Time-Optimal, Space-Efficient Single-Scanner Snapshots & Efficient Multi-Scanner Snapshots using CAS

P. Fatourou, N.D. Kallimanis

TECHNICAL REPORT
DCS 2007-02
Time-Optimal, Space-Efficient Single-Scanner Snapshots & Efficient Multi-Scanner Snapshots using CAS

Panagiota Fatourou  
Department of Computer Science  
University of Ioannina  
faturo@cs.uoi.gr

Nikolaos D. Kallimanis  
Department of Computer Science  
University of Ioannina  
nkallima@cs.uoi.gr

Abstract

Snapshots are fundamental shared objects which provide consistent views of blocks of shared memory. A snapshot object consists of an array of \( m \) memory cells and allows processes to execute UPDATEs to write new values in any of the snapshot cells, and SCANs to return consistent views of all \( m \) cells. An interesting (weaker) form of snapshot with several applications is a single-scanner snapshot which allows to only one process, called scanner, to execute SCANs (UPDATEs can still be executed concurrently).

We present the first time-optimal, linearizable, wait-free, single-scanner snapshot implementations from read-write registers for an asynchronous system of \( n \) processes. Our first algorithm is very simple and has time complexity \( O(1) \) for UPDATE and \( O(m) \) for SCAN (which is optimal). However, in systems with no garbage collector, the number of registers it uses is proportional to the number of executed SCANs. Our second implementation employs an interesting recycling technique to reduce the space complexity to \( O(mn) \) bounded-size registers still achieving optimal time complexities for both operations. For systems that provide stronger primitives, like Compare-And-Swap (CAS), we provide a multi-scanner snapshot implementation that uses \( m \) CAS registers and \( m \) read-write registers, and achieves time complexity \( O(1) \) for UPDATE and \( O(m) \) for SCAN. The presented algorithms are simple and practical, and improve upon all previously presented algorithms in terms of time and/or space complexity.

Keywords: snapshots, single-scanner, multi-writer, linearizable objects, wait-free implementations, Compare-And-Swap (CAS), asynchronous shared memory systems, distributed algorithms
1 Introduction

A fundamental problem in asynchronous, shared-memory systems is to obtain an instantaneous view of a block of shared memory while concurrently processes may be updating its cells. Snapshots are shared objects aiming in providing such consistent views. A snapshot object consists of an array of m components and supports two operations, UPDATES to change the value of any component and SCANS to obtain instantaneous views of all components. Snapshots can be used to record the state of a system as it is changing, so they facilitate problems that need to perform an action when the state of the system satisfies some condition [23]; such problems are termination detection, garbage collection, and dynamic adaptation of a program's configuration, e.g., for load balancing. Snapshots have been extensively used for the design and verification of several distributed algorithms, e.g., the construction of concurrent timestamps [14], approximate agreement [8], randomized consensus [4], check-pointing and restarting [23], and the design of complex distributed data structures [5].

In their general form, snapshots are multi-writer (each process can UPDATE any component). The more restricted form of single-writer snapshots (where only one process can UPDATE a component) has been also studied [1, 2, 9, 16, 17], but recently more attention has been given to implementations of multi-writer snapshots from registers [3, 10, 11, 12, 18, 19]. A read-write register stores a value that supports the atomic read and write of its contents. A register can be multi-writer if all processes can write it or single-writer if only one process writes it.

A snapshot implementation is evaluated in terms of the number (and size) of registers it requires, and of its time complexity, expressed by the maximum number of steps taken by a process in any execution to perform a SCAN or an UPDATE. The advantages of snapshots can be exploited only if it is possible to design implementations that achieve good complexity. All current snapshot implementations (single-writer or multi-writer) from read-write registers have time complexity for SCAN and UPDATE at least linear to n, where n is the number of processes in the system. The best multi-writer snapshot implementation [7] from read-write registers uses $O(n^2)$ registers and has time complexity $O(n)$ for SCAN and UPDATE. Recently, it has been proved [12] that any multi-writer snapshot implementation from any fixed number of multi-writer read-write registers, the time required to do SCANS grows without bound as n increases. This result implies that the time complexity of snapshot implementations from read-write registers cannot be better than a function of n even in systems where the number of snapshot components m is much smaller than n.

This lower bound can be beaten if one restricts to the weaker form of single-scanner snapshots [22, 24, 19, 13]. In a single-scanner snapshot, only one process, the scanner, performs SCANS at any point in time. Single-scanner snapshots have several applications like garbage collection, generating backups, etc., and therefore studying their complexity is very interesting. Jayanti, Tan and Toueg [21] have presented a lower bound of $\Omega(n)$ on the time complexity of SCAN for implementations of perturbable objects from read-write registers. They prove that single-writer snapshots are perturbable [21]. Their proof does not use more than one scanner. A multi-writer snapshot trivially implements a single-writer snapshot for m processes. This implies a lower bound of $\Omega(m)$ on the time complexity of single-scanner, multi-writer snapshots.

The main contribution of this paper is the presentation of the first time-optimal single-scanner, multi-writer snapshot implementations from read-write registers. Our implementations have time complexity $O(m)$ for SCAN and $O(1)$ for UPDATE and use bounded-size registers. The first implementation (Section 3), called T-Opt (Time-Optimal), is the simplest but in systems with no garbage collector, the number of registers it employs is $O(m \cdot \kappa)$, where $\kappa$ is the number of executed SCANS (which might be unbounded). In order to design a more space efficient algorithm, we first employ a relatively simple recycling technique to get an implementation (Section 4) from $O(mn)$ bounded-size registers, called RT, which still has time complexity $O(1)$ for UPDATE but the time complexity
for SCAN increases to \( O(n) \). We then introduce a more advanced recycling technique to get an implementation (Section 5), called RT-Opt (Recycled T-Opt), that uses \( O(mn) \) bounded-size read-write registers and achieves optimal time complexity. RT is middle ground between T-Opt and RT-Opt; its design provides intuition for RT-Opt and simplifies its presentation.

The first single-scanner, multi-writer snapshot implementations from read-write registers were presented by Kirsous, Spirakis and Tsigas [22]. Their first algorithm uses an unbounded number of registers and has unbounded time complexity for SCAN. A register recycling technique which leads to an implementation that uses \( O(mn) \) bounded-size registers and has time complexity \( O(mn) \) for SCAN and \( O(1) \) for UPDATE, is also presented in [22]. For single-writer snapshots, a simplified version of this algorithm uses \( n+1 \) single-writer registers of unbounded size and has time complexity \( O(n) \) for SCAN and \( O(1) \) for UPDATE [24]. Jayanti [19] has presented a single-scanner, single-writer snapshot implementation from \( O(n) \) bounded-size single-writer registers that has time complexity \( O(n) \) for SCAN and \( O(1) \) for UPDATE.

Fatourou and Kallimanis [13] have presented the first implementations of single-scanner, multi-writer snapshots from read-write registers whose time complexities are (linear or quadratic) functions only of the number of components and not of the number of processes (as for the algorithms above). Their first algorithm, called SweepLine, has time complexity \( O(m^2) \) for SCAN and UPDATE and uses \( m+1 \) registers of unbounded size (each of them stores a sequence number that can be as large as the number of executed SCANs); moreover, \( m \) of those registers store a vector of \( m \) values. The second algorithm [13], called Linear, achieves time complexity \( O(m) \) for both SCAN and UPDATE but uses an unbounded number of registers, most of which store a vector of \( m \) values. These implementations are theoretically interesting since they are the first to achieve time complexities that are not functions of \( n \), but they are impractical due to the size of the registers they use. The design of practical single-scanner multi-writer algorithms is the challenge that we undertake in this work. T-Opt and RT-Opt employ only bounded-size registers each of which stores a value; they also significantly improve upon these algorithms in terms of time complexity, achieving optimal time complexity for both SCAN and UPDATE. A practical snapshot implementation should keep the time complexity of UPDATE within a small constant of that of a simple memory write since it is not often desirable to increase the complexity of updating memory for supporting SCANS. Our algorithms respect this property by having UPDATEs perform a small number of shared memory accesses.

Like in Linear (and in the algorithms of [22]), SCANS in T-Opt have the responsibility of determining the appropriate registers where UPDATEs should write their values. T-Opt borrows also the idea of using an array of an unbounded number of registers from these algorithms. However, both SCANS and UPDATEs in T-Opt employ significantly simpler techniques than Linear to achieve optimal time complexity and use registers that store only a single value. RT-Opt has additionally the advantage of using only a bounded number of registers. The recycling technique employed by RT-Opt is based on the philosophy of recycling blocks of \( m \) registers, and not one register for each component as proposed in [22]. Our technique is completely different and much simpler than the recycling technique of [22]. It allows sacrificing space for time and vice versa, and we believe that it is of independent interest.

Attiya, Ellen and Fatourou [6] proved a lower bound of \( \Omega(m) \) on the time 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 but has time complexity for UPDATE \( O(1) \). So, the lower bound of [6] can be beaten if we restrict to single-scanner snapshot implementations.

If stronger primitives than read-write registers are employed, like Compare-And-Swap (CAS) and Load-Link/Store-conditional (LL/SC), the lower bound on the time complexity of multi-writer
snapshots presented in [12] does not hold [19]. A CAS register \( O \) stores a value and supports the atomic operation \( \text{CAS}(O, v, v_n) \) which reads the value of \( O \) and if it equals to \( v_n \), it changes it to \( v \). An LL/SC register \( O \) supports the atomic operations LL and SC. LL(\( O \)) returns the current value of \( O \); the execution of SC(\( O, v \)) by a process \( p \) must follow the execution of LL(\( O \)) by \( p \), and it is successful only if no process performed a successful SC on \( O \) since the execution of \( p \)'s latest LL on \( O \); if SC(\( O, v \)) is successful the value of \( O \) changes to \( v \) and true is returned. Otherwise, the value of \( O \) does not change and false is returned.

Based on T-Opt, we design a multi-scanner, multi-writer snapshot implementation employing CAS registers. The implementation, called C-Snap (CAS-Snapshot), has time complexity \( O(1) \) for \text{UPDATE} and \( O(m) \) for \text{SCAN} and uses only \( m + 1 \) CAS registers and \( m \) read-write registers. The \( m \) read-write registers store just a single value, and the \( m \) out of the \( m + 1 \) CAS registers store a sequence number and a value. The last register used by C-Snap is big. It stores a vector of \( m \) values and a sequence number.

Jayanti [18] has presented a multi-writer snapshot implementation that has time complexity \( O(nm) \) for \text{UPDATE} and \text{SCAN} and uses \( O(mn^2) \) CAS registers. Recently, Jayanti [19] has presented another multi-writer snapshot implementation that achieves time complexity \( O(m) \) for \text{SCAN} and \( O(1) \) for \text{UPDATE}, and uses \( O(mn) \) LL/SC registers. One of the LL/SC registers is big, since it stores a vector of values. Using the algorithm in [20], these LL/SC registers can be simulated using \( O(mn^2) \) single-word CAS registers. C-Snap achieves the same time complexity as Jayanti's algorithm [19] but improves upon it on the number of CAS registers it uses. C-Snap is the first multi-writer snapshot implementation that employs a number of CAS registers that does not depend on \( n \). However, C-Snap has not avoided the use of a big register. Designing an algorithm that achieves time complexity \( O(1) \) for \text{UPDATE} and \( O(m) \) for \text{SCAN} using \( O(m) \) single-word CAS registers is an interesting open problem.

C-Snap has been designed based on T-Opt which is a multi-writer snapshot implementation. The high-level idea is to have multiple scanners coordinate to appear like if just a single \text{SCAN} is active at any time. This approach was introduced in [24] to design a multi-scanner, single-writer snapshot implementation with time complexity \( O(n) \) for \text{SCAN} and \( O(1) \) for \text{UPDATE} from \( O(n^2) \) CAS registers; it is also employed by Jayanti's algorithm [19]. Despite the same methodology of all three algorithms, the design of the individual steps of the algorithms are different.

We remark that for T-Opt and C-Snap processes do not require to have unique identifiers. On the contrary, the snapshot implementations using CAS discussed above [24, 19] are not anonymous. Moreover, T-Opt and C-Snap work even if the number of participating processes is infinite. All our single-scanner algorithms work under the weaker assumption that several processes perform \text{SCANS} although not at the same time. T-Opt and RT does not require any changes but in order for RT-Opt to work some of the scanner persistent (static) variables should now be read by any process performing a \text{SCAN}, although these variables will never be accessed concurrently.

2 Model

Consider an asynchronous system with \( n \) processes that communicate by accessing shared registers. A register stores a value and supports atomic operations to access it. A read-write register \( R \) supports the operations \text{write}(R, v) \) which writes \( v \) into \( R \), and \text{read}(R) \) which returns the value of \( R \). A CAS register \( O \) supports the operations \text{read} \text{and CAS}; CAS(\( O, v, v_n \)) compares the current value of \( O \) with \( v \) and if they are equal it changes the value of \( O \) to \( v_n \). In this case we say that the CAS is successful; otherwise, the CAS fails. A register is multi-writer if all processes can change its content; on the contrary, a single-writer register can be modified only by one process.
A snapshot object consists of an array of \( m \) components, \( A_1, \ldots, A_m \), each of which stores a value (initially \( \bot \)). It supports the operations \textsc{scan} and \textsc{update}, which can be executed concurrently. An \textsc{update} \((i, v)\) updates the value of component \( A_i \) to \( v \), while \textsc{scan} returns a consistent vector of \( m \) values, one for each component. Multi-writer snapshots allow to all processes to \textsc{update} each of the components, while in a single-writer snapshot only one process can update a component. A snapshot implementation from registers simulates the components using registers and provides an algorithm to implement \textsc{scan} and an algorithm to implement \textsc{update} by accessing these registers.

A configuration is a vector whose elements are the states of processes and the values of registers; it describes the system at some point in time. In the initial configuration each process is in its initial state and each register contains an initial value. A process takes a step each time it accesses one of the shared registers; a step also involves the execution of any local operations required before the process accesses some shared register again (this may cause the state of the process to change). An execution \( \alpha \) is a sequence of steps starting from an initial configuration. An execution fragment of \( \alpha \) is a part of \( \alpha \) consisting of any number of consecutive steps. Let \( s_1 \) and \( s_2 \) be two steps of \( \alpha \) such that \( s_1 \) is executed before \( s_2 \). We denote by \( \alpha(s_1, s_2) \) the execution interval that starts with \( s_1 \) and ends with \( s_2 \). The execution interval of a \textsc{scan} (\textsc{update}) is the execution fragment that starts with the first and ends with the last step executed by the algorithm of the \textsc{scan} (\textsc{update}).

An implementation is single-scanner if in any execution of the implementation there is only one process that performs \textsc{scans}. The time complexity of a \textsc{scan} (\textsc{update}) of an implementation is the maximum number of steps performed by a process to perform a \textsc{scan} (\textsc{update}, respectively) in any execution of the implementation. The time complexity of the implementation is the maximum between the time complexities of \textsc{scan} and \textsc{update}.

We assume that processes may fail by crashing. We study wait-free implementations where each process completes the execution of any \textsc{scan} or \textsc{update} within a bounded number of its own steps independently of the speeds or the failure of other processes. All implementations we study are linearizable [15]. Linearizability guarantees that in any execution \( \alpha \) of the implementation, each \textsc{scan} or \textsc{update} appears to take effect at some point, called linearization point, within its execution interval. Thus, linearizability imposes a total order, called linearization order, to all \textsc{scans} and \textsc{updates} performed in \( \alpha \). We say that a \textsc{scan} \( S \) returns a consistent vector if for each component \( A_i, 1 \leq i \leq m \), \( S \) returns for \( A_i \) the value of the \textsc{update} on \( A_i \) that appears last in the linearization order among the \textsc{updates} on \( A_i \) that are linearized before \( S \).

