the invocation of some other `SCAN` (or `UPDATE`) op', then *op precedes* op'. We say that *op* is *pending* at some configuration C if the process p executing *op*, has performed at least one step of the algorithm of *op* at C but has not yet completed executing *op*. If a process p has a pending operation at C, p is called *active*; otherwise, p is *inactive* at C. In a *sequential* execution, only one process is active at each configuration of the execution.

A snapshot implementation is *single-scanner* if in each execution produced by the implementation, there is only one process, called the *scanner*, that performs `SCANS` at each point in time. We remark that the implementations we present work correctly even if several scanners perform `SCANS` provided that the execution intervals of `SCANS` do not overlap with one another. Our lower bounds are also true in this case.

We study implementations that are *linearizable* [18]. An execution α starting from an initial configuration is *linearizable* if for each completed `SCAN` or `UPDATE` operation *op* in α (and for some of those that are not completed), we can choose a point in its execution interval, called *linearization point*, such that the response returned by *op* in α is the same as the response *op* would return if these operations were executed sequentially in the order determined by their linearization points. The sequence of these linearization points is called a *linearization* of the execution. An implementation is *linearizable* if all its executions are linearizable. If L is a linearization of α, we say that the response of *op* is *consistent* in α with respect to L (for simplicity, we sometimes omit reference to α and L whenever they are clear from the context).

Additionally, our implementations are *wait-free*. *Wait-freedom* requires that every non-faulty process should complete the execution of any `SCAN` or `UPDATE` it initiates within a finite number of steps, independently of whether other processes crash or run at arbitrary speeds.

The *step complexity* of `SCAN` (`UPDATE`) for a snapshot implementation I is the maximum number of steps executed by a process to perform `SCAN` (`UPDATE`, respectively) in any execution produced by I. The *step complexity* of I is the maximum between the step complexity of `SCAN` and the step complexity of `UPDATE` for I. The *space complexity* of I is determined by the maximum number of registers (and their sizes) used in any execution produced by I.

4 Lower Bound

In this section, we present lower bounds for single-scanner snapshot implementations. First, we argue that in a single-scanner implementation of an m-component multi-writer snapshot object for $n > m + 1$ processes using only m multi-writer registers, processes must access the registers in a very constrained way.

In *multi-scanner* (multi-writer) m-component snapshot implementations that use only m registers, Fatourou, Fich and Ruppert have proved in [12] that (1) `SCANS` do not write (Lemma 1), (2) unless every process has taken steps, each `UPDATE` operation writes to only one register (Lemma 2), and (3) processes that perform `UPDATE` operations to different snapshot components must write to different registers (Lemma 3). It is easy to verify that the proofs of these lemmas for the single-scanner case are similar to those presented in [12] for multi-scanner snapshots. For the sake of completeness, the proofs of Lemmas 1-3 (as well as of others that they depend on) are provided in the appendix.

Let m, n be integers such that $m < n - 1$ and fix any execution α of an n-process single-scanner, multi-writer, m-component snapshot implementation from m registers starting from an initial configuration, C_0.

Lemma 1 (Fatourou, Fich and Ruppert) *No* `SCAN` *operation ever performs* `writes` *in α.*

Consider any process p_i, $1 \leq i \leq n$, other than the scanner, and any component A_j, $1 \leq j \leq m$. For any value v different from the initial value of A_j, consider an execution where p_i runs solo from C_0 to perform an `UPDATE` on A_j with value v. It has been proved in [12] (see appendix) that this execution contains at least one `write` to some register and the first such `write` is performed to the same register independently of the process that executes the `UPDATE` and the value used. Denote this register by R_j.

Lemma 2 (Fatourou, Fich and Ruppert) *If there is a process, other than the scanner, that takes no steps in α, then for each $j \in \{1, \ldots, m\}$,* `UPDATE` *operations to component A_j write only to R_j.*

Lemma 3 (Fatourou, Fich and Ruppert) *For distinct $j_1, j_2 \in \{1, \ldots, m\}$, $R_{j_1} \neq R_{j_2}$.*

Lemma 3 implies the following lower bound on the space complexity of single-scanner, multi-writer implementations of snapshot objects.

Theorem 1 *Any n-process implementation of a multi-writer, single-scanner snapshot object with $m < n - 1$ components from multi-writer registers requires at least m registers.*

We next employ Lemmas 1-3 to prove our lower bound on step complexity. To do so, we construct an execution in which the scanner, p_s, takes $\Omega(m^2)$ steps to perform a single `SCAN` operation S. The construction is inductive, constructing executions $\alpha_0, \alpha_1, \ldots, \alpha_{m-2}$, in which p_s takes more and more steps to complete S. The key part of the induction step is to show that, for each index i, $0 \leq i \leq m-2$, p_s must read at least $(m - i)$ registers after α_i to complete S. We prove that if p_s completes S without executing that many steps, then p_s returns an incorrect response. Thus, in α_{m-2}, p_s performs at least $\Omega(m^2)$ steps to execute S.

Theorem 2 *Any n-process implementation of a multi-writer, single-scanner snapshot object with $m < n - 1$ components using m multi-writer registers has step complexity $\Omega(m^2)$.*

Proof Let $p_s, p_u \in P$, $p_s \neq p_u$, be any two processes; p_s will play the role of the scanner. We assume that the initial value of every component is $\perp$. Let B_0 be the empty set, let π_0 be the empty sequence, and let α_0 be the empty execution. For $0 < i < m-2$, we inductively construct a sequence of indices ℓ_i, where $1 \leq \ell_i \leq m$, a sequence of sets of registers B_i, a sequence of processes p_i, and a sequence of schedules:

$$\begin{aligned}\pi_i = \rho(\ell_i, 0) \cdot \ldots \cdot \rho(\ell_1, 0) \cdot \\ \sigma_1 \cdot w_1 \cdot r_1 \cdot \\ \sigma_2 \cdot w_2 \cdot r_2 \cdot \\ \vdots \\ \sigma_i \cdot w_i \cdot r_i \,,\end{aligned}$$

where for each $1 \leq j \leq i$:

- p_j is a process not in $\{p_s, p_u\}$ that does not take any step in π_{j-1},
- $\rho(\ell_j, 0)$ is the schedule of the biggest prefix of the solo execution of `UPDATE`$(\ell_j, 0)$ by p_j starting from C_0 that does not contain any `write`,
- w_j is the `write` that p_j is poised to perform after $\rho(\ell_j, 0)$,
- σ_j is a sequence of `read` steps by p_s, and
- r_j is a single `read` step of R_{ℓ_j} by p_s.

The construction is done in such a way that, for each $0 \leq i < m - 2$, the following claims hold:

1. if $i > 0$, $R_{\ell_i} \notin B_{i-1}$, $B_i = B_{i-1} \cup \{R_{\ell_i}\}$, and $|B_i| = i$,
2. if $i > 0$, for every register R, if R is not in $B_{i-1} \cup \{R_{\ell_i}\}$, then σ_i contains a `read` of R,
3. σ_i does not contain `writes` to any register, nor does it contain any `reads` of register R_{ℓ_i},
4. π_i is legal starting from C_0,
5. if α_i is the execution we get when we apply π_i from C_0, all steps by p_s in α_i are part of a single `SCAN` operation S, and

6. in a solo execution by p_s starting from the final configuration C_i of α_i, all registers apart from those in B_i are read.

The proof is by induction on i. For the base case, where $i = 0$, Claims 1-5 hold vacuously. We prove claim 6. Let γ be the execution where p_s executes a `SCAN` operation solo starting from C_0. We prove that in γ all m registers are read. To derive a contradiction, assume that there is an integer ℓ, $1 \leq \ell \leq m$, such that R_ℓ is not read by p_s during γ. Let $p \in P - \{p_s, p_u\}$ be any process and let γ' be the execution where p performs an `UPDATE` on component A_ℓ with the value 1 solo starting from C_0. Let γ'' be the execution we get when $\pi(\gamma') \cdot \pi(\gamma)$ is applied starting from C_0. Execution γ'' is legal because Lemma 2 implies that all `writes` in γ' are on register R_ℓ and by assumption, register R_ℓ is not read during γ. Thus, γ'' is indistinguishable from γ to process p_s. Therefore, p_s returns the same vector of values in both executions. Since p_s's `SCAN` starts after p's `UPDATE` has terminated in γ'', process p_s must return the value 1 for component A_ℓ in γ''. Thus, p_s must return 1 for A_ℓ in γ. However, no `UPDATE` with value 1 is executed in γ. This contradicts linearizability for γ.

Induction Hypothesis: Fix any integer i, $0 < i < m - 2$. Assume that we have defined ℓ_{i-1}, and we have constructed B_{i-1} and π_{i-1} so that the claims hold; let α_{i-1} be the execution we get when π_{i-1} is applied starting from C_0.

Induction Step: We choose ℓ_i and we show how to construct B_i and π_i so that the claims hold.

By induction hypothesis (claim 6), in a solo execution by p_s from the final configuration C_{i-1} of α_{i-1}, all registers apart from those in B_{i-1} are read. By induction hypothesis (claim 1), $|B_{i-1}| = (i-1)$. Since $i < m-2$, it follows that there is more than one register that does not belong to B_{i-1}. Let σ_i be the sequence of steps performed by p_s when it runs solo from C_{i-1} until it has read all but one register outside B_{i-1} and it is poised to read this last register for the first time. Let R_{ℓ_i} be this register and let $B_i = B_{i-1} \cup \{R_{\ell_i}\}$. By definition of B_i, it follows that $|B_i| = i$. Thus, claim 1 holds. By definition of σ_i and R_{ℓ_i} and by Lemma 1, claims 2 and 3 also hold.

Let p_i be a process not in $\{p_s, p_u\}$, which has not taken any steps during α_{i-1}. Let

$$\begin{aligned}\pi_i = \rho(\ell_i, 0) \cdot \\ \pi_{i-1} \cdot \\ \sigma_i \cdot w_i \cdot r_i\,,\end{aligned}$$

where:

- $\rho(\ell_i, 0)$ is the schedule of the biggest prefix of the solo execution of `UPDATE`$(\ell_i, 0)$ by p_i from C_0 that does not contain any `write` primitive,
- w_i is the `write` primitive that p_i is poised to perform after $\rho(\ell_i, 0)$ has been applied starting from C_0 (recall that the solo execution of `UPDATE`$(\ell_i, -)$ starting from C_0 contains at least one `write`, and all `writes` contained in it are to register R_{ℓ_i} by Lemma 2), and
- r_i is the `read` of register R_{ℓ_i} that process p_s is poised to perform after $\rho(\ell_i, 0) \cdot \pi_{i-1} \cdot \sigma_i \cdot w_i$ has been applied starting from C_0.

Let α_i be the execution we get when π_i is applied starting from C_0. Since $\rho(\ell_i, 0)$ does not contain any `write` primitives, $\rho(\ell_i, 0) \cdot \pi_{i-1} \cdot \sigma_i$ is legal starting from C_0. Moreover, w_i and r_i are just the steps that processes p_i and p_s, respectively, are poised to perform after $\rho(\ell_i, 0) \cdot \pi_{i-1} \cdot \sigma_i$ has been applied starting from C_0. Thus, α_i is legal starting from C_0, and claim 4 holds.

By definition of σ_i and r_i, claim 5 holds. We next prove claim 6. To derive a contradiction, assume that in a solo execution by p_s starting from the final configuration C_i of α_i, there exists some register $R_\ell \notin B_i$ that is not read. Denote by σ the sequence of steps performed by p_s when it runs solo to complete its active `SCAN` starting from C_i. Let p be any process not in $\{p_s, p_u\}$ that does not take any step in α_i, and let

$$\begin{aligned}\tau &= \rho(\ell, 0) \cdot \\ &\quad \pi_i \cdot \\ &\quad \sigma \cdot w \cdot S' \\ &= \rho(\ell, 0) \cdot \\ &\quad \rho(\ell_i, 0) \cdot \ldots \cdot \rho(\ell_1, 0) \cdot \\ &\quad \sigma_1 \cdot w_1 \cdot r_1 \cdot \\ &\quad \sigma_2 \cdot w_2 \cdot r_2 \cdot \\ &\quad \vdots \\ &\quad \sigma_i \cdot w_i \cdot r_i \cdot \\ &\quad \sigma \cdot w \cdot S'\,,\end{aligned}$$

where:

- $\rho(\ell, 0)$ is the schedule of the biggest prefix of the solo execution of `UPDATE`$(\ell, 0)$ by p from C_0 that does not contain any `write` primitive,
- w is the `write` primitive that process p is poised to perform after $\rho(\ell, 0)$ has been applied starting from C_0, and
- S' is the sequence of steps by process p_s for executing one more `SCAN` operation (other than S) solo starting from the configuration that we

get when $\rho(\ell,0) \cdot \pi_i \cdot \sigma \cdot w$ is applied starting from C_0.

Let γ be the execution we get when τ is applied starting from C_0. We argue that γ is legal. By definition, $\rho(\ell,0)$ contains no `write` primitives, so the execution we get when $\rho(\ell,0) \cdot \pi_i \cdot \sigma$ is applied starting from C_0 is legal. Since w is the step p is poised to perform after $\rho(\ell,0)$ has been applied starting from C_0, γ is legal.

We aim at constructing another execution γ' such that γ' is indistinguishable from γ to process p_s and still p_s must return different vectors of values in these two executions.

Execution γ' is constructed by adding a number of `UPDATES`, each with value 1, executed by process p_u. More specifically, an `UPDATE` operation $U(\ell_1,1)$ by process p_u on component ℓ_1 with value 1 is executed before σ_1. For each $1 \le j < i$, a sequence of two `UPDATES` $U(\ell_{j+1},1) \cdot U'(\ell_j,1)$ by process p_u on components ℓ_{j+1} and ℓ_j with value 1 is executed after σ_j. A sequence of two `UPDATES` $U(\ell,1) \cdot U'(\ell_i,1)$ by process p_u on components ℓ and ℓ_i with value 1 is executed after σ_i. An `UPDATE` operation $U'(\ell,1)$ with value 1 by process p_u is executed after σ. Let

$$
\begin{aligned}
\tau' = {} & \rho(\ell,0) \cdot \rho(\ell_i,0) \cdot \ldots \cdot \rho(\ell_1,0) \cdot \\
& U(\ell_1,1) \cdot \\
& \sigma_1 \cdot U(\ell_2,1) \cdot U'(\ell_1,1) \cdot w_1 \cdot r_1 \cdot \\
& \sigma_2 \cdot U(\ell_3,1) \cdot U'(\ell_2,1) \cdot w_2 \cdot r_2 \cdot \\
& \vdots \\
& \sigma_{i-1} \cdot U(\ell_i,1) \cdot U'(\ell_{i-1},1) \cdot w_{i-1} \cdot r_{i-1} \cdot \\
& \sigma_i \cdot U(\ell,1) \cdot U'(\ell_i,1) \cdot w_i \cdot r_i \cdot \\
& \sigma \cdot U'(\ell,1) \cdot w \cdot S' .
\end{aligned}
$$

Let γ' be the execution we get when τ' is applied starting from C_0. We first prove that γ' is legal. After the beginning of the execution of $U(\ell_1,1)$ (that is the first `UPDATE` by p_u), each of the processes $p_1,\ldots,p_i,p$ executes just the `write` primitive that it is poised to perform. By Lemma 2, for each $1 \le j \le i$, all `writes` contained in $U(\ell_j,1)$ are to register R_{ℓ_j}. By induction hypothesis (claim 3), register R_{ℓ_j} is not read during σ_j. Moreover, w_j overwrites any value that was written to R_{ℓ_j} during $U(\ell_j,1)$. By Lemma 2, all `writes` contained in $U'(\ell_j,1)$ are to register R_{ℓ_j}. Register R_{ℓ_j} is overwritten by w_j before p_s executes any `read` primitive. By Lemma 2, all `writes` contained in $U(\ell,1)$ and $U'(\ell,1)$ are to register R_ℓ. By assumption, register R_ℓ is not read during σ; moreover, register R_ℓ is overwritten by w. Thus, none of the values written to a register by p_u is ever read by p_s. It follows that γ' is legal starting from C_0. Notice that γ' is indistinguishable from γ to all processes other than p_u. (We remark that for defining τ' we abuse notation and use U and U' to denote both the `UPDATE` operations and the sequence of steps these `UPDATES` perform in γ'.)

From now on we call process p_u *invisible*, while the rest of the processes that perform `UPDATES` are *visible*. `UPDATES` performed by visible processes are called *visible* `UPDATES`, whereas those performed by p_u are called *invisible* `UPDATES`. We remark that all visible `UPDATES` use the value 0, whereas all invisible `UPDATES` use the value 1.

The operations by p_u and the final `SCAN` S' are executed serially in γ', so they are linearized according to the order of their execution:

$$
\begin{aligned}
& U(\ell_1,1), \\
& U(\ell_2,1), U'(\ell_1,1), \\
& U(\ell_3,1), U'(\ell_2,1), \\
& \vdots \\
& U(\ell_i,1), U'(\ell_{i-1},1) \\
& U(\ell,1), U'(\ell_i,1) \\
& U'(\ell,1), S'.
\end{aligned}
$$

We next argue about the order in which visible `UPDATES` are linearized. By claim 1, for all j,k, $1 \le j,k \le i$, $j \ne k$, $\ell_j \ne \ell_k$. Moreover, $\ell \ne \ell_j$ and $\ell \ne \ell_k$ since $R_\ell \notin B_i$. Thus, every visible `UPDATE` is executed on a different component from any other visible `UPDATE` in γ and γ'.

