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

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

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

February 19, 2007

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 tecnique 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

*__Eligible__ for the best student paper award since N. Kallimanis is a full time MsC student. **Number of pages**: 12 (plus an appendix of 17 pages with formal proofs).

†**Contact Author**. Address: Department of Computer Science, University of Ioannina, P.O. Box 1186, Ioannina, GR 45100, GREECE. Tel: +30 26510 98808, Fax: +30 26510 98890, E-mail: nkallima@cs.uoi.gr

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 in 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 prsented a lower bound of $\Omega(n)$ on the time complexity of `SCAN` for implementations of *pertubable* objects from read-write registers. They prove that single-writer snapshots are pertubale [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 $O(m * \kappa)$, where κ 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 Kirousis, 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 these 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 CAS(O, v_o, v_n) which reads the value of O and if it equals to v_o, it changes it to v_n. 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 `UPDATE` and $O(m)$ for `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 $O(m)$ for `UPDATE` and `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 `SCAN` and $O(1)$ for `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 compexity 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 `UPDATE` and $O(m)$ for `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 `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 `SCAN` and $O(1)$ for `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 `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 `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 write(R, v) which writes v into R, and read(R) which returns the value of R. A *CAS* register O supports the operations read and CAS; CAS(O, v_o, v_n) compares the current value of O with v_o 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 $\perp$). It supports the operations, SCAN and UPDATE, which can be executed concurrently. An UPDATE(i, v) updates the value of component A_i to v, while SCAN returns a consistent vector of m values, one for each component. *Multi-writer* snapshots allow to all processes to 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 SCAN and an algorithm to implement 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 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* α is a sequence of steps starting from an initial configuration. An *execution fragment* of α is a part of α consisting of any number of consecutive steps. Let s_1 and s_2 be two steps of α 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 SCAN (UPDATE) is the execution fragment that starts with the first and ends with the last step executed by the algorithm of the SCAN (UPDATE).

An implementation is *single-scanner* if in any execution of the implementation there is only one process that performs SCANS. The *time complexity* of a SCAN (UPDATE) of an implementation is the maximum number of steps performed by a process to perform a SCAN (UPDATE, respectively) in any execution of the implementation. The *time comlexity* of the implementation is the maximum between the time complexities of SCAN and UPDATE.

We assume that processes may fail by crashing. We study *wait-free* implementations where each process completes the execution of any SCAN or 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 α of the implementation, each SCAN or 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 SCANS and UPDATES performed in α. We say that a 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 UPDATE on A_i that appears last in the linearization order among the 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 *pre* of m registers, one for each component. Any UPDATE on A_i, $1 \leq i \leq m$, writes its value to $pre[i]$ (line 6). Before doing so, the UPDATE stores (line 5) the previous value of $pre[i]$ in some appropriate register of an array, called *post*, of registers to help the scanner discover a consistent vector. Array *post* is a 2-dimensional array with each row having m registers; the number of its rows depends on the maximum number of SCANS performed in any execution (and therefore it might be unbounded).

The scanner uses register *seq* to set up a new sequence number each time a SCAN starts executed (line 7). Each UPDATE U on any component A_i starts by reading *seq* (line 1). The sequence number found there is used to index (line 5) the row of post in the ith entry of which the previous value of $pre[i]$ is written before U overwrites it (line 6). To find a consistent vector, a SCAN S should not return the values written by UPDATES that start after S. To achieve this S reads all m registers of *pre* (line 9), and the m registers $post[seq]$ (line 10). Notice that only UPDATES that have started after the beginning of S may write to this row of *post*. UPDATES that write to smaller rows of *post*

have started their execution before S, so if S reads in pre 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 $post[seq][i] \neq \perp$ 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*.)

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

    void update(data value, int i){
       int curr_seq;
       data d1, d2;
1      curr_seq=seq;
2      d1=pre[i];
3      d2=post[curr_seq][i];
4      if(d2==⊥)
5        post[curr_seq][i]=d1;
6      pre[i]=value;
    }

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

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 ith entry of the block of shared memory pointed to by the pointer read in $sptr$. In order to compute the vector to return, S reads the m positions of the block pointed to by $sptr$ in addition to the m registers of pre. Due to lack of space, pseudo-code for the improved version of T-Opt is presented at the appendix.

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 α be an execution of T-Opt and let S be any SCAN performed in α. Let w_S be the write performed by S (line 7), and let seq_S be the value written to seq by w_S. For each $i \in \{1, \ldots, m\}$, denote by r_i^S the read of $pre[i]$ by S (line 9), and by $\tilde{r}_i^S$ the read of $post[seq_S][i]$ by S (line 10). Let v_i be the value that S returns for component A_i. In case S reads $\perp$ in $post[seq_S][i]$ and v_i in $pre[i]$, we denote by U_i^S the UPDATE that writes v_i to $pre[i]$ and its write to $pre[i]$ is the last write to $pre[i]$ that precedes r_i^S. If S reads v_i in $post[seq_S][i]$, we introduce the following notation. We denote by V_i^S the UPDATE that writes v_i to register $post[seq_S][i]$ and its write to $post[seq_S][i]$

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

We now assign linearization points to SCANS and UPDATES. Each SCAN S is linearized at w_S. For each $i \in \{1, \ldots, m\}$, 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 $pre[i]$ between w_S and w_i^S just before w_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 $pre[i]$.

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 $pre[i]$ of those UPDATES have been performed, and (2) by definition of U_i^S, no other UPDATE on A_i writes to $pre[i]$ 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 seq_S in seq and attempts to perform its write to $post[seq_S][i]$ before V_i^S reads this register.Thus, V_i^S finds a value other than $\perp$ in $post[seq_S][i]$ 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. Due to lack of space, the formal proof for the linearizability of T-Opt is presented in the appendix.

Theorem 3.1 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 *post*, it uses only $n + 2$ such rows. To achieve this, it employs another array, called *state*, of n registers, one for each process, which are written when UPDATES are performed. Pseudo-code for RT is presented in Algorithm 2. An UPDATE by some process p records in $state[p]$ the value it read in *seq* (line 2). A SCAN S reads all n registers of *state* and chooses as its sequence number seq_S some index not appearing in any of these registers (lines 10-12).

The main goal of the algorithm is to guarantee that only UPDATES that perform the biggest part of their execution after the write w_S to *seq* by S (line 14) write to registers of $post[seq_S]$. 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 UPDATE U, it uses $state[p]$ to inform the scanner of the value it read in *seq* (lines 1-2). Then, it reads *seq* again (line 3) and only if it sees the same value in *seq* twice (line 6), it attempts to write to *post* (line 7).