## 3 T-Opt Algorithm

Pseudo-code for T-Opt is presented in Algorithm 1. The algorithm uses an array \( \text{pre} \) of \( m \) registers, one for each component. Any \textsc{update} on \( A_i, 1 \leq i \leq m \), writes its value to \( \text{pre}[i] \) (line 6). Before doing so, the \textsc{update} stores (line 5) the previous value of \( \text{pre}[i] \) in some appropriate register of an array, called \text{post}, of registers to help the scanner discover a consistent vector. Array \text{post} is a 2-dimensional array with each row having \( m \) registers; the number of its rows depends on the maximum number of \textsc{scans} performed in any execution (and therefore it might be unbounded).

The scanner uses register \( \text{seq} \) to set up a new sequence number each time a \textsc{scan} starts executed (line 7). Each \textsc{update} \( U \) on any component \( A_i \) starts by reading \( \text{seq} \) (line 1). The sequence number found there is used to index (line 5) the row of \text{post} in the \( i \)th entry of which the previous value of \( \text{pre}[i] \) is written before \( U \) overwrites it (line 6). To find a consistent vector, a \textsc{scan} \( S \) should not return the values written by \textsc{updates} that start after \( S \). To achieve this \( S \) reads all \( m \) registers of \( \text{pre} \) (line 9), and the \( m \) registers \( \text{post}[\text{seq}] \) (line 10). Notice that only \textsc{updates} that have started after the beginning of \( S \) may write to this row of \text{post}. \textsc{updates} that write to smaller rows of \text{post}
have started their execution before $S$, so if $S$ reads in $pre[i]$ a value of such an UPDATE it can include it to the vector it returns (line 12). Thus, $S$ does not need to read rows of $post$ other than $seq$. On the contrary, the values written to $pre$ by UPDATES that start after $S$ should be ignored. These UPDATES write to some register of row $seq$ of $post$. Thus, if $post[seq][i] \neq \bot$ for some $i$, $1 \leq i \leq m$, $S$ returns for component $A_i$ the old value of $pre[i]$ found in $post[seq][i]$ (line 12).

**Algorithm 1** Pseudo-code for T-Opt. (We assume that components store values of type data.)

```c
shared int seq=1;
shared data post[1..n][1..m]={⊥, ⊥, ⊥}; /* n is the number of executed SCANS*/
shared data pre[1..m] = {⊥, ⊥, ⊥};
data =scan(void){
data view[1..m], d1, d2;
int j;
void update(data value, int i){
  int curr_seq;
data d1, d2;
  curr_seq=seq;
d1=pre[i];
d2=post[curr_seq][i];
if(d2==⊥){
  return view;
}
  if(d2==1)
    view[i]=d1;
else view[i]=d2;
  for(j=1;j<=m;j++){
    seq=seq+1;
    d1=pre[j];
d2=post[seq][j];
    if(d2==⊥)
      view[j]=d1;
else view[j]=d2;
  }
}
}
```

**Time and Space Complexity.** By the code, it is obvious that the time complexity of UPDATE is $O(1)$, and the time complexity of SCAN is $O(m)$. Thus, T-Opt is a time-optimal algorithm.

All registers used by T-Opt other than $seq$ store just a single value. However, $seq$ is of unbounded size since its value is increased each time a SCAN takes place. Moreover, the number of registers used by T-Opt is only bounded by the maximum number of SCANS performed in any execution. So, in a first glance, T-Opt does not seem to be a space-efficient algorithm. However, it is easy to implement T-Opt more efficiently as follows. Each time a SCAN $S$ starts executing, the scanner dynamically asks for the allocation of a new block of $m$ positions in shared memory and sets a pointer (let it be $sptr$) to this block of memory. An UPDATE on $A_i$ starts by reading $sptr$ (which plays the role of $seq$); it then writes the value it read in $pre[i]$ to the $i$th 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 $pre$. Pseudo-code for the improved version of T-Opt is presented in Algorithm 2.

In the new implementation, $seq$ has been replaced by a memory pointer and a garbage collector can be used to de-allocate blocks of memory that are not used by any of the processes (notice that at most $n$ of the allocated blocks are pointed to by some process at each point in time). For systems with no garbage collector, implementations that are more space efficient are presented in later sections. The code presented in Algorithm 1 has been written to be consistent with the codes of these algorithms (part of the analysis of which is similar to the analysis of T-Opt).

**Linearizability.** Let $a$ be an execution of T-Opt and let $S$ be any SCAN performed in $a$. Let $w_S$ be the write performed by $S$ (line 7), and let $seg_S$ be the value written to $seq$ by $w_S$. For each $i \in \{1, \ldots, m\}$, denote by $r^{seg}_S$ the read of $pre[i]$ by $S$ (line 9), and by $r^{post}_S$ the read of $post[seg_S][i]$ by $S$ (line 10). Let $v_i$ be the value that $S$ returns for component $A_i$. In case $S$ reads $\bot$ in $post[seg_S][i]$ and $v_i$ in $pre[i]$, we denote by $U^S_\bot$ the UPDATE that writes $v_i$ to $pre[i]$ and its write to $pre[i]$ that precedes $r^{seg}_S$. If $S$ reads $v_i$ in $post[seg_S][i]$, we introduce the following notation. We denote by $V^S_{\bot}$ the UPDATE that writes $v_i$ to register $post[seg_S][i]$ and its write to $post[seg_S][i]$. 


is the last write to this register that precedes $r_i^S$. By the code, $V_i^S$ must have read the value $v_i$ in $\text{pre}[i]$. We denote by $U_i^S$ the UPDATE on $A_i$ which writes $v_i$ to $\text{pre}[i]$ and its write to $\text{pre}[i]$ is the last write to $\text{pre}[i]$ before $V_i^S$ reads $\text{pre}[i]$. We denote by $w_i^S$ the write to $\text{pre}[i]$ by $U_i^S$ (line 6).


```plaintext
shared pointer sptr[i]=new data[m];
shared data pre[1..m] = {1,...,1};

void update(data value, int i){
    data *lptr;
data d1, d2;
    1 lptr=sptr[i];
    2 d1=pre[i];
    3 d2=*ptr[i];
    4 if(d2==1)
        5 lptr[i]=d1;
    6 pre[i]=value;
}

data *scan(void){
    data view[1..m], d1, d2;
    int j;
    7 sptr=new data[m];
    8 for(j=1;j<=m;j++){
        9 d1=pre[j];
        10 d2=*ptr[j];
        11 if(d2==1) view[j]=d1;
        12 else view[j]=d2;
    }
    return view;
}
```

We now assign linearization points to SCANS and UPDATES. Each SCAN $S$ is linearized at $w_S$. For each $i \in \{1,...,m\}$, if $w_i^S$ (performed by $U_i^S$) follows $w_S$, we place the linearization point of $U_i^S$ just before $w_S$. We also place the linearization point of each UPDATE on $A_i$ that performs its write to $\text{pre}[i]$ between $w_S$ and $w_i^S$ just before $w_S$; ties are broken by the order that the writes to register $\text{pre}[i]$ 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 at its write to $\text{pre}[i]$.

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 $w_S$, 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 $S$. The same is true if $U_i^S$ executes $w_i^S$ before $w_S$. Intuitively, this holds for the following reasons: (1) by the way linearization points are assigned to UPDATES, for each $i$, $1 \leq i \leq m$, the linearization order of UPDATES on $A_i$ respects the order in which the writes to $\text{pre}[i]$ of those UPDATES have been performed, and (2) by definition of $U_i^S$, no other UPDATE on $A_i$ writes to $\text{pre}[i]$ between $w_i^S$ and $w_S$. Thus, $S$ returns a consistent vector.

It is also remarkable that if $w_i^S$ follows $w_S$, then $U_i^S$ and all UPDATES that perform their writes between $w_S$ and $w_i^S$ start their executions before $w_S$. Otherwise, each of them reads $\text{seqs[i]}$ in $\text{seq}$ and attempts to perform its write to $\text{post[seqs][i]}$ before $V_i^S$ reads this register. Thus, $V_i^S$ finds a value other than $\bot$ in $\text{post[seqs][i]}$ and does not write in it (which contradicts its definition). Thus, UPDATES are linearized within their execution intervals. Obviously, this is also true for SCANS.

We continue with the formal proof for the linearizability of T-Opt. Let $\alpha$ be any execution of T-Opt and let $\beta$ be any SCAN executed in $\alpha$. Assume that $S$ returns $v_i$ for component $A_i$. We first prove two simple technical lemmas.

Lemma 3.1 For each $i \in \{1,...,m\}$, $r_i^S$ follows $w_i^S$.

Proof: If $w_i^S$ precedes $w_S$ the claim holds since $S$ first executes $w_S$ and then $r_i^S$. So, let $w_i^S$ follow $w_S$. Assume first that $S$ reads $\bot$ in $\text{post[seqs][i]}$ and $v_i$ in $\text{pre}[i]$. By definition of $U_i^S$, $w_i^S$ writes the value read in $\text{pre}[i]$ by $S$, so $w_i^S$ precedes $r_i^S$. By the code, $r_i^S$ follows $r_i^S$. So, $r_i^S$ follows $w_i^S$.

Assume finally that $S$ reads $v_i$ in $\text{post[seqs][i]}$. Then, $V_i^S$ is well-defined. By definitions of $V_i^S$ and $U_i^S$, the following hold: (1) $V_i^S$ reads in $\text{pre}[i]$ the value written there by $w_i^S$, so the read of $\text{pre}[i]$ by $V_i^S$ occurs after $w_i^S$, and (2) $r_i^S$ reads in $\text{post[seqs][i]}$ the value written there by $V_i^S$, so $r_i^S$
follows the write to \( \text{post[seqs]}[i] \) by \( V^S_i \). By the code, the write to \( \text{post[seqs]}[i] \) by \( V^S_i \) follows its read of \( \text{pre}[i] \). It follows that \( r^S_i \) follows \( w^S_i \).

**Lemma 3.2** Assume that \( S \) reads \( u_i \) in register \( \text{post[seqs]}[i] \), and let \( r_{\text{pre}} \) be the read of \( \text{pre}[i] \) by \( V^S_i \). Then, \( r_{\text{pre}} \) is executed after \( w_S \).

**Proof:** Assume, by the way of contradiction, that \( r_{\text{pre}} \) is executed before \( w_S \). Then, the read of \( \text{seq} \) by \( V^S_i \) (let it be \( r_{\text{seq}} \)) precedes \( w_S \). By the code, \( \text{seq} \) increases each time a new SCAN is executed. Since there is only one SCAN active at each point in time, it follows that \( r_{\text{seq}} \) which is executed before \( w_S \), reads a value \( t < \text{seqs} \). By the code, it follows that \( V^S_i \) writes \( u_i \) to register \( \text{post[t]}[i] \). This a contradiction, since (by definition) \( V^S_i \) writes \( v_i \) to register \( \text{post[seqs]}[i] \neq \text{post[t]}[i] \).

We are now ready to prove that if \( w^S_i \) follows \( w_S \), then the execution of an UPDATE that performs its write to \( \text{pre}[i] \) between \( w_S \) and \( w^S_i \) (including \( U^S_i \)) starts before \( w_S \).

**Lemma 3.3** For each \( i \in \{1, \ldots, m\} \) such that \( w^S_i \) follows \( w_S \), it holds that any UPDATE on \( A_i \) that performs its write to \( \text{pre}[i] \) between \( w_S \) and \( w^S_i \) (including \( U^S_i \)) begins its execution before \( w_S \).

**Proof:** Assume, by the way of contradiction, that there is an UPDATE \( U \) on \( A_i \) that starts its execution after \( w_S \) and performs its write to \( \text{pre}[i] \) (let it be \( w^S_i \)) before \( w^S_i \). By Lemma 3.1, \( w^S_i \) precedes \( r^S_i \), and therefore \( U \) ends its execution before the end of \( S \). Since \( U \) starts its execution after \( w_S \), \( U \) reads \( \text{seqs} \) in \( \text{seq} \). By the code, \( U \) first reads register \( \text{pre}[i] \) and then register \( \text{post[seqs]}[i] \). Moreover, in case \( U \) reads \( \perp \) in \( \text{post[seqs]}[i] \), it writes in \( \text{post[seqs]}[i] \) the value read in \( \text{pre}[i] \).

By Lemma 3.1, \( w^S_i \) precedes \( r^S_i \). By the code, the execution of code lines 4-5 by \( U \) precedes \( w_S \). Thus, the execution of lines 4-5 precedes \( r^S_i \). Thus, \( r^S_i \) reads a value other than \( \perp \) in \( \text{post[seqs]}[i] \), so \( V^S_i \) is well-defined. By definition, the following are true for \( V^S_i \) and \( w^S_i \):

1. \( V^S_i \) writes to register \( \text{post[seqs]}[i] \), so it reads \( \perp \) in \( \text{post[seqs]}[i] \) (line 4), and \( \text{seqs} \) in register \( \text{seq} \) (line 1), and
2. The read in \( \text{pre}[i] \) by \( V^S_i \) follows \( w^S_i \) since \( V^S_i \) reads in \( \text{pre}[i] \) the value written by \( w^S_i \).

It follows that the read of \( \text{post[seqs]}[i] \) by \( V^S_i \), which by the code follows its read to \( \text{pre}[i] \), comes after the execution of lines 4-5 and the possible write to \( \text{post[seqs]}[i] \) by \( U \). Thus, \( V^S_i \) reads a value other than \( \perp \) in \( \text{post[seqs]}[i] \). A contradiction.

Using Lemma 3.3, it can be easily proved that the linearization point of each UPDATE is within its execution interval.

**Lemma 3.4** Let \( \alpha \) be any execution of T-Opt. The linearization point of any SCAN or UPDATE executed in \( \alpha \) 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 UPDATES that are linearized at their writes to \( \text{pre} \).

Let \( U \) be an UPDATE on \( A_i \) which is not linearized at its write to \( \text{pre}[i] \). By the way that linearization points are assigned, there is a SCAN \( S \) such that \( w^S_i \) of \( U^S_i \) is executed after \( w_S \). The write to \( \text{pre}[i] \) by \( U \) is executed between \( w_S \) and \( w^S_i \), and \( U \) is linearized just before \( w_S \). Obviously, the execution of \( U \) ends after \( w_S \). Lemma 3.3 implies that \( U \) begins its execution before \( w_S \). Thus \( U \) is linearized within its execution interval.

To prove that SCANS return consistent vectors, 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 \( \text{pre}[i] \).
Lemma 3.5 Let $U_1$, $U_2$ be two \textsc{Updates} on some component $A_i$, $1 \leq i \leq m$. Denote by $w_1$ the write to $\text{pre[i]}$ by $U_1$ and by $w_2$ the write to $\text{pre[i]}$ by $U_2$. If $w_1$ precedes $w_2$, the linearization point of $U_1$ precedes the linearization point of $U_2$.

Proof: Assume, by the way of contradiction, that the claim does not hold. If $U_1$ and $U_2$ are linearized at their writes to $\text{pre[i]}$, then the claim holds trivially. Therefore, assume that at least one of the $U_1$, $U_2$ is not linearized at its write to $\text{pre[i]}$. We consider the following cases.

1. $U_2$ is linearized at $w_2$. Lemma 3.4 implies that $U_1$ is linearized within its execution interval, so $U_1$ is linearized before $w_1$. Thus, $U_1$ is linearized before $U_2$. A contradiction.

2. $U_1$ is linearized at $w_1$. Since we have assumed that $U_2$ is linearized before $U_1$, $U_2$ cannot be linearized at $w_2$. By the way linearization points are assigned, there is a \textsc{Scan} $S$ such that $w_2$ has been performed between $w_S$ and $w^{S}_1$. Since $w_1$ precedes $w_2$, $w_1$ has been executed before $w^{S}_1$. In case $w_2$ follows $w_S$, both $U_1$ and $U_2$ are linearized just before $w_S$ in the order that they perform their writes to $\text{pre[i]}$. Thus, $U_1$ is linearized before $U_2$, which is a contradiction. So, assume $w_1$ precedes $w_S$. Lemma 3.4 implies that $U_1$ is linearized the latest at $w_1$. By the way linearization points are assigned, $U_2$ is linearized just before $w_S$. Thus, $U_1$ is linearized before $U_2$. A contradiction.