Since γ is indistinguishable from γ' to process p_s, S' returns the same vector of values in γ and γ'. In the partial linearization order presented above, no invisible `UPDATE` on A_{ℓ_j} is linearized between $U'(\ell_j,1)$ and S'. Thus, for each $1 \le j \le i$, S' returns either 1 for component A_{ℓ_j} or the value 0 of some visible `UPDATE` that is linearized after $U'(\ell_j,1)$. However, S' cannot return 1 for A_{ℓ_j} because S' returns the same vector of values in γ and γ', and no `UPDATE` with value 1 is executed in γ. Thus, $U(\ell_j,0)$ (the unique visible `UPDATE` on A_{ℓ_j}) must be linearized between $U'(\ell_j,1)$ and S' (see Fig. 1).

We next prove that in any linearization order of γ', the snapshot object always contains the value 1 in at least one of its components after the execution of $U(\ell_1,1)$. For each $1 \le j \le i$, the value 1, written by $U(l_j,1)$, is in A_{ℓ_j} from the execution of $U(\ell_j,1)$ until the execution of $U'(\ell_j,1)$. Moreover, the value 1 is in A_ℓ from the execution of $U(\ell,1)$ until the execution of $U'(\ell,1)$. Notice that S, the first `SCAN` by p_s, starts its execution in γ' after $U(l_1,1)$ and terminates before $U'(\ell,1)$. Thus, its linearization point must be placed between $U(l_1,1)$

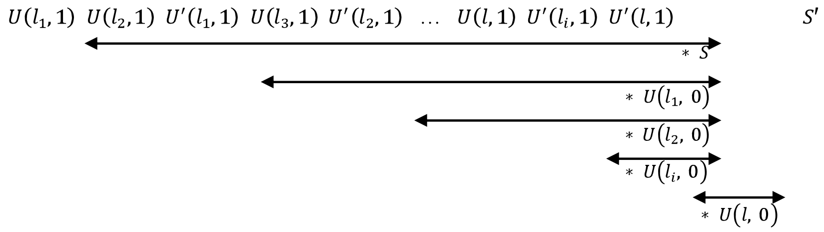

Fig. 1 Proof of Theorem 2: Linearization points for SCANS and UPDATES in γ'.

and $U'(\ell, 1)$ (see Fig. 1). It follows that S must return the value 1 for at least one component in γ' (independently of where exactly it is linearized). However, p_s must return the same vector of values in executions γ and γ', and no UPDATE with value 1 is executed in γ. This contradicts the fact that γ is linearizable.

We conclude that claim 6 holds. The proof of the induction step is now complete.

Claims 1 and 2 imply that for each i, $1 \leq i < m-2$, p_s performs $m-i$ `read` primitives during σ_i and r_i. Thus, in α_{m-1}, p_s performs at least $(m + (m-1) + \ldots + 3) \in \Omega(m^2)$ `read` primitives.

5 The Checkmarking Algorithm

In this section, we present Checkmarking, a single-scanner, multi-writer m-component snapshot implementation from $m+1$ registers. Checkmarking is linearizable and has step complexity $O(m^2)$.

A description of Checkmarking is provided in Section 5.1. In Section 5.2, we prove that Checkmarking is linearizable, and in Section 5.3, we study its space complexity and its step complexity.

5.1 Description

Checkmarking uses $m+1$ registers, denoted $R_1, \ldots, R_{m+1}$; these are the only shared variables used by the algorithm. Each component A_i, $1 \leq i \leq m$, is associated with a register R_i and processes updating A_i write only to R_i. Register R_{m+1} is written when some SCAN takes place (i.e., it is written by the process executing the SCAN). Notice that if we assume that all SCANS are performed by a single process (that is, there is a single scanner in the system), then R_{m+1} is a single-writer register. We remark that the algorithm is correct even if SCAN operations are executed by different processes provided that no pair of SCANS overlaps.

Checkmarking is based on the well-known technique [1] in which a scanner repeatedly reads the m registers written by the updaters until it sees the same values in all registers in two consecutive sets of reads. To achieve wait-freedom, a process executing UPDATE helps SCAN by calculating a vector of values and storing it in the appropriate register together with the new value. In contrast to what happens in [1], Checkmarking avoids paying a step complexity cost of $\Omega(n)$ by introducing a new efficient termination technique for SCANS, which takes into consideration the fact that Checkmarking is single-scanner.

For each $1 \leq i \leq m$, register R_i stores the following information: (1) the *value* of component A_i, (2) the identifier *id* of the process p that performed the last `write` to R_i,(3) a *timestamp*, which is used by p to distinguish its UPDATES, (4) a sequence number, *curr_seq*, that p read in R_{m+1} at the beginning of the execution of the UPDATE operation that last wrote in R_i, and (5) a vector *view* containing a value for each of the m components. Register R_{m+1} stores only an integer, *curr_seq*, which has the initial value 1 and is increased by one each time a new SCAN operation starts executing.

Each SCAN and UPDATE operation tries to obtain a consistent vector by executing GetVector. Each time a SCAN S is executed by some process p, the following actions take place. Process p increases by one the *curr_seq* field of register R_{m+1}. Then, p executes GetVector and returns the vector calculated by it.

Each process has a local variable ts, with initial value 0, which is incremented every time the process executes an UPDATE operation. During the execution of an UPDATE on some component A_i by some process p, the following actions take place. Process p first reads the value of *curr_seq* from register R_{m+1}. To help SCANS complete, the UPDATE then tries to obtain a consistent vector by executing GetVector. Finally, p writes the new value of A_i, its identifier, its increased timestamp, the value of *curr_seq* it read in R_{m+1}, and the vector of values calculated by GetVector to register R_i. The pseudocode for SCAN and UPDATE is presented in Algorithm 1. For ease of presentation, we assume

Algorithm 1 Pseudocode for `UPDATE` and `SCAN`. (We assume that components store values of type *data*.)

```
struct register {
    data value;
    int id;
    int timestamp;
    int curr_seq;
    data view[1..m];
}

shared struct register R_1, R_2 ..., R_{m+1};
// initially, R_{m+1}.curr_seq = 1

void UPDATE(int i, data value, int id, int ts) {
    data view[1..m];
    int curr_seq;

    curr_seq = R_{m+1}.curr_seq;
    view = GetVector(curr_seq);
    R_i = <value, id, ts, curr_seq, view>;
}

data *SCAN(void) {
    data view[1..m];
    int curr_seq;

    curr_seq = R_{m+1}.curr_seq + 1;
    R_{m+1}.curr_seq = curr_seq;
    view = GetVector(curr_seq);
    return view;
}
```

that R_{m+1} has the same structure as the rest of the registers and all its fields other than *curr_seq* are unused.

Any instance g of `GetVector` performs consecutive *sets of reads* of $R_1, \ldots, R_m$ until one of the following conditions is satisfied:

1. If the *curr_seq* field of some register R_i has a value larger than or equal to the *curr_seq* parameter of g, then the `UPDATE` operation that wrote this value to R_i started its execution after the beginning of the operation that invoked g and finished it before the completion of g. In this case, g returns the vector of values read in the *view* field of R_i.
2. Assume that there exist integers $\ell > 1$ and $j \geq \ell$, for which the following hold: there exists an integer $d \geq 0$, such that (a) during the ℓ-th set of reads, d registers $R_{x_1}, \ldots, R_{x_d}$ (and no others) have different values than those read during the $(\ell - 1)$st set of reads, (b) g has seen each of these d registers change at least once between the ℓ-th set of reads and the j-th set of reads, and (c) j is the smallest and ℓ is the largest integer for which conditions (a) and (b) hold. Then, g responds by returning the *value* fields of $R_1, \ldots, R_m$ read during the ℓ-th set of reads. We remark that if $d = 0$, then g terminates by observing, for each register, the same values in two consecutive sets of reads, namely the $(\ell - 1)$-st and the ℓ-th set of reads; we remark that in this case, $j = \ell$.

	A_1	A_2	A_3	A_4
1				
2		✓		
3	✓		✓	
4				✓
5			✓	
6	✓			

Fig. 2 An example of an execution of an instance of `GetVector` in Checkmarking that terminates by evaluating the second termination condition to `true`.

To better illustrate the second termination condition, Fig. 2 shows array *history* for a snapshot object of four components in an execution of a `SCAN` S where six sets of reads take place before the second termination condition becomes `true` and S terminates. A ✓ appears in those elements of the array whose value has changed from the $(j - 1)$-st set of reads to the j-th set of reads. Termination condition (2) is satisfied for the first time when $j = 6$ and $\ell = 3$ because components A_1 and A_3, which are seen by S to have changed from the 2nd to the 3rd set of reads, have changed once more between the 3rd set of reads and the 6th set of reads. Notice that, for all smaller values of j, there is no value of ℓ that satisfies the required property.

The pseudocode for `GetVector` is presented in Algorithm 2. Each of the processes maintains a local array of two dimensions, called *history*. Row 1 of *history* stores the information that is read during the initial set of reads (line 18 of the pseudocode). Specifically, for each i, $1 \leq i \leq m$, each element of $history[1][i]$ has two fields; these are r, which stores the value read in R_i during the first set of reads, and a boolean variable *change*, which is equal to `false`. For each $1 < j \leq m + 2$, row j of *history* stores the information that is read during the j-th set of reads. We will prove in Section 5.3 that at most $m + 2$ sets of reads may take place in any execution of `GetVector`. Moreover, $history[j][i].change$, $1 \leq i \leq m$, is a boolean variable that indicates whether register R_i has been found to have a different value when it was read during the j-th set of reads from the value read in it during the $(j - 1)$-st set of reads. The number of registers that have been found to indeed have a different value is stored in $checkmarks[j]$.

Specifically, *checkmarks*$[j]$ stores a counter of the number of checkmarks in row j that have no later checkmark in the same column (see Fig. 2).

To check whether condition (2) is satisfied, each time a checkmark is added in *history*$[j][i]$, where $2 \leq j \leq m+2$, $1 \leq i \leq m$, the algorithm walks up column i starting from row $j-1$ until it reaches an earlier checkmark in column i (or the beginning of the column). If an earlier checkmark is reached on row ℓ of *history*, *checkmarks*$[\ell]$ is decreased by one and that checkmark is removed. If a row's counter becomes equal to zero, condition (2) is satisfied and the algorithm terminates and returns the vector of values stored in that row.

Consider any `SCAN` S and let U be an `UPDATE` that performs its `write` primitive in the execution interval of S. Notice that if U starts its execution after S writes into R_{m+1}, then the execution interval of U is contained in the execution interval of S. A `SCAN` that sees such an `UPDATE` borrows the vector written by it. Condition (2) guarantees that S terminates even if it does not ever see such an `UPDATE`.

5.2 Linearizability

Consider any execution α of Checkmarking. By inspection of the pseudocode (lines 1-5), it follows that no `UPDATE` ever writes into register R_{m+1}. Thus, R_{m+1} is written only by `SCANS`. Moreover, the integers stored in R_{m+1} (lines 8-9) are strictly increasing.

Observation 3 *The following hold:*

1. *no* `UPDATE` *ever writes to register* R_{m+1}*, and*
2. *the values written into* R_{m+1} *are strictly increasing.*

Let S be any `SCAN` executed in α and denote by g the instance of `GetVector` executed by S.

Lemma 4 *Suppose that g terminates after epoch iterations of the while loop of line 20. For each integer ℓ, $1 \leq \ell \leq epoch$, at the beginning of the ℓ-th iteration, it holds that for each integer i, $1 < i < \ell$, checkmarks$[i] > 0$.*

Proof The proof is a direct induction on ℓ. The claim holds vacuously when $\ell = 1$. Fix any ℓ, $1 \leq \ell < epoch$ and suppose that the claim is true for ℓ. We prove that the claim holds for $\ell + 1$.

Since $\ell < epoch$, it follows that g does not terminate during the ℓ-th iteration of the while loop. By inspection of the pseudocode, it follows that the condition of the if statement of line 26 evaluates to `true` at least once during the execution of the ℓ-th iteration (otherwise, g would return on line 38 before the end of the ℓ-th iteration). It follows that *checkmarks*$[\ell] > 0$ at the end of the ℓ-th iteration.

The induction hypothesis implies that at the beginning of the ℓ-th iteration, for each j, $2 \leq j < \ell$, *checkmarks*$[j] > 0$. By inspection of the pseudocode, it follows that *checkmarks*$[j]$ is reduced only whenever line 30 is executed. However, the if statement of line 32 implies that the first time that *checkmarks*$[j]$ becomes equal to zero for some j, g terminates. Since $\ell < epoch$, this does not occur during the ℓ-th iteration of the while loop. It follows that at the beginning of the $(\ell+1)$-st iteration, it holds that for each j, $2 \leq j < \ell + 1$, *checkmarks*$[j] > 0$.

By inspection of the pseudocode (lines 34 and 38) and by Lemma 4, we get the following observation.

Observation 4 *Suppose that g returns on line 34 or 38. Then, the following hold:*

1. *at the point that g returns, there is an integer ℓ, $1 < \ell \leq epoch$, such that checkmarks$[\ell] = 0$, and for each integer j, $1 < j \leq epoch$, $j \neq \ell$, checkmarks$[j] \neq 0$,*
2. *g returns the values read during the ℓ-th iteration of the while loop of line 20.*

We split the execution interval of g into epochs as follows. Epoch 1 starts with the first and ends with the last `read` primitive of the initial set of reads. Similarly, for each $i > 1$, the i-th epoch (or epoch i) starts with the first and ends with the last `read` primitive of the i-th set of reads.

Assume that g completes on line 25 of the pseudocode. We denote by k_g the largest integer for which the following holds: there exists a sequence $U_1, \ldots, U_{k_g}$ of `UPDATES` such that:

- U_{k_g} is the `UPDATE` operation that writes the vector of values returned by g, and
- for each ℓ, $1 < \ell \leq k_g$, the instance g_ℓ of `GetVector` that is executed by U_ℓ returns (on line 25) the vector of values written by $U_{\ell-1}$.

Let $SU(g) = U_1, \ldots, U_{k_g}$ and let $SG(g) = g_1, \ldots, g_{k_g}$. In case g does not terminate on line 25, $SU(g) = SG(g) = \lambda$ (where λ is the empty sequence). Notice that g and each of the $g_1, \ldots, g_{k_g} \in SG(g)$ return the same vector of values. Moreover, g_1 returns by executing line 34 or line 38 of the pseudocode, while $g_2, \ldots, g_{k_g}, g$ return by executing line 25.

Algorithm 2 Pseudocode for `GetVector`.

```
    struct info {
12      struct register r;
13      boolean change;
    }

    data *GetVector(int curr_seq) {
14      struct info history[1..m + 2][1..m];
15      int epoch = 1, i, ℓ, k, checkmarks[1..m + 2] = {0, . . . , 0};
16      data view[1..m];
17      struct register mp;

        // initial set of reads
18      for (i = 1; i ≤ m; i++) history[epoch][i] = ⟨read(R_i), false⟩;
19      epoch = epoch + 1;

20      while (true) {
            // perform the next set of reads checking if condition 1 is satisfied
21          for (i = 1; i ≤ m; i++) {
22              mp = read(R_i);
23              history[epoch][i] = ⟨mp, false⟩;
24              if (mp.curr_seq ≥ curr_seq)
25                  return mp.view;
26              if (history[epoch-1][i].r ≠ history[epoch][i].r) {
27                  history[epoch][i].change = true;
                    checkmarks[epoch]++;

28                  for (ℓ = epoch-1; ℓ ≥ 2; ℓ−−) {
29                      if (history[ℓ][i].change == true) {
30                          (checkmarks[ℓ])−−;
31                          history[ℓ][i].change = false;
32                          if (checkmarks[ℓ] == 0) {
33                              for (k =1; k ≤ m; k++) view[k] = history[ℓ][k].r.value ;
34                              return view;
                            } // if
35                          break;  // stop executing the for loop of line 28
                        } // if
                    } // for
                } // if
            } // for
36          if (checkmarks[epoch] == 0) {
37              for (k = 1; k ≤ m; k++) view[k] = history[epoch][k].r.value;
38              return view;
            } // if
39          epoch = epoch + 1;
        } //while
    }
```

Let $g(S) = g_1$ if $SG(g) \neq \lambda$, and let $g(S) = g$ otherwise. For clarity of presentation, Table 2 summarizes the notation used in this section.

The following observation is a consequence of the above definitions.

Observation 5 *For any* `SCAN` *operation S in α, the following hold:*

1. *$g(S)$ returns on line 34 or line 38,*
2. *S returns the same vector of values as $g(S)$, and*
3. *if g is the instance of* `GetVector` *invoked by S and $SG(g) \neq \lambda$, then g returns by executing line 25. Moreover, for each instance g' of* `GetVector` *in $SG(g)$ other than $g(S)$, g' returns by executing line 25.*

We next assign linearization points to `SCANS` that complete in α and to `UPDATES` that perform the `write` of line 5 in α.