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

Algorithm 2 Pseudo-code for RT (process p, $1 \leq p \leq n$).

```
   shared int seq=1;
   shared int state[1..n]={1,..,1};
   shared data post[1..n+2][1..m]={⊥,...,⊥};
   shared data pre[1..m]={⊥,...,⊥ };

   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;
   }
```

```
   data *scan(void){
      data view[1..m], d1, d2;
      set act_set;
      int lseq, j;
9     act_set={seq};
10    for(j=1;j≤n;j++)
11       act_set=act_set∪{state[j]};
12    lseq=any int of {1,..,n+2}−act_set;
13    for(j=1;j≤m;j++) post[lseq][j]=⊥;
14    seq=lseq;
15    for(j=1;j≤m;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 UPDATE is $O(1)$, and the time complexity of SCAN is $O(n)$. RT uses only $(n+3)m+n+1$ registers; $(n+3)m$ out of these registers (namely, the registers of arrays *pre* and *post*) store just a single value, while the size of the rest $n+1$ registers (namely, *seq* and the registers of array *state*) is $O(\log n)$ bit (since each of them stores values from the set $\{1,\ldots,n+2\}$).

Linearizability. Let α be an execution of RT and let S be any SCAN performed in α. Let w_S be the write to *seq* performed by S (line 14), and let seq_S be the value written to *seq* by w_S. For each $i \in \{1,\ldots,m\}$, we introduce the notation r_i^S, $\tilde{r}_i^S$, v_i, U_i^S, V_i^S and w_i^S, and assign linearization points to SCANS and UPDATES in exactly the same way as we did for T-Opt. The proof of the linearizability of RT is in its biggest part similar to the proof of T-Opt. Due to lack of space, more details are provided in the appendix.

Theorem 4.1 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 3. 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 w_S 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 w_S.

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 incurrs 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*. So, $\lceil n/m \rceil$ consecutive SCANS are required to read all n registers of

Algorithm 3 Pseudo-code for RT-Opt (process p).

```
    constant PACE = m;
    constant PERIODS = ⌈n/PACE⌉;
    shared int seq = 1;
    shared int states[1..PERIODS*PACE]={1,..,1};
    shared data pre[1..m]={⊥,..,⊥},
           post[1..n+2*PERIODS+1][1..m]={⊥,..,⊥};

    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;
    }

    data *scan(void){
       data view[1..m], d1, d2;
       int lseq, j;
       static int cur_period=0;
       static set free=∅,
          candidates={2,...,n+2*PERIODS+1};
9      if(cur_period==0){
10        free=(free∪candidates);
11        candidates={1,...,n+2*PERIODS+1};
       }
12     lseq=any element of set free;
13     for(j=1;j≤m;j++) post[lseq][j]=⊥;
14     free=free-{lseq};
15     candidates=candidates-{lseq};
16     cur_period=(cur_period+1)%PERIODS;
17     seq=lseq;
18     for(j=1;j≤PACE;j++)
19        candidates=candidates-
                {state[cur_period*PACE+j]};
20     for(j=1;j≤m;j++){
21        d1=pre[j];
22        d2=post[seq][j];
23        if(d2==⊥) view[j]=d1;
24        else view[j]=d2;
       }
       return view;
    }
```

state; each execution α of RT-Opt can be partitioned into execution fragments, called *epochs*, each containing $\lceil n/m \rceil$ consecutive SCANS. Moreover, sequence numbers are now chosen from the set $\{1, \ldots, n + 2\lceil n/m \rceil + 1\}$ which is larger than the set $\{1, \ldots, n + 2\}$ used in RT.

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 $\lceil n/m \rceil$ 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 α of RT-Opt, *candidates* contain $n + 2*$PERIODS different sequence numbers, where PERIODS $= \lceil n/m \rceil$ (see initialization of PERIODS and *candidates*). During E_1, at most n+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 $n+2*$PERIODS+1 different sequence numbers. During E_j, at most n+PERIODS sequence numbers are removed from *candidates*. Thus, at least 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 $n + 2 * \lceil n/m \rceil + 1$ 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 $PACE = m$), m registers of *post*, and m registers of *pre*; the rest of the SCAN computation is on local variables. RT-Opt uses $O(mn)$ registers; most of them (the registers of arrays *pre* and *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 PACE$= n$, RT-Opt works in the same way as RT, while if PACE$\leq m$, RT-Opt achieves optimal time complexity.

Linearizability. Let α be an execution of RT-Opt and let S be any SCAN performed in α. Let w_S be the write to *seq* performed by S (line 17), and let seq_S be the value written to *seq* by w_S. For each $i \in \{1, \ldots, m\}$, we introduce the notation r_i^S (read by S, line 21), $\tilde{r}_i^S$ (read by S, line 22), v_i, U_i^S, V_i^S and w_i^S, 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 seq_S of *post* have executed their biggest part after the write to *seq* by S. Due to lack of space, the proof is provided in the appendix.

Theorem 5.1 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* SCAN *and $O(1)$ for* UPDATE.

6 C-Snap Algorithm

Pseudo-code for C-Snap is presented in Algorithm 4. C-Snap works in the same spirit as T-Opt. The arrays *pre* and *post* play a similar role as for T-Opt. However, *post* consists of only one row of m registers, each containing a component value and a sequence number. The use of sequence numbers and CAS to change *post* (line 4) guarantee that an 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 $post[i] = \perp$ (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 18 CAS 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 24 CAS 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 4 C-Snap Algorithm

```
   struct sq {
      int tm;
      boolean grab;
      data view[1..m];
   };
   struct post_data {
      int tm;
      data value;
   };
   shared sq seq=<1,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=<s.tm-1, ⊥>;
4     CAS(post[i],lpost,<lpost.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        d2=post[j].value;
9        if(d2==⊥) view[j]=d1;
10       else view[j]=d2;
       }
11     return view;
    }