3. None of $U_1$, $U_2$ is linearized at its write to $\text{pre[i]}$. By the way linearization points are assigned, there are two \textsc{Scan} operations $S_1$ and $S_2$ such that $w_1$ has been performed between $w_S$ and $w^{S}_1$, and $w_2$ has been performed between $w_S$ and $w^{S}_2$; moreover, $U_1$ is linearized just before $w_S$, and $U_2$ is linearized just before $w_S$. Lemma 3.1 implies that $w^{S}_1$ precedes the end of $S_1$, and $w^{S}_2$ precedes the end of $S_2$.

If $S_1 = S_2$, both $U_1$ and $U_2$ are linearized just before $w_{S_1} = w_{S_2}$ in the order they perform their writes to $\text{pre[i]}$. So, $U_1$ is linearized before $U_2$, which is a contradiction.

If $S_1$ precedes $S_2$, the linearization point of $U_1$ which is placed just before $w_{S_1}$ precedes the linearization point of $U_2$ which is placed just before $w_{S_2}$, which is a contradiction.

Assume finally that $S_1$ follows $S_2$. Recall that $w^{S}_1$ is executed after $w_{S_1}$ and before the end of $S_1$; similarly, $w^{S}_2$ is executed after $w_{S_2}$ and before the end of $S_2$. Thus, $w_1$ which is executed between $w_{S_1}$ and $w^{S}_1$ follows $w_2$ which is executed between $w_{S_2}$ and $w^{S}_2$, which is a contradiction.

In all cases we derived a contradiction. Thus, we conclude that the linearization point of $U_1$ precedes the linearization point of $U_2$, as needed.

We finally use Lemma 3.5 to prove consistency.

Lemma 3.6 Let $\alpha$ be an execution of $T$-Opt. Any \textsc{Scan} executed in $\alpha$ returns a consistent vector.

Proof: Assume that $S$ returns the vector $v = (v_1, \ldots, v_m)$. Recall that, for each $i \in \{1, \ldots, m\}$, $U^S_i$ writes $v_i$ to $\text{pre[i]}$, and therefore it uses $v_i$ as a parameter. In case $w^S$ precedes $w_S$, Lemma 3.4 implies that $U^S_i$ is linearized before $S$. In case $w^S$ follows $w_S$, by the way linearization points are assigned, the linearization point of $U^S_i$ precedes the linearization point of $S$. Thus, $U^S_i$ stores $v_i$ in component $A_i$ and its linearization point precedes the linearization point of $S$. We prove that there is no $\textsc{Update}$ on component $A_i$ that is linearized between $U^S_i$ and $S$, so that $S$ returns a consistent value for $A_i$.  

9
Assume, by the way of contradiction, that there is an integer \( i \in \{1, \ldots, m\} \) such that the last UPDATE on \( A_i \) which has been linearized before \( S \) is not \( U_i^S \). Denote by \( U \) this UPDATE and let \( w \) be the write to \( \text{pre}[i] \) by \( U \). We consider the following cases.

1. Assume that \( S \) reads \( l \) in register \( \text{post}[\text{seqs}][i] \) and \( v_i \) in register \( \text{pre}[i] \). In case \( w \) precedes \( w_i^S \), Lemma 3.5 implies that \( U \) is linearized before \( U_i^S \), which is a contradiction. Thus, assume that \( w \) follows \( w_i^S \). By the definition of \( w_i^S \), \( r_i^S \) reads the value that \( w_i^S \) writes to register \( \text{pre}[i] \). Therefore, \( w \) must follow \( r_i^S \). Since \( U \) is linearized before \( S \), and \( S \) is linearized at \( w_S \), \( U \) cannot be linearized at \( w \). Therefore, there is a SCAN \( S' \) such that \( w \) is performed between \( w_S \) and \( w_i^S \), and \( U \) is linearized just before \( w_S \). Since \( w \) follows \( w_i^S \), \( S \neq S' \). Lemma 3.1 implies that \( w_i^S \) precedes the end of the execution interval of \( S' \). If \( S' \) precedes \( S \), \( w \) which is performed between \( w_S \) and \( w_i^S \), precedes \( w_S \), which is a contradiction. If \( S' \) follows \( S \), the linearization point of \( U \) which is placed at \( w_S \) follows the linearization point of \( S \) which is placed at \( w_S \), which is a contradiction.

2. Assume that \( S \) reads \( v_i \) in register \( \text{post}[\text{seqs}][i] \). Then, \( V_i^S \) is well-defined and let \( r_{\text{pre}} \) be the read of \( \text{pre}[i] \) by \( V_i^S \). By the definitions of \( U_i^S \) and \( V_i^S \), \( S \) returns the value that \( U_i^S \) uses as a parameter and therefore \( S \) returns the value that has been written to \( \text{pre}[i] \) by \( U_i^S \). In case \( w \) precedes \( w_i^S \), Lemma 3.5 implies that \( U \) is linearized before \( U_i^S \), which is a contradiction. Thus, assume that \( w \) follows \( w_i^S \). Then, \( w \) must follow \( r_{\text{pre}} \) since (by the definition of \( U_i^S \)) \( V_i^S \) reads the value that \( w_i^S \) writes. By Lemma 3.2, \( r_{\text{pre}} \) follows \( w_S \). Therefore, \( w \) follows \( w_S \). Since \( U \) is linearized before \( S \), and \( S \) is linearized at \( w_S \), \( U \) cannot be linearized at \( w \). Thus, there is a SCAN \( S' \) such that \( w_i^S \) follows \( w_S \), \( w \) is performed between \( w_S \) and \( w_i^S \), and \( U \) is linearized just before \( w_S \). Since \( w \) is performed after \( w_i^S \), \( S' \neq S \).

If \( S' \) follows \( S \), the linearization point of \( U \) which is placed at \( w_S \), follows the linearization point of \( S \) which is placed at \( w_S \). This is a contradiction.

Thus, assume that \( S' \) precedes \( S \). Lemma 3.1 implies that \( w_i^S \) precedes the end of the execution interval of \( S' \). Thus, \( w \) which is performed between \( w_S \) and \( w_i^S \) precedes \( w_S \), which is a contradiction.

In all cases we derived 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.

Theorem 3.7 T-Opt is a linearizable, wait-free, single-scanner, multi-writer snapshot implementation from an unbounded number of registers that achieves time complexity \( O(m) \) for SCAN and \( O(1) \) for UPDATE.

4 RT Algorithm

RT attempts to reduce the number of registers used by T-Opt; instead of requiring an unbounded number of rows for \( \text{post} \), it uses only \( n + 2 \) such rows. To achieve this, it employs another array, called \( \text{state} \), of \( n \) registers, one for each process, which are written when \( \text{UPDATES} \) are performed. Pseudo-code for RT is presented in Algorithm 3. An UPDATE by some process \( p \) records in \( \text{state}[p] \) the value it read in \( \text{seq} \) (line 2). A SCAN \( S \) reads all \( n \) registers of \( \text{state} \) and chooses as its sequence number \( \text{seqs} \) some index not appearing in any of these registers (lines 10-12).

The main goal of the algorithm is to guarantee that only \( \text{UPDATES} \) that perform the biggest part of their execution after the write \( w_S \) to \( \text{seq} \) by \( S \) (line 14) write to registers of \( \text{post}[\text{seqs}] \). This is achieved by employing a technique that reminds handshaking between the scanner and each of the
updaters. Each time some process \( p \) performs an \texttt{UPDATE} \( U \), it uses \( \text{state}[p] \) to inform the scanner of the value it read in \( \text{seq} \) (lines 1-2). Then, it reads \( \text{seq} \) again (line 3) and only if it sees the same value in \( \text{seq} \) twice (line 6), it attempts to write to \( \text{post} \) (line 7).

If \( U \) has performed its write to \( \text{state}[p] \) before \( S \) reads \( \text{state}[p] \), \( S \) will choose a sequence number other than the one read by \( U \) in \( \text{seq} \). The only other interesting case is if \( U \) writes to \( \text{state}[p] \) after \( S \) has read it, and \( U \) performs its second read of \( \text{seq} \) before \( \text{ws} \). Then, the second read of \( \text{seq} \) by \( U \) reads the sequence number of the \texttt{SCAN} that precedes \( S \) (or the initial value of \( \text{seq} \)). However, \( \text{RT} \) guarantees that each \texttt{SCAN} chooses a sequence number different than the one chosen by its previous \texttt{SCAN} (and than the initial value of \( \text{seq} \)). This discussion implies that we need to store \( n+2 \) different values in \( \text{seq} \) (and to have \( n+2 \) rows in \( \text{post} \)).

Algorithm 3 Pseudo-code for \texttt{RT} (process \( p, 1 \leq p \leq n \)).

```c
shared int seq=1; data =scan(void){
shared int state[1..n]=1,..,n];
data view[1..m], d1, d2;
shared data post[i..m+2][1..m]=⊥,..,⊥;
set act.set;
shared data pre[1..m]=⊥,..,⊥ ;
int lseq, j;
void update(data value, int i){
sequence curr.seq1, curr.seq2;
data d1, d2;
1 curr.seq1=seq;
2 state[p]=curr.seq1;
3 curr.seq2=seq;
4 d1=pre[i];
5 d2=post[curr.seq1][i];
6 if(d2=⊥ & curr.seq1=curr.seq2)
7 post[curr.seq1][i]=d1;
8 pre[i]=value;
9 act.set={seq};
10 for(i=1; j≤n; j++)
11 act.set=act.setU{state[j]};
12 lseq=any int of {1,..,n+2}-act.set;
13 for(j=1; j≤n; j++) post[lseq][j]=⊥;
14 seq=lseq;
15 for(i=1; j≤n; j++)
16 d1=pre[j];
17 d2=post[seq][j];
18 if(d2=⊥) view[j]=d1;
19 else view[j]=d2;
}
return view;
}
```

Time and Space Complexity. By the code, it is obvious that the time complexity of \texttt{UPDATE} is \( O(1) \), and the time complexity of \texttt{SCAN} is \( O(n) \). \texttt{RT} uses only \( (n+3)m+n+1 \) registers; \( (n+3)m \) out of those registers (namely, the registers of arrays \( \text{pre} \) and \( \text{post} \) store just a single value, while the size of the rest \( n+1 \) registers (namely, \( \text{seq} \) and the registers of array \( \text{state} \)) is \( O(\log n) \) bit (since each of them stores values from the set \{1,..,n+2\}).

Linearizability. Let \( \alpha \) be an execution of \texttt{RT} and let \( S \) be any \texttt{SCAN} performed in \( \alpha \). Let \( ws \) be the write to \( \text{seq} \) performed by \( S \) (line 14), and let \( seqs \) be the value written to \( \text{seq} \) by \( ws \). For each \( i \in \{1,..,m\} \), we introduce the notation \( r_{i}^{S}, p_{i}^{S}, v_{i}, U_{i}^{S}, V_{i}^{S} \) and \( w_{i}^{S} \), and assign linearization points to \texttt{SCAN}s and \texttt{UPDATES} in exactly the same way as we did for \( \text{T-Opt} \). The proof of the linearizability of \texttt{RT} is in its biggest part similar to the proof of \( \text{T-Opt} \).

The arguments for overcoming the difficulties encountered by \texttt{T-Opt} discussed above are formalized in the following lemma (which corresponds to Lemma 3.2 of Section 3).

Lemma 4.1 Assume that \( S \) reads \( v_{i} \) in register \( \text{post}[seqs][i] \), and let \( r_{pre} \) be the read of \( \text{pre}[i] \) by \( V_{i}^{S} \). Then, \( r_{pre} \) is executed after \( ws \).

Proof. Assume, by the way of contradiction, that \( r_{pre} \) is executed before \( ws \). Denote by \( r_{seq} \) the first read of \( \text{seq} \) by \( V_{i}^{S} \) (line 1), and by \( r_{seq}^{'} \) its second read of \( \text{seq} \) (line 3). Let \( p \) be the process
that executes $V^S_i$, let $w_p$ be the write to $\text{state}[p]$ by $V^S_i$ (line 2), and let $r_p$ be the read of $\text{state}[p]$ by $S$ (line 11). Since $r_{pre}$ precedes $w_S$, the same is true for $r'_{seq}$ (which precedes $r_{pre}$).

Assume that $r'_{seq}$ precedes $r_p$. Since $w_p$ precedes $r'_{seq}$, it follows that $w_p$ precedes $r_p$. Thus, the value $t$ written to $\text{state}[p]$ by $w_p$ (line 2) is read by $r_p$. By the code (line 11), it follows that $\text{seqS} \neq t$. Since $V^S_i$ read $t$ in seq, it follows by the code that $V^S_i$ cannot write to any other register than $\text{post}[i][i] \neq \text{post}[\text{seqS}][i]$, which is a contradiction.

Assume now that $r'_{seq}$ follows $r_p$. Since $r'_{seq}$ precedes $r_{pre}$, $r'_{seq}$ precedes $w_S$. Let $S'$ be the SCAN executed just before $S$ in $\alpha$ (or a fictitious SCAN that writes to seq the initial value if no such SCAN exists). By the code (line 9), it follows that $\text{seqS} \neq \text{seqS'}$. Since $r'_{seq}$ follows $r_p$ and precedes $w_S$, $r'_{seq}$ reads $\text{seqS'}$ in seq. Thus, it follows by the code that $V^S_i$ does not write in $\text{post}[\text{seqS}][i]$, which is a contradiction.

The proof of the next lemma follows similar arguments as the proof of Lemma 3.3 but we include it below because it slightly differentiates at some places. In the proof below we apply Lemma 3.1 (presented in Section 3) which holds also for RT (its proof is exactly the same as for T-Opt).

**Lemma 4.2** For each $i \in \{1, \ldots, m\}$ such that $w^S_i$ follows $w_S$, it holds that any UPDATE on $A_i$ that performs its write to $\text{pre}[i]$ between $w_S$ and $w^S_i$ (including $U^S_i$) begins its execution before $w_S$.

**Proof:** Assume, by the way of contradiction, that there is an UPDATE $U$ on $A_i$ that starts its execution after $w_S$ and performs its write to $\text{pre}[i]$ (let it be $w$) before $w^S_i$. By Lemma 3.1, $w^S_i$ precedes $r^S_i$, and therefore $U$ ends its execution before the end of $S$. Since $U$ starts its execution after $w_S$, $U$ reads $\text{seqS}$ in seq both times (lines 1 and 3). So, the second condition of the if statement of line 6 is evaluated to true. Thus, in case $U$ reads $\bot$ in $\text{post}[\text{seqS}][i]$, it writes to $\text{post}[\text{seqS}][i]$ the value read in $\text{pre}[i]$.

By the code (line 13), $S$ initializes the $m$ registers of $\text{post}[\text{seqS}]$ to $\bot$ before $w_S$. Since $U$ starts after $w_S$, the execution of code lines 6-7 (if statement and possible write to $\text{post}[\text{seqS}][i]$) by $U$ follows the initialization of $\text{post}[\text{seqS}][i]$ to $\bot$ by $S$. By Lemma 3.1, $w^S_i$ precedes $r^S_i$. By the code, the execution of code lines 6-7 by $U$ precedes $w$ (which precedes $w^S_i$). Thus, the execution of lines 6-7 precedes $r^S_i$. Thus, $r^S_i$ reads a value other than $\bot$ in $\text{post}[\text{seqS}][i]$, so $V^S_i$ is well-defined. By definition, the following are true for $V^S_i$ and $w^S_i$: (1) $V^S_i$ writes to register $\text{post}[\text{seqS}][i]$, so it reads $\bot$ in $\text{post}[\text{seqS}][i]$ (line 5), and $\text{seqS}$ in register seq (lines 3 and 1), and (2) the read in $\text{pre}[i]$ by $V^S_i$ follows $w^S_i$ since $V^S_i$ reads in $\text{pre}[i]$ the value written by $w^S_i$.