Consider any `SCAN` operation S that completes in α. We find it helpful to assign a linearization point not only to S but also to $g(S)$. Assume that $g(S)$ returns after having executed *epoch* ≥ 1 iterations of the while loop of line 20. By Observation 5 (claim 1), $g(S)$ terminates by executing line 34 or line 38. By Observation 4, at the point that $g(S)$

Notation	Description
A_i, $1 \leq i \leq m$	The i-th component of the snapshot object
R_i, $1 \leq i \leq m$	Register that is associated with component A_i; `UPDATES` to component A_i write only to R_i
R_{m+1}	Register that is written by `SCANS`
α	An execution of Checkmarking
S	A `SCAN` operation in α
g	The instance of `GetVector` executed by S
U_{k_g}	The `UPDATE` that writes the vector of values returned by g (if g returns on line 25)
$SU(g) = U_1, \ldots, U_{k_g}$	The instance g_ℓ, $1 < \ell \leq k_g$, of `GetVector` executed by U_l terminates on line 25 and returns the vector of values written by U_{l-1}; $k_g \geq 0$ is the length of this sequence
$SG(g) = g_1 \ldots, g_{k_g}$	The sequence of instances of `GetVector` that are invoked by $U_1, \ldots, U_{k_g}$, respectively
λ	The empty sequence
$g(S)$	$g(S) = g_1$ if $SU(g) \neq \lambda$, $g(S) = g$ otherwise

Table 2 Notation used in the proof of Checkmarking.

terminates, there is a unique integer ℓ, $1 < \ell \leq epoch$, for which it holds that $checkmarks[\ell] = 0$; moreover, $g(S)$ returns the values it read during its ℓ-th epoch. We insert the linearization point for $g(S)$ immediately before the point that performs the first `read` primitive of the ℓ-th epoch. The linearization point for S is inserted at the same place as that for $g(S)$.

Let $d \geq 0$ be the number of registers whose values have changed from the $(\ell-1)$-st to the ℓ-th epoch of $g(S)$. Denote by $R_{x_1}, \ldots, R_{x_d}$ these registers, denote by $v_{x_1}, \ldots, v_{x_d}$ the values read by $g(S)$ in $R_{x_1}, \ldots, R_{x_d}$, respectively, during the ℓ-th epoch, and let $U_{x_1}, \ldots, U_{x_d}$ be the `UPDATES` that wrote the values $v_{x_1}, \ldots, v_{x_d}$ to $R_{x_1}, \ldots, R_{x_d}$. Notice that $U_{x_1}, \ldots, U_{x_d}$ update different components. For those `UPDATES` of the $U_{x_1}, \ldots, U_{x_d}$ that performed their `write` primitives after the first `read` primitive r_1 of the ℓ-th epoch of $g(S)$, we insert their linearization points immediately before the linearization point of $g(S)$ (in any order since they all update different components). After we have assigned linearization points to all `SCANS` (and to some `UPDATES`) according to the rules described above, we linearize each `UPDATE` that has not yet been assigned a linearization point at the point where its `write` occurs. Let L be the linearization of α determined by assigning linearization points to operations as described above.

Intuitively, it turns out that all values read by $g(S)$ in the ℓ-th epoch have been written by `UPDATES` that have started their execution before S writes to R_{m+1}. Some of them perform their `write` before r_1 (where S is linearized), whereas others after it. Let U be such an `UPDATE` that performs its `write` before r_1. Let A_j be the component that U updates. We argue that U is linearized at its `write` and no other `UPDATE` has written to R_j between U's `write` and r_1. This implies the consistency of the value returned for A_j by $g(S)$ (as well as by S).

To guarantee the consistency of those values returned by S that have been written by `UPDATES`, which perform their `writes` after r_1, we have to move the linearization points of these `UPDATES` immediately before r_1 (that is earlier than the point where their `write` primitives occur). Let U be an `UPDATE` whose linearization point has been placed immediately before r_1. Assume that U updates component A_j and let w be the `write` primitive performed by U. Notice that w may obliterate the evidence of some other `UPDATE` U' on A_j that performs its `write` between r_1 and r_j. Since S does not see the value that U' writes for A_j, U' is linearized at its `write` primitive and therefore after the linearization point of U. This might cause problems to the consistency of `SCANS` that follow S (see Fig. 3). For this reason, termination condition (2) requires that S sees the value written by one more `UPDATE` on A_j (let it be U'') after the ℓ-th epoch of its execution; we argue that U'' is linearized at its `write` primitive, which occurs after r_j and before the end of S, and therefore U'' is linearized after U and U'. Thus, in order to be consistent, `SCANS` that are subsequent to S must return the value of U'' or some later `UPDATE` (and not that of U', which they cannot be aware of).

By the way linearization points are assigned, each `SCAN` and each `UPDATE` that is linearized at the `write` primitive it performs on line 5 is assigned a unique linearization point. Consider an `UPDATE` U that is not linearized at the `write` primitive it per-

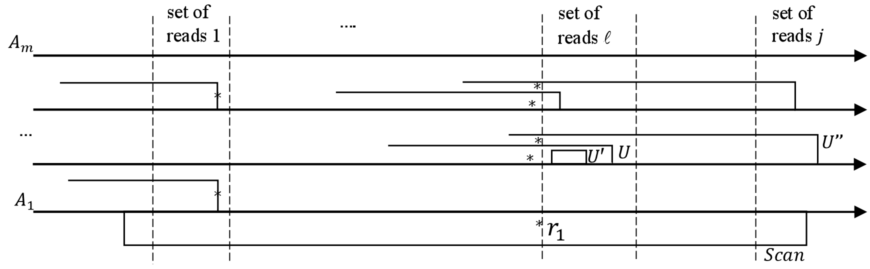

Fig. 3 An example of an execution of Checkmarking.

forms on line 5; let w be this primitive. By the way linearization points are assigned, there is a SCAN S such that w is performed within the execution interval of S. Since there is a single active SCAN at each point in time, it follows that the linearization point of U is unique.

To argue that for each SCAN and UPDATE, its linearization point is within its execution interval, we first prove the following technical lemma.

Lemma 5 *Consider any* SCAN *operation S in α and denote by g the instance of* GetVector *that is executed by S. Let $SU(g) = U_1, \ldots, U_{k_g} \neq \lambda$ and let $SG(g) = g_1, \ldots, g_{k_g}$. Then, for each $1 \leq j \leq k_g$, it holds that:*

1. *if $j > 1$, U_{j-1} performs its* write *primitive before the* write *primitive of U_j,*
2. *the value of curr_seq read in R_{m+1} by U_j is the value written there by S, and*
3. *the execution interval of U_j (and therefore also of g_j) is within the execution interval of S and starts after S has written to R_{m+1}.*

Proof We start by proving claim 1. By definition of $SU(g)$ and $SG(g)$, for each $1 < j \leq k_g$, g_j (that is invoked by U_j) returns the vector of values written by U_{j-1}. Thus, U_{j-1} executes its (unique) write primitive before the end of g_j and therefore before the write primitive of U_j.

Next, we prove claim 2. Let $g_{k_g+1} = g$. Consider any j, $1 < j \leq k_g+1$. Observation 5 (claim 3) implies that g_j terminates by executing line 25 of the pseudocode. Therefore, the condition of the if statement on line 24 is evaluated to true by g_j. Since (1) g_j returns *mp.view*, (2) *mp* is the value written by U_{j-1}, and (3) *mp.curr_seq* is greater than or equal to the *curr_seq* parameter of g_j, it follows that U_{j-1} has read a value for *curr_seq* in R_{m+1} that is greater than or equal to that read by U_j (if $j \leq k_g$), or to the value written there by S (if $j = k_g + 1$). (Notice that the value written to R_{m+1} by S is equal to the *curr_seq* parameter of $g = g_{k_g+1}$.)

By definition of $SU(g)$, g returns the vector of values written by U_{k_g}. Thus, U_{k_g} terminates before the end of g (and therefore also before the end of S). By claim 1, for each $1 \leq j < k_g$, U_j terminates before U_{k_g}. Therefore, U_j also terminates before the end of S. Since there is just a single active SCAN in the system at each point in time and for each j, $1 \leq j \leq k_g$, S terminates after the end of U_j, Observation 3 (claim 1) implies that U_j cannot read a value for *curr_seq* in R_{m+1} greater than that written there by S. Thus, U_j reads the value written to R_{m+1} by S for *curr_seq*.

We next prove claim 3. It suffices to prove that U_j (and therefore also g_j, which is invoked by U_j) starts its execution after S has written to R_{m+1}. This is so because U_j starts its execution by reading R_{m+1} and reads there the value written by S.

The next lemma is a consequence of Lemma 5 and the definition of $g(S)$.

Lemma 6 *Consider any* SCAN *operation S in α. Then, the following hold:*

1. *the execution interval of $g(S)$ is contained in the execution interval of S and starts after S has written to R_{m+1}, and*
2. *the curr_seq parameter of $g(S)$ has the value written to R_{m+1} by S.*

Proof Let g be the instance of GetVector executed by S. Consider first the case where $SG(g) = \lambda$. Then, $g(S) = g$ (by definition). Obviously, the execution interval of g is contained in the execution interval of S (since g is executed by S). By inspection of the pseudocode (lines 8-10), S invokes g after it writes to R_{m+1}. Also, notice that the *curr_seq* parameter of g has the value written to R_{m+1} by S (because g is called by S with this parameter).

Consider now the case where $SG(g) \neq \lambda$. Then, $g(S) = g_1$ (by definition). By Lemma 5 (claim 3), it follows that the execution interval of the first instance g_1 of GetVector in $SG(g)$ is within the execution interval of S and starts after S has written to R_{m+1}. Lemma 5 (claim 2) implies that the *curr_seq* parameter of g_1 has the value written to R_{m+1} by S.

We next prove that the linearization point of any `SCAN` that terminates in α is within its execution interval.

Lemma 7 *For each* `SCAN` *operation* S *that terminates in* α*, the following hold:*

1. *the linearization point of* $g(S)$ *is within its execution interval, and*
2. *the linearization point of* S *is within its execution interval.*

Proof Consider any `SCAN` S that terminates in α. By Observation 5 (claim 1), $g(S)$ returns by executing line 34 or line 38. By the way linearization points are assigned, it follows that the linearization point of $g(S)$ is within its execution interval. Moreover, the linearization point of S is placed at the same point as that of $g(S)$. By Lemma 6 (claim 1), the execution interval of $g(S)$ is within the execution interval of S. Therefore, the linearization point of S is within its execution interval.

Next, we study properties of an `UPDATE` operation whose linearization point has not been inserted at its `write` primitive.

Lemma 8 *Consider any* `UPDATE` *operation* U *on a component* A_j *such that the linearization point of* U *is not at the point that it performs its* `write` *primitive. Then, there exists a* `SCAN` *operation* S *such that:*

1. *the linearization point of* U *is contained in the execution interval of* S*,*
2. S *returns the value* v *written by* U *for component* A_j*, and*
3. *the* `write` *primitive of* U *follows the linearization point of* $g(S)$ *and precedes the end of* S*.*

Proof We start by proving claim 1. Since the linearization point of U has not been inserted at the point that U performs its `write`, by the way linearization points are assigned, there is some `SCAN` operation S such that U is linearized immediately before $g(S)$. The linearization point of S is placed at the same point as the linearization point of $g(S)$. By Lemma 7 (claim 2), the linearization point of S is within its execution interval. It follows that the linearization point of U is within the execution interval of S.

Next, we prove claim 2. Assume that $g(S)$ returns after *epoch* iterations of the while loop of line 20. By Observation 5 (claim 1), $g(S)$ terminates by executing line 34 or line 38. By Observation 4, at the point that $g(S)$ terminates, there is a unique integer ℓ, $1 < \ell \leq epoch$, for which it holds that $checkmarks[\ell] = 0$; moreover, $g(S)$ returns the values it read during its ℓ-th epoch. Let $d \geq 0$ be the number of registers whose values have changed from the $(\ell - 1)$-st to the ℓ-th epoch of $g(S)$. Denote by $R_{x_1}, \ldots, R_{x_d}$ these registers, denote by $v_{x_1}, \ldots, v_{x_d}$ the values read by $g(S)$ in $R_{x_1}, \ldots, R_{x_d}$, respectively, during the ℓ-th epoch, and let $U_{x_1}, \ldots, U_{x_d}$ be the `UPDATES` that wrote the values $v_{x_1}, \ldots, v_{x_d}$ to $R_{x_1}, \ldots, R_{x_d}$. By the way linearization points are assigned, U must be one of the $U_{x_1}, \ldots, U_{x_d}$. Thus, in the ℓ-th epoch, $g(S)$ reads the value written by U for A_j. Since $g(S)$ returns the vector of values read in the ℓ-th epoch, it follows that $g(S)$ returns the value written by U for A_j. By Observation 5 (claim 2), S returns the same vector of values as $g(S)$. Thus, S returns the value written by U for A_j.

We next prove claim 3. Let r_1 be the first `read` primitive executed by $g(S)$ at the ℓ-th epoch, and let r_j be the `read` primitive executed by $g(S)$ on register R_j (that corresponds to component A_j) during the ℓ-th epoch. By the way linearization points are assigned, the `write` primitive w of U follows r_1. Since $g(S)$ is linearized immediately before r_1, it follows that the `write` primitive of U follows the linearization point of $g(S)$. Since S returns the value v written by U for A_j, it follows that w is performed before the end of S.

We next prove that the linearization point of each `UPDATE` operation is within its execution interval.

Lemma 9 *For each* `UPDATE` *operation* U *that performs its* `write` *in* α*, the linearization point of* U *is within its execution interval.*

Proof If the linearization point of U has been inserted at the point where its `write` occurs, then it is obviously within its execution interval. Assume that this is not the case. Then, there exists a `SCAN` operation S such that the linearization point of U has been inserted immediately before the linearization point of $g(S)$. By Lemma 7 (claim 1), the linearization point of $g(S)$ is within its execution interval. By Lemma 8 (claim 3), U has performed its `write` primitive after the linearization point of $g(S)$.

By Lemma 6 (claim 1), the execution interval of $g(S)$ starts after S has written to R_{m+1}. It suffices to prove that U has started its execution before the `write` of S to R_{m+1}. Suppose not. By Lemma 8 (claim 2), S returns the value written by U. By Observation 5 (claim 2), S returns the same vector of values as $g(S)$. It follows that $g(S)$ reads the value written by U. By inspection of the pseudocode (lines 22-25), it follows that if U reads a value for *curr_seq* greater than or equal to that written to R_{m+1} by S, $g(S)$ will terminate by executing line 25 returning the vector of values written by U. This contradicts Observation 5 (claim 1). Thus, U reads in R_{m+1} a value for *curr_seq* less than that written there by S. Therefore, Observation 3 implies that U starts its execution before S writes to R_{m+1}.

Consider any `UPDATE` operation U on a component A_j such that the linearization point of U has not been inserted at the point that it performs its `write` primitive. By Lemma 8, there exists a `SCAN` operation, which we will denote by $S(U)$, such that the linearization point of U has been inserted immediately before the linearization point of $g(S(U))$. To prove consistency of `SCANS` (with respect to L), we first prove the following technical lemma.

Lemma 10 *Consider any* `UPDATE` *operation* U *on a component* A_j *such that the linearization point of* U *has not been inserted at the point that it performs its* `write` *primitive. Then, there exists an* `UPDATE` *operation* U' *such that:*

1. *the* `write` *primitive* w *of* U *precedes the* `write` *primitive* w' *of* U',
2. w' *precedes the end of* $S(U)$, *and*
3. *the linearization point of* U' *follows the linearization point of* $S(U)$.

Proof Assume that $g(S(U))$ returns after *epoch* iterations of the while loop (line 20). By Observation 5 (claim 1), $g(S(U))$ terminates by executing line 34 or line 38. By Observation 4, at the point that $g(S(U))$ terminates, there is a unique integer ℓ, $1 < \ell \leq epoch$, for which it holds that $checkmarks[\ell] = 0$; moreover, $g(S(U))$ returns the values it read during its ℓ-th epoch. By the way linearization points are assigned, $g(S(U))$ reads the value written by U for A_j during the ℓ-th epoch.

Let r_1 be the first `read` executed by $g(S(U))$ at the ℓ-th epoch, and let r_j be the `read` of $g(S(U))$ on register R_j in the ℓ-th epoch. By the way linearization points are assigned, $g(S(U))$ is linearized

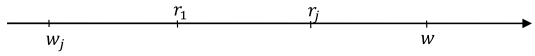

Fig. 4 Case 2 of Lemma 11.

before r_1, U is linearized before $g(S(U))$, and U's `write` is performed after r_1 and before r_j. Therefore, the first time $g(S(U))$ reads the value written to R_j by U is by executing r_j (i.e., in the ℓ-th epoch). By the pseudocode, it follows that $history[\ell][j].change ==$ `true` at the point that r_j is executed and therefore $checkmarks[\ell]$ is greater than zero at that point. By definition of ℓ, it follows that $checkmarks[\ell]$ is equal to zero at the point that $g(S(U))$ terminates. By inspection of the pseudocode, it follows that there is some integer $\ell' > \ell$ such that $history[\ell'][j].change ==$ `true`. Therefore, R_j that has changed from the $(\ell-1)$-st to the ℓ-th epoch of $g(S(U))$ changes again from the $(\ell'-1)$-st to the ℓ'-th epoch. So, there exists some `UPDATE` operation U', which performs its `write` primitive w' after the `write` primitive w of U and before the end of the execution interval of $g(S(U))$ (and therefore also of $S(U)$). So, claims 1 and 2 hold.