   data [] scan(void){
12    grab_scan();
13    grab_scan();
14    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 s){
      int j;
      post_data p, lpost;
25    for(j=1;j≤m;j++){
26       p=post[j];
27       lpost=<s.tm-1, p.value>;
28       CAS(post[j], lpost, <s.tm, ⊥>);
29       p=post[j];
30       lpost=<s.tm-1, p.value>;
31       CAS(post[j], lpost, <s.tm, ⊥>)
      }
   }
```

Each execution α 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 (beacuse 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 $\perp$ 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 $\perp$ 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 $\perp$ to $post[i]$. Thus, if the grab_scan that succeeds to write the new vector in the current phase, reads a value other than $\perp$ 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_S 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 α be an execution of C-Snap and let S be any `SCAN` performed in α. 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 $\perp$ in $post[i]$ and some value v_i in $pre[i]$, let U_i^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 $\perp$ in $post[i]$, we denote by V_i^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_i^S must have read the value v_i in $pre[i]$. We denote by U_i^S the `UPDATE` on A_i that last writes to $pre[i]$ before V_i^S reads $pre[i]$. Let w_i^S be the write to $pre[i]$ by U_i^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_i^S has not already been linearized)[1], we place the linearization point of U_i^S just before C_1^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 $pre[i]$. Due to lack of space, the proof of the linearizability of C-Snap is provided in the appendix.

Theorem 6.1 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 $O(m)$ for* `SCAN` *and $O(1)$ for* `UPDATE`.

[1]It might happen that for two `SCANS` S and S', $g_0^S = g_c^{S'}$, so that both `SCANS` constraint the linearization points of the same `UPDATES`. This is why the parenthesis is needed.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. of the ACM*, 40(4):873–890, Sept. 1993.

[2] J. H. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, Apr. 1993.

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

[4] J. Aspnes. Time- and space-efficient randomized consensus. *J. of Algorithms*, 14(3):414–431, May 1993.

[5] J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Proc. of the 2nd ACM Symp. on Parallel Algorithms and Architectures*, pp. 340–349, 1990.

[6] H. Attiya, F. Ellen, and P. Fatourou. The Complexity of Updating Multi-writer Snapshot Objects. In *Proc. of 8th Int. Conf. on Distributed Computing and Networking*, pp. 319–330, Dec. 2006.

[7] H. Attiya and A. Fouren. Adaptive and Efficient Algorithms for lattice agreememt and renaming. *SICOMP*, 31(2):642–664, Oct. 2001.

[8] H. Attiya, N. Lynch, and N. Shavit. Are wait-free algorithms fast? *J. of the ACM*, 41(4):725–763, 1994.

[9] H. Attiya and O. Rachman. Atomic snapshots in $O(n \log n)$ operations. *SICOMP*, 27(2):319–340, 1998.

[10] P. Fatourou, F. Fich, and E. Ruppert. Space-optimal multi-writer snapshot objects are slow. In *Proc. of the 21th ACM Symp. on Principles of Distributed Computing*, pp. 13–20, 2002.

[11] P. Fatourou, F. Fich, and E. Ruppert. A tight time lower bound for space-optimal implementations of multi-writer snapshots. In *Proc. of the 35th ACM Symp. on Theory of Computing*, pp. 259–268, 2003.

[12] P. Fatourou, F. Fich, and E. Ruppert. Time-space tradeoffs for implementations of snapshots. In *Proc. of the 38th ACM Symp. on Theory of Computing*, 2006.

[13] P. Fatourou, and N. D. Kallimanis. Single-Scanner Snapshot Implementations are Fast! In *Proc. of the 25th ACM Symp. on Principles of Distributed Computing*, pp. 228-237, July 2006.

[14] R. Gawlick, N. Lynch, and N. Shavit. Concurrent timestamping made simple. In *Proc. of the Israel Symp. on the Theory of Computing and Systems*, *LLNCS* # 601, pp. 171–183, 1992.

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

[16] M. Inoue, W. Chen, T. Masuzawa, and N. Tokura. Linear time snapshots using multi-writer multi-reader registers. In *8th Int. Workshop on Distributed Algorithms*, *LLNCS* # 857, pp. 130–140, 1994.

[17] A. Israeli, A. Shaham, and A. Shirazi. Linear-time snapshot implementations in unbalanced systems. *Mathematical Systems Theory*, 28(5):469–486, Sept./Oct. 1995.

[18] P. Jayanti. f-Arrays: Implementation and Applications. In *Proc. of the 21th ACM Symp. on Principles of Distributed Computing*, pp. 270–279, 2002.

[19] P. Jayanti. An Optimal Multi-writer Snapshot Algorithm. In *Proc. of the 37th ACM Symp. on Theory of Computing*, pp. 723–732, 2005.

[20] P. Jayanti, and S. Petrovic. Efficient wait-free implementation of multiword ll/sc variables. In *Proc. of the 25th Int. Conf. on Distributed Computing Systems*, pp. ??–??, 2005.

[21] P. Jayanti, K. Tan, and S. Toueg. Time and Space Lower Bounds for non-blocking implementations. *SICOMP*, 30(2):438–456, June 2000.

[22] L. M. Kirousis, P. Spirakis, and P. Tsigas. Reading many variables in one atomic operation: solutions with linear or sublinear complexity. *IEEE Trans. Parallel and Distributed Systems*, 5(7):688–696, 1994.

[23] S. Mullender. *Distributed Systems*. Addison-Wesley, 1994.

[24] Y. Riany, N. Shavit, and D. Touitou. Towards a practical snapshot algorithm. *Theoretical Computer Science*, 269(1–2):163–201, 2001.

Appendix

A Linearizability of T-Opt

Algorithm 5 Pseudo-code for improved version of T-Opt.

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

    void update(data value, int i){
       data lptr[];
       data d1, d2;
1      lptr=sptr;
2      d1=pre[i];
3      d2=lptr[i];
4      if(d2==⊥)
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=sptr[j];
11       if(d2==⊥) view[j]=d1;
12       else view[j]=d2;
       }
       return view;
    }
```

Pseudocode for the improved version of T-Opt is presented in Algorithm 5. Let α be any execution of T-Opt and let S be any SCAN executed in α. Assume that S returns v_i for component A_i. We first prove two simple technical lemmas.

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

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

Assume finally that S reads v_i in $post[seq_S][i]$. 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 $pre[i]$ the value written there by w_i^S, so the read of $pre[i]$ by V_i^S occurs after w_i^S, and (2) $\tilde{r}_i^S$ reads in $post[seq_S][i]$ the value written there by V_i^S, so $\tilde{r}_i^S$ follows the write to $post[seq_S][i]$ by V_i^S. By the code, the write to $post[seq_S][i]$ by V_i^S follows its read of $pre[i]$. It follows that $\tilde{r}_i^S$ follows w_i^S. ■

Lemma A.2 *Assume that S reads v_i in register $post[seq_S][i]$, and let r_{pre} be the read of $pre[i]$ by V_i^S. Then, r_{pre} is executed after w_S.*

Proof: Assume, by the way of contradiction, that r_{pre} is executed before w_S. Then, the read of seq by V_i^S (let it be r_{seq}) precedes w_S. By the code, 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_{seq} which is executed before w_S, reads a value $t < seq_S$. By the code, it follows that V_i^S writes v_i to register $post[t][i]$. This a contradiction, since (by definition) V_i^S writes v_i to register $post[seq_S][i] \neq post[t][i]$. ■

We are now ready to prove that if w_i^S follows w_S, then the execution of an UPDATE that performs its write to $pre[i]$ between w_S and w_i^S (including U_i^S) starts before w_S.

Lemma A.3 *For each $i \in \{1,\ldots,m\}$ such that w_i^S follows w_S, it holds that any* UPDATE *on A_i that performs its write to $pre[i]$ between w_S and w_i^S (including U_i^S) 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 $pre[i]$ (let it be w) before w_i^S. By Lemma A.1, w_i^S precedes $\tilde{r}_i^S$, and therefore U ends its execution before the end of S. Since U starts its execution after w_S, U reads seq_S in seq. By the code, U first reads register $pre[i]$ and then register $post[seq_S][i]$. Moreover, in case U reads $\perp$ in $post[seq_s][i]$, it writes in $post[seq_S][i]$ the value read in $pre[i]$.