It follows that the read of $\text{post}[\text{seqS}][i]$ by $V^S_i$, which by the code follows its read to $\text{pre}[i]$, comes after the execution of lines 6-7 and the possible write to $\text{post}[\text{seqS}][i]$ by $U$. Thus, $V^S_i$ reads a value other than $\bot$ in $\text{post}[\text{seqS}][i]$. A contradiction.

To prove the rest of the lemmas presented in Section 3, exactly the same arguments as for T-Opt are applied for RT.

**Theorem 4.3** RT is a linearizable, wait-free, single-scanner, multi-writer snapshot implementation from $(n+2)m + n + 1$ bounded-size registers that achieves time complexity $O(n)$ for SCAN and $O(1)$ for UPDATE.

5 RT-Opt Algorithm

Pseudo-code for RT-Opt is presented in Algorithm 4. The UPDATE for RT-Opt is exactly the same as for RT. The major goal of any SCAN $S$ for both RT and RT-Opt is to keep track of the different
rows of post where old UPDATEs (that is, those that have performed some part of their executions before the write wag of S to seq) may write. S must choose a row of post where no such UPDATE could possibly write, in order to ensure that all values other than \( \perp \) that it reads in post have been written by UPDATEs that have performed the biggest part of their execution after wag.

In RT this is achieved by having each SCAN S read all \( n \) registers of state and choosing some value other than those read there. Unfortunately, this incurs an overhead on the time complexity of SCAN. To keep the time complexity of SCAN low, each SCAN of RT-Opt reads only \( m \) of the \( n \) registers of array state; each execution \( \alpha \) of RT-Opt can be partitioned into execution fragments, called epochs, each containing \( [n/m] \) consecutive SCANs. Moreover, sequence numbers are now chosen from the set \([1, \ldots, n+2[n/m]+1]\) which is larger than the set \([1, \ldots, n+2]\) used in RT.


```plaintext
constant \( \text{PACE} = n \);
constant \( \text{PERIODS} = \lceil n/\text{PACE} \rceil \);
shared int \( \text{seq} = 1 \);
shared int states[1..PERIODS*n] = \{1, \ldots, 1\};
shared data pre[1..n] = \{1, \ldots, 1\},
post[1..n+2*PERIODS+1] = \{1, \ldots, 1\};
void update(data value, int i){
    sequence curr_seq1, curr_seq2;
data d1, d2;
data view[1..n], d1, d2;
    int lseq, j;
    static int cur.period=0;
    static set free=8,
candidates=\{2, \ldots, n+2*PERIODS+1\};
    if(cur.period==0)
        free=\{free\}candidates;
    candidates=\{1, \ldots, n+2*PERIODS+1\};
    lseq=any element of set free;
    for(\( j=1; j\leq lseq; j++ \))
        post[lseq][j]=l;
    free=\{lseq\};
    candidates=\{lseq\};
cur.period=(cur.period+1)%PERIODS;
    seq=lseq;
    for(\( j=1; j\leq \text{PACE}; j++ \))
        candidates=\{state[cur.period*\text{PACE}+j]\};
    for(\( j=1; j\leq lseq; j++ \))
        d1=pre[j];
    d2=post[seq][j];
    if(d2=\perp) view[j]=d1;
    else view[j]=d2;
}
return view;
```

Set free keeps track of the values that can be used as sequence numbers by SCANs of each epoch. During any epoch \( E_j \), \( j \geq 1 \), all sequence numbers chosen by SCANs are distinct (line 14). For the first epoch \( E_1 \), all these values are additionally different than the initial value of seq (see initialization of free and seq). Consider a later epoch \( E_j \), \( j > 1 \). Recall that all registers of array state have been read once during \( E_{j-1} \). All the values read in these registers index rows of post 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 state[\( p \)] after the read of state[\( p \)] 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 post 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 11). Then, during the execution of the $[n/m]$ SCANS of the epoch, all values read in registers of array state, as well as those chosen as sequence numbers by the SCANS of the epoch, are removed from candidates (lines 15 and 19). At the beginning of the next epoch, the values remaining in candidates can be moved to the set free of available sequence numbers (line 10) for the epoch. We remark that no other elements are added to free during the epoch. So, free correctly keeps track of the set of available sequence numbers for the SCANS of each epoch.

At the beginning of any execution $\alpha$ of RT-Opt, candidates contain $n + 2 \cdot \text{PERIODS}$ different sequence numbers, where PERIODS = $[n/m]$ (see initialization of PERIODS and candidates).

During $E_1$, at most $n + \text{PERIODS}$ sequence numbers are extracted from candidates. So at the end of $E_1$, candidates contain at least PERIODS values, which are added to free at the beginning of $E_2$. So, free contains enough sequence numbers for the PERIODS SCANS that are executed during $E_2$. Consider now any epoch $E_j$, $j > 1$. At the beginning of $E_j$, candidates contain $n + 2 \cdot \text{PERIODS} + 1$ different sequence numbers. During $E_j$, at most $n + \text{PERIODS}$ sequence numbers are removed from candidates. Thus, at least $\text{PERIODS} + 1$ sequence numbers are added to free at the beginning of $E_{j+1}$, which are enough for the SCANS of epoch $E_{j+1}$. From this discussion, it follows that $n + 2 \cdot [n/m] + 1$ different sequence numbers are required by RT-Opt.

**Time and Space Complexity.** By the code, it is obvious that RT-Opt has $O(1)$ time complexity for UPDATE. The time complexity for SCAN is $O(m)$ since each SCAN reads $3m$ shared registers, namely, $m$ registers of state (since $\text{PACE} = m$), $m$ registers of post, and $m$ registers of $\text{pre}$; the rest of the SCAN computation is on local variables. RT-Opt uses $O(mn)$ registers; most of them (the registers of arrays $\text{pre}$ and $\text{post}$) store just a value, while the size of the rest registers is $O(\log n)$ bit. It is remarkable that PACE can take any value between 1 and $n$. If $\text{PACE} = n$, RT-Opt works in the same way as RT, while if $\text{PACE} \leq m$, RT-Opt achieves optimal time complexity.

**Linearizability.** Let $\alpha$ be an execution of RT-Opt and let $S$ be any SCAN performed in $\alpha$. Let $w_S$ be the write to $\text{seq}$ performed by $S$ (line 17), and let $\text{seq}_S$ be the value written to seq by $w_S$. For each $i \in \{1, \ldots, m\}$, we introduce the notation $r^S_i$ (read by $S$, line 21), $r^S_i$ (read by $S$, line 22), $v_i$, $U^S_i$, $V^S_i$ and $w^S_i$, and assign linearization points to SCANS and UPDATES in exactly the same way as we did for T-Opt (and RT). A part of the proof of the linearizability of RT-Opt is similar to the proof for T-Opt (and RT). As for RT, the main difficulty is in proving that the UPDATES that write values to row $\text{seq}_S$ of post have executed their biggest part after the write to $\text{seq}$ by $S$.

Let $\alpha$ be an execution of RT-Opt and let $S$ be any SCAN performed in $\alpha$. We split $\alpha$ into epochs so that each epoch contains exactly $[n/m]$ SCANS. Denote by $E_i$ the $i$-th epoch of $\alpha$, $i \geq 1$. Epoch $E_i$ starts with the first instruction of the execution and ends with the last instruction of the $[n/m]$-th SCAN (or equals to $\alpha$ if not that many SCANS occur in $\alpha$). For each $i > 1$, epoch $E_i$ starts at the point that the execution of the $(i-1)[n/m]$-th SCAN ends and finishes with the last instruction executed by the $(i[n/m])$-th SCAN (or equals to the suffix of $\alpha$ which starts at the point that the execution of the $(i-1)[n/m]$-th SCAN ends, if less than $(i[n/m])$ SCANS occur in $\alpha$.

Assume that $\alpha$ contains $(c_1 [n/m] + c_2)$ SCANS, where $c_1 \geq 0$ and $0 \leq c_2 < [n/m]$ are constants. We remark that if $c_2$ equals zero then $\alpha$ contains $c_1$ epochs. However, if $c_2 > 0$ then $\alpha$ consists of $c_1 + 1$ epochs where the $(c_1 + 1)$-th epoch contains $c_2 < [n/m]$ SCANS. Notice also that if $\alpha$ is an infinite execution, the execution interval of epoch $c_1 + 1$ is infinite.

Let $k$ be the number of epochs in $\alpha$. For each $i \in \{1, \ldots, k\}$, denote by $SN_i$ the set of values written in register seq by any SCAN of epoch $E_i$, and by free, the set free at the end of $E_i$. Denote
by *candidates*; the set *candidates* at the end of *E*$_j$.

The proof of the linearizability of RT-Opt is in its biggest part similar to the proof of T-Opt (and RT). Consider some SCAN *S* that writes *seq* in register *seq*. As for RT, the main difficulty is to prove that the *updates* that write values to the *seq* row of *post* have executed their biggest part after the write to *seq* by *S*. We first prove four simple technical lemmas which are basically direct consequences of the code.

**Lemma 5.1** For each *j* \(\in \{1, \ldots, k - 1\}\) and for each *p* \(\in \{1, \ldots, n\}\), there is a unique SCAN that reads register *state*[$p$] in *E*$_j$.

**Proof:** By definition of *E*$_j$, exactly \([n/m]\) SCANs are performed during it. Each if these SCANs reads *m* distinct registers of *state* (lines 16, 18, 19). Thus, all \([n/m] * m\) registers of *state* are read once in *E*$_j$.

**Lemma 5.2** 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 the code (line 14), each value chosen as the sequence number of some SCAN in *E*$_j$ is extracted from *free* (line 14); the same is true for *candidates* (line 15). 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 the code (lines 9-11 and line 16), it follows that lines 10-11 are executed only by the 1-st, \([(n/m) + 1]\)-th \(\ldots\) \(((k - 1)(n/m) + 1)\)-th SCAN of *α*, as stated by the following lemma.

**Lemma 5.3** For each *j* \(\in \{1, \ldots, k\}\), the following hold for the first SCAN *S* executed in *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 11 initializing *candidates*.

Next lemma states that each SCAN of some epoch writes to *seq* a value different than the values written to *seq* by the other SCANs of the epoch.

**Lemma 5.4** For each *j* \(\in \{1, \ldots, k\}\), each SCAN of epoch *E*$_j$ writes a distinct value to *seq*.

**Proof:** Lemma 5.3 implies that elements are added in *free* only by the first SCAN of each epoch. By the code (line 12), each SCAN *S* chooses as its sequence number some element of *free*. This element is removed by *S* from *free* (line 14), so that later SCANs of the same epoch choose to write to *seq* different values.

We next prove that the SCANs of an epoch choose different sequence numbers than the SCANs of the previous epoch.

**Lemma 5.5** For each *j* \(\in \{2, \ldots, k\}\), \(SN_{j-1} \cap SN_j = \emptyset\).

**Proof:** Fix any *j* \(\in \{2, \ldots, k\}\). By Lemma 5.2, \(free_{j-1} \cap SN_{j-1} = \emptyset\), and \(candidates_{j-1} \cap SN_{j-1} = \emptyset\). Thus, at the end of epoch *E*$_j$, neither set *free* nor *candidates* have common elements with *SN*$_{j-1}$.

By Lemma 5.3, the only SCAN of *E*$_j$ that adds elements in *free* (by executing line 10) is the first SCAN of the epoch (let is be *S*). This SCAN adds the elements of *candidates*$_{j-1}$ in *free*$_{j-1}$. Denote by *free*$_j$ the set *free* after line 10 has been executed by *S*. Clearly, *free*$_j$ = *free*$_{j-1}$ \(\cup\) *candidates*$_{j-1}$.

Since *free*$_{j-1} \cap SN_{j-1} = \emptyset$, and *candidates*$_{j-1} \cap SN_{j-1} = \emptyset$, it follows that *free*$_j \cap SN_{j-1} = \emptyset$. Hence, *free*$_j \cap SN_{j-1} = \emptyset$.

By the code, all elements of *SN*$_j$ are chosen by *free*$_j$. Thus, *SN*$_j \cap SN_{j-1} = \emptyset$.
Lemma 5.6 Assume that $S$ reads $v_i$ in register $post[seqs][i]$, and let $r_{pre}$ be the read of $pre[i]$ by $V^S_i$. Then, $r_{pre}$ is executed after $w_S$.

Proof: Assume, by the way of contradiction, that $r_{pre}$ is executed before $w_S$. Denote by $r_{seq}$ the first read of $seq$ by $V^S_i$ (line 1), and by $r'_{seq}$ the second read of $seq$ by $V^S_i$ (line 3). Let $p$ be the process that executes $V^S_i$ and let $w_p$ be the write to $state[p]$ by $V^S_i$ (line 2). Since $r_{pre}$ precedes $w_S$, the same is true for $r'_{seq}$ (which precedes $r_{pre}$). Assume that $S$ is executed in epoch $E_j$, $j \geq 1$.

1. Assume that $j = 1$. By the code (line 10) and by Lemma 5.3, at each point during the first epoch, $free$ does not contain the initial value of $seq$. Since SCANs of each epoch choose elements of $free$ as their sequence numbers, $S$ chooses a sequence number different than the initial value of $seq$. Lemma 5.4 implies that no SCAN that precedes $S$ chooses the same sequence number as $S$. By definition, $V^S_i$ writes in row $seqs$ of $post$. By the code (line 6), this write is performed only if both $r_{seq}$ and $r'_{seq}$ read $seqs$ in $seq$. It follows that $r_{seq}$ and $r'_{seq}$ are both performed after $w_S$. Since $r_{pre}$ follows $r'_{seq}$, $r_{pre}$ follows $w_S$, which is a contradiction.

2. Assume that $j > 1$. By Lemma 5.1, there is a unique SCAN $S'$ that reads $state[p]$ during $E_{j-1}$. Denote by $r_p$ the read of $state[p]$ by $S'$ and by $w_{S'}$ the write to $seq$ by $S'$.

By the code (line 13), $S$ initializes all $m$ registers of $post[seqs]$ to $\bot$. By definition of $V^S_i$, $S$ reads the value that $V^S_i$ writes to $post[seqs][i]$. Thus, $V^S_i$ writes to $post[seqs][i]$ after the initialization of $post[seqs][i]$ to $\bot$ by $S$. Since $state[p]$ is written only by $p$, $state[p]$ contains the value written by $V^S_i$ from $w_p$ until at least the initialization of $post[seqs][i]$ by $S$. By the code, $r_{pre}$ is executed after $r'_{seq}$. Recall that $r_p$ is the read of $state[p]$ by $S'$. We consider the following cases.

(a) $r'_{seq}$ follows $r_p$. By definition, $V^S_i$ writes to $post[seqs][i]$, so (by the code) it must be that $r_{seq}$ reads $seqs$ in $seq$. By Lemma 5.5, $SN_{j-1} \cap SN_j = \emptyset$. Thus, since $seqs \in SN_j$, $seqs \notin SN_{j-1}$ (that is, no SCAN of epoch $E_{j-1}$ writes $seqs$ to $seq$). By Lemma 5.4, each SCAN of epoch $E_j$ writes a distinct value to $seq$. So, no SCAN of epoch $E_j$ that precedes $S$ writes $seqs$ to $seq$. Since $r'_{seq}$ follows $r_p$ and (by the code) $r_p$ follows $w_{S'}$, the only way for $r'_{seq}$ to read $seqs$ is if it occurs after $w_{S'}$. Since $r_{pre}$ follows $r'_{seq}$, it follows that $r_{pre}$ follows $w_S$, which is a contradiction.