Next, we prove claim 3. We argue that the linearization point of U' is inserted at its `write` primitive and therefore it follows the linearization point of $g(S(U))$ and $S(U)$. Suppose not. Then, by the way linearization points are assigned, there exists some `SCAN` operation S' such that S' returns U''s value for A_j and the linearization point of U' is placed immediately before that of $g(S')$ (and S'). Since $S(U)$ returns U's value and not U''s value for A_j, S' is different from S. Since, by claim 2, w' is performed before the end of the execution interval of $S(U)$, Lemma 9 implies that the linearization point of U' is placed before the end of $S(U)$. Thus, U' and therefore also S' is linearized before $S(U)$. Lemma 7 (claim 1), Lemma 6 (claim 1), and Lemma 8 (claim 3) imply that w' occurs in the execution interval of S'. Since w occurs in the execution interval of $S(U)$ (between r_1 and r_j), it follows that w' precedes w. This contradicts claim 1 proved above.

We are now ready to prove that `SCANS` return consistent vectors with respect to L.

Lemma 11 *For each* `SCAN` *operation* S *that terminates in* α, *the vector of values returned by* S *is consistent with respect to* L.

Proof By the way linearization points are assigned, S is linearized at the same place as $g(S)$. By Observation 5 (claim 2), S returns the same vector of values as $g(S)$. Thus, it suffices to prove that $g(S)$ returns a consistent vector of values with respect to L. Assume that $view = \langle v_1, \ldots, v_m \rangle$ is the vector of values returned by $g(S)$. To derive a contradiction, assume that there is some integer $j \in \{1, \ldots, m\}$ such that the value parameter of the last `UPDATE` U on A_j, which is linearized before $g(S)$ is not v_j. Assume that the value of U is v and let U_j be the `UPDATE` operation, which writes the value v_j read by $g(S)$ to R_j.

By Observation 5 (claim 1), $g(S)$ returns by executing line 34 or line 38 of the pseudocode. Assume that $g(S)$ returns after having executed *epoch* iterations of the while loop. By Observation 4, at the point that $g(S)$ terminates, there is a unique integer ℓ, $1 < \ell \leq epoch$, for which it holds that $checkmarks[\ell] = 0$; moreover, $g(S)$ returns the values it read during its ℓ-th epoch. Let r_1 be the first `read` of $g(S)$ in the ℓ-th epoch and let r_j be the `read` of $g(S)$ on register R_j (that corresponds to component A_j) in the ℓ-th epoch. We proceed by case analysis.

1. Assume first that U_j performs its `write` primitive w_j after r_1. Since $g(S)$ returns v_j, it follows that U_j performs its `write` primitive between r_1 and r_j. By the way linearization points are assigned, the linearization point of U_j is inserted immediately before the linearization point of $g(S)$ and no other `UPDATE` operation on A_j is linearized between U_j and S. (Recall that all `UPDATES` that are linearized immediately before $g(S)$ are on distinct components.) This contradicts our assumption that U is linearized between U_j and $g(S)$.
2. Assume now that w_j precedes r_1. Assume first that U's `write` primitive w follows w_j. Since $g(S)$ returns v_j for A_j, the last `write` primitive to R_j that precedes r_j is w_j. Since w follows w_j, it follows that w follows r_j (see Fig. 4). Since $g(S)$ is linearized immediately before r_1, the linearization point of $g(S)$ precedes w. Since U is linearized before S (and therefore before $g(S)$), it follows that U is not linearized at w. By the way linearization points are assigned, there exists some `SCAN` operation S' such that S' returns v and the linearization point of U is placed immediately before that of $g(S')$. Since S returns v_j and not v for A_j, S' is different from S. Lemma 8 (claim 3) implies that w precedes the end of the execution interval of S'. Because w follows r_1 (and precedes the end of the execution interval of S'), and there is just

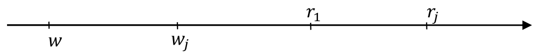

Fig. 5 Case 3 of Lemma 11.

a single `SCAN` operation active at each point in time, it follows that S' follows S.
By Lemma 7 (claim 2), the linearization point of S is within its execution interval. By Lemma 8 (claim 1), the linearization point of U is within the execution interval of S'. It follows that the linearization point of U follows the linearization point of S. This is a contradiction.
Assume next that w precedes w_j (see Fig. 5). By Lemma 9, U is linearized within its own execution interval. Thus, the latest point at which U can be linearized is at its `write` primitive w. Since w precedes w_j and U is linearized between U_j and S (and therefore after U_j), it follows that U_j is not linearized at its `write` primitive. Lemma 8 implies that there exists some `SCAN` operation S'' such that U_j is linearized within the execution interval of S''. If $S = S''$, by the way linearization points are assigned, the linearization point of U_j is placed immediately before that of S and no other `UPDATE` on component A_j can be linearized in between. Since U is linearized between U_j and S, it follows that $S \neq S''$.
By Lemma 9, U_j is linearized within its own execution interval. Since w_j is the last instruction executed by U_j and w_j precedes r_1, it follows that S'' precedes S. Lemma 10 (claims 1 and 2) implies that there exists some `UPDATE` operation U' whose `write` primitive w' follows w_j and precedes the end of the execution interval of S''. It follows that S does not read the value written by U_j in R_j, so it does not return v_j for A_j. A contradiction.

The following theorem is an immediate consequence of Lemmas 7, 9 and 11.

Theorem 6 Checkmarking *is linearizable.*

5.3 Space and Step Complexity

In this section, we study the step complexity of Checkmarking. By inspection of the pseudocode, each `SCAN` and `UPDATE` operation performs only a constant number of shared memory accesses in addition to executing `GetVector`. Therefore, it suffices to prove that the step complexity of `GetVector` is $O(m^2)$.

Consider any execution α of Checkmarking and let g be any instance of `GetVector` executed in α. We prove that g does not execute more than $m+1$ iterations of the while loop of line 20. To derive a contradiction, assume that g executes $m+1$ iterations of the while loop without having terminated. By the pseudocode, it follows that for each i, $2 \le i \le m+2$, there is an integer j_i, $1 \le j_i \le m$, such that $history[i][j_i].change = \texttt{true}$ and no other change has been observed on column j_i of *history* after row i. Therefore, j_i must be distinct for each i. Since the snapshot object has only m components, this is a contradiction. Thus, after at most $m+1$ iterations of the while loop, g completes its execution.

In each iteration of the while loop, m registers are read. `GetVector` additionally executes a set of m reads at the beginning of its execution. Thus, the step complexity of `GetVector` is $O(m^2)$. We remark that the number of instructions executed by each instance of `GetVector` as local computation is also in $O(m^2)$.

Theorem 7 *Checkmarking uses $m+1$ registers and has step complexity $O(m^2)$.*

Algorithm 3 Pseudo-code for T-Opt. (We assume that components store values of type *data*.)

```
shared int seq = 1;
   shared data preVal[1..κ][1..m] =
         {{⊥, ..., ⊥}, ..., {⊥, ..., ⊥}};
   // κ is the number of executed SCANS
   shared data Val[1..m] = {⊥, ..., ⊥};

   void UPDATE(data value, int i) {
      int curr_seq;
      data v1, v2;

1     curr_seq = seq;
2     v1 = Val[i];
3     v2 = preVal[curr_seq][i];
4     if (v2 == ⊥)
5        preVal[curr_seq][i] = v1;
6     Val[i] = value;
   }

   data *SCAN(void) {
      data view[1..m], v1, v2;
      int i;

7     seq = seq+1;
8     for (i = 1; i ≤ m; i++) {
9        v1 = Val[i];
10       v2 = preVal[seq][i];
11       if(v2 == ⊥) view[i] = v1;
12       else view[i] = v2;
      }
      return view;
   }
```

6 The T-Opt Algorithm

In this section, we present T-Opt, the first of the implementations of the Time-efficient family of algorithms. T-Opt is optimal in terms of its step complexity, i.e., it has step complexity $O(m)$ for `SCANS` and $O(1)$ for `UPDATES`. The number of registers that T-Opt uses is linear in the number of `SCANS` it performs in each execution and therefore it is unbounded.

The pseudocode for T-Opt is given in Algorithm 3. T-Opt is described in Section 6.1. Its correctness proof is provided in Section 6.2 and its space and step complexity are studied in Section 6.3

6.1 Description

Each time a `SCAN` starts its execution, the scanner stores a new sequence number in a register *seq* (line 7). In addition, T-Opt uses an array Val of m registers, one for each component.

Any `UPDATE` U on a component A_i, $1 \le i \le m$, writes its value into $Val[i]$ (line 6). Before doing so, it stores (line 5) the current value of $Val[i]$ in some appropriate element of an array of registers, called $preVal$, to help `SCANS` be consistent. Array $preVal$ is a two-dimensional array with each row having m registers; the number of its rows depends on the maximum number of `SCANS` performed in an execution. Specifically, U starts by reading *seq* (line 1) and uses the sequence number that it reads there to determine the row of $preVal$ in the ith entry of which it stores the value of $Val[i]$ (line 5) before it overwrites it (line 6).

We will place the linearization point of each `SCAN` operation, S, at line 7. To ensure consistency, S must ignore the values written by `UPDATES` that start their execution after the beginning of S. To achieve this, S reads all m registers of Val (line 9) and the m registers of $preVal[seq]$ (line 10), where *seq* contains the value written to it by S. We remark that `UPDATES`, which start their execution after S has written to *seq* and before the end of S, write to some register of row *seq* of $preVal$. Therefore, if $preVal[seq][i] \neq \perp$ for some i, $1 \le i \le m$, S should return the old value of $Val[i]$ for component A_i, which is stored in $preVal[seq][i]$ (line 12). `UPDATES` that write to smaller rows of $preVal$ have started their execution before S, so if S reads in Val the value written by such an `UPDATE`, it can include it to the vector it returns (line 11).

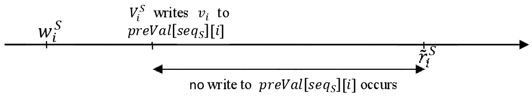

Fig. 6 V_i^S writes v_i to register $preVal[seq_S][i]$ and this `write` is the last to $preVal[seq_S][i]$ that precedes $\tilde{r}_i^S$.

6.2 Linearizability

Let α be any execution of T-Opt and let S be any `SCAN` performed in α. We start by introducing some useful notation. Let w_{seq}^S be the `write` to *seq* performed by S (line 7) and let seq_S be the value written to *seq* by S. Since there is a single-scanner active at each point in time, by inspection of the pseudocode (lines 7-12), we get the following:

Observation 8 *The initial value of seq is* 1 *and seq's value is incremented each time a* `SCAN` *executes line 7.*

For each $i \in \{1, \ldots, m\}$, denote by r_i^S the `read` of $Val[i]$ by S (line 9), and by $\tilde{r}_i^S$ the `read` of $preVal[seq_S][i]$ by S (line 10).

Observation 9 *For each* $i \in \{1, \ldots, m\}$, w_{seq}^S *precedes* r_i^S, *which precedes* $\tilde{r}_i^S$.

Let v_i be the value that S returns for component A_i. In case S reads $\perp$ in $preVal[seq_S][i]$, we denote by U_i^S the `UPDATE` such that U_i^S writes v_i to $Val[i]$ and this `write` is the last to $Val[i]$ that precedes r_i^S. If S reads v_i in $preVal[seq_S][i]$, we introduce the following notation. We denote by V_i^S the `UPDATE` such that V_i^S writes v_i to register $preVal[seq_S][i]$ and this `write` is the last to $preVal[seq_S][i]$ that precedes $\tilde{r}_i^S$ (see Fig. 6). By inspection of the pseudocode (lines 2-5), V_i^S reads the value v_i in $Val[i]$. We denote by U_i^S the `UPDATE` on A_i such that U_i^S writes v_i to $Val[i]$ and this `write` is the last `write` to $Val[i]$ before V_i^S reads $Val[i]$. In either case, let w_i^S be the `write` to $Val[i]$ by U_i^S (line 6). For clarity of presentation, Table 3 summarizes the notation used in this section.

By definition of V_i^S, V_i^S writes into register $preVal[seq_S][i]$. By inspection of the pseudocode (lines 1, 4-5), it follows that it reads a value equal to $\perp$ in $preVal[seq_S][i]$ (line 3) and seq_S in register *seq* (line 1). Moreover, by definition of V_i^S and U_i^S, the `read` of $Val[i]$ by V_i^S follows w_i^S since V_i^S reads in $Val[i]$ the value written there by w_i^S; additionally, $\tilde{r}_i^S$ reads in $preVal[seq_S][i]$ the value written there by V_i^S, so $\tilde{r}_i^S$ follows the `write` to $preVal[seq_S][i]$ by V_i^S.

Observation 10 *For every* $i \in \{1, \ldots, m\}$, *if* S *reads* v_i *in* $preVal[seq_S][i]$, *the following hold:*

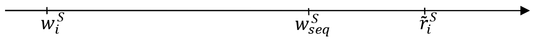

Fig. 7 Proof of Lemma 12, w_i^S precedes w_{seq}^S.

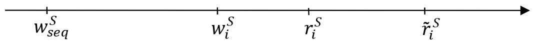

Fig. 8 Proof of Lemma 12, S reads $\perp$ in $preVal[seq_S][i]$.

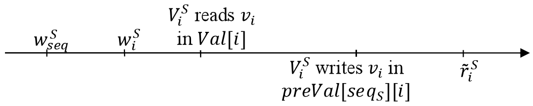

Fig. 9 Proof of Lemma 12, w_i^S precedes r_i^S and S reads v_i in $Val[i]$.

1. *V_i^S reads the value seq_S in register seq and the value $\perp$ in $preVal[seq_S][i]$,*
2. *the* `read` *of $Val[i]$ by V_i^S follows w_i^S, and*
3. *$\tilde{r}_i^S$ follows the* `write` *to $preVal[seq_S][i]$ by V_i^S.*

We now assign linearization points. Each `SCAN` S that terminates in α is linearized at w_{seq}^S. For each $i \in \{1, \ldots, m\}$, if w_i^S (performed by U_i^S) follows w_{seq}^S, we place the linearization point of U_i^S immediately before w_{seq}^S. We also place the linearization point of each `UPDATE` on A_i that performs its `write` to $Val[i]$ between w_{seq}^S and w_i^S immediately before w_{seq}^S; ties are broken by the order that the `write`s to register $Val[i]$ occur. After assigning linearization points to all `SCANS` and to some `UPDATES` (following the rules just described), we linearize each of the rest of the `UPDATES` that performs the `write` to $Val[i]$ (line 6) in α, at this `write`. Let L be the linearization of α determined by assigning linearization points to operations as described above.

We remark that U_i^S uses as a parameter the value v_i returned by S for A_i. Notice that in case w_i^S is executed after w_{seq}^S, we assign linearization points to `UPDATES` in such a way that U_i^S is the last `UPDATE` on A_i that is linearized before S. We later prove (in Lemma 14) that U_i^S and all `UPDATES` that perform their `write`s between w_{seq}^S and w_i^S start their execution before w_{seq}^S. In case U_i^S executes w_i^S before w_{seq}^S, we argue that U_i^S is the last `UPDATE` on A_i that is linearized before S. Intuitively, this is so for the following reasons: (1) by the way linearization points are assigned, for each i, $1 \leq i \leq m$, the linearization order of `UPDATES` on A_i respects the order in which the `write`s to $Val[i]$ of those `UPDATES` have been performed, and (2) by definition of U_i^S, no other `UPDATE` on A_i writes to $Val[i]$ between w_i^S and w_{seq}^S. Thus, S returns a consistent vector with respect to L.

Notation	Description
α	An execution of T-Opt
L	The linearization of α
S	A `SCAN` that terminates in α
w^S_{seq}	The `write` to register *seq* (line 7) performed by S
seq_S	The value written to register *seq* by S
v_i	The value that S returns for component A_i
r^S_i	The `read` of $Val[i]$ performed by S (line 9)
$\tilde{r}^S_i$	The `read` of $preVal[seq_S][i]$ performed by S (line 10)
V^S_i	The `UPDATE` that writes v_i to $preVal[seq_S][i]$; this `write` is the last to $preVal[seq_S][i]$ that precedes $\tilde{r}^S_i$.
U^S_i	The `UPDATE` that writes v_i to $Val[i]$.
w^S_i	The `write` to $Val[i]$ performed by U^S_i (line 6)

Table 3 Notation used in the proof of T-Opt.

We start by proving two simple technical lemmas.

Lemma 12 *For each $i \in \{1, \ldots, m\}$, w^S_i precedes $\tilde{r}^S_i$.*

Proof If w^S_i precedes w^S_{seq} (see Fig. 7), the claim holds because, by Observation 9, w^S_{seq} precedes $\tilde{r}^S_i$. So, assume that w^S_i follows w^S_{seq}. Assume first that S reads $\perp$ in $preVal[seq_S][i]$ and v_i in $Val[i]$. By definition of U^S_i, w^S_i writes the value v_i to $Val[i]$, which is read by S. So, w^S_i precedes r^S_i (see Fig. 8). By Observation 9, r^S_i precedes $\tilde{r}^S_i$. So, w^S_i precedes $\tilde{r}^S_i$.

Assume now that S reads v_i in $preVal[seq_S][i]$. Then, V^S_i is well-defined. Observation 10 (claims 2 and 3) implies that the `read` of $Val[i]$ by V^S_i follows w^S_i and $\tilde{r}^S_i$ follows the `write` primitive to $preVal[seq_S][i]$ by V^S_i (see Fig. 9). By inspection of the pseudocode (lines 2, 5), the `write` primitive to $preVal[seq_S][i]$ by V^S_i follows its `read` of $Val[i]$. Therefore, w^S_i precedes $\tilde{r}^S_i$.