By Lemma A.1, w_i^S precedes $\tilde{r}_i^S$. By the code, the execution of code lines 4-5 by U precedes w (which precedes w_i^S). Thus, the execution of lines 4-5 precedes $\tilde{r}_i^S$. Thus, $\tilde{r}_i^S$ reads a value other than $\perp$ in $post[seq_S][i]$, so V_i^S is well-defined. By definition, the following are true for V_i^S and w_i^S: (1) V_i^S writes to register $post[seq_S][i]$, so it reads $\perp$ in $post[seq_S][i]$ (line 9), and seq_S in register seq (line 7), and (2) The read in $pre[i]$ by V_i^S follows w_i^S since V_i^S reads in $pre[i]$ the value written by w_i^S.

It follows that the read of $post[seq_S][i]$ by V_i^S, which by the code follows its read to $pre[i]$, comes after the execution of lines 4-5 and the possible write to $post[seq_S][i]$ by U. Thus, V_i^S reads a value other than $\perp$ in $post[seq_S][i]$. A contradiction. ■

Using Lemma A.3, it can be easily proved that the linearization point of each `UPDATE` is within its execution interval.

Lemma A.4 *Let α be any execution of* T-Opt. *The linearization point of any* `SCAN` *or* `UPDATE` *executed in α 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 pre.

Let U be an `UPDATE` on A_i 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_i^S of U_i^S is executed after w_S, the write to $pre[i]$ by U is executed between w_S and w_i^S, and U is linearized just before w_S. Obviously, the execution of U ends after w_S. Lemma A.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 $pre[i]$.

Lemma A.5 *Let U_1, U_2 be two* `UPDATES` *on some component A_i, $1 \leq i \leq m$. Denote by w_1 the write to $pre[i]$ by U_1 and by w_2 the write to $pre[i]$ 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 $pre[i]$, then the claim holds trivially. Therefore, assume that at least one of the U_1, U_2 is not linearized at its write to $pre[i]$. We consider the following cases.

1. U_2 is linearized at w_2. Lemma A.4 implies that U_1 is linearized within its execution interval, so U_1 is linearized the latest 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 `SCAN` S such that w_2 has been performed between w_S and w_i^S. Since w_1 precedes w_2, w_1 has been executed before w_i^S. In case w_1 follows w_S, both U_1 and U_2 are linearized just before w_S in the order that they perform their writes to $pre[i]$. Thus, U_1 is linearized before U_2, which is a contradiction.

So, assume w_1 precedes w_S. Lemma A.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 $pre[i]$. By the way linearization points are assigned, there are two `SCAN` operations S_1 and S_2 such that w_1 has been performed between w_{S_1} and $w_i^{S_1}$, and w_2 has been performed between w_{S_2} and $w_i^{S_2}$; moreover, U_1 is linearized just before w_{S_1}, and U_2 is linearized just before w_{S_2}. Lemma A.1 implies that $w_i^{S_1}$ precedes the end of S_1, and $w_i^{S_2}$ 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 $pre[i]$. 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 $w_i^{S_1}$ is executed after w_{S_1} and before the end of S_1; similarly, $w_i^{S_2}$ is executed after w_{S_2} and before the end of S_2. Thus, w_1 which is executed between w_{S_1} and $w_i^{S_1}$ follows w_2 which is executed between w_{S_2} and $w_i^{S_2}$, 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 A.5 to prove consistency.

Lemma A.6 *Let α be an execution of* T-Opt. *Any* `SCAN` *executed in α returns a consistent vector.*

Proof: Assume that S returns the vector $\mathbf{v} = \langle v_1, ..., v_m \rangle$. Recall that, for each $i \in \{1, \ldots, m\}$, U_i^S writes v_i to $pre[i]$, and therefore it uses v_i as a parameter. In case w_i^S precedes w_S, Lemma A.4 implies that U_i^S is linearized before S. In case w_i^S follows w_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 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` and let w be the write to $pre[i]$ by U. We consider the following cases.

1. Assume that S reads $\perp$ in register $post[seq_S][i]$ and v_i in register $pre[i]$. In case w precedes w_i^S, Lemma A.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 $pre[i]$. 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 A.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 read v_i in register $post[seq_S][i]$. Then, V_i^S is well-defined and let r_{pre} be the read of $pre[i]$ 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 $pre[i]$ by U_i^S. In case w precedes w_i^S, Lemma A.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_{pre} since (by the definition of U_i^S) V_i^S reads the value that w_i^S writes. By Lemma A.2, r_{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 A.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 conculde that no `UPDATE` on component A_i is linearized between U_i^S and S. Thus, S returns a consistent vector. ■

B Linearizability of RT

The arguments for overcoming the difficulties encountered by RT discussed in Section 4 are formalized in the following lemma (which corresponds to Lemma A.2 of Section 3).

Lemma B.1 *Assume that S reads v_i in register $post[seq_S][i]$, and let r_{pre} be the read of $pre[i]$ by V_i^S. 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_i^S (line 1), and by r'_{seq} its second read of seq (line 3). Let p be the process that executes V_i^S, let w_p be the write to $state[p]$ by V_i^S (line 2), and let r_p be the read of $state[p]$ 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 $state[p]$ by w_p (line 2) is read by r_p. By the code (line 11), it follows that $seq_S \neq t$. Since V_i^S read t in seq, it follows by the code that V_i^S cannot write to any other register than $post[t][i] \neq post[seq_S][i]$, 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 α (or a ficticious `SCAN` that writes to seq the initial value if no such `SCAN` exists). By the code (line 9), it follows that $seq_{S'} \neq seq_S$. Since r'_{seq} follows r_p and precedes w_S, r'_{seq} reads $seq_{S'}$ in seq. Thus, it follows by the code that V_i^S does not write in $post[seq_S][i]$, which is a contradiction. ■

The proof of the next lemma follows similar arguments as the proof of Lemma A.3 but we include it below because it slightly differentiates at some places. In the proof below we apply Lemma A.1 (presented in Section 3) which holds also for RT (its proof is exactly the same as for T-Opt).

Lemma B.2 *For each $i \in \{1, \ldots, m\}$ such that w_i^S follows w_S, it holds that any `UPDATE` on A_i that performs its write to $pre[i]$ between w_S and w_i^S (including U_i^S) 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 $pre[i]$ (let it be w) before w_i^S. By Lemma A.1, w_i^S precedes $\tilde{r}_i^S$, and therefore U ends its execution before the end of S. Since U starts its execution after w_S, U reads seq_S 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 $\perp$ in $post[seq_s][i]$, it writes to $post[seq_S][i]$ the value read in $pre[i]$.