(b) $r'_{seq}$ precedes $r_p$. Recall that $r_{seq}$ reads $seqs$ in $seq$. Assume that the value $seqs$ read by $r'_{seq}$ has been written to $seq$ by some SCAN $S_i$ that is executed in epoch $E_l$, $l \geq 1$. If no such SCAN exists, then $r_{seq}$ reads the initial value of $seq$, so $seqs = 1$; moreover, $r_{seq}$ is executed before the write to $seq$ by the first SCAN of the execution (which is also the first SCAN of $E_1$). In this case, let $l = 0$, $free_0 = 0$, and $candidates = \{2, \ldots, n + 2 \cdot PERIODS + 1\}$ (that is, $free_0$ and $candidates$ denote sets $free$ and $candidates$, respectively, in initial state). Let $l > 0$. Since $seqs \in SN_j$, Lemma 5.5 implies that $seqs \notin SN_{j-1}$. Thus, $1 \leq l < j - 1$. Let $w_{S_i}$ be the write to $seq$ by $S_i$. Since $r'_{seq}$ reads the value written by $w_{S_i}$, $r'_{seq}$ is performed between $w_{S_i}$ and the write to $seq$ by the next to $S_i$ SCAN (since $l < j - 1$, such a SCAN exists). So, $r'_{seq}$ is executed either during $E_l$ or at the beginning of epoch $E_{l+1}$, before the write to $seq$ by the first SCAN of $E_{l+1}$ (this situation may occur if $S_i$ is the last SCAN of $E_l$). (Notice that since $l < j - 1$, $E_{l+1}$ is either $E_{j-1}$ or an earlier epoch.)

We prove the following claims.

Claim 5.7 For each $f \in \{l, \ldots, j - 1\}$, $seqs \notin candidates_f$. 

16
Proof: Assume first that \( f = l \). In case \( l = 0 \), recall that \( \textit{seqs} = 1 \) (the initial value of \( \textit{seq} \)), and \( \textit{candidates}_0 = \{ 2, \ldots, n + 2 \cdot \text{PERIODS} + 1 \} \). So, \( \textit{seqs} \notin \textit{candidates}_0 \). Assume now that \( l > 0 \). Since \( S_l \) is executed in epoch \( E_l \) and chooses \( \textit{seqs} \) as its sequence number, \( \textit{seqs} \in SN_l \). By Lemma 5.5, \( \textit{candidates}_l \cap SN_l = \emptyset \). Thus, \( \textit{seqs} \notin \textit{candidates}_l \).

Assume now that \( f > l \). Recall that \( \textit{state}[p] \) does not change from \( w_p \) until at least the beginning of \( S \). Recall also that \( r'_{\textit{seq}} \) (and therefore also \( w_p \) which precedes \( r'_{\textit{seq}} \)) is executed before the write to \( \textit{seq} \) by the first \( \text{SCAN} \) of epoch \( E_{l+1} \). By the code (lines 17-19), each \( \text{SCAN} \) first writes to \( \textit{seq} \) and then reads some of the registers of array \( \textit{state} \).

By Lemma 5.1, \( \textit{state}[p] \) is read by a unique \( \text{SCAN} \) of \( E_f \). So, \( \textit{seqs} \) is read in \( \textit{state}[p] \) during \( E_f \). Thus, it follows by the code (line 19) that \( \textit{seqs} \) is removed from \( \textit{candidates} \) during \( E_f \). By Lemma 5.3, no elements are added in \( \textit{candidates} \) after the execution of line 11 by the first \( \text{SCAN} \) of epoch \( E_f \). Since line 19 follows line 11, we conclude that \( \textit{seqs} \notin \textit{candidates}_f \).

Claim 5.8 For each \( f \in \{ l, \ldots, j - 1 \} \), \( \textit{seqs} \notin \textit{free}_f \).

Proof: By induction on \( f \). We first prove the claim for \( f = l \). In case \( l = 0 \), \( \textit{free}_0 = \emptyset \), so \( \textit{seqs} \notin \textit{free}_0 \). Assume that \( l > 0 \). Since \( S_l \) chooses \( \textit{seqs} \) as its sequence number, \( \textit{seqs} \in SN_l \). Lemma 5.2 implies that \( \textit{free}_l \cap SN_l = \emptyset \). Thus, \( \textit{seqs} \notin \textit{free}_l \).

Fix some \( f, l < f \leq j - 1 \), and assume that the claim holds for \( f - 1 \). We prove that the claim holds for \( f \).

By the induction hypothesis, \( \textit{seqs} \notin \textit{free}_{f-1} \). By claim 5.7 it follows that \( \textit{seqs} \notin \textit{candidates}_{f-1} \). Let \( \textit{free}_f \) denote set \( \textit{free} \) after the execution of line 10 by the first \( \text{SCAN} \) of epoch \( E_f \). By the code (line 10), \( \textit{free}_f = \textit{free}_{f-1} \cup \textit{candidates}_{f-1} \). It follows that \( \textit{seqs} \notin \textit{free}_{f} \). No other element is added to \( \textit{free} \) until the end of \( E_f \). Thus, \( \textit{seqs} \notin \textit{free}_f \), as needed.

For \( f = j - 1 \), Claim 5.7 implies that \( \textit{seqs} \notin \textit{candidates}_{j-1} \), and Claim 5.8 implies that \( \textit{seqs} \notin \textit{free}_{j-1} \). By Lemma 5.3, only the first \( \text{SCAN} \) of epoch \( E_j \) adds elements to \( \textit{free} \) by executing line 10. Let \( \textit{free}_f \) denote set \( \textit{free} \) after the execution of this line. By the code, \( \textit{free}_f = \textit{free}_{j-1} \cup \textit{candidates}_{j-1} \). It follows that \( \textit{seqs} \notin \textit{free}_f \). All \( \text{SCANS} \) of epoch \( E_j \) (including \( S \)) choose their sequence numbers from \( \textit{free}_f \). However, \( S \) choose \( \textit{seqs} \) as its sequence number which does not exist in \( \textit{free}_f \). This is a contradiction.

Lemma 4.2 applies for RT-Opt and its proof is similar as for RT. To prove the rest of the lemmas presented in Section 3, exactly the same arguments as for T-Opt are applied for RT-Opt.

Theorem 5.9 RT-Opt is a linearizable, wait-free, single-scanner, multi-writer snapshot implementation from \( O(mn) \) bounded-size registers that achieves time complexity \( O(m) \) for \textit{SCAN} and \( O(1) \) for \textit{UPDATE}.

6 C-Snap Algorithm

Pseudo-code for C-Snap is presented in Algorithm 5. C-Snap works in the same spirit as T-Opt. The arrays \( \textit{pre} \) and \( \textit{post} \) play a similar role as for T-Opt. However, \( \textit{post} \) consists of only one row of \( m \) registers, each containing a component value and a sequence number. The use of sequence numbers and \textit{CAS} to change \( \textit{post} \) (line 4) guarantee that an \textit{UPDATE} on some component \( A_{i,1} \leq i \leq m \),
writes a value to post[i] only if the UPDATE is recent enough to write useful information. Notice that the UPDATE for C-Snap is almost similar to the UPDATE for T-Opt. We remark that a CAS on post[i] by some UPDATE succeeds only if post[i] = ⊥ (lines 3 and 4).

Since C-Snap is multi-scanner, many different processes may execute SCANS concurrently. The important work of a SCAN is performed by grab.scan. Each grab.scan tries to obtain a consistent vector (line 16). This is done in exactly the same way as for T-Opt (compare take.view of C-Snap with code lines 8-12 of T-Opt). Then, it tries to store this vector into seq (which now stores more information than simply a sequence number) using CAS (lines 17-18). We call the CAS executed in line 18CAS of type 0. The last task of grab.scan is to increase the sequence number of seq (line 20) using CAS (line 24). We call the CAS executed in line 24CAS of type 1. The code has been designed so that successful CAS of type 0 alternate with successful CAS of type 1. This is achieved by using the grab bit of seq. CAS of type 0 succeed only if the grab bit of seq is true (line 17) and change it to false (line 18). On the contrary, CAS of type 1 succeed only if this bit is false (line 21) and change it to true (line 24). Moreover, CAS of type 0 change only the view field of seq, while CAS of type 1 change only seq.tm writing there the next sequence number (lines 20, 24).

Algorithm 5 C-Snap Algorithm

```c
struct sq {
    int tm;
    bool scan grab;
    data view[1..m];
};

struct post_data {
    int tm;
    data value;
};
shared sq seq=c1,true,<⊥,...,⊥>; //
shared post_data post[1..m]=
    {<0, ⊥,...,<0, ⊥>}; //
shared data pre[1..m]=<⊥,...,⊥>;

void update(data value, int i){
    sq s;
    data d1;
    post_data lpost;
    1 s=seq;
    2 d1=pre[i];
    3 lpost=<tm-1, ⊥>;
    4 CAS(post[i], lpost, <post.tm, value>);
    5 pre[i]=value;
}

data [] take.view( void){
    int j;
    data view[1..m], d1, d2;
    6 for(j=1; j≤ m; j++){
        7 d1=pre[j];
        8 if(d2=⊥) view[j]=d1;
        9 else view[j]=d2;
    }
    11 return view;
}

data [] scan( void){
    grab_scan();
    grab_scan();
    return seq.view;
}

void grab_scan( void){
    sq curr_seq;
    data view[1..m], lview[1..m];
    int ltm;
    15 curr_seq=seq;
    16 view=take.view();
    17 if(curr_seq.grab=true)
        18 CAS(seq, curr_seq, <curr_seq.tm, false, view>);
    19 clear_registers(curr_seq);
    20 ltm=curr_seq.tm+1;
    21 curr_seq.grab=false;
    22 lview=seq.view;
    23 curr_seq.view=lview;
    24 CAS(seq, curr_seq, <ltm, true, lview>);
}

void clear_registers( sequence a){
    int j;
    post_data p, lpost;
    25 for(j=1; j≤ m; j++)
        26 p=post[j];
        27 lpost=<tm-1, p.value>;
        28 CAS(post[j], lpost, <ts, ⊥>);
        29 p=post[j];
        30 lpost=<tm-1, p.value>;
        31 CAS(post[j], lpost, <ts, ⊥>)
}
```

Each execution $\alpha$ of C-Snap can be split into phases as follows. The first phase starts with the initial configuration. A phase ends just before a (new) sequence number is written to $seq$.tm by a successful CAS of type 1 and the next phase starts with this CAS. Roughly speaking, the code has been designed so that each phase emulates the execution of a SCAN by the single-scanner T-Opt.

The use of sequence numbers and CAS guarantee that no SCAN or UPDATE that has started its execution in a previous phase succeeds in writing any of the CAS registers ($seq$ and $post$) in the current phase. Only SCANS starting in the current phase may write into $seq$ and out of these only one will succeed to do so (because of the grab bit) writing a new vector of values into $seq$. Once this is performed, the system enters an initialization period for $post$ registers which ends at the beginning of the next phase (that is, the system is in this period as long as the grab bit equals $false$). Once the $value$ field of some register $post[i]$ is changed to $\bot$ during this period, the use of sequence numbers and CAS ensure that no other CAS on this register succeeds until the beginning of the next phase. Thus, all $m$ registers of $post$ have the value $\bot$ stored in its $value$ fields when a new phase starts. After the new sequence number has been written for this phase and up to the point that the new vector is written to $seq$ (by the subsequent successful CAS of type 0), no clear_registers succeeds to write $\bot$ to $post[i]$. Thus, if the grab.scan that succeeds to write the new vector in the current phase, reads a value other than $\bot$ in $post[i]$, then this value has been written by an UPDATE on $A_i$ that has started its execution in the current phase. For such components, the grab.scan uses the value found in $post[i]$ (as was done by T-Opt).

A SCAN $S$ returns the vector contained in $seq$ by executing line 14. This vector has been written there by the last successful CAS of type 0 (denote by $C_0^S$ this CAS) that precedes the execution of line 14 by $S$. Since $C_0^S$ is of type 0, it does not change $seq$.tm, the value of which has been written by the last CAS of type 1 that precedes $C_0^S$ (denote by $C_1^S$ this CAS). For C-Snap, $C_1^S$ plays the same role as $w_g$ for T-Opt. To guarantee that $C_1^S$ is within the execution interval of $S$, grab.scan is called twice by $S$ (lines 12-13). It can be proved that the execution interval of each of these grab.scans contains a successful CAS of type 1. Between these successful CAS of type 1, a successful CAS of type 0 has been executed. So, $C_1^S$ is executed in the execution interval of $S$.

**Time and Space Complexity.** By the code, it is obvious that C-Snap has $O(1)$ time complexity for UPDATE and $O(m)$ time complexity for SCAN. C-Snap uses $m+1$ CAS registers and $m$ read-write registers. All read-write registers store just a single value. The $m$ CAS registers of $post$ store a pair of a value and a sequence number which can be as big as the number of executed SCANS; $seq$ stores a vector of $m$ values, a sequence number and a bit.

**Linearizability.** Let $\alpha$ be an execution of C-Snap and let $S$ be any SCAN performed in $\alpha$. Denote by $g_0^S$ the grab.scan that performs $C_0^S$. Fix any $i$, $1 \leq i \leq m$. In case $g_0^S$ reads $\bot$ in $post[i]$ and some value $v_i$ in $pre[i]$, let $U_0^S$ be the UPDATE that last writes to $pre[i]$ (line 4) before $g_0^S$ reads $pre[i]$ (line 7). If $g_0^S$ reads a value $v_i$ other than $\bot$ in $post[i]$, we denote by $V_1^S$ the UPDATE whose CAS on $post[i]$ (line 4) is the last successful CAS on $post[i]$ before $g_0^S$ reads $post[i]$ (line 8). By the code, it follows that $V_1^S$ must have read the value $v_i$ in $pre[i]$. We denote by $U_1^S$ the UPDATE on $A_i$ that last writes to $pre[i]$ before $V_1^S$ reads $pre[i]$. Let $w_i^S$ be the write to $pre[i]$ by $U_1^S$ (line 5).

Each SCAN $S$ is linearized at $C_1^S$. For each $i \in \{1, \ldots, m\}$, if $w_i^S$ follows $C_1^S$ (and $U_1^S$ has not already been linearized)\(^1\), we place the linearization point of $U_1^S$ just before $C_0^S$. We also place the linearization point of each UPDATE on $A_i$ that performs its write to $pre[i]$ between $C_1^S$ and $w_i^S$ (and has not been linearized yet) just before $C_1^S$; ties are broken by the order that the writes to register $pre[i]$ 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 at its write to $pre[i]$.

\(^1\)It might happen that for two SCANS $S$ and $S'$, $g_0^S = g_0^{S'}$, so that both SCANS constraint the linearization points of the same UPDATES. This is why the parenthesis is needed.
The proof of linearizability of C-Snap requires to prove all claims stated in the description of the algorithm provided above. So, the linearizability proof of C-Snap is much longer than the proofs of the other algorithms.

Let $\alpha$ be an execution of C-Snap and let $S$ be any SCAN performed in $\alpha$. Recall that the CAS instructions of line 18 (executed on seq by grab_scans) are called CAS instructions of type 0, and the CAS instructions of line 24 are called CAS of type 1. Recall also that $C_0^S$ is the last successful CAS instruction of type 0 which precedes the end of $S$ (line 14), and $C_1^S$ is the last successful CAS instruction of type 1 which precedes $C_0^S$. Denote by $\bar{r}_i^S$ the read of $\text{pre}[i]$ by $g_0^S$, and by $\bar{r}_i^S$ the read of $\text{post}[i]$ by $g_0^S$. If $g_0^S$ reads a value $u_i$ other than $\bot$ in $\text{pre}[i]$, let $c_{\text{pre}}$ be the read of $\text{pre}[i]$ by $V_i^S$ (line 2). Denote by $\alpha^C$ the subsequence of $\alpha$ which contains all the successful CAS instructions on $\text{seq}$ (type 0 and type 1) in the order they appear in $\alpha$; let $|\alpha^C|$ denote the number of CAS instructions in $\alpha^C$. Denote by $\alpha_i^C (\alpha_i^C)$, the subsequence of $\alpha^C$ that contains all the successful CAS of type 1 (type 0, respectively) in the order they are performed in $\alpha$; let $|\alpha_i^C|$ (|$\alpha_i^C|$) denote the number of CAS in $\alpha_i^C (\alpha_i^C)$, respectively.

The following lemma is a direct consequence of the definition of $U_t^S$. It implies that $w_t^S$ precedes $C_0^S$ and therefore also the end of $g_0^S$.