Lemma 13 *Fix any $i \in \{1, \ldots, m\}$ such that S reads v_i in $preVal[seq_S][i]$. If r_v is the* `read` *of $Val[i]$ by V^S_i, then r_v is executed after w^S_{seq}.*

Proof To derive a contradiction, assume that r_v is executed before w^S_{seq}. By inspection of the pseudocode (lines 1-2), the `read` r_s of *seq* by V^S_i precedes r_v. It follows that r_s precedes w^S_{seq}. Since there is only one `SCAN` active at each point in time, Observation 8 implies that r_s reads a value $t < seq_S$. This contradicts Observation 10 (claim 1).

Lemma 14 *For each $i \in \{1, \ldots, m\}$ such that w^S_i follows w^S_{seq}, it holds that any* `UPDATE` *on A_i that performs its* `write` *to $Val[i]$ between w^S_{seq} and w^S_i (including U^S_i) begins its execution before w^S_{seq}.*

Proof To derive a contradiction, assume that there is an `UPDATE` U on A_i that starts its execution after w^S_{seq} and performs its `write` w to $Val[i]$ at or

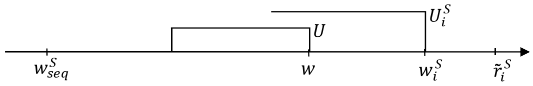

Fig. 10 U starts its execution after w^S_{seq}.

before w^S_i (see Fig. 10). By Lemma 12, w^S_i precedes $\tilde{r}^S_i$ and therefore U ends its execution before the end of S. Since U starts its execution after w^S_{seq}, Observation 8 implies that U reads seq_S in *seq*. By inspection of the pseudocode (lines 2-3), U first reads register $Val[i]$ and then register $preVal[seq_S][i]$. Moreover, in case U reads $\perp$ in $preVal[seq_s][i]$, it writes in $preVal[seq_S][i]$ the value it read in $Val[i]$.

By inspection of the pseudocode, lines 4-5 are executed by U before w and therefore before w^S_i. Since w^S_i precedes $\tilde{r}^S_i$, it follows that the execution of lines 4-5 precedes $\tilde{r}^S_i$. Thus, $\tilde{r}^S_i$ reads a value other than $\perp$ in $preVal[seq_S][i]$, so V^S_i is well-defined. By Observation 10 (claim 2), the `read` of $Val[i]$ by V^S_i follows w^S_i. It follows that the `read` of $preVal[seq_S][i]$ by V^S_i, which (by inspection of the pseudocode) follows its `read` to $Val[i]$, comes after U's execution of lines 4-5 and the possible `write` to $preVal[seq_S][i]$ by U. Thus, V^S_i reads a value other than $\perp$ in $preVal[seq_S][i]$. This contradicts Observation 10 (claim 1).

Using Lemma 14, it can be easily proved that the linearization point of each operation is within its execution interval.

Lemma 15 *The linearization point of each* `SCAN` *that terminates in α and each* `UPDATE` *that executes the* `write` *of line 6 is within its execution interval.*

Proof By the way that linearization points are assigned to `SCANS`, a `SCAN` is linearized within its execution interval. The same is true for each `UPDATE` that is linearized at its `write` primitive to Val.

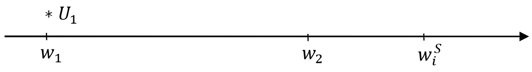

Fig. 11 Case 2 of proof of Lemma 16.

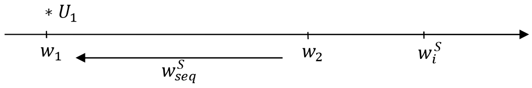

Fig. 12 Case 2 of proof of Lemma 16.

Let U be an `UPDATE` on A_i, which is not linearized at its `write` to $Val[i]$. By the way that linearization points are assigned, there is a `SCAN` S' such that (1) $w_i^{S'}$ of $U_i^{S'}$ is executed after $w_{seq}^{S'}$, (2) the `write` to $Val[i]$ by U is executed between $w_{seq}^{S'}$ and $w_i^{S'}$, and (3) U is linearized immediately before $w_{seq}^{S'}$. Obviously, the execution of U ends after $w_{seq}^{S'}$. Lemma 14 implies that U begins its execution before $w_{seq}^{S'}$. Thus, U is linearized within its execution interval.

To prove that `SCANS` return consistent vectors with respect to L, we first prove that the linearization order of the `UPDATES` on any component A_i respects the order in which these `UPDATES` perform their `writes` to $Val[i]$.

Lemma 16 *Let U_1, U_2 be two* `UPDATE` *operations on some component A_i, $1 \leq i \leq m$. Denote by w_1 the* `write` *to $Val[i]$ by U_1 and by w_2 the* `write` *to $Val[i]$ by U_2. If w_1 precedes w_2, the linearization point of U_1 precedes the linearization point of U_2.*

Proof We consider the following cases.

1. U_2 is linearized at w_2. Lemma 15 implies that U_1 is linearized within its execution interval, so U_1 is linearized at or before w_1. Since w_1 precedes w_2, U_1 is linearized before U_2.
2. U_1 is linearized at w_1 and U_2 is not linearized at w_2 (see Fig. 11). By the way linearization points are assigned, there is a `SCAN` S' such that w_2 has been performed between $w_{seq}^{S'}$ and $w_i^{S'}$. Since w_1 precedes w_2, w_1 has been executed before $w_i^{S'}$. Since U_1 is linearized at w_1, it follows that $w_{seq}^{S'}$ follows w_1 (see Fig. 12), since otherwise U_1 would be linearized at $w_{seq}^{S'}$. Lemma 15 implies that U_1 is linearized the latest at w_1. By the way linearization points are assigned, U_2 is linearized immediately before $w_{seq}^{S'}$. Thus, U_1 is linearized before U_2.
3. Neither U_1 nor U_2 is linearized at its `write` to $Val[i]$. By the way linearization points are assigned, there are two `SCAN` operations S_1 and S_2 such that w_1 has been performed between $w_{seq}^{S_1}$ and $w_i^{S_1}$, and w_2 has been performed between $w_{seq}^{S_2}$ and $w_i^{S_2}$. Since w_1 precedes w_2 and there is just a single `SCAN` active at each point in time, it follows that either $S_1 = S_2$ or S_1 precedes S_2.
   If $S_1 = S_2$, both U_1 and U_2 are linearized immediately before $w_{seq}^{S_1} = w_{seq}^{S_2}$ in the order they perform their `writes` to $Val[i]$. So, U_1 is linearized before U_2.
   If S_1 precedes S_2, the linearization point of U_1, which is placed immediately before $w_{seq}^{S_1}$, precedes the linearization point of U_2, which is placed immediately before $w_{seq}^{S_2}$.

We finally use Lemma 16 to prove consistency.

Lemma 17 *Every* `SCAN` *operation that terminates in α returns a consistent vector with respect to L.*

Proof Consider any `SCAN` operation S that terminates in α. Assume that S returns $\mathbf{v} = \langle v_1, ..., v_m \rangle$. By definition, for each $i \in \{1, \ldots, m\}$, U_i^S writes v_i to $Val[i]$ and therefore it uses v_i as a parameter. In case w_i^S precedes w_{seq}^S, Lemma 15 implies that U_i^S is linearized before S. In case w_i^S follows w_{seq}^S, by the way linearization points are assigned, the linearization point of U_i^S precedes the linearization point of S. Thus, in either case, the linearization point of U_i^S precedes the linearization point of S. We prove that there is no `UPDATE` on component A_i that is linearized between U_i^S and S. This implies that S returns a consistent value for A_i with respect to L.

To derive a contradiction, assume that there is an integer $i \in \{1, \ldots, m\}$ such that the last `UPDATE` on A_i linearized before S is not U_i^S. Denote by U this `UPDATE` and let w be the `write` to $Val[i]$ by U. In case w precedes w_i^S, Lemma 16 implies that U is linearized before U_i^S. This is a contradiction. Thus, assume that w follows w_i^S. We argue that w follows w_{seq}^S by considering the following cases.

1. Assume first that S reads a value equal to $\perp$ in register $preVal[seq_S][i]$. By the definition of w_i^S, r_i^S reads the value that w_i^S writes to register $Val[i]$. Since w follows w_i^S, it follows that w follows r_i^S and therefore also w_{seq}^S (see Fig. 13).
2. Assume next that S reads v_i in $preVal[seq_S][i]$. In this case, V_i^S is well-defined and let r_v be the read of $Val[i]$ by V_i^S. By the definitions of U_i^S and V_i^S, r_v returns the value written by w_i^S and therefore r_v follows w_i^S. Since w follows w_i^S, it follows that w must follow r_v. By Lemma 13, r_v follows w_{seq}^S (see Fig. 14). Therefore, w follows w_{seq}^S.

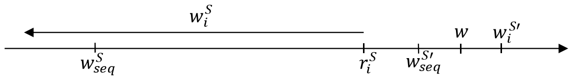

Fig. 13 Case 1 of proof of Lemma 17.

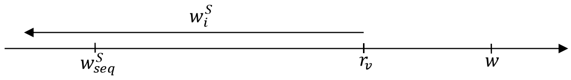

Fig. 14 Case 2 of proof of Lemma 17.

Since U is linearized before S and S is linearized at w_{seq}^S, U cannot be linearized at w. Thus, there is a SCAN S' such that $w_i^{S'}$ follows $w_{seq}^{S'}$, w is performed between $w_{seq}^{S'}$ and $w_i^{S'}$, and U is linearized immediately before $w_{seq}^{S'}$. Since w is performed after w_i^S, $S' \neq S$. Because (1) w is performed between $w_{seq}^{S'}$ and $w_i^{S'}$, (2) w follows w_{seq}^S, and (3) there is just a single SCAN active at each point in time, it follows that S' follows S. Thus, the linearization point of U, which is placed at $w_{seq}^{S'}$, follows the linearization point of S, which is placed at w_{seq}^S. This is a contradiction. We conclude that no UPDATE on component A_i is linearized between U_i^S and S. Thus, S returns a consistent vector with respect to L.

Theorem 11 T-Opt *is linearizable.*

Algorithm 4 Pseudocode for improved version of T-Opt.

```
// initially all m elements are equal to ⊥
    shared pointer sptr[]=new data[m];
    shared data Val[1..m] = {⊥, . . . , ⊥};

    void UPDATE(data value, int i) {

        data *lptr;
        data v1, v2;

        lptr = sptr;
        v1 = Val[i];
        v2 = lptr[i];
        if (v2 == ⊥)
            lptr[i] = v1;
        Val[i] = value;
    }

    data *SCAN(void) {
        data view[1..m], v1, v2;
        int i;

        sptr = new data[m];
        for (i = 1; i ≤ m; i++) {
            v1 = Val[i];
            v2 = sptr[i];
            if(v2 == ⊥) view[i] = v1;
            else view[i] = v2;
        }
        return view;
    }
```

6.3 Step and Space Complexity

By inspection of the pseudocode, it follows that the step complexity of UPDATE is $O(1)$, and the step complexity of SCAN is $O(m)$. Thus, T-Opt is an optimal implementation in terms of its step complexity.

Every register used by T-Opt, other than *seq*, stores just a single value and *seq* stores an integer (that is incremented each time a SCAN takes place). The number of registers used by T-Opt is linear in the maximum number of SCANS performed in any execution and it is therefore unbounded. Thus, in a first glance, T-Opt does not seem to be space-efficient.

Theorem 12 T-Opt *has optimal step complexity,* $O(1)$ *for* UPDATE *and* $O(m)$ *for* SCAN.

We remark that it is easy to implement T-Opt in a more space efficient way as follows. Each time a SCAN S starts executing, the scanner dynamically allocates a new block of m positions in shared memory and sets a pointer *sptr* to point to this block of memory. An UPDATE on A_i starts by reading *sptr* (that plays the role of *seq*); it then saves the value it read in $Val[i]$ in the ith entry of the block of shared memory pointed to by the pointer read in *sptr*. In order to compute the vector to return, S reads the m positions of the block pointed to by *sptr* in addition to the m registers of Val. The pseudocode for the improved version of T-Opt is presented in Algorithm 4.

In Algorithm 4, *seq* has been replaced by a memory pointer and a garbage collector can be used to de-allocate blocks of memory that are not referenced to by the processes. We remark that the total number of allocated blocks that are referenced by all processes at each point in time is at most n. For systems with no garbage collector, more space efficient implementations are presented in later sections.

7 The RT Algorithm

In this section, we present RT, the second implementation of the Time-efficient family. (RT stands for Time-efficient algorithm with Recycling.) RT

Algorithm 5 Pseudocode for RT (process p, $1 \le p \le n$).

```
   shared int seq = 1;
      shared int SeqNums[1..n] = {1,..,1};
      shared data preVal[1..n+2][1..m] = {⊥, . . . , ⊥};
      shared data Val[1..m] = {⊥, . . . , ⊥};

      void UPDATE(data value, int i){
         int curr_seq1, curr_seq2;
         data v1, v2;

1        curr_seq1 = seq;
2        SeqNums[p] = curr_seq1;
3        curr_seq2 = seq;
4        v1 = Val[i];
5        v2 = preVal[curr_seq1][i];
6        if (v2 == ⊥ AND curr_seq1 == curr_seq2)
7           preVal[curr_seq1][i] = v1;
8        Val[i] = value;
      }

      data *SCAN(void) {
         data view[1..m], v1, v2;
         set seq_nums;
         int curr_seq, i;

9        seq_nums = {seq};
10       for (i = 1; i ≤ n; i++)
11          seq_nums = seq_nums ∪ {SeqNums[i]};
12       curr_seq = any int in set ({1,..,n+2} - seq_nums);
13       for (i = 1; i ≤ m; i++) preVal[curr_seq][i] = ⊥;
14       seq = curr_seq;
15       for (i = 1; i ≤ m; i++) {
16          v1 = Val[i];
17          v2 = preVal[seq][i];
18          if(v2 == ⊥) view[i] = v1;
19          else view[i] = v2;
         }
         return view;
      }
```

makes an attempt to reduce the number of registers used by T-Opt. In RT, array $preVal$ has only $n+2$ rows. To achieve this, RT employs an additional array $SeqNums$, of n single-writer registers, one for each process, which are written when `UPDATES` are performed. The pseudocode for RT is presented in Algorithm 5.

An `UPDATE` by some process p records the value it read in seq into register $SeqNums[p]$ (line 2). A `SCAN` S reads all n registers of $SeqNums$ and chooses as its sequence number, seq_S, some index not appearing in any of these registers (lines 10-12). Thus, RT trades the step complexity of `SCANS` (that is now not optimal) for better space complexity.

The main goal of the implementation is to guarantee that only those `UPDATES` that perform the biggest part of their execution after the `write` primitive, w^S_{seq}, to seq by S (line 14), write to registers of row seq_S of $preVal$. This is achieved by employing a technique that resembles handshaking between the scanner and each of the updaters. Each time some process p performs an `UPDATE` operation U, it uses $SeqNums[p]$ to inform the scanner of the value it read in seq (lines 1-2). Then, it reads seq again (line 3) and only if it sees the same value in seq (line 6), does it attempt to write to $preVal$ (line 7).

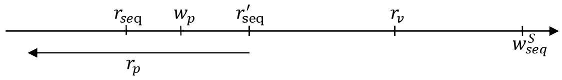

Fig. 15 Lemma 19. r_v is executed before w^S_{seq}.

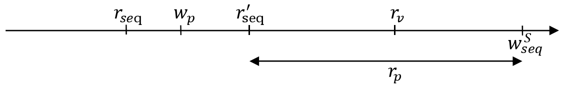

Fig. 16 Lemma 19. r'_{seq} precedes r_p.

If U performs its `write` to $SeqNums[p]$ before S reads $SeqNums[p]$, S will choose a sequence number other than that read by U in seq. If U writes to $SeqNums[p]$ after S has read it and performs its second `read` of seq before w^S_{seq}, then the second `read` of seq by U reads the sequence number of the `SCAN` that precedes S (or the initial value of seq if such a `SCAN` does not exist).

RT guarantees that S chooses a sequence number different from the n numbers that S read in $SeqNums$, and from that chosen by the previous `SCAN` to S, as well as from the initial value of seq. It follows that the number of different values that may be stored into seq is $n+2$ and, therefore, $preVal$ now has $n+2$ different rows.

7.1 Linearizability

Let α be an execution of RT and let S be any `SCAN` performed in α. Let w^S_{seq} be the `write` to seq performed by S (line 14) and let seq_S be the value written to seq by w^S_{seq}. For each $i \in \{1, \dots, m\}$, we introduce the notation r^S_i, $\tilde{r}^S_i$, v_i, U^S_i, V^S_i and w^S_i, and assign linearization points to `SCANS` and `UPDATES` in exactly the same way as we did for T-Opt. Let L be the resulting linearization of α.

The proof of the linearizability of RT is, in its biggest part, similar to the proof of T-Opt (Section 6.2). Specifically, the following two observations, which are similar to Observations 9 and 10, also hold for RT. Lemma 18, which is similar to Lemma 12 from Section 6, also holds for RT.