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

It follows that the read of $post[seq_S][i]$ by V_i^S, which by the code follows its read to $pre[i]$, comes after the execution of lines 6-7 and the possible write to $post[seq_S][i]$ by U. Thus, V_i^S reads a value other than $\perp$ in $post[seq_S][i]$. 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.

C Linearizability of RT-Opt

Let α be an execution of RT-Opt and let S be any `SCAN` performed in α. We split α into epochs so that each epoch contains exactly $\lceil n/m \rceil$ `SCANS`. Denote by E_i the i-th epoch of α, $i \geq 1$. Epoch E_1 starts with the first instruction of the execution and ends with the last instruction of the $\lceil n/m \rceil$-th `SCAN` (or equals to α if not that many `SCANS` occur in α). For each $i > 1$, epoch E_i starts at the point that the execution of the $((i-1)\lceil n/m \rceil)$-th `SCAN` ends and finishes with the last instruction executed by the $(i\lceil n/m \rceil)$-th `SCAN` (or equals to the suffix of α which starts at the point that the execution of the $((i-1)\lceil n/m \rceil)$-th `SCAN` ends, if less than $(i\lceil n/m \rceil)$ `SCANS` occur in α). Assume that α contains $(c_1\lceil n/m \rceil + c_2)$ `SCANS`, where $c_1 \geq 0$ and $0 \leq c_2 < \lceil n/m \rceil$ are constants. We remark that if c_2 equals zero then α contains c_1 epochs. However, if $c_2 > 0$ then α consists of $c_1 + 1$ epochs where the (c_1+1)-th epoch contains $c_2 < \lceil n/m \rceil$ `SCANS`. Notice also that if α is an infinite execution, the execution interval of epoch $c_1 + 1$ is infinite.

Let k be the number of epochs in α. 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 $free_i$ the set $free$ at the end of E_i. Denote by $candidates_i$ the set $candidates$ at the end of E_i.

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_S in register seq. As for RT, the main difficulty is to prove that the `UPDATES` that write values to the seq_S 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 C.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 $\lceil n/m \rceil$ `SCANS` are performed during it. Each if these `SCANS` reads m distinct registers of $state$ (lines 16,18,19). Thus, all $\lceil n/m \rceil * m$ registers of $state$ are read once in E_j. ■

Lemma C.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, $(\lceil n/m \rceil + 1)$-th, ..., $((k-1)\lceil n/m \rceil + 1)$-th `SCAN` of α, as stated by the following lemma.

Lemma C.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 C.4 *For each $j \in \{1, \ldots, k\}$, each `SCAN` of epoch E_j writes a distinct value to seq.*

Proof: Lemma C.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 C.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 C.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 C.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 $free_j^s$ the set $free$ after line 10 has been executed by S. Clearly, $free_j^s = free_{j-1} \cup candidates_{j-1}$. Since $free_{j-1} \cap SN_{j-1} = \emptyset$, and $candidates_{j-1} \cap SN_{j-1} = \emptyset$, it follows that $free_j^s \cap SN_{j-1} = \emptyset$. By the code, all elements of SN_j are chosen by $free_j^s$. Thus, $SN_j \cap SN_{j-1} = \emptyset$. ■

Lemma C.6 *Assume that S reads v_i in register $post[seq_S][i]$, and let r_{pre} be the read of $pre[i]$ by V_i^S. 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_i^S (line 1), and by r'_{seq} the second read of *seq* by V_i^S (line 3). Let p be the process that executes V_i^S and let w_p be the write to $state[p]$ by V_i^S (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, $j \leq 1$.

1. Assume that $j = 1$. By the code (line 10) and by Lemma C.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 C.4 implies that no `SCAN` that precedes S chooses the same sequence number as S. By definition, V_i^S writes in row seq_S of *post*. By the code (line 6), this write is performed only if both r_{seq} and r'_{seq} read seq_S in *seq*. It follows that r_{seq} and r'_{seq} are both performed after w_S. Since r_{pre} follows r'_{seq}, r_{pre} follows w_S, which is a contradiction.

2. Assume that $j > 1$. By Lemma C.1, there is a unique SCANS 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 $post[seq_S]$ to $\perp$. By definition of V_i^S, S reads the value that V_i^S writes to $post[seq_S][i]$. Thus, V_i^S writes to $post[seq_S][i]$ after the initialization of $post[seq_S][i]$ to $\perp$ by S. Since $state[p]$ is written only by p, $state[p]$ contains the value written by V_i^S from w_p until at least the initialization of $post[seq_S][i]$ 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_i^S writes to $post[seq_S][i]$, so (by the code) it must be that r'_{seq} reads seq_S in seq. By Lemma C.5, $SN_{j-1} \cap SN_j = \emptyset$. Thus, since $seq_S \in SN_j$, $seq_S \notin SN_{j-1}$ (that is, no SCAN of epoch E_{j-1} writes seq_S to seq). By Lemma C.4, each SCAN of epoch E_j writes a distinct value to seq. So, no SCAN of epoch E_j that precedes S writes seq_S 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 seq_S 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 seq_S in seq. Assume that the value seq_S read by r'_{seq} has been written to seq by some SCAN S_l 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 $seq_S = 1$; moreover, r'_{seq} is executed before the write to seq by the first SCAN of the execution (which is also the first SCAN of epoch E_1). In this case, let $l = 0$, $free_0 = \emptyset$, and $candidates_0 = \{2, \ldots, n + 2{*}\text{PERIODS}{+}1\}$ (that is, $free_0$ and $candidates_0$ denote sets $free$ and $candidates$, respectively, in initial state). Let $l > 0$. Since $seq_S \in SN_j$, Lemma C.5 implies that $seq_S \notin SN_{j-1}$. Thus, $1 \leq l < j - 1$. Let w_{S_l} be the write to seq by S_l. Since r'_{seq} reads the value written by w_{S_l}, r'_{seq} is performed between w_{S_l} and the write to seq by the next to S_l 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_l 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 C.7 *For each $f \in \{l, \ldots, j-1\}$, $seq_S \notin candidates_f$.*

   Proof: Assume first that $f = l$. In case $l = 0$, recall that $seq_S = 1$ (the initial value of seq), and $candidates_0 = \{2, \ldots, n + 2{*}\text{PERIODS}{+}1\}$. So, $seq_S \notin candidates_0$. Assume now that $l > 0$. Since S_l is executed in epoch E_l and chooses seq_S as its sequence number, $seq_S \in SN_l$. By Lemma C.5, $candidates_l \cap SN_l = \emptyset$. Thus, $seq_S \notin candidates_l$.