**Lemma 6.1** For each $i \in \{1, \ldots, m\}$, (1) $\bar{r}_i^S$ follows $w_i^S$, and (2) $C_0^S$ follows $w_i^S$.  

**Proof:** Assume first that $S$ reads $\bot$ in $\text{post}[i]$. Then, by the definition of $U_t^S$, $w_i^S$ precedes $r_i^S$ (since $r_i^S$ reads the value written to $\text{pre}[i]$ by $w_i^S$). Since $r_i^S$ precedes $\bar{r}_i^S$ and $\bar{r}_i^S$ precedes $C_0^S$, the claim follows.

Assume now that $S$ reads a value $u_i$ other than $\bot$ in $\text{post}[i]$. Then, $V_i^S$ is well-defined. By the definitions of $V_i^S$ and $U_i^S$, the following hold: (1) $V_i^S$ reads in $\text{pre}[i]$ the value written there by $w_i^S$, so $\text{pre}_i$ occurs after $w_i^S$, and (2) $\bar{r}_i^S$ reads in $\text{post}[i]$ the value written there by $V_i^S$, so $\bar{r}_i^S$ follows the CAS on $\text{post}[i]$ by $V_i^S$. By the code, the CAS on $\text{post}[i]$ by $V_i^S$ follows $\text{pre}_i$. So, $\bar{r}_i^S$ follows $w_i^S$, as needed by (1). Since $g_0^S$ first executes $\bar{r}_i^S$ and then $C_0^S$, it follows that $C_0^S$ follows $w_i^S$, as needed by (2).

Consider any execution $\alpha$ of C-Snap, and let $\alpha^C = C_0^S C_1^S \ldots C_k^S$, where $k = |\alpha^C| - 1$. The following lemma states that in $\alpha^C$ CAS instructions of type 0 alternate with CAS instructions of type 1 starting with a CAS of type 0.

**Lemma 6.2** For each $i \in \{0, \ldots, k\}$, $C_i^S$ is of type $(i \mod 2)$.

**Proof:** By induction on $i$.

**Base Case.** We prove that $C_0^S$ is of type 0. $C_0^S$ is the first successful CAS on $\text{seq}$ in $\alpha$. Assume, by the way of contradiction, that $C_0^S$ is of type 1. By the code (line 21), it follows that $\text{curr.seq.grab} = \text{false}$ just before $C_0^S$ is executed. Since $C_0^S$ is successful, it must be that $\text{seq.grab} = \text{false}$ just before $C_0^S$. However the initial value of $\text{seq.grab}$ is $\text{true}$ and register $\text{seq}$ is modified only by successful CAS instructions. Thus $C_0^S$ cannot be successful, which is a contradiction.

**Induction hypothesis.** Fix some $i$, $1 \leq i \leq k$, and assume that the claim holds for $i - 1$; that is, $C_{i-1}^S$ is of type $((i - 1) \mod 2)$.

**Induction step.** We prove that the claim holds for $i$. Let $p'$ be the process that performs $C_{i-1}^S$, and let $p$ be the process that performs $C_i^S$ (notice that it might be $p = p'$). Denote by $g_p$ the $\text{grab}$.\text{scan}$ that performs $C_i^S$. We consider the following cases.

Let $C_{i-1}^S$ be of type 0. Assume, by the way of contradiction, that $C_i^S$ is also of type 0. Then, it must be that $g_p$ reads in $\text{seq}$ (line 15) the value true for $\text{seq.grab}$ since otherwise the condition of the if statement (line 17) would be evaluated to false (and $C_i^S$ would not be performed). Thus, 20
curr_seq.grab = true when \( C^\alpha_i \) is executed. Since \( C^{\alpha}_{l-1} \) is of type 0, all CAS instructions of type 0 write false to seq.grab, and no successful CAS instruction on seq is executed between \( C^{\alpha}_{l-1} \) and \( C^\alpha_i \), seq.grab is false when \( C^\alpha_i \) is executed. Thus, \( C^\alpha_i \) cannot be successful, which is a contradiction.

Let now \( C^{\alpha}_{l-1} \) be of type 1. Assume, by the way of contradiction, that \( C^\alpha_i \) is also of type 1. Then, by the code (line 21), it holds that curr_seq.grab = false when \( C^\alpha_i \) is executed. Since \( C^{\alpha}_{l-1} \) is of type 1, all CAS instructions of type 1 write true to seq.grab, and no successful CAS on seq is executed between \( C^{\alpha}_{l-1} \) and \( C^\alpha_i \), seq.grab is true when \( C^\alpha_i \) is executed. Thus, \( C^\alpha_i \) cannot be successful, which is a contradiction.

The following lemma states that seq.tm changes only by CAS of type 1, while seq.view changes only by CAS of type 0.

**Lemma 6.3** The following hold:

1. \( C^\alpha_i \) writes in seq the initial value for seq.tm.
2. For each \( i, 1 \leq i \leq k \), it holds that:
   
   (a) If \( i \mod 2 = 0 \), \( C^\alpha_i \) stores in seq.tm the same value as \( C^\alpha_{i-1} \);
   
   (b) If \( i \mod 2 = 1 \), \( C^\alpha_i \) stores in seq.view the same vector as \( C^\alpha_{i-1} \).

**Proof:** We first prove 1. Recall that \( C^\alpha_0 \) is the first successful CAS on seq in \( \alpha \), so seq has its initial value before \( C^\alpha_0 \). Thus, if \( p \) is the process that executes \( C^\alpha_0 \), \( p \) reads in seq its initial value (line 15). Lemma 6.2 implies that \( C^\alpha_0 \) is of type 0. By the code (line 18), \( C^\alpha_0 \) writes in seq.tm the value of curr_seq.tm. So, \( C^\alpha_0 \) writes in seq.tm its initial value, as needed.

We continue to prove claim 2a. Consider any \( i, 1 \leq i \leq k \) such that \( i \mod 2 == 0 \). Lemma 6.2 implies that \( C^\alpha_i \) is of type 0. Suppose that \( C^{\alpha}_{l-1} \) writes \( t \) to seq.tm. Since there are no successful CAS on seq between \( C^{\alpha}_{l-1} \) and \( C^\alpha_i \), seq.tm = \( t \) when \( C^\alpha_i \) is executed. Since \( C^\alpha_i \) is successful, it follows by the code (line 18) that curr_seq.tm = \( t \), as needed.

We finally prove claim 2b. Consider any \( i, 1 \leq i \leq k \) such that \( i \mod 2 == 1 \). Lemma 6.2 implies that \( C^\alpha_i \) is of type 1. Suppose that \( C^{\alpha}_{l-1} \) writes the vector view to seq.view. Since there are no successful CAS on seq between \( C^{\alpha}_{l-1} \) and \( C^\alpha_i \), seq.view = view when \( C^\alpha_i \) is executed. Since \( C^\alpha_i \) is successful, it follows by the code (line 24) that curr_seq.view = seq.view when \( C^\alpha_i \) is executed. By the code (line 23), curr_seq.view = lview, so \( C^\alpha_i \) does not change the value of seq.view, as needed.

We next prove that a successful CAS of type 1 increases the value of seq.tm by 1.

**Lemma 6.4** Let \( C_1 \) and \( C'_1 \) be two successful CAS instructions of type 1 in \( \alpha \) such that there is no successful CAS of type 1 between \( C_1 \) and \( C'_1 \). If \( C_1 \) writes \( t \) in seq.tm, then \( C'_1 \) writes \( t + 1 \) in seq.tm.

**Proof:** Assume, by the way of contradiction, that \( C'_1 \) does not write \( t + 1 \) to seq.tm. Let \( g' \) be the grab.scan that performs \( C'_1 \), and let \( t' \) be the value \( g' \) reads in seq.tm (line 15). By the code (lines 20, 24), \( C'_1 \) writes \( t' + 1 \) \( \neq t + 1 \) to seq.tm. Lemma 6.3 implies that seq.tm changes only by CAS of type 1. Since no successful CAS of type 1 is performed between \( C_1 \) and \( C'_1 \), it follows that seq.tm = \( t \) when \( C'_1 \) is performed. However, by the code it follows that curr_seq.tm = \( t' \neq t \) when \( C'_1 \) is performed. Thus, \( C'_1 \) cannot be successful, which is a contradiction.
Assume that \(|I_0^C| = k_1\), and let \(I_0^C = C_1^C \ldots C_{k_1}^C\). Let \(C_1^0\) denote the initial configuration of the system. Notice that the initial value of \(seq.tm\) is 1. By Lemma 6.3, this value does not change until \(C_1^1\). By Lemma 6.3, \(seq.tm\) changes only by successful CAS of type 1. These and Lemma 6.4 imply:

**Corollary 6.5** For each \(j, 1 \leq j \leq k_1\), it holds that \(seq.tm = j\) at all points in time between \(C_1^{j-1}\) and \(C_1^j\).

Let \(S\) be any scan operation executed by some process \(p\) in \(\alpha\), and let \(g\) be any of the two grab scans executed by \(S\). Denote by \(r_{seq}\) the read of \(seq\) by \(g\) (line 15), and by \(C_1\) the CAS of type 1 executed by \(g\) (line 24). In order to prove that SCANS are linearized within their execution intervals, we first prove that a successful CAS of type 1 is executed within the execution interval of \(g\).

**Lemma 6.6** The execution interval of \(g\) contains at least one successful CAS of type 1.

**Proof:** Recall that \(g\) starts by executing \(r_{seq}\) and ends by executing \(C_1\). If \(C_1\) is a successful CAS, the claim follows. Suppose that \(C_1\) fails. Assume, by the way of contradiction, that there is no successful CAS of type 1 executed between \(r_{seq}\) and \(C_1\). We consider the following cases.

Assume that \(curr.seq.tm \neq seq.tm\) when \(C_1\) is executed. Lemma 6.3 implies that \(seq.tm\) changes only by the execution of successful CAS of type 1. Thus, there is some successful CAS of type 1 that is executed between \(r_{seq}\) and \(C_1\). This is a contradiction (to our assumption above).

Assume now that \(curr.seq.tm = seq.tm\) when \(C_1\) is executed. Let \(curr.seq.grab \neq seq.grab\) when \(C_1\) is executed. Since \(C_1\) is of type 1, by the code (line 21) it follows that \(curr.seq.grab = false\) when \(C_1\) is executed. Thus, \(seq.grab = true\) when \(C_1\) is executed. By the code, only CAS of type 1 write the value \(true\) to \(seq.grab\). It follows that the last CAS executed on \(seq\) before \(C_1\) must be a CAS of type 1. Let \(C_1'\) be this CAS. If \(C_1'\) is executed between \(r_{seq}\) and \(C_1\), the claim holds.

So, assume that \(C_1'\) is executed before \(r_{seq}\). Then, \(r_{seq}\) reads \(true\) in \(seq.grab\) (line 15), so that the condition of the if statement of line 17 is evaluated to \(true\) and the CAS of type 0 (line 18) is executed by \(g\); let \(C_0\) be this CAS. Since the last successful CAS on \(seq\) before \(C_1\) is \(C_1'\) which is a CAS of type 1, no CAS of type 0 has been executed between \(r_{seq}\) and \(C_0\), so \(curr.seq = seq\) when \(C_0\) is executed. Therefore, \(C_0\) succeeds. This is a contradiction since the last CAS on \(seq\) that is executed before \(C_1\) is \(C_1'\).

Assume now that \(curr.seq.grab = seq.grab\) when \(C_1\) is executed. Recall that additionally, \(curr.seq.tm = seq.tm\) when \(C_1\) is executed. Since \(C_1\) fails, it must be that \(curr.seq.view \neq seq.view\). Lemma 6.3 implies that \(seq.view\) changes only by successful CAS of type 0. Since \(C_1\) fails, it holds that a successful CAS of type 0, let it be \(C_0'\), is executed between the execution of line 22 and line 24 by \(g\). Clearly, \(C_0'\) is executed by a process \(p'\) other than \(p\). By Lemma 6.2, the last successful CAS before \(C_0'\) is of type 1. Let \(C_1'\) be this CAS. If \(C_1'\) is executed between \(r_{seq}\) and \(C_1\), then the claim holds. So, assume that \(C_1'\) is executed before \(r_{seq}\). Recall that there are no successful CAS on \(seq\) from \(r_{seq}\) (executed by \(p\)) to \(C_0'\) (executed by \(p'\)). Since \(C_1\) writes \(true\) in \(seq.grab\), \(r_{seq}\) reads \(true\) in \(seq.grab\). So, the condition of the if statement of line 17 is evaluated to \(true\), and the CAS of type 0 (let it be \(C_0\)) is executed by \(p\). Moreover, since \(C_0\) precedes \(C_0'\), it follows that there are no successful CAS on \(seq\) from \(r_{seq}\) to \(C_0\). Thus, \(curr.seq = seq\) when \(C_0\) is executed. Thus, \(C_0\) is a successful CAS of type 0. \(C_0\) occurs before \(C_0'\) since \(C_0'\) is performed between the execution of line 22 and line 24 by \(p\) (which come after the execution of \(C_0\)). This is a contradiction (since the last successful CAS on \(seq\) that precedes \(C_0'\) is \(C_1'\) which is a CAS of type 1).
Lemma 6.7 Let $S$ by any SCAN executed in $\alpha$. Then, $S$ is linearized within its execution interval.

Proof: By the code (lines 12-13), $S$ executes twice function \texttt{grab.scan}. Let $g$ and $g'$ be the execution of the first and second \texttt{grab.scan}, respectively, by $S$. Denote by $r_{\text{seq}}$ and $r'_{\text{seq}}$ the reads of \texttt{seq} (line 15) performed by $g$ and $g'$, respectively, and let $C_1$ and $C'_1$ be the CAS of type 1 (line 24) executed by $g$ and $g'$, respectively. Lemma 6.6 implies that the execution interval which starts with $r_{\text{seq}}$ and ends with $C_1$, contains a successful CAS $C_{\text{suc}}$ of type 1. Similarly, the execution interval that starts with $r'_{\text{seq}}$ and ends with $C'_1$ contains a successful CAS $C'_{\text{suc}}$ of type 1. Lemma 6.2 implies that there is a successful CAS of type 0 between $C_{\text{suc}}$ and $C'_{\text{suc}}$. So, by the way linearization points are assigned, $S$ is linearized at $C_{\text{suc}}$ or at some later point. Thus, $S$ is linearized after the beginning of its execution interval. Furthermore, by the way linearization points are assigned, it is obvious that $S$ is linearized before the end of its execution interval. We conclude that $S$ is linearized within its execution interval. ■

The next lemma proves that exactly one CAS on a register \texttt{post[i]}, $1 \leq i \leq m$, writes $\bot$ to the value field of \texttt{post[i]} during each phase; moreover, after the execution of this CAS, the value field of \texttt{post[i]} remains $\bot$ up to the beginning of the next phase. Some useful properties of sequence numbers are also proved.

Lemma 6.8 Consider any $j$, $1 \leq j \leq k_1$ and any $i$, $1 \leq i \leq m$. Then,

1. between $C_1^{j-1}$ and $C_1^j$, exactly one CAS \texttt{Cpost} on \texttt{post[i]} by a clear_registers is successful,

2. $\texttt{post[i].tm} = j - 1$ between $C_1^{j-1}$ and $C_1^j$, and

3. $\texttt{post[i].tm} = j$ and $\texttt{post[i].value} = \bot$ between $C_{\text{post}}$ and $C_1^j$.

Proof: By induction on $j$, $1 \leq j \leq k_1$. Fix any $j$, $1 < j \leq k_1$ and assume that the claim holds for $j - 1$. We prove that the claim holds for $j$.