Observation 13 *For each $i \in \{1, \ldots, m\}$, w_{seq}^S precedes r_i^S and r_i^S precedes $\tilde{r}_i^S$.*

Observation 14 *For every $i \in \{1, \ldots, m\}$ such that S reads v_i in $preVal[seq_S][i]$, the following hold:*

1. *V_i^S reads the value seq_S in register seq and the value $\perp$ in $preVal[seq_S][i]$,*
2. *the `read` of $Val[i]$ by V_i^S follows w_i^S, and*
3. *$\tilde{r}_i^S$ follows the `write` to $preVal[seq_S][i]$ by V_i^S.*

Lemma 18 *For each $i \in \{1, \ldots, m\}$, w_i^S precedes $\tilde{r}_i^S$.*

Lines 9-12 and 14 of the pseudocode imply the following observation.

Observation 15 *Let S and S' be two consecutive `SCANS` in α, it holds that $seq_{S'} \neq seq_S$.*

The statement of the following lemma is similar to that of Lemma 13 but its proof is different than that of Lemma 13, so we present it below.

Lemma 19 *Fix any $i \in \{1, \ldots, m\}$ such that S reads v_i in $preVal[seq_S][i]$. If r_v is the `read` of $Val[i]$ by V_i^S, then r_v is executed after w_{seq}^S.*

Proof To derive a contradiction, assume that r_v is executed before w_{seq}^S (see Fig. 15). Denote by r_{seq} the first `read` of *seq* by V_i^S (line 1) and by r'_{seq} the second `read` of *seq* by V_i^S (line 3). Let p be the process that executes V_i^S, let w_p be the `write` to $SeqNums[p]$ by V_i^S (line 2), and let r_p be the `read` of $SeqNums[p]$ by S (line 11). Since r_v precedes w_{seq}^S, the same is true for r_{seq}, w_p and r'_{seq} (since V_i^S executes these instructions before r_v).

Assume first that r'_{seq} follows r_p (see Fig. 15). Since r'_{seq} precedes r_v, r'_{seq} precedes w_{seq}^S. Let S' be the `SCAN` executed immediately before S in α (or a fictitious `SCAN` that writes the initial value to *seq* if no such `SCAN` exists). By Observation 15, it follows that $seq_{S'} \neq seq_S$. Since r_p and w_{seq}^S are executed by S and r'_{seq} follows r_p and precedes w_{seq}^S, it follows that r'_{seq} reads $seq_{S'}$ in *seq*. By inspection of the pseudocode (lines 1-3, 7), it follows that V_i^S does not write to $preVal[seq_S][i]$. This contradicts the definition of V_i^S.

Assume now that r'_{seq} precedes r_p (Fig. 16). By definition, V_i^S writes to $preVal[seq_S][i]$ the value v_i that is read from there by S. After r_p, S initiates $preVal[seq_S][i]$ to $\perp$ (line 13). Thus, the `write` to $preVal[seq_S][i]$ by V_i^S occurs after r_p. Since w_p precedes r'_{seq}, it follows that w_p precedes r_p. Since V_i^S executes w_p before r_p and its `write` to $preVal[seq_S][i]$ after r_p, it follows that the value t written to $SeqNums[p]$ by w_p (line 2) is the value read by r_p. By the pseudocode (lines 11, 12 and 14), it follows that $seq_S \neq t$. By inspection of the pseudocode (lines 1-3, 7), it follows that V_i^S writes to register $preVal[t][i] \neq preVal[seq_S][i]$. This contradicts the definition of V_i^S.

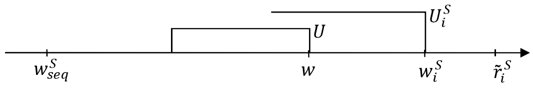

Fig. 17 U starts its execution after w_{seq}^S and finishes before w_i^S.

A big part of the proof of the next lemma follows similar arguments as the proof of Lemma 14.

Lemma 20 *For each $i \in \{1, \ldots, m\}$ such that w_i^S follows w_{seq}^S, it holds that any `UPDATE` on A_i that performs its `write` to $Val[i]$ between w_{seq}^S and w_i^S (including U_i^S) begins its execution before w_{seq}^S.*

Proof To derive a contradiction, assume that there is an `UPDATE` U on A_i that starts its execution after w_{seq}^S and performs its `write` w to $Val[i]$ before w_i^S (see Fig. 17). By Lemma 18, w_i^S precedes $\tilde{r}_i^S$ and therefore U ends its execution before the end of S. Since U starts its execution after w_{seq}^S and ends before the end of S, by inspection of the pseudocode, it follows that U reads seq_S in *seq* both times (on lines 1 and 3). So, the second condition of the if statement of line 6 is evaluated to `true`. By inspection of the pseudocode (lines 4-5), U first reads register $Val[i]$ and then register $preVal[seq_S][i]$. Moreover, in case U reads $\perp$ in $preVal[seq_s][i]$, it writes the value it read in $Val[i]$ to $preVal[seq_S][i]$.

By inspection of the pseudocode (line 13), S initializes the m registers of row seq_S of $preVal$ to the value $\perp$ before w_{seq}^S. Since U starts after w_{seq}^S, the execution of lines 6-7 (i.e., the if statement and the possible `write` to $preVal[seq_s][i]$) by U follows the initialization of $preVal[seq_S][i]$ to $\perp$ by S. By inspection of the pseudocode, lines 6-7 are executed by U before w and therefore before w_i^S. By Lemma 18, w_i^S precedes $\tilde{r}_i^S$. Thus, $\tilde{r}_i^S$ reads a value other than $\perp$ in $preVal[seq_S][i]$, so V_i^S is well-defined. By Observation 14 (claim 2), the `read` of $Val[i]$ by V_i^S follows w_i^S. It follows that the `read` of $preVal[seq_S][i]$ by V_i^S, which (by inspection of the pseudocode) follows its `read` to $Val[i]$, comes after U's execution of lines 6-7 and the possible `write` to $preVal[seq_S][i]$ by U. Thus, V_i^S reads a value other than $\perp$ in $preVal[seq_S][i]$. This contradicts Observation 14 (claim 1).

Lemmas 15-17, which we have proved in Section 6 for T-Opt, hold also for RT without any modification to their proofs.

Lemma 21 *The linearization point of each* `SCAN` *that terminates in α and each* `UPDATE` *that executes the* `write` *of line 8 in α is within its execution interval.*

Lemma 22 *Let U_1, U_2 be two* `UPDATES` *on some component A_i, $1 \leq i \leq m$. Denote by w_1 the* `write` *to $Val[i]$ by U_1 and by w_2 the* `write` *to $Val[i]$ by U_2. If w_1 precedes w_2, the linearization point of U_1 precedes the linearization point of U_2.*

Lemma 23 *Every* `SCAN` *operation that terminates in α returns a consistent vector with respect to L.*

Lemma 23 implies that the following theorem holds for RT.

Theorem 16 RT *is linearizable.*

7.2 Step and Space Complexity

By inspection of the pseudocode, it is obvious that the step complexity of `UPDATE` is $O(1)$ and the step complexity of `SCAN` is $O(n)$.

RT uses $(n+3)m+n+1$ registers; $(n+3)m$ out of these registers (namely, the registers of arrays Val and $preVal$) store just a single value, while the remaining $n+1$ registers (namely, seq and the registers of array $SeqNums$) store $O(\log n)$ bits each (since each of them stores values from the set $\{1, \ldots, n+2\}$).

Theorem 17 RT *uses $(n+3)m+n+1$ registers of bounded size and has step complexity $O(n)$ for* `SCAN` *and $O(1)$ for* `UPDATE`.

8 The RT-Opt Algorithm

In this section, we present RT-Opt, the last implementation of the Time-efficient family of single-scanner, multi-writer snapshots. RT-Opt has step complexity $O(m)$ for `SCAN`, $O(1)$ for `UPDATE` and uses $O(mn)$ registers of bounded size. Thus, RT-Opt improves upon T-Opt in terms of its space complexity. It also improves upon RT in terms of its step complexity.

The pseudocode for RT-Opt is presented in Algorithm 6. RT-Opt is described in Section 8.1. The correctness proof of RT-Opt is provided in Section 8.2 and its space and step complexity are studied in Section 8.3

8.1 Description

The `UPDATE` in RT-Opt is exactly the same as in RT. The major goal of any `SCAN` operation, S, for both RT and RT-Opt is to keep track of the different rows of $preVal$ where old `UPDATES` (i.e., those that have performed some part of their execution before the `write` primitive, w^S_{seq}, of S to seq) may write. S must choose a row of $preVal$ where no such `UPDATE` could possibly write, in order to ensure that all values other than $\perp$ that it reads in $preVal$ have been written by `UPDATES` that have performed the biggest part of their execution after w^S_{seq}.

In RT, this is achieved by having each `SCAN` S read all n registers of $SeqNums$ and choose some value to write into seq other than those read in these registers. Unfortunately, this results in some overhead on the step complexity of `SCAN`. To keep the step complexity of `SCAN` optimal, each `SCAN` in RT-Opt reads only m of the n registers of array $SeqNums$. So, $\lceil n/m \rceil$ consecutive `SCANS` are required to read all n registers of $SeqNums$. We remark that sequence numbers in RT-Opt are chosen from the set $\{1, \ldots, n+2\lceil n/m \rceil+1\}$, which is larger than the set $\{1, \ldots, n+2\}$ used in RT.

We partition each execution α of RT-Opt into execution fragments, called *epochs*, each containing $\lceil n/m \rceil$ consecutive `SCANS`. The scanner keeps track of the values that can be used, as sequence numbers, by `SCANS` of each epoch, in a persistent local variable, called $free$, which implements a set. All sequence numbers chosen by the `SCANS` of an epoch E_j, $j \geq 1$, are distinct (line 25). For the first epoch E_1, all these values are additionally different from the initial value of seq (see initialization of seq and $free$ on lines 3 and 19-21). Consider a later epoch E_j, $j > 1$. Recall that all registers of array $SeqNums$ have been read once during E_{j-1}. All the values read in these registers index rows of $preVal$ where old `UPDATES` may write. So, none of these values should be chosen, as a sequence number, by any `SCAN` of epoch E_j. However, excluding only these values from the set of available sequence numbers for epoch E_j is not sufficient, since some of these values may be already obsolete. This occurs if some process p has started a new `UPDATE` and has written (again) to $SeqNums[p]$ after the `read` of $SeqNums[p]$ during E_{j-1}. Notice that such an `UPDATE` will read in seq the value written there by some `SCAN` of epoch E_{j-1}. So, values chosen as sequence numbers by `SCANS` of epoch E_{j-1} may also index rows of $preVal$ that can be written by old UPDATES, and should be excluded from the set of available sequence numbers for the SCANS of epoch E_j.

Set *candidates* keeps track of all the values that are allowed to be chosen as sequence numbers by SCANS of the next epoch. Notice that at the beginning of each epoch, *candidates* is initialized to contain all possible sequence numbers (line 22). Then, during the execution of the $\lceil n/m \rceil$ SCANS of the epoch, all values read in registers of array $SeqNums$, as well as those chosen as sequence numbers by the SCANS of the epoch, are removed from *candidates* (lines 26 and 29-30). At the beginning of the next epoch, the values remaining in *candidates* can be moved to the set $free$ of available sequence numbers for the epoch (line 21). We remark that no other element is added to $free$ during the epoch.

At the beginning of any execution α of RT-Opt, *candidates* contains $n+2*ScansPerEpoch$ different sequence numbers, where $ScansPerEpoch = \lceil n/m \rceil$. During E_1, at most $n + ScansPerEpoch$ sequence numbers are removed from *candidates*. This is so because the $ScansPerEpoch$ SCAN operations that are executed during E_1, read the n sequence numbers recorded in $SeqNums$ and remove them from *candidates*. The $ScansPerEpoch$ sequence numbers chosen by these SCANS are also removed from *candidates*. So, at the end of epoch E_1, *candidates* contains $ScansPerEpoch$ values, which are added to $free$ at the beginning of E_2. So, $free$ contains enough sequence numbers for the $ScansPerEpoch$ SCANS that are executed during E_2. Consider now any epoch E_j, $j > 1$. At the beginning of E_j (specifically, after line 21 has been executed by the first SCAN of the epoch), *candidates* contains $n+2*ScansPerEpoch+1$ different sequence numbers. During E_j, at most $n + ScansPerEpoch$ sequence numbers are removed from *candidates*. Thus, at least $ScansPerEpoch+1$ sequence numbers are added to $free$ at the beginning of E_{j+1}, which are enough for the SCANS of epoch E_{j+1}. We remark that when line 21 is executed, $free$ and *candidates* may contain elements that are common to both sets. For instance, at the end of E_1, *candidates* is a subset of $free$. From this discussion, it follows that $n + 2 * \lceil n/m \rceil + 1$ different sequence numbers are required in order for RT-Opt to be correct.

8.2 Linearizability

Let α be an execution of RT-Opt and let S be any SCAN performed in α. Let w^S_{seq} be the `write` to seq performed by S (line 28), and let seq_S be the value written to seq by w^S_{seq}. For each $i \in \{1, \ldots, m\}$, we introduce the notation r^S_i, $\tilde{r}^S_i$, v_i, U^S_i, V^S_i and w^S_i, and assign linearization points to SCANS and UPDATES in exactly the same way as we did for T-Opt. Let L be the resulting linearization of α.

The proof of the linearizability of RT-Opt is in its biggest part similar to the proof of T-Opt. Specifically, the following two observations, which are similar to Observations 9 and 10 from Section 6, hold for RT-Opt. The same holds for Lemma 24, which is similar to Lemma 12 (from Section 6) and Lemma 18 (from Section 7).

Observation 18 *For each* $i \in \{1, \ldots, m\}$, w^S_{seq} *precedes* r^S_i *and* r^S_i *precedes* $\tilde{r}^S_i$.

Observation 19 *For every* $i \in \{1, \ldots, m\}$ *such that* S *reads* v_i *in* $preVal[seq_S][i]$, *the following hold:*

1. V^S_i *reads the value* seq_S *in register seq (lines 7 and 9) and the value* $\perp$ *in* $preVal[seq_S][i]$ *on line 11,*
2. *the* `read` *of* $Val[i]$ *by* V^S_i *follows* w^S_i, *and*
3. $\tilde{r}^S_i$ *follows the* `write` *to* $preVal[seq_S][i]$ *by* V^S_i.

Lemma 24 *For each* $i \in \{1, \ldots, m\}$, w^S_i *precedes* $\tilde{r}^S_i$.

As in the correctness proof of RT, the main difficulty in proving that RT-Opt is linearizable is to prove that, for any SCAN S, the UPDATES that write values to row seq_S of $preVal$ have executed the biggest part of their execution after the `write` w^S_{seq} to seq by S. To prove this we need to introduce the following notation.

We split α into epochs so that each epoch contains exactly $\lceil n/m \rceil$ SCANS. Denote by E_j the j-th epoch of α, $j \geq 1$. Epoch E_1 starts with the first instruction of the execution and ends with the last instruction of the $\lceil n/m \rceil$-th SCAN (or $E_1 = \alpha$ if fewer than $\lceil n/m \rceil$ SCANS occur in α). For each $j > 1$, epoch E_j starts at the point that the execution of the $((j-1)\lceil n/m \rceil)$-th SCAN ends and finishes with the last instruction executed by the $(j\lceil n/m \rceil)$-th SCAN (or E_j is the suffix of α, which starts at the point that the execution of the $((j-1)\lceil n/m \rceil)$-th SCAN ends, if fewer than $(j\lceil n/m \rceil)$ SCANS occur in α). Notice that if α contains $(c_1\lceil n/m \rceil + c_2)$ SCANS, where $c_1 \geq 0$ and

Algorithm 6 Pseudocode for RT-Opt (process p).

```
constant ReadsPerScan = m;
constant ScansPerEpoch = ⌈n/ReadsPerScan⌉;

shared int seq = 1;
shared int SeqNums[1..ScansPerEpoch*m]={1,..,1};
shared data Val[1..m]={⊥,..,⊥};
shared data preVal[1..n+2*ScansPerEpoch+1][1..m]={⊥,..,⊥};

void UPDATE(data value, int i){
    int curr_seq1, curr_seq2;
    data v1, v2;

    curr_seq1 = seq;
    SeqNums[p] = curr_seq1;
    curr_seq2 = seq;
    v1 = Val[i];
    v2 = preVal[curr_seq1][i];
    if(v2 == ⊥ && curr_seq1 == curr_seq2)
        preVal[curr_seq1][i] = v1;
    Val[i] = value;
}

data *SCAN(void){
    data view[1..m], v1, v2;
    int curr_seq, i;
    static int cur_period = 0;          // variables that are declared as static
    static set free = ∅;                // are not re-initialized each time SCAN is called
    static set candidates = {2, ..., n+2*ScansPerEpoch+1;}

    if (cur_period == 0) {
        free = free ∪ candidates;
        candidates = {1, ..., n+2*ScansPerEpoch+1};
    }
    curr_seq = any element of set free;
    for (i = 1; i≤m; i++) preVal[curr_seq][i] = ⊥;
    free = free - {curr_seq};
    candidates = candidates - {curr_seq};
    cur_period = (cur_period+1)  mod  ScansPerEpoch;

    seq = curr_seq;
    for (j = 1; j ≤ ReadsPerScan; j++)
        candidates = candidates - { SeqNums[cur_period*ReadsPerScan+j] };
    for (i = 1; i ≤ m; i++) {
        v1 = Val[i];
        v2 = preVal[seq][i];
        if (v2 == ⊥) view[i] = v1;
        else view[i] = v2;
    }
    return view;
}
```

$0 \le c_2 < \lceil n/m \rceil$ are constants, then α contains c_1+1 epochs, where the (c_1+1)-st epoch contains $c_2 < \lceil n/m \rceil$ SCANS. We remark that the (c_1+1)-st epoch may contain only steps by UPDATE operations (if $c_2 = 0$) or may be empty. Let k be the number of epochs in α. (We remark that if α is infinite, the number of epochs in it may be infinite.) For each $j \in \{1, \ldots, k\}$, denote by SN_j the set of values written in register *seq* by any SCAN of epoch E_j and by $free_j$ the set $free$ at the end of E_j. Denote by $candidates_j$ the set *candidates* at the end of E_j.