   Assume now that $f > l$. Recall that $state[p]$ does not change from w_p until at least the beginning of S. Recall also that r'_{seq} (and therefore also w_p which precedes r'_{seq}) is executed before the write to seq by the first SCAN of epoch E_{l+1}. By the code (lines 17-19), each SCAN first writes to seq and then reads some of the registers of array $state$. By Lemma C.1, $state[p]$ is read by a unique SCAN of E_f. So, seq_S is read in $state[p]$ during E_f. Thus, it follows by the code (line 19) that seq_S is removed from $candidates$ during E_f. By Lemma C.3, no elements are added in $candidates$ after the execution of line 11 by the first SCAN of epoch E_f. Since line 19 follows line 11, we conclude that $seq_S \notin candidates_f$. ■

   Claim C.8 *For each $f \in \{l, \ldots, j-1\}$, $seq_S \notin free_f$.*

Proof: By induction on f. We first prove the claim for $f = l$. In case $l = 0$, $free_0 = \emptyset$, so $seq_S \notin free_0$. Assume that $l > 0$. Since S_l chooses seq_S as its sequence number, $seq_S \in SN_l$. Lemma C.2 implies that $free_l \cap SN_l = \emptyset$. Thus, $seq_S \notin free_l$.

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, $seq_S \notin free_{f-1}$. By claim C.7 it follows that $seq_S \notin candidates_{f-1}$. Let $free^s_f$ denote set $free$ after the execution of line 10 by the first SCAN of epoch E_f. By the code (line 10), $free^s_f = free_{f-1} \cup candidates_{f-1}$. It follows that $seq_S \notin free^s_f$. No other element is added to $free$ until the end of E_f. Thus, $seq_S \notin free_f$, as needed. ■

For $f = j - 1$, Claim C.7 implies that $seq_S \notin candidates_{j-1}$, and Claim C.8 implies that $seq_S \notin free_{j-1}$. By Lemma C.3, only the first SCAN of epoch E_j adds elements to $free$ by executing line 10. Let $free^s_j$ denote set $free$ after the execution of this line. By the code, $free^s_j = free_{j-1} \cup candidates_{j-1}$. It follows that $seq_S \notin free^s_j$. All SCANS of epoch E_j (including S) choose their sequence numbers from $free^s_j$. However, S choose seq_S as its sequence number which does not exist in $free^s_j$. This is a contradiction.

■

Lemma B.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.

D Linearizability of C-Snap

Let α be an execution of C-Snap and let S be any SCAN performed in α. 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^S_0 is the last successful CAS instruction of type 0 which precedes the end of S (line 14), and C^S_1 is the last successful CAS instruction of type 1 which precedes C^S_0. Denote by r^S_i the read of $pre[i]$ by g^S_0, and by $\tilde{r}^S_i$ the read of $post[i]$ by g^S_0. If g^S_0 reads a value v_i other than $\perp$ in $post[i]$, let r_{pre} be the read of $pre[i]$ by V^S_i (line 2). Denote by α^C the subsequence of a which contains all the successful CAS instructions on *seq* (type 0 and type 1) in the order they appear in α; let $|\alpha^C|$ denote the number of CAS instructions in α^C. Denote by α^C_1 (α^C_0), the subsequence of α^C that contains all the successful CAS of type 1 (type 0, respectively) in the order they are performed in α; let $|\alpha^C_1|$ ($|\alpha^C_0|$) denote the number of CAS in a^C_1 (α^C_0, respectively).

The following lemma is a direct consequence of the definition of U^S_i. It implies that w^S_i precedes C^S_0 and therefore also the end of g^S_0.

Lemma D.1 *For each $i \in \{1, \ldots, m\}$, (1) $\tilde{r}^S_i$ follows w^S_i, and (2) C^S_0 follows w^S_i.*

Proof: Assume first that S reads $\perp$ in $post[i]$. Then, by the definition of U^S_i, w^S_i precedes r^S_i (since r^S_i reads the value written to $pre[i]$ by w^S_i). Since r^S_i precedes $\tilde{r}^S_i$ and $\tilde{r}^S_i$ precedes C^S_0, the claim follows.

Assume now that S reads a value v_i other than $\perp$ in $post[i]$. Then, V^S_i is well-defined. By the definitions of V^S_i and U^S_i, the following hold: (1) V^S_i reads in $pre[i]$ the value written there by w^S_i, so r_{pre} occurs after w^S_i, and (2) $\tilde{r}^S_i$ reads in $post[i]$ the value written there by V^S_i, so $\tilde{r}^S_i$ follows the CAS on $post[i]$ by V^S_i. By the code, the CAS on $post[i]$ by V^S_i follows r_{pre}. So, $\tilde{r}^S_i$ follows w^S_i, as

needed by (1). Since g_0^S first executes $\tilde{r}_i^S$ and then C_0^S, it follows that C_0^S follows w_i^S, as needed by (2). ■

Consider any execution α 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 α^C CAS instructions of type 0 alternate with CAS instructions of type 1 starting with a CAS of type 0.

Lemma D.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 *seq* in α. Assume, by the way of contradiction, that C_0^s is of type 1. By the code (line 21), it follows that $curr\_seq.grab = false$ just before C_0^s is executed. Since C_0^s is successful, it must be that $seq.grab = false$ just before C_0^s. However the initial value of *seq.grab* is *true* and register *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 *grab_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 *seq* (line 15) the value true for *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, $curr\_seq.grab = true$ when C_i^s is executed. Since C_{i-1}^s 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_{i-1}^s and C_i^s, *seq.grab* is $false$ when C_i^s is executed. Thus, C_i^s cannot be successful, which is a contradiction.

Let now C_{i-1}^s be of type 1. Assume, by the way of contradiction, that C_i^s is also of type 1. Then, by the code (line 21), it holds that $curr\_seq.grab = false$ when C_i^s is executed. Since C_{i-1}^s 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_{i-1}^s and C_i^s, *seq.grab* is *true* when C_i^s is executed. Thus, C_i^s 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 D.3 *The following hold:*

1. *C_0^s 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_i^s stores in seq.tm the same value as C_{i-1}^s;*
   (b) *If $i \mod 2 = 1$, C_i^s stores in seq.view the same vector as C_{i-1}^s.*

Proof: We first prove 1. Recall that C_0^s is the first successful CAS on *seq* in a, so *seq* has its initial value before C_0^s. Thus, if p is the process that executes C_0^s, p reads in *seq* its initial value (line 15). Lemma D.2 implies that C_0^s is of type 0. By the code (line 18), C_0^s writes in *seq.tm* the value of *curr_seq.tm*. So, C_0^s writes in *seq.tm* its initial value.

We continue to prove claim 2a. Consider any i, $1 \leq i \leq k$ such that $i \mod 2 == 0$. Lemma D.2 implies that C_i^s is of type 0. Suppose that C_{i-1}^s writes t to *seq.tm* Since there are no successful CAS

on *seq* between C^s_{i-1} and C^s_i, $seq.tm = t$ when C^s_i is executed. Since C^s_i is successful, it follows by the code (line 18) that $curr\_seq.tm = t$.