Lemma 6.3 implies that only CAS of type 1 change \texttt{seq.tm}. In case $j = 1$, since $C_1^1$ is the first CAS of type 1 executed in $\alpha$, it follows that \texttt{seq.tm} has its initial value 1 at all points in time between $C_1^0$ and $C_1^1$. Thus, any process that starts before $C_1^1$ reads 1 in \texttt{seq.tm} (lines 1, 15). Recall also that, for each $i$, $1 \leq i \leq m$, initially, $\texttt{post[i].tm} = 0$ and $\texttt{post[i].value} = \bot$. In case $j > 1$, Corollary 6.5 implies that immediately after $C_1^{j-1}$, it holds that \texttt{seq.tm} = $j$. Since no CAS of type 1 is successful between $C_1^{j-1}$ and $C_1^j$, \texttt{seq.tm} = $j$ at all points in time between $C_1^{j-1}$ and $C_1^j$. Moreover, Corollary 6.5 implies that all operations whose executions have started before $C_1^j$ read a value less or equal to $j$ in \texttt{seq.tm} (lines 1 and 15). Moreover, by induction hypothesis, immediately after $C_1^{j-1}$ it holds that $\texttt{post[i].tm} = j$ and $\texttt{post[i].value} = \bot$. Thus, in either case we have that \texttt{seq.tm} equals $j$ at all times between $C_1^{j-1}$ and $C_1^j$, and all operations that start before $C_1^j$ read a value less than or equal to $j$ in \texttt{seq.tm}. Additionally, immediately after $C_1^{j-1}$, it holds that $\texttt{post[i].tm} = j - 1$ and $\texttt{post[i].value} = \bot$.

Let $p$ be the process that performs $C_1^j$. Since $C_1^j$ is successful and \texttt{seq.tm} = $j$ when $C_1^j$ is executed, it follows by the code that $p$ reads the value $j$ in \texttt{seq.tm} (line 15).

Fix any $i$, $1 \leq i \leq m$. We prove the following two useful claims.

Claim 6.9 Let $C$ be the first successful CAS on \texttt{post[i]} that is executed between $C_1^{j-1}$ and $C_1^j$ by some clear_registers. Then,

1. all successful CAS on \texttt{post[i]} executed between $C_1^{j-1}$ and $C$ does not change the value of \texttt{post[i].tm}, and

24
2. no CAS on post[i] succeeds between C and C\textsubscript{\textup{\textEnc}}.

\textbf{Proof:} We start by proving 1. Since \emph{C} is the first successful CAS on \emph{post[i]} executed between \emph{C_{\textsubscript{\textEnc}}} and \emph{C_{\textsubscript{\textEnc}}} by clear_registers, any successful CAS on \emph{post[i]} between \emph{C_{\textsubscript{\textEnc}}} and \emph{C} is by an UPDATE. Recall that \emph{post[i]}:tm = \textit{j} – 1 immediately after \emph{C_{\textsubscript{\textEnc}}} is executed. Let \emph{U} be the first UPDATE that executes a successful CAS \emph{C_{U}} on \emph{post[i]} between \emph{C_{\textsubscript{\textEnc}}} and \emph{C_{\textsubscript{\textEnc}}}. Then, \emph{post[i]}:tm = \textit{j} – 1 when \emph{C_{U}} is executed. By the code (line 4), \emph{C_{U}} does not change the value of \emph{post[i]}:tm. Thus, \emph{seq.tm} equals \textit{j} – 1 after \emph{C_{U}}. Applying the above argument inductively, it follows that no CAS on \emph{post[i]} executed between \emph{C_{\textsubscript{\textEnc}}} and \emph{C} changes the value of \emph{post[i]}:tm, as needed.

We continue to prove 2. Claim 1 (proved above) implies that \emph{post[i]}:tm = \textit{j} – 1 when \emph{C} is executed. Assume that \emph{C} is executed by some process \emph{q}. Since \emph{C} is successful, it follows by the code (lines 27-28 and 30-31) that \emph{q} must have read \textit{j} in \emph{seq} (line 15), and that it writes \textit{j} in \emph{post[i]}:tm. Assume, by the way of contradiction, that there is at least one successful CAS on \emph{post[i]} between \emph{C} and \emph{C_{\textsubscript{\textEnc}}}. If \emph{C'} is the first such CAS, then when \emph{C'} is executed it holds that \emph{post[i]}:tm = \textit{j}. Since \emph{C'} precedes \emph{C_{\textsubscript{\textEnc}}}, the operation that executes \emph{C'} has started its execution before \emph{C_{\textsubscript{\textEnc}}}. Recall that all operations that start before \emph{C_{\textsubscript{\textEnc}}} read a value less than or equal to \textit{j} in \emph{seq.tm}. Thus, by the code (lines 3, 27 and 30), \emph{lpost.tm} \leq \textit{j} – 1 when \emph{C'} is executed. Therefore, \emph{C'} cannot be a successful CAS. This is a contradiction. Applying the above argument inductively, it follows that no CAS on \emph{post[i]} executed between \emph{C} and \emph{C_{\textsubscript{\textEnc}}} is successful, as needed. \hfill \triangle

\textbf{Claim 6.10} For each \emph{i}, 1 \leq \emph{i} \leq \textit{m}, there is some successful CAS on \emph{post[i]} executed by a clear_registers between \emph{C_{\textsubscript{\textEnc}}} and \emph{C_{\textsubscript{\textEnc}}}.

\textbf{Proof:} Fix any \emph{i}, 1 \leq \emph{i} \leq \textit{m}. By the code (line 19), \emph{p} calls clear_registers before it executes \emph{C_{\textsubscript{\textEnc}}}. Let \emph{C_{\textsubscript{\textEnc}}} and \emph{C_{\textsubscript{\textEnc}}} be the two CAS on \emph{post[i]} executed by \emph{p} (lines 28 and 31). Assume, by the way of contradiction, that no CAS on \emph{post[i]} by clear_registers succeeds between \emph{C_{\textsubscript{\textEnc}}} and \emph{C_{\textsubscript{\textEnc}}}. Claim 6.9 implies that \emph{post[i]}:tm = \textit{j} – 1 when \emph{C_{\textsubscript{\textEnc}}} is executed. Recall that \emph{p} reads \textit{j} in \emph{seq.tm} (line 15), so, the code (line 27), \emph{lpost.tm} = \textit{j} – 1 when \emph{C_{\textsubscript{\textEnc}}} is executed. Since we have assumed that \emph{C_{\textsubscript{\textEnc}}} fails and \emph{seq.tm} = \emph{lpost.tm} when \emph{C_{\textsubscript{\textEnc}}} is executed, it must be that \emph{lpost.value} \neq \emph{post[i].value} when \emph{C_{\textsubscript{\textEnc}}} is executed. Recall that \emph{post[i].value} = \perp immediately after \emph{C_{\textsubscript{\textEnc}}}. Thus, there is an UPDATE whose CAS on \emph{post[i]} succeeds between \emph{C_{\textsubscript{\textEnc}}} and \emph{C_{\textsubscript{\textEnc}}}. Let \emph{U} be the first such UPDATE and let \emph{C_{U}} be the CAS executed by \emph{U}. Claim 6.9 implies that \emph{C_{U}} writes the value \textit{j} – 1 to \emph{post[i].tm}; let \emph{v} be the value that \emph{C_{U}} writes to \emph{post[i].value}.

We argue that all CAS on \emph{post[i]} that are executed between \emph{C_{U}} and \emph{C_{\textsubscript{\textEnc}}} fail. Recall that all these CAS are executed by UPDATES. By the code (line 3), for each of these UPDATES, \emph{lpost.value} = \perp. Notice that \emph{post[i].value} = \emph{v} \neq \perp after the execution of \emph{C_{U}}, and therefore the CAS by each of these UPDATES fails. It follows that \emph{post[i].tm} = \textit{j} – 1 and \emph{post[i].value} = \emph{v} at all points between \emph{C_{U}} and \emph{C_{\textsubscript{\textEnc}}}. Thus, when \emph{p} performs its second read of \emph{post[i]} (line 29), it reads \emph{v} in \emph{post[i].value}, so \emph{lpost.value} = \emph{v} when \emph{C_{\textsubscript{\textEnc}}} is executed by \emph{p}. Recall that \emph{p} reads \textit{j} in \emph{seq.tm}. Thus, by the code, \emph{lpost.tm} = \textit{j} – 1 when \emph{C_{\textsubscript{\textEnc}}} is executed. Thus, \emph{C_{\textsubscript{\textEnc}}} is successful, which is a contradiction. \hfill \triangle

Claim 6.10 implies that there is a successful CAS on \emph{post[i]} executed by a clear_registers between \emph{C_{\textsubscript{\textEnc}}} and \emph{C_{\textsubscript{\textEnc}}}. By Claim 6.9, it follows that there is exactly one such CAS \emph{C_{post}}, as needed by 1. Recall that \emph{post[i].tm} = \textit{j} – 1 immediately after \emph{C_{\textsubscript{\textEnc}}} at all points between \emph{C_{\textsubscript{\textEnc}}} and \emph{C_{post}}, as needed by 2. It follows that \emph{post[i].tm} = \textit{j} – 1 when \emph{C_{post}} is executed. Therefore, by the code, it holds that \emph{lpost.tm} = \textit{j} – 1 and \emph{lpost.value} = \perp when \emph{C_{post}} is executed. So, \emph{C_{post}} writes the value \textit{j} to \emph{post[i].tm} and \perp to \emph{post[i].value}. Claim 6.9 implies that these values do not change since \emph{C_{\textsubscript{\textEnc}}}, as needed by 3. \hfill \blacksquare
We next prove that grab_scans which have started their execution in previous phases than the current one cannot change the value of any of the CAS registers in the current or later phases.

Lemma 6.11 Let $C_i^j$, $1 \leq j \leq k_1$, be any successful CAS of type 1 and let $g$ be a grab_scan that reads seq (line 15) before $C_i^j$. Then, after the execution of $C_i^j$, no CAS (on seq or on any of the post) by $g$ (and the function clear_registers called by $g$) is successful.

Proof: By Corollary 6.5, $C_i^j$ writes the value $j + 1$ to seq.tm. Let $q$ be the process that executes $g$. Since the execution of $g$ starts before $C_i^j$, $q$ reads in seq.tm a value less than or equal to $j$. Denote by $C$ any of the two CAS on post[i] executed by $q$ (during clear_registers). We first prove that after the execution of $C_i^j$, no CAS on seq by $g$ is successful. Corollary 6.5 implies that all values written to seq.tm after $C_i^j$ are larger than $j + 1$. Recall that $g$ reads in seq a value less than or equal to $j$. Thus, for $g$ it holds that curr.seq.tm $\leq j$, while seq.tm $> j$ at all points in time after $C_i^j$. It follows that if $g$ executes a CAS on seq after $C_i^j$ it will fail.

Fix any $i$, $1 \leq i \leq m$. We continue to prove that after the execution of $C_i^j$, no CAS on post[i] by $g$ is successful. Let $C$ be any such CAS. By Lemma 6.8 it follows that post[i].tm $\geq j$ at all points in time after $C_i^j$. Recall that $g$ reads a value less than $j + 1$ in seq.tm, so lpost.tm $< j$ when $C$ is executed. It follows that $C$ fails.

Next lemma proves that the initialization period of post registers for each phase starts after the execution of the successful CAS of type 0 of the phase. Consider any $j$, $1 \leq j \leq k_1$ and any $i$, $1 \leq i \leq m$. Denote by $C_0$ the successful CAS of type 0 that is executed between $C_i^{j-1}$ and $C_i^j$. Lemma 6.8 implies that there is a successful CAS on post[i] between $C_i^{j-1}$ and $C_i^j$. Denote by $C_{post}$ this CAS.

Lemma 6.12 $C_{post}$ is executed after $C_0$.

Proof: Assume first $j = 1$. Then, $C_0$ is the first CAS on seq (line 18) executed by any process. By the code, grab_scans first execute their CAS of type 0 and then call clear_registers. So, no clear_registers starts its execution before $C_0$. Therefore, $C_{post}$ follows $C_0$.

Let now $j > 1$. Lemma 6.11 implies that for all grab_scans that start their execution before $C_i^{j-1}$, their CAS (on any register) are not successful. Thus, $C_0$ and $C_{post}$ are executed by grab_scans that start their execution after $C_i^{j-1}$ (or by functions called by them). Let $G$ be the set of grab_scans that start their execution after $C_i^{j-1}$ and let $g$ be the grab_scan that first executes its CAS $C_0$ of type 0. Clearly, $C_0$ is executed before or at $C_0$. Thus, $C_0$ precedes $C_i^j$. By Lemma 6.3, seq does not change between $C_i^{j-1}$ and $C_i^j$. Thus, curr_seq = seq when $C_0$ is executed. So, $C_0$ is successful. Lemma 6.2 implies that the only successful CAS between $C_i^{j-1}$ and $C_i^j$ is $C_0$. Thus, it must be that $C_0 = C_0$, so $C_0$ is the first CAS of type 0 executed by any grab_scan of $G$. By the code, it follows that all clear_registers called by grab_scans in $G$ start their execution after $C_0$. We conclude that $C_{post}$ is executed after $C_0$.

We are now ready to prove that those UPDATES that perform their write to pre[i] between $C_i^S$ and $w_i^S$ have started their execution before $C_i^S$. This lemma is essential to prove that each UPDATE is linearized within its execution interval.

Lemma 6.13 For each $i \in \{1, \ldots, m\}$ such that $w_i^S$ follows $C_i^S$, it holds that any UPDATE on $A_i$ that performs its write to pre[i] between $C_i^S$ and $w_i^S$ (including $U_i^S$) begins its execution before $C_i^S$. 

25
Proof: Assume that $C^S_j = C^S_i$ for some $j, 1 \leq j \leq k_1$. Let $C_{post}$ be the first successful CAS on $post[i]$ that is executed by a clear_registers after $C^S_i$. Lemma 6.12 implies that $C_{post}$ is executed after $C^S_j$. By Corollary 6.5, $C^S_j$ writes the value $j + 1$ in seq tm. Lemma 6.8 implies that $post[i].tm = j$ between $C^S_j$ and $C_{post}$. Since $C_{post}$ follows $C^S_j$, it follows that $post[i].tm = j$ between $C^S_j$ and $C^S_i$.

Assume, by the way of contradiction, that there is at least one UPDATE on $A_1$ that starts its execution after $C^S_j$ and performs its write to $pre[i]$ before $w^S_j$. Denote by $U$ the set of UPDATES that start their executions after $C^S_j$ and perform their writes to $pre[i]$ before $w^S_j$. Let $U \in U$ be the UPDATE of $U$ that executes first its CAS $C$ on $post[i]$. By Lemma 6.1, $C^S_i$ follows $w^S_j$, so $C$ precedes $C^S_j$.

Recall that $C^S_j$ writes $j + 1$ to seq tm. Lemmas 6.2 and 6.3 imply that $seq \cdot tm = j + 1$ from $C^S_j$ to $C^S_{j+1}$ (or to the end of the execution if $j = k_1$). Since $U$ starts after $C^S_j$ and performs $C$ before $C^S_i$, it follows that $U$ reads $j + 1$ in $seq \cdot tm$. Thus, by the code, $l_{post}.tm = j$ when $C$ is executed. Recall that $post[i].tm = j$ between $C^S_j$ and $C^S_i$. Since $C$ precedes $C^S_j$, $post[i].tm = j$ when $C$ is executed. So, $post[i].tm = l_{post}.tm$ when $C$ is executed. By Lemma 6.8, $post[i].value = \bot$ immediately after $C^S_i$. Lemma 6.8 implies that the only CAS on $post[i]$ executed by clear_registers between $C^S_j$ and $C^S_{j+1}$ (or the end of the execution if $j = k_1$) is $C_{post}$ which follows $w^S_j$ and therefore also $C$. Since $C$ is the first CAS on $post[i]$ executed by an UPDATE after $C^S_j$, it follows that $post[i].value = \bot$ when $C$ is executed. By the code (line 3), $l_{post}.value = \bot$ when $C$ is executed. It follows that $C$ is a successful CAS on $post[i]$.

By Lemma 6.1, $w^S_j$ precedes $r^S_j$. So, $C$ precedes $r^S_j$. Lemma 6.12 implies that no clear_registers writes the value $\bot$ in $post[i]$ between $C^S_j$ and $C^S_i$. Since $r^S_j$ precedes $C^S_j$, $r^S_j$ reads a value other than $\bot$ in $post[i]$. Thus, $V^S_j$ is well-defined. Moreover, $V^S_j \notin U$ since $V^S_j$ reads the value written to $pre[i]$ by $w^S_j$, and therefore it performs its write to $pre[i]$ after $w^S_j$. So, $V^S_j \neq U$.