Next, we prove four simple technical lemmas that are basically direct consequences of the pseudocode.

Lemma 25 *For each $j \in \{1, \ldots, k-1\}$ and for each $p \in \{1, \ldots, n\}$, there is a unique* SCAN *that reads register $SeqNums[p]$ during E_j.*

Proof By definition of E_j, exactly $\lceil n/m \rceil$ SCANS are performed during E_j. Each of these SCANS reads m distinct registers of $SeqNums$ (lines 27, 29, 30). Thus, each of the n registers of $SeqNums$ is read exactly once during E_j.

Lemma 26 *For each $j \in \{1, \ldots, k\}$, it holds that (1) $free_j \cap SN_j = \emptyset$, and (2) $candidates_j \cap SN_j = \emptyset$.*

Proof By inspection of the pseudocode (line 25), each value chosen as the sequence number of some SCAN during E_j is removed from $free$ (line 25); the same is true for *candidates* (line 26). Thus, at the end of epoch E_j it holds that $free_j \cap SN_j = \emptyset$, and $candidates_j \cap SN_j = \emptyset$.

By inspection of the pseudocode (lines 20-22 and 27), lines 21-22 are executed only by the 1-st, $(\lceil n/m \rceil + 1)$-st, ..., $((k-1)\lceil n/m \rceil + 1)$-st SCAN of α, as stated by the following lemma.

Lemma 27 *For each $j \in \{1, \ldots, k\}$, the following hold for the first* SCAN *S executed during E_j:*

1. *S is the only* SCAN *in E_j that adds elements to $free$, and*
2. *S is the only* SCAN *in E_j that executes line 22 to initialize candidates.*

The next lemma states that each SCAN operation S writes to *seq* a value different from the values written to *seq* by the other SCANS of the epoch to which S belongs.

Lemma 28 *For each $j \in \{1, \ldots, k\}$, each* SCAN *of epoch E_j writes a distinct value to seq.*

Proof Fix any $j \in \{1, \ldots, k\}$. Lemma 27 implies that elements are added into $free$ only by the first SCAN of epoch E_j. By inspection of the pseudocode (line 23), each SCAN S of epoch E_j chooses as its sequence number some element of $free$. This element is removed from $free$ when S executes line 25. Thus, SCANS of E_j that are executed after S choose to write different values into *seq*.

The next lemma proves that the SCANS of an epoch choose different sequence numbers than the SCANS of the previous epoch.

Lemma 29 *For each $j \in \{2, \ldots, k\}$, it holds that $SN_{j-1} \cap SN_j = \emptyset$.*

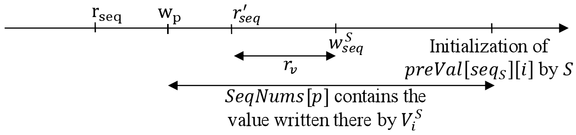

Fig. 18 Case 2 in proof of Lemma 30.

Proof Fix any $j \in \{2, \ldots, k\}$. By Lemma 27, the only SCAN of E_j that adds elements to $free$ is the first SCAN S of E_j. Specifically, S adds the elements of $candidates_{j-1}$ to $free_{j-1}$ by executing line 21. Denote by $free^s_j$ the set $free$ after line 21 has been executed by S. Clearly, $free^s_j = free_{j-1} \cup candidates_{j-1}$. Lemma 26 implies that $free_{j-1} \cap SN_{j-1} = \emptyset$, and $candidates_{j-1} \cap SN_{j-1} = \emptyset$. It follows that $free^s_j \cap SN_{j-1} = \emptyset$. By inspection of the pseudocode (line 23), all elements of SN_j are chosen by $free^s_j$. Thus, $SN_j \cap SN_{j-1} = \emptyset$.

We are now ready to prove a lemma similar to Lemma 19.

Lemma 30 *Fix any $i \in \{1, \ldots, m\}$ such that S reads v_i in $preVal[seq_S][i]$. If r_v is the* `read` *of $Val[i]$ by V_i^S, then r_v is executed after w^S_{seq}.*

Proof To derive a contradiction, assume that r_v is executed before w^S_{seq}. Denote by r_{seq} the first `read` of *seq* by V_i^S (line 7), and by r'_{seq} the second `read` of *seq* by V_i^S (line 9). Let p be the process that executes V_i^S and let w_p be the `write` to $SeqNums[p]$ by V_i^S (line 8). Since r_v precedes w^S_{seq}, the same is true for r'_{seq} (that is executed by V_i^S before r_v). Assume that S is executed in epoch E_j, $j \geq 1$. We proceed by case analysis.

1. Assume first that $j = 1$. By inspection of the pseudocode (line 21), by the way $free$ and *candidates* are initialized (lines 18, 19), and by Lemma 27 (claim 1), it follows that $free$ does not contain the initial value of *seq* during the first epoch. Since SCANS of each epoch choose elements from $free$ as their sequence numbers, S chooses a sequence number different from the initial value of *seq*. Lemma 28 implies that no SCAN that precedes S chooses the same sequence number as S. By definition, V_i^S writes in row seq_S of $preVal$. By inspection of the pseudocode (lines 12-13), this `write` is performed only if both r_{seq} and r'_{seq} read seq_S in *seq*. It follows that r_{seq} and r'_{seq} are both performed after w^S_{seq}. Since r_v follows r'_{seq}, r_v follows w^S_{seq}. This contradicts our assumption that r_v precedes w^S_{seq}.

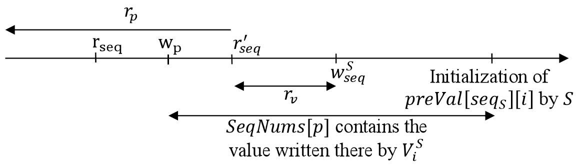

Fig. 19 Case 2a in proof of Lemma 30.

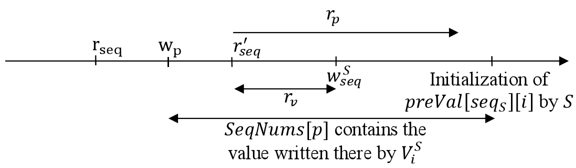

Fig. 20 Case 2b in proof of Lemma 30.

2. Assume now that $j > 1$. By inspection of the pseudocode (line 24), S initializes all m registers of $preVal[seq_S]$ to $\perp$. By definition of V_i^S, S reads the value that V_i^S writes to register $preVal[seq_S][i]$. Thus, V_i^S writes to register $preVal[seq_S][i]$ after the initialization of $preVal[seq_S][i]$ to $\perp$ by S. Since $SeqNums[p]$ is written only by p and V_i^S does not terminate before the initialization of $preVal[seq_S][i]$ by S, it follows that $SeqNums[p]$ contains the value seq_S written by V_i^S from w_p until (at least) the initialization of $preVal[seq_S][i]$ by S.
By Lemma 25, there is a unique `SCAN` operation S' that reads $SeqNums[p]$ during E_{j-1}. Denote by r_p the `read` of $SeqNums[p]$ by S' (see Fig. 18). We consider the following cases.
   (a) r'_{seq} follows r_p (see Fig. 19). By Observation 19 (claim 1), r'_{seq} reads seq_S in seq. Since r'_{seq} follows r_p and r_p is executed by a `SCAN` of epoch E_{j-1}, it follows that r'_{seq} is executed after the beginning of epoch E_{j-1}. By Lemma 29, $SN_{j-1} \cap SN_j = \emptyset$. Thus, since $seq_S \in SN_j$, $seq_S \notin SN_{j-1}$, that is, no `SCAN` of epoch E_{j-1} writes seq_S to seq. By Lemma 28, each `SCAN` of epoch E_j writes a distinct value to seq. So, no `SCAN` of epoch E_j other than S writes seq_S to seq. It follows that the only way for r'_{seq} to read seq_S is if it occurs after w^S_{seq}. Since r_v follows r'_{seq}, it follows that r_v follows w^S_{seq}. This contradicts our assumption that r_v precedes w^S_{seq}.
   (b) r'_{seq} precedes r_p (Fig. 20). Observation 19 (claim 1) implies that r'_{seq} reads seq_S in seq. Assume that S_ℓ is the `SCAN` that writes the value seq_S read by r'_{seq} to seq and let E_ℓ, $\ell \geq 1$, be the epoch in which S_ℓ is executed. If no such `SCAN` exists, then r'_{seq} reads the initial value of seq, so it holds that $seq_S = 1$; moreover, r'_{seq} is executed before the `write` to seq by the first `SCAN` of epoch E_1. In this case, let $\ell = 0$, let $free_0 = \emptyset$, and let $candidates_0 = \{2, \ldots, n + 2 * ScansPerEpoch + 1\}$ (i.e., sets $free_0$ and $candidates_0$ are the initial values of sets $free$ and $candidates$, respectively). Since $seq_S \in SN_j$, Lemma 29 implies that $seq_S \notin SN_{j-1}$. Thus, $0 \leq \ell < j - 1$.
   If $\ell > 0$, let $w^{S_\ell}_{seq}$ be the `write` to seq by S_ℓ. Since r'_{seq} reads the value written by $w^{S_\ell}_{seq}$, r'_{seq} is performed between $w^{S_\ell}_{seq}$ and the `write` to seq by the next `SCAN` after S_ℓ (since $l < j - 1$, such a `SCAN` exists). So, r'_{seq} is executed either during E_ℓ or at the beginning of epoch $E_{\ell+1}$, before the `write` to seq by the first `SCAN` of $E_{\ell+1}$ (this situation may occur if S_ℓ is the last `SCAN` of E_ℓ). (Notice that since $\ell < j - 1$, $E_{\ell+1}$ is either E_{j-1} or an earlier epoch.)
   We prove the following claims.

   Claim 1 For each $f \in \{\ell, \ldots, j - 1\}$, $seq_S \notin candidates_f$.

   Proof Assume first that $f = \ell$. In case $\ell = 0$, recall that $seq_S = 1$ and $candidates_0 = \{2, \ldots, n + 2 * ScansPerEpoch + 1\}$. So, $seq_S \notin candidates_0$. Assume now that $\ell > 0$. Since S_ℓ is executed in epoch E_ℓ and chooses seq_S as its sequence number, $seq_S \in SN_\ell$. By Lemma 29, $candidates_\ell \cap SN_\ell = \emptyset$. Thus, it holds that $seq_S \notin candidates_\ell$.
   Assume now that $f > \ell$. By Lemma 25, $SeqNums[p]$ is read by a unique `SCAN` S_f of E_f. Recall that $SeqNums[p]$ stores the value seq_S from w_p until at least the beginning of S; moreover, r'_{seq} (and therefore also w_p, which is performed by V_i^S before r'_{seq}) is executed before the `write` to seq by the first `SCAN` of epoch $E_{\ell+1}$. By inspection of the pseudocode (lines 28-30), a `SCAN` first writes to seq and then reads some of the registers of array $SeqNums$. Since $\ell < f \leq j - 1$ and S occurs in epoch E_j, it follows that $SeqNums[p]$ contains the value seq_S when S_f reads $SeqNums[p]$. By inspection of the pseudocode (line 30), it follows that seq_S is removed from $candidates$ during E_f. By Lemma 27 (claim 1), no elements are added in $candidates$ after the execution of line 22 by the first `SCAN` of epoch

E_f. Since line 30 follows line 22, it follows that $seq_S \notin candidates_f$.

Claim 2 For each $f \in \{\ell, \ldots, j-1\}$, $seq_S \notin free_f$.

Proof We first prove the claim for $f = \ell$. In case $\ell = 0$, $free_0 = \emptyset$, so $seq_S \notin free_0$. Assume that $\ell > 0$. Since S_ℓ chooses seq_S as its sequence number, $seq_S \in SN_\ell$. Lemma 26 implies that $free_\ell \cap SN_\ell = \emptyset$. Thus, $seq_S \notin free_\ell$.
To derive a contradiction, assume that f, where $\ell < f \leq j-1$, is the smallest integer for which the claim does not hold, i.e., $seq_S \in free_f$. Since the claim holds for $f-1$, it follows that $seq_S \notin free_{f-1}$. By Claim 1, it follows that $seq_S \notin candidates_{f-1}$. Let $free_f^s$ denote set $free$ after the execution of line 21 by the first `SCAN` of epoch E_f. By inspection of the pseudocode (line 21), $free_f^s = free_{f-1} \cup candidates_{f-1}$. It follows that $seq_S \notin free_f^s$. Lemma 27 (claim 1) implies that no elements are added to $free$ after the execution of line 21 and until the end of E_f. Thus, $seq_S \notin free_f$. This is a contradiction.

For $f = j-1$, Claim 1 implies that $seq_S \notin candidates_{j-1}$, and Claim 2 implies that $seq_S \notin free_{j-1}$. By Lemma 27, only the first `SCAN` of epoch E_j adds elements to $free$ by executing line 21 of the pseudocode. Let $free_j^s$ denote set $free$ after the execution of this line. By the pseudocode, it follows that $free_j^s = free_{j-1} \cup candidates_{j-1}$. It follows that $seq_S \notin free_j^s$. All `SCANS` of epoch E_j (including S) choose their sequence numbers from $free_j^s$. Since seq_S does not exist in $free_j^s$, it follows that S cannot choose seq_S as its sequence number. This is a contradiction.

The statement (and the proof) of the following lemma is the same as that of Lemma 20.

Lemma 31 *For each $i \in \{1, \ldots, m\}$ such that w_i^S follows w_{seq}^S, it holds that any `UPDATE` on A_i that performs its `write` to $Val[i]$ between w_{seq}^S and w_i^S (including U_i^S) begins its execution before w_{seq}^S.*

Lemmas 15-17, which we have proved in Section 6 for T-Opt, hold also for RT-Opt without requiring any modification in their proofs:

Lemma 32 *The linearization point of each `SCAN` that terminates in α and each `UPDATE` that executes the `write` of line 14 in α is within its execution interval.*

Lemma 33 *Let U_1, U_2 be two `UPDATES` on some component A_i, $1 \leq i \leq m$. Denote by w_1 the `write` to $Val[i]$ by U_1 and by w_2 the `write` to $Val[i]$ by U_2. If w_1 precedes w_2, the linearization point of U_1 precedes the linearization point of U_2.*

Lemma 34 *Every `SCAN` operation that terminates in α returns a consistent vector with respect to L.*

Lemma 34 implies that the following theorem holds for RT-Opt.

Theorem 20 RT-Opt *is linearizable.*

8.3 Step and Space Complexity

By inspection of the pseudocode, it is obvious that the step complexity of `UPDATE` in RT-Opt is $O(1)$. If in each execution α of RT-Opt, just a single process (always the same) performs the `SCANS` in α, then RT-Opt's step complexity for `SCAN` is $O(m)$. Specifically, each `SCAN` reads $3m$ shared registers, namely, m registers of $SeqNums$ (since it holds that $ReadsPerScan = m$), m registers of $preVal$, and m registers of Val; the rest of the `SCAN` computation is on local variables.

Remarkably, the value of $ReadsPerScan$ can be chosen to be any value between 1 and n. If $ReadsPerScan = n$, RT-Opt works in the same way and has the same step complexity for `SCAN` and `UPDATE` as RT. If $ReadsPerScan = m$ and a single process plays the role of the scanner in the system, RT-Opt achieves optimal step complexity.

RT-Opt uses $O(mn)$ registers. Most of these registers (e.g., the registers of Val and $preVal$) store just one value. The size of each of the rest of the registers is $O(\log n)$ bits.

Theorem 21 RT-Opt *uses $O(mn)$ registers that have bounded size and has step complexity $O(m)$ for `SCAN` and $O(1)$ for `UPDATE`.*

9 Discussion

This paper presents a collection of lower and upper bounds for single-scanner multi-writer snapshot implementations from registers, including the first such implementations that are optimal in terms of step complexity.

An object is called *historyless* if the current state of the object depends only on the last nontrivial primitive that was performed on the object [15]; *nontrivial* is a primitive that can change the state of the object. An example of a historyless object is a swap object. A *swap* object supports, in addition to `read`, the primitive `swap`(v) that changes the state of the object to v and returns the previous value stored in the object. It was proved in [14] that any type of historyless object can be implemented by a `swap` object with the same set of possible states. Moreover, each operation of the historyless object can be simulated by a single access to the swap object. As a consequence, proving a complexity lower bound for implementations from swap objects implies the same lower bound for implementations from historyless objects. It is easy to verify that the proof of our lower bound holds for implementations from swap objects. Thus, our lower bound holds for implementations from historyless objects as well.