We finally prove claim 2b. Consider any i, $1 \leq i \leq k$ such that $i \mod 2 == 1$. Lemma D.2 implies that C^s_i is of type 1. Suppose that C^s_{i-1} writes the vector *view* to *seq.view*. Since there are no successful CAS on *seq* between C^s_{i-1} and C^s_i, $seq.view = view$ when C^s_i is executed. Since C^s_i is successful, it follows by the code (line 24) that $curr\_seq.view = seq.view$ when C^s_i is executed. By the code (line 23), $curr\_seq.view = lview$, so C^s_i does not change the value of *seq.view*. ■

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

Lemma D.4 *Let C_1 and C'_1 be two successful CAS instructions of type 1 in α 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 D.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 $|\alpha^C_1| = k_1$, and let $\alpha^C_1 = C^1_1 C^2_1 \ldots C^{k_1}_1$. Let C^0_1 denote the initial configuration of the system. Notice that the initial value of $seq.tm = 1$. By Lemma D.3, this value does not change until C^1_1. By Lemma D.3, *seq.tm* changes only by successful CAS of type 1. These and Lemma D.4 imply:

Corollary D.5 *For each j, $1 \leq j \leq k_1$, it holds that $seq.tm = j$ at all points in time between C^{j-1}_1 and C^j_1.*

Let S be any scan operation executed by some process p in α, 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 D.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 D.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 exeuted 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 D.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 D.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 D.7 *Let S by any* `SCAN` *executed in α. Then, S is linearized within its execution interval.*

Proof: By the code (lines 12-13), S executes twice function grab_scan. Let g and g' be the execution of the first and second grab_scan, respectively, by S. Denote by r_{seq} and r'_{seq} the reads of 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 D.6 implies that the execution interval which starts with r_{seq} and ends with C_1, contains a successful CAS C_{suc} of type 1. Similarly, the execution interval that starts with r'_{seq} and ends with C'_1 contains a successful CAS C'_{suc} of type 1. Lemma D.2 implies that there is a successful CAS of type 0 between C_{suc} and C'_{suc}. So, by the way linearization points are assigned, S is linearized at C_{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 $post[i]$, $1 \leq i \leq m$, writes $\perp$ to the $value$ field of $post[i]$ during each phase; moreover, after the execution of this CAS, the $value$ field of $post[i]$ remains $\perp$ up to the beginning of the next phase. Some useful properties of sequence numbers are also proved.

Lemma D.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 C_{post} on $post[i]$ by a clear_registers is successful,*
2. *$post[i].tm = j - 1$ between C_1^{j-1} and C_{post}, and*
3. *$post[i].tm = j$ and $post[i].value = \perp$ between C_{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 D.3 implies that only CAS of type 1 change *seq.tm*. In case $j = 1$, since C_1^1 is the first CAS of type 1 executed in α, it follows that *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 *seq.tm* (lines 1, 15). Recall also that, for each i, $1 \leq i \leq m$, initially, $post[i].tm = 0$ and $post[i].value = \perp$. In case $j > 1$, Corollary D.5 implies that immediately after C_1^{j-1}, it holds that $seq.tm = j$. Since no CAS of type 1 is successful between C_1^{j-1} and C_1^j, $seq.tm = j$ at all points in time between C_1^{j-1} and C_1^j. Moreover, Corollary D.5 implies that all operations whose executions have started before C_1^j read a value less or equal to j in *seq.tm* (lines 1 and 15). Moreover, by induction hypothesis, immediately after C_1^{j-1} it holds that $post[i].tm = j$ and $post[i].value = \perp$. Thus, in either case we have that *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 *seq.tm*. Additionally, immediately after C_1^{j-1}, it holds that $post[i].tm = j-1$ and $post[i].value = \perp$.

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

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

Claim D.9 *Let C be the first successful CAS on $post[i]$ that is executed between C_1^{j-1} and C_1^j by some clear_registers. Then,*

1. *all successful CAS on $post[i]$ executed between C_1^{j-1} and C does not change the value of $post[i].tm$, and*
2. *no CAS on $post[i]$ succeeds between C and C_1^j.*

Proof: We start by proving 1. Since C is the first successful CAS on $post[i]$ executed between C_1^{j-1} and C_1^j by clear_registers, any successful CAS on $post[i]$ between C_1^{j-1} and C is by an `UPDATE`. Recall that $post[i].tm = j-1$ immediately after C_1^{j-1}. Let U be the first `UPDATE` that executes a successful CAS C_U on $post[i]$ between C_1^{j-1} and C_1^j. Then, $post[i].tm = j-1$ when C_U is executed. By the code (line 4), C_U does not change the value of $post[i].tm$. Thus, *seq.tm* equals $j-1$ after C_U. Applying the above argument inductively, it follows that no CAS on $post[i]$ executed between C_1^{j-1} and C changes the value of $post[i].tm$, as needed.

We continue to prove 2. Claim 1 (proved above) implies that $post[i].tm = j-1$ when C is executed. Assume that C is executed by some process q. Since C is successful, it follows by the code (lines 27-28 and 30-31) that q must have read j in *seq* (line 15), and that it writes j in $post[i].tm$. Assume, by the way of contradiction, that there is at least one successful CAS on $post[i]$ between C and C_1^j. If C' is the first such CAS, then when C' is executed it holds that $post[i].tm = j$. Since C' precedes C_1^j, the operation that executes C' has started its execution before C_1^j. Recall that all operations that start before C_1^j read a value less than or equal to j in *seq.tm*. Thus, by the code (lines 3, 27 and 30), $lpost.tm \leq j-1$ when C' is executed. Therefore, C' cannot be a successful CAS. This is a contradiction. Applying the above argument inductively, it follows that no CAS on $post[i]$ executed between C and C_1^j is successful, as needed. △

Claim D.10 *For each i, $1 \leq i \leq m$, there is some successful CAS on $post[i]$ executed by a clear_registers between C_1^{j-1} and C_1^j.*

Proof: Fix any i, $1 \leq i \leq m$. By the code (line 19), p calls clear_registers before it executes C_1^j. Let C_p^1 and C_p^2 be the two CAS on $post[i]$ executed by p (lines 28 and 31). Assume, by the way of

contradiction, that no CAS on $post[i]$ by clear_registers succeeds between C_1^{j-1} and C_1^j. Claim D.9 implies that $post[i].tm = j-1$ when C_p^1 is executed. Recall that p reads j in $seq.tm$ (line 15), so, by the code (line 27), $lpost.tm = j-1$ when C_p^1 is executed. Since we have assumed that C_p^1 fails and $seq.tm = lpost.tm$ when C_p^1 is executed, it must be that $lpost.value \neq post[i].value$ when C_p^1 is executed. Recall that $post[i].value = \perp$ immediately after C_1^{j-1}. Thus, there is an `UPDATE` whose CAS on $post[i]$ succeeds between C_1^{j-1} and C_p^1. Let U be the first such `UPDATE` and let C_U be the CAS executed by U. Claim D.9 implies that C_U writes the value $j-1$ to $post[i].tm$; let v be the value that C_U writes to $post[i].value$.