By definition, $V^S_j$ performs its CAS $C'$ on $post[i]$ before $r^S_j$ (since $r^S_j$ reads the value written in $post[i]$ by $C'$). Thus, $C'$ precedes $C^S_j$. Moreover, $C'$ follows $w^S_j$ and therefore $C'$ follows $C$. It follows that $post[i].value \neq \bot$ when $C'$ is executed. By the code (line 3), $l_{post}.value = \bot$ when $C'$ is executed. Thus, $C'$ is not a successful CAS, which is a contradiction.

Lemma 6.14 Let $U$ be any UPDATE executed in $\alpha$. Then, $U$ is linearized within its execution interval.

Proof: Let $U$ be an UPDATE on $A_1$ which is not linearized at its write to $pre[i]$. By the way that linearization points are assigned, there is a SCAN $S$ such that $w^S_j$ of $U^S_j$ is executed after $C^S_j$, the write to $pre[i]$ by $U$ is executed between $C^S_j$ and $w^S_j$, and $U$ is linearized just before $C^S_j$. Obviously, the execution of $U$ ends after $C^S_j$. Lemma 6.13 implies that $U$ begins its execution before $C^S_j$. Thus, $U$ is linearized within its execution interval.

Consider two SCANS that return vectors written by different grab/scans. The CAS of type 0 that writes the first vector is executed at a different phase than the CAS of type 0 that writes the other vector (since it is not possible to have two successful CAS of type 0 in the same phase).

Lemma 6.15 Let $S_1$ and $S_2$ be two SCANS such that $g^{S_1}_c \neq g^{S_2}_c$. Then, $\alpha(C^S_1, C^{S_1}_0)$ and $\alpha(C^S_2, C^{S_2}_0)$ do not intersect.

Proof: Since $g^{S_1}_c \neq g^{S_2}_c$, $C^S_0 \neq C^{S_2}_0$. Without loss of generality, assume that $C^S_0$ precedes $C^{S_2}_0$. Lemma 6.2 implies that there is at least one successful CAS of type 1 between $C^S_0$ and $C^{S_2}_0$. Thus, $C^S_0 \neq C^S_1$ and $C^S_1$ follows $C^S_0$. Since $C^S_1$ precedes $C^{S_1}_0$, it follows that $\alpha(C^S_1, C^{S_1}_0)$ and $\alpha(C^S_0, C^{S_2}_0)$ do not intersect, as needed.
Next lemma proves that the linearization order of the \texttt{UPDATES} on any component \( A_i \) respects the order in which these \texttt{UPDATES} perform their writes to \texttt{pre[i]}. Its proof is in its biggest part identical to the proof of Lemma 3.5.

**Lemma 6.16** Let \( U_1, U_2 \) be two \texttt{UPDATES} on some component \( A_i \), \( 1 \leq i \leq m \). Denote by \( w_1 \) the write to \texttt{pre[i]} by \( U_1 \) and by \( w_2 \) the write to \texttt{pre[i]} by \( U_2 \). If \( w_1 \) precedes \( w_2 \), the linearization point of \( U_1 \) precedes the linearization point of \( U_2 \).

**Proof:** Assume, by the way of contradiction, that the claim does not hold. If \( U_1 \) and \( U_2 \) are linearized at their writes to \texttt{pre[i]}, then the claim holds trivially. Therefore, assume that at least one of the \( U_1, U_2 \) is not linearized at its write to \texttt{pre[i]}. We consider the following cases.

1. \( U_2 \) is linearized at \( w_2 \). Lemma 6.14 implies that \( U_1 \) is linearized within its execution interval, so \( U_1 \) is linearized at \( w_1 \). Thus, \( U_1 \) is linearized before \( U_2 \). A contradiction.

2. \( U_1 \) is linearized at \( w_1 \). Since we have assumed that \( U_2 \) is linearized before \( U_1 \), \( U_2 \) cannot be linearized at \( w_2 \). By the way linearization points are assigned, there is a \texttt{SCAN} \( S \) such that \( w_1 \) has been performed between \( C^S_0 \) and \( w_0^S \). Since \( w_0 \) precedes \( w_2 \), \( w_1 \) has been executed before \( w_0^S \). In case \( w_1 \) follows \( C^S_0 \), both \( U_1 \) and \( U_2 \) are linearized just before \( C^S_0 \) in the order that they perform their writes to \texttt{pre[i]}. Thus, \( U_1 \) is linearized before \( U_2 \), which is a contradiction. So, assume \( w_1 \) precedes \( C^S_0 \). Lemma 6.14 implies that \( U_1 \) is linearized the latest at \( w_1 \). By the way linearization points are assigned, \( U_2 \) is linearized just before \( C^S_0 \). Thus, \( U_1 \) is linearized before \( U_2 \). A contradiction.

3. None of \( U_1, U_2 \) is linearized at its write to \texttt{pre[i]}. By the way linearization points are assigned, there are \texttt{SCANS} \( S_1 \) and \( S_2 \) such that \( w_1 \) has been performed between \( C^S_1 \) and \( w_1^{S_1} \), and \( w_2 \) has been performed between \( C^S_2 \) and \( w_1^{S_2} \); moreover, \( U_1 \) is linearized just before \( C^S_0 \), and \( U_2 \) is linearized just before \( C^S_0 \). Lemma 6.1 implies that \( w_1^{S_1} \) precedes \( C^S_0 \), and \( w_1^{S_2} \) precedes \( C^S_0 \).

If \( g_0^{S_1} = g_0^{S_2} \), then \( C^S_0 = C^S_0 \), so \( C^S_0 = C^S_0 \). Thus, both \( U_1 \) and \( U_2 \) are linearized just before \( C^S_1 = C^S_2 \) in the order they perform their writes to \texttt{pre[i]}. So, \( U_1 \) is linearized before \( U_2 \), which is a contradiction.

If \( g_0^{S_1} \neq g_0^{S_2} \), Lemma 6.15 implies that \( \alpha(C^S_1, C^S_0) \) and \( \alpha(C^S_2, C^S_0) \) do not intersect. If \( C^S_1 \) precedes \( C^S_2 \), the linearization point of \( U_1 \) which is placed just before \( C^S_1 \) precedes the linearization point of \( U_2 \) which is placed just before \( C^S_2 \), which is a contradiction.

Assume finally that \( C^S_1 \) follows \( C^S_2 \). By Lemma 6.1, \( w_1^{S_2} \) precedes \( C_1^S \). and \( w_1^{S_2} \) precedes \( C_0^S \). Thus, \( w_1 \) which is executed between \( C_1^S \) and \( w_1^{S_2} \) follows \( w_2 \) which is executed between \( C_1^S \) and \( w_1^{S_2} \), which is a contradiction.

In all cases we derived a contradiction. Thus, we conclude that the linearization point of \( U_1 \) precedes the linearization point of \( U_2 \), as needed.

The following two technical lemmas are useful for proving the consistency of C-Snap.

**Lemma 6.17** For each \( i \in \{1, \ldots, m\} \), \( r_i^S \) follows \( C_i^S \).

**Proof:** Recall that \( r_i^S \) is the read of \texttt{pre[i]} by \( g_0^S \) which executes \( C_i^S \). By definition, \( C_i^S \) is the last successful \texttt{CAS} of type 1 that precedes \( C_i^S \). Assume that \( C_i^S \) writes the value \( j \) to \texttt{seq.trm}. Then,
Lemma 6.3 implies that $C_i^S$ also writes the value $j$ to $\text{seq.tm}$. Thus, by the code, $g_i^S$ reads $j$ in $\text{seq}$ (line 15). Corollary 6.5 implies that at all points in time before $C_i^S$, the value of $\text{seq.tm}$ is at most $j - 1$. Thus, $g_i^S$ performs its read of $\text{seq}$ (let it be $r_{seq}$) after $C_i^S$. Since $r_i^S$ follows $r_{seq}$, it follows that $r_i^S$ follows $C_i^S$, as needed.

Lemma 6.18 Assume that $S$ is a SCAN in $\alpha$ such that $V_i^S$ is well-defined, and let $r_{pre}$ be the read of $\text{pre[i]}$ by $V_i^S$. Then, $r_{pre}$ is executed after $C_i^S$.

Proof: Assume, by the way of contradiction, that $r_{pre}$ is executed before $C_i^S$.

Suppose that $C_j^S = C_i^j$, for some $j$, $1 \leq j \leq k_1$. Then, Corollary 6.5 implies that $C_j^S$ writes $j + 1$ to $\text{seq.tm}$. By definition of $V_i^S$, the CAS $C$ on $\text{post[i]}$ by $V_i^S$ is a successful CAS, and the value written in $\text{post[i].value}$ by $C$ is read by $r_i^S$. By Lemma 6.17, $r_i^S$ follows $C_i^S$. Since $r_i^S$ follows $r_i^S$, $r_i^S$ follows $C_i^S$. By Lemma 6.8, $\text{post[i].tm = p}$ immediately after $C_i^S$. By definition of $V_i^S$, the value written to $\text{post[i]}$ by $C$ is read by $r_i^S$. Since $r_i^S$ follows $C_i^S$, it must be that $C$ is executed after $C_i^S$. By Lemma 6.8, it follows that $\text{post[i].tm} \geq j$ at all points in time after $C_i^S$. Thus, $\text{post[i].tm} \geq j$ when $C$ is executed.

Since the read $r$ of $\text{seq}$ by $V_i^S$ precedes $r_{pre}$ and we have assumed that $r_{pre}$ precedes $C_i^S$, $r$ precedes $C_i^S$. Corollary 6.5 implies that $\text{seq.tm} \leq j$ at all points in time before $C_i^S$. Thus, $r$ reads a value for $\text{seq.tm}$ smaller than or equal to $j$. By the code, it follows that $\text{ipost.tm} < j$ when $C$ is executed. Since $\text{post[i].tm} \geq j$ and $\text{ipost.tm} < j$ when $C$ is executed, it follows that $C$ is not successful, which is a contradiction.

We are now ready to prove the consistency of C-Snap. Its proof is in its biggest part similar to the proof of Lemma 3.6.

Lemma 6.19 Let $\alpha$ be an execution of T-Opt. Any SCAN executed in $\alpha$ returns a consistent vector.

Proof: Assume that $S$ returns the vector $v = (v_1, \ldots, v_m)$. By definition of $C_i^S$ and by Lemma 6.2, it follows that $S$ returns the vector written in $\text{seq}$ by $C_i^S$. Thus, the vector calculated by $g_i^S$ is $v$. So, either $g_i^S$ reads $\bot$ in $\text{post[i].value}$ (line 8) and $v_i$ in $\text{pre[i]}$, $1 \leq i \leq m$ or $g_i^S$ reads $v_i$ in $\text{post[i].value}$. In either case, by definition of $U_i^S$, $U_i^S$ must write $v_i$ to $\text{pre[i]}$. Thus, $U_i^S$ uses $v_i$ as a parameter. In case $w_i^S$ precedes $C_i^S$, Lemma 6.14 implies that $U_i^S$ is linearized before $C_i^S$, where $S$ is linearized. In case $w_i^S$ follows $C_i^S$, by the way linearization points are assigned, the linearization point of $U_i^S$ precedes the linearization point of $S$. Thus, $U_i^S$ stores $v_i$ in component $A_i$ and its linearization point precedes the linearization point of $S$. We prove that there is no UPDATE on component $A_i$ with value $v \neq v_i$ that is linearized between $U_i^S$ and $S$, so that $S$ returns a consistent value for $A_i$.

Assume, by the way of contradiction, that there is an integer $i \in \{1, \ldots, m\}$ such that the last UPDATE on $A_i$ which has been linearized before $S$ is not $U_i^S$. Denote by $U$ this UPDATE, let $v \neq v_i$ be the value it writes to $\text{pre[i]}$, and let $w$ be the write to $\text{pre[i]}$ by $U$. We consider the following cases.

1. Assume that $g_i^S$ reads $\bot$ in $\text{post[i]}$ (line 8) and $v_i$ in $\text{pre[i]}$ (line 7). In case $w$ precedes $w_i^S$, Lemma 6.16 implies that $U$ is linearized before $w_i^S$, which is a contradiction. Thus, assume that $w$ follows $w_i^S$. By the definition of $w_i^S$, $r_i^S$ reads the value that $w_i^S$ writes to register $\text{pre[i]}$. Therefore, $w$ must follow $r_i^S$. By Lemma 6.17, $r_i^S$ follows $C_i^S$. So, $w$ follows $C_i^S$. Since $U$ is linearized before $S$, and $S$ is linearized at $C_i^S$, $U$ cannot be linearized at $w$. Therefore, there is a SCAN $S'$ such that $w$ is performed between $C_i^S$ and $w_i^S$, and $U$ is linearized just before $C_i^S$. Since $w$ follows $w_i^S$, $g_i^S \neq g_i^S$. Lemma 6.15 implies that $\alpha(C_i^S, C_i^S)$ and $\alpha(C_i^S, C_i^S)$ do not intersect. If $C_i^S$ precedes $C_i^S$, the linearization point of $U$ which is placed at $C_i^S$ follows
the linearization point of \( S \) which is placed at \( C^S_1 \), which is a contradiction. Assume finally that \( C^S_1 \) follows \( C^S_0 \); Lemma 6.1 implies that \( w^S_1 \) precedes \( C^S_0 \). Thus, \( w \) which is performed between \( C^S_1 \) and \( w^S_2 \), precedes \( C^S_1 \), which is a contradiction.

2. Assume now that \( g^S \) reads \( u_i \) in \( \text{post}[i] \) (line 8). Then, \( V^S_i \) is well-defined and let \( r_{\text{pre}} \) be the read of \( \text{pre}[i] \) by \( V^S_i \) (line 7). Recall that \( S \) returns the value that \( U^S_i \) uses as a parameter and therefore \( S \) returns the value that has been written to \( \text{pre}[i] \) by \( U^S_i \). In case \( w \) precedes \( w^S_1 \), Lemma 6.16 implies that \( U \) is linearized before \( U^S \), which is a contradiction. Thus, assume that \( w \) follows \( w^S_1 \). Then, \( w \) must follow \( r_{\text{pre}} \) since (by the definition of \( U^S \)) \( V^S_i \) reads the value that \( w^S \) writes. By Lemma 6.18, \( r_{\text{pre}} \) follows \( C^S \). Therefore, \( w \) follows \( C^S \). Since \( U \) is linearized before \( S \), and \( S \) is linearized at \( C^S_1 \), \( U \) cannot be linearized at \( w \). Thus, there is a scan \( S' \) such that \( w^S \) follows \( C^S_1 \), \( w \) is performed between \( C^S_1 \) and \( w^S_2 \), and \( U \) is linearized just before \( C^S_1 \). Since \( w \) is performed after \( w^S_1 \), \( g^S \neq g^S \).

Lemma 6.15 implies that \( \alpha(C^S_1, C^S_0) \) and \( \alpha(C^S_1, C^S_0) \) do not intersect. If \( C^S_1 \) precedes \( C^S_0 \), the linearization point of \( U \) which is placed at \( C^S_0 \) follows the linearization point of \( S \) which is placed at \( C^S_1 \), which is a contradiction. Assume finally that \( C^S_1 \) follows \( C^S_0 \). Lemma 6.1 implies that \( w^S_1 \) precedes \( C^S_0 \). Thus, \( w \) which is performed between \( C^S_0 \) and \( w^S_2 \), precedes \( C^S_1 \), which is a contradiction.

In all cases we derived a contradiction. We conclude that no update on component \( A_i \) is linearized between \( U^S \) and \( S \). Thus, \( S \) returns a consistent vector.

Theorem 6.20 C-Snap is an anonymous, linearizable, wait-free, multi-scanner, multi-writer snapshot implementation from \( m \) read-write registers and \( m + 1 \) CAS registers that achieves time complexity \( O(n) \) for scan and \( O(1) \) for update.

References