Dwork and Waarts [10], have proposed a primitive, called a *traceable register*, which provides the capability of tracing the values that are still active (i.e., those that are currently stored in the shared variables of the system or the local variables of the processes) among those that have been written in the traceable register. A *traceable register* stores a value and supports the operations `tread`, `twrite`, and `garbage-collect`; `tread` and `twrite` are used for reading and writing the register, while `garbage-collect` allows a process to find out which values that have been written into the traceable register are still active. The first traceable register implementation [10] uses $O(n)$ registers and employs handshaking techniques [26] to have processes notify others when they access the register. Specifically, each time `twrite` is invoked by a writer to store a new value into the traceable register, the writer handshakes with all the readers and sets aside (i.e., stores in some of the $O(n)$ registers) the old value of the traceable register for each one of them with which handshaking succeeds. Readers handshake with the writer and depending on whether the handshaking succeeds, they decide whether to return the current value of the register or one of the old values set aside for them. A `garbage-collect` reads all the registers and returns a set of all the values stored in them. This implementation has time complexity $O(n)$ for `twrite`.

Traceable registers could be employed to design a version of T-Opt that is space-bounded. In this version, a `SCAN` would writes into traceable registers in order to specify into which registers `UPDATES` may write their values. These traceable registers are tread by `UPDATE`. Then, the scanner would be able to find out which of the values ever written in the traceable register are still active by performing `garbage-collect`. However, this would lead to an implementation where the time complexity of `SCAN` is $O(n)$ (due to the handshaking). RT-Opt uses a much simpler recycling technique that is based on the standard read-write-read approach. This avoids handshaking and leads to optimal `SCAN` complexity.

Garbage collection in [10] is an expensive task because there are many processes that can perform `twrite`; so a value that appears for the first time in a traceable register may be later written to some other traceable register and be read from there (i.e., the degree of *indirection* can be greater than one). Dwork and Waarts [10] remark that `garbage-collect` can be executed more efficiently if values that are supposed to be active are gradually collected during the execution of more than one `twrite`. In a space bounded version of T-Opt using a traceable register, the degree of indirection is one. As a consequence of this, all the information collected during a `garbage-collect` is local to `SCAN`, so that the execution of `garbage-collect` has no influence on the time complexity of `SCAN` even if it is not performed gradually (despite this, the time complexity of `SCAN` is linear in n due to handshaking). However, the technique of gradually collecting information about values written by `SCANS` that may still be active is useful in RT-Opt, which owes its good time complexity mainly to such a technique. However, space bounded versions of T-Opt employing this implementation do not achieve time complexity less than $\Theta(n)$ for `SCAN`.

An interesting problem left open by our work is to derive a lower bound on the number of read-write registers that are needed to design an implementation that ensures step complexity $O(m)$ for `SCAN` and $O(1)$ for `UPDATE`. Is there an algorithm with this step complexity that uses less than $\Theta(mn)$ registers?

Checkmarking has the same step complexity as a space-optimal single-scanner snapshot implementation. However, in contrast to such an implementation, it uses an additional single-writer register and allows `SCANS` to write to this register. It is interesting to investigate whether the $\Omega(m^2)$ lower bound still holds on the step complexity of `SCAN`, for single-scanner implementations in which `SCAN` is allowed to write to a single-writer register.

Acknowledgments. This research has been supported by the European Commission through the TransForm, Euroserver, and HiPEAC3 projects. It has also been supported by the ARISTEIA Action of the Operational Programme Education and Lifelong Learning through the GreenVM project, and by the project "Computer Science Studies at the University of Ioannina" of the Operational Program for Education and Initial Vocational Training funded by the 3rd Community Support Framework and the Hellenic Ministry of Education.

We would like to thank Faith Ellen for her valuable comments on an early version of this work. We would also like to thank Prasad Jayanti for pointing out some related work.

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic Snapshots of Shared Memory. Journal of the ACM (JACM) **40**(4), 873–890 (1993)
2. Anderson, J.H.: Composite registers. Distributed Computing **6**(3), 141–154 (1993)
3. Anderson, J.H.: Multi-writer composite registers. Distributed Computing **7**(4), 175–195 (1994)
4. Aspnes, J.: Time-and space-efficient randomized consensus. In: Proceedings of the ninth annual ACM symposium on Principles of distributed computing, pp. 325–331. ACM (1990)
5. Aspnes, J., Herlihy, M.: Wait-Free Data Structures in the Asynchronous PRAM Model. In: Proc. 2nd ACM Symposium on Parallel Algorithms and Architectures, pp. 340–349 (1990)
6. Attiya, H., Ellen, F., Fatourou, P.: The complexity of updating snapshot objects. J. Parallel Distrib. Comput. **71**(12), 1570–1577 (2011)
7. Attiya, H., Lynch, N., Shavit, N.: Are Wait-free Algorithms Fast? Journal of the ACM (JACM) **41**(4), 725–763 (1994)
8. Attiya, H., Rachman, O.: Atomic Snapshots in $O(n \log n)$ Operations. SIAM J. Comput. **27**(2), 319–340 (1998)
9. Burns, J., Lynch, N.: Bounds on shared memory for mutual exclusion. Information and Computation **107**(2), 171–184 (1993)
10. Dwork, C., Waarts, O.: Simple and efficient bounded concurrent timestamping and the traceable use abstraction. Journal of the ACM (JACM) **46**(5), 633–666 (1999)
11. Ellen, F., Fatourou, P., Ruppert, E.: Time lower bounds for implementations of multi-writer snapshots. Journal of the ACM (JACM) **54**(6), 30 (2007)
12. Fatourou, P., Fich, F., Ruppert, E.: Space-optimal multi-writer snapshot objects are slow. In: Proceedings of the twenty-first annual symposium on Principles of distributed computing, pp. 13–20. ACM (2002)
13. Fatourou, P., Fich, F., Ruppert, E.: A tight time lower bound for space-optimal implementations of multi-writer snapshots. In: Proceedings of the 35th ACM Symposium on Theory of Computing, pp. 259–268 (2003)
14. Fatourou, P., Fich, F., Ruppert, E.: Time-space tradeoffs for implementations of snapshots. In: Proceedings of the 38th ACM Symposium on Theory of Computing (2006)
15. Fich, F., Herlihy, M., Shavit, N.: On the space complexity of randomized synchronization. Journal of the ACM (JACM) **45**(5), 843–862 (1998)
16. Gafni, E., Merritt, M., Taubenfeld, G.: The concurrency hierarchy, and algorithms for unbounded concurrency. In: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing, pp. 161–169. ACM (2001)
17. Gawlick, R., Lynch, N., Shavit, N.: Concurrent timestamping made simple. In: Proceedings of the Israel Symposium on the Theory of Computing and Systems, *LNCS*, vol. 601, pp. 171–183 (1992)
18. Herlihy, M., Wing, J.M.: Linearizability: A Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems **12**(3), 463–492 (1990)
19. Inoue, M., Chen, W., Masuzawa, T., Tokura, N.: Linear Time Snapshots Using Multi-writer Multi-reader Registers. In: Distributed Algorithms, 8th International Workshop, *LNCS*, vol. 857, pp. 130–140 (1994)
20. Israeli, A., Shaham, A., Shirazi, A.: Linear-time snapshot implementations in unbalanced systems. Mathematical Systems Theory **28**(5), 469–486 (1995)
21. Jayanti, P.: F-arrays: implementation and applications. In: Proceedings of the 21th ACM Symposium on Principles of Distributed Computing, pp. 270–279 (2002)
22. Jayanti, P.: An Optimal Multi-Writer Snapshot Algorithm. In: Proceedings of the 37th ACM Symposium on Theory of Computing, pp. 723–732 (2005)
23. Jayanti, P., Tan, K., Toueg, S.: Time and Space Lower Bounds for Nonblocking Implementations. SIAM J. Comput. **30**(2), 438–456 (2000)
24. Kirousis, L.M., Spirakis, P., Tsigas, P.: Reading many variables in one atomic operation: solutions with linear or sublinear complexity. IEEE Transactions on Parallel and Distributed Systems **5**(7), 688–696 (1994)
25. Mullender, S.: Distributed Systems. Addison-Wesley (1994)
26. Peterson, G.L.: Concurrent reading while writing. ACM Transactions on Programming Languages and Systems (TOPLAS) **5**(1), 46–55 (1983)
27. Rianny, Y., Shavit, N., Touitou, D.: Towards a practical snapshot algorithm. Theoretical Computer Science **269(1-2)**, 163–201 (2003)

A Proofs of Lemmas 1, 2, and 3

Consider any implementation of an m-component multi-writer snapshot object shared by $n > m+1$ processes from a set of m multi-writer read/write registers. The statements of Lemmas 35 and 36 and their proofs are slightly modified versions of similar lemmas that appear in [12]. Lemmas 37, 38, 39 and 40 and their proofs are exactly the same as their analogs from [12]. For the shake of simplicity, our proofs below assume that there is a unique process p_s that performs `SCAN` operations in the system. (We remark that the lemmas hold even if `SCAN` operations are executed by different processes provided that no pair of `SCANS` overlap.)

For the shake of simplicity, in this section, we assume that an execution is a sequence of steps. Fix any execution α of a single-scanner, multi-writer, m-component snapshot implementation from m registers starting from C_0.

Lemma 35 *Suppose that, in configuration C, a set P_O of at most $n-1$ processes covers a set of registers O, and all processes not in P_O are inactive. Furthermore, suppose there is some component A_i such that no process has a pending* `UPDATE` *to A_i in configuration C. Consider an execution starting from C in which the processes in P_O execute a step each to perform their* `writes` *and, immediately afterwards, the scanner p_s performs a solo execution in which it finishes its pending operation (if it has one) and then performs a complete* `SCAN`. *Let v be the value that this* `SCAN` *returns for component A_i. Then, for all $p \notin P_O \cup \{q\}$ and all $v' \neq v$, the solo execution by p of* `UPDATE`(i, v') *starting from C must perform a* `write` *to a register outside O.*

Proof Suppose not. Let C' be the configuration obtained from C when the processes in P_O perform one step each and let β be the solo execution by p_s starting from C'. Let C'' be the configuration obtained when p performs a solo execution of `UPDATE`(i, v') starting from C and then the processes in P_O execute a step each to perform their `writes`. By our assumption, p does not write to any register outside O, so each register has the same value in C'' that it has in C'. Furthermore, p_s is in the same state in C' and C''. Therefore, the solo execution β by p_s starting from C'' is legal and p_s's `SCAN` returns the value v for component A_i. However, the execution β starting from C'' must return the value $v' \neq v$ for component A_i, since p completed its `UPDATE`(i, v') before the `SCAN` began and no process has a pending `UPDATE` to A_i at C. This is a contradiction.

For any configuration C and for any set of processes P', the set of components with a pending `UPDATE` in C by a process in P' is denoted $CPU(C, P')$.

Definition 1 Consider any integer ℓ, where $1 \leq \ell \leq m < n$. A configuration C is *ℓ-fatal* if there exists a subset O of ℓ registers and a set P_O of ℓ processes such that P_O covers O in C and $|CPU(C, P_O)| < \ell$.

Lemma 36 *No implementation for n processes of an m-component snapshot object from m registers has a reachable ℓ-fatal configuration, for $1 \leq \ell \leq m < n-1$.*

Proof Suppose the lemma is false. Let ℓ be the largest integer such that there is a reachable ℓ-fatal configuration, C_1. Then there is a set O of ℓ registers and a set P_O of ℓ processes such that P_O covers O and $|CPU(C_1, P_O)| < \ell$. Let C be the configuration obtained from C_1 by running all processes not in P_O until they are inactive. Since it holds that $|CPU(C, P_O)| = |CPU(C_1, P_O)| < \ell \leq m$, there exists a component $A_i \notin CPU(C, P_O)$.

Let $p \notin P_O$ be any process other than p_s. These process exists because $|P_O| = \ell$ and $1 \leq \ell < n-1$. Consider the execution starting from C in which the processes in P_O execute a step each to perform their `writes`, p_s finishes its pending operation (if any), and then p_s performs a complete `SCAN`. Let v be the value that this `SCAN` returns for component A_i. By Lemma 35, for all $v' \neq v$, the solo execution of `UPDATE`(i, v') to A_i by p starting from C contains a `write` to a register $R \notin O$.

If $\ell = m$, then we have a contradiction, since all registers are in O. Otherwise, $l < m$. In this case, let C_2 be the reachable configuration obtained by performing p's solo execution of `UPDATE`(i, v') starting from C until just before p writes to R for the first time. Let $O' = O \cup \{R\}$ and let $P'_O = P_O \cup \{p\}$. Then $|O'| = |P'_O| = \ell + 1$, P'_O covers O' in C_2, and $CPU(C_2, P'_O) = CPU(C, P_O) \cup \{A_i\}$, so $|CPU(C_2, P'_O)| < \ell+1$. Thus, C_2 is a reachable $(\ell+1)$-fatal configuration, contradicting the maximality of ℓ.

Lemma 37 `SCAN` *operations never perform* `writes`.

Proof Suppose there is an execution of a `SCAN` operation by process p_s that contains a `write` to a register R. Consider the configuration C that occurs just before this `write` is performed. Since $\{q\}$ covers $\{R\}$ and $CPU(C, \{q\})$ is empty, this configuration is 1-fatal, contradicting Lemma 36.

A solo `SCAN` starting from C_0 returns $\perp$ for every component. For each process p_i other than p_s, each component A_j, and each possible value $v \neq \perp$, consider the solo execution of an `UPDATE` of component A_j with value v by process p_i starting from C_0. Since all processes are inactive in C_0, we can apply Lemma 35 with $O = P_O = \emptyset$ to see that this execution by p_i contains at least one `write` to a register. Denote by $R_i(j, v)$ the first register written by p_i and denote by $\rho_i(j, v)$ the prefix of this execution up to, but not including this first `write`. (The sequence $\rho_i(j, v)$ may be empty.)

Lemma 38 *Consider any component A_j. For any processes p_{i_1} and p_{i_2} other than p_s, and for any non-$\perp$ values v_1 and v_2, $R_{i_1}(j, v_1) = R_{i_2}(j, v_2)$.*

Proof Assume first that $p_{i_1} \neq p_{i_2}$. Consider the execution $\rho_{i_1}(j, v_1) \cdot \rho_{i_2}(j, v_2)$ starting from C_0 and let C be the resulting configuration. This execution is legal since p_{i_1} performs no `writes` during $\rho_{i_1}(j, v_1)$. Note that $\{p_{i_1}, p_{i_2}\}$ covers $\{R_{i_1}(j, v_1), R_{i_2}(j, v_2)\}$ in C and $CPU(C, \{p_{i_1}, p_{i_2}\}) = \{A_j\}$. If $R_{i_1}(j, v_1) \neq R_{i_2}(j, v_2)$, then C is 2-fatal. This contradicts Lemma 36. Hence $R_{i_1}(j, v_1) = R_{i_2}(j, v_2)$.

Assume now that $p_{i_1} = p_{i_2}$. Let p_i be any other process. By the argument above, $R_i(j, v_1) = R_{i_1}(j, v_1)$ and $R_i(j, v_1) = R_{i_2}(j, v_2)$. Hence $R_{i_1}(j, v_1) = R_{i_2}(j, v_2)$.

Lemma 38 allows us to define R_j to be the register $R_i(j, v)$ covered by each process p_i other than p_s, immediately after it executes $\rho_i(j, v)$ starting from C_0, for any value $v \neq \perp$. That is, every process (other than p_s) does its first `write` to R_j when it performs any solo `UPDATE` to A_j (with a non-$\perp$ value) starting from C_0.

Lemma 39 *Let α be an execution starting from C_0 in which some process other than p_s takes no steps. Then, for each $j \in \{1, \ldots, m\}$,* `UPDATE` *operations to component A_j in α write only to R_j.*

Proof Suppose there is a process p_i other than p_s that performs a `write` to a register $R \neq R_j$ during the execution of an `UPDATE` to component A_j in α. Let α' denote the prefix of α up to, but not including this `write` by p_i to register R.

Let p_k be a process other than p_s that takes no steps in α and let v be a non-$\perp$ value. Consider the execution $\rho_k(j, v) \cdot \alpha'$ and let C' be the resulting configuration. This execution is legal since p_k performs no `writes` during $\rho_k(j, v)$. Note that $\{p_i, p_k\}$ covers $\{R, R_j\}$ in C' and since it holds that $CPU(C', \{p_i, p_k\}) = \{A_j\}$, it follows that C' is 2-fatal. This contradicts Lemma 36.

The next result shows that processes, which perform `UPDATE` operations to different snapshot components must write to different registers.

Lemma 40 $R_{j_1} \neq R_{j_2}$ *for distinct $j_1, j_2 \in \{1, \ldots, m\}$.*

Proof To derive a contradiction, suppose $R_{j_1} = R_{j_2}$ for some $j_1 \neq j_2$. Let p_{k_1} and p_{k_2} be two distinct processes other than p_s. Let v be some non-$\perp$ value. Let C be the configuration that results when $\rho_{k_2}(j_2, v)$ is performed by p_{k_2} starting from C_0. In configuration C, $\{p_{k_2}\}$ covers $\{R_{j_2}\}$, all other processes are inactive, and no process has a pending `UPDATE` to A_{j_1}. Let C' be the configuration obtained from C by allowing p_{k_2} to do its pending `write`. A solo `SCAN` by process p_s starting from C' returns $\perp$ for component A_{j_1}, since no `UPDATES` to A_{j_1} have been started in this execution. Let α be the solo execution of `UPDATE`(j_1, v) by p_{k_1} starting from C. By Lemma 35, p_{k_1} must write to some register other than $R_{j_1} = R_{j_2}$ during α.

Since p_{k_2} performs no `writes` during $\rho_{k_2}(j_2, v)$, it is also the case that α is a legal execution starting from C_0. Process p_{k_2} takes no steps during α, so Lemma 39 implies that p_{k_1} writes only to R_{j_1} during α. This is a contradiction.