We argue that all CAS on $post[i]$ that are executed between C_U and C_1^j fail. Recall that all these CAS are executed by `UPDATES`. By the code (line 3), for each of these `UPDATES`, $lpost.value = \perp$. Notice that $post[i].value = v \neq \perp$ after the execution of C_U, and therefore the CAS by each of these `UPDATES` fails. It follows that $post[i].tm = j-1$ and $post[i].value = v$ at all points between C_U and C_1^j. Thus, when p performs its second read of $post[i]$ (line 29), it reads v in $post[i].value$, so $lpost.value = v$ when C_p^2 is executed by p. Recall that p reads j in $seq.tm$. Thus, by the code, $lpost.tm = j-1$ when C_p^2 is executed. Thus, C_p^2 is successful, which is a contradiction. △

Claim D.10 implies that there is a successful CAS on $post[i]$ executed by a clear_registers between C_1^{j-1} and C_1^j. By Claim D.9, it follows that there is exactly one such CAS C_{post}, as needed by 1. Recall that $post[i].tm = j-1$ immediately after C_1^{j-1}. Claim D.9 implies that $post[i].tm = j-1$ at all points between C_1^{j-1} and C_{post}, as needed by 2. It follows that $post[i].tm = j-1$ when C_{post} is executed. Therefore, by the code, it holds that $lpost.tm = j-1$ and $lpost.value = \perp$ when C_{post} is executed. So, C_{post} writes the value j to $post[i].tm$ and $\perp$ to $post[i].value$. Claim D.9 implies that these values do not change since C_1^j, as needed by 3. ■

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 D.11 *Let C_1^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_1^j. Then, after the execution of C_1^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 D.5, C_1^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_1^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_1^j, no CAS on seq by g is successful. Corollary D.5 implies that all values written to $seq.tm$ after C_1^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_1^j. It follows that if g executes a CAS on seq after C_1^j it will fail.

Fix any i, $1 \leq i \leq m$. We continue to prove that after the execution of C_1^j, no CAS on $post[i]$ by g is successful. Let C be any such CAS. By Lemma D.8 it follows that $post[i].tm \geq j$ at all points in time after C_1^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_1^{j-1} and C_1^j. Lemma D.8 implies that there is a successful CAS on $post[i]$ between C_1^{j-1} and C_1^j. Denote by C_{post} this CAS.

Lemma D.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 D.11 implies that for all grab_scans that start their execution before C_1^{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_1^{j-1} (or by functions called by them). Let $\mathcal{G}$ be the set of grab_scans that start their execution after C_1^{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_1^j. By Lemma D.3, *seq* does not change between C_1^{j-1} and C_1^j. Thus, $curr\_seq = seq$ when C_0' is executed. So, C_0' is successful. Lemma D.2 implies that the only successful CAS between C_1^{j-1} and C_1^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 $\mathcal{G}$. By the code, it follows that all clear_registers called by grab_scans in $\mathcal{G}$ 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_1^S and w_i^S have started their execution before C_1^S. This lemma is essential to prove that each `UPDATE` is linearized within its execution interval.

Lemma D.13 *For each $i \in \{1, \ldots, m\}$ such that w_i^S follows C_1^S, it holds that any* `UPDATE` *on A_i that performs its write to $pre[i]$ between C_1^S and w_i^S (including U_i^S) begins its execution before C_1^S.*

Proof: Assume that $C_1^S = C_1^j$ 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_1^S. Lemma D.12 implies that C_{post} is executed after C_0^S. By Corollary D.5, C_1^S writes the value $j+1$ in $seq.tm$. Lemma D.8 implies that $post[i].tm = j$ between C_1^S and C_{post}. Since C_{post} follows C_0^S, it follows that $post[i].tm = j$ between C_1^S and C_0^S.

Assume, by the way of contradiction, that there is at least one `UPDATE` on A_i that starts its execution after C_1^S and performs its write to $pre[i]$ before w_i^S. Denote by $\mathcal{U}$ the set of `UPDATES` that start their executions after C_1^S and perform their writes to $pre[i]$ before w_i^S. Let $U \in \mathcal{U}$ be the `UPDATE` of $\mathcal{U}$ that executes first its CAS C on $post[i]$. By Lemma D.1, C_0^S follows w_i^S, so C precedes C_0^S.

Recall that C_1^j writes $j+1$ to $seq.tm$. Lemmas D.2 and D.3 imply that $seq.tm = j+1$ from C_1^j to C_1^{j+1} (or to the end of the execution if $j = k_1$). Since U starts after C_1^S and performs C before C_0^S, it follows that U reads $j+1$ in $seq.tm$. Thus, by the code, $lpost.tm = j$ when C is executed. Recall that $post[i].tm = j$ between C_1^S and C_0^S. Since C precedes C_0^S, $post[i].tm = j$ when C is executed. So, $post[i].tm = lpost.tm$ when C is executed. By Lemma D.8, $post[i].value = \perp$ immediately after C_1^S. Lemma D.8 implies that the only CAS on $post[i]$ executed by clear_registers between C_1^S and C_1^{j+1} (or the end of the execution if $j = k_1$) is C_{post} which follows w_i^S and therefore also C. Since C is the first CAS on $post[i]$ executed by an `UPDATE` after C_1^S, it follows that $post[i].value = \perp$ when C is executed. By the code (line 3), $lpost.value = \perp$ when C is executed. It follows that C is a successful CAS on $post[i]$.

By Lemma D.1, w_i^S precedes $\tilde{r}_i^S$. So, C precedes $\tilde{r}_i^S$. Lemma D.12 implies that no clear_registers writes the value $\perp$ in $post[i]$ between C_1^S and C_0^S. Since $\tilde{r}_i^S$ precedes C_0^S, $\tilde{r}_i^S$ reads a value other than $\perp$ in $post[i]$. Thus, V_i^S is well-defined. Moreover, $V_i^S \notin \mathcal{U}$ since V_i^S reads the value written to $pre[i]$ by w_i^S, and therefore it performs its write to $pre[i]$ after w_i^S. So, $V_i^S \neq U$.

By definition, V_i^S performs its CAS C' on $post[i]$ before $\tilde{r}_i^S$ (since $\tilde{r}_i^S$ reads the value written in $post[i]$ by C'). Thus, C' precedes C_0^S. Moreover, C' follows w_i^S and therefore C' follows C. It

follows that $post[i].value \neq \perp$ when C' is executed. By the code (line 3), $lpost.value = \perp$ when C' is executed. Thus, C' is not a successful CAS, which is a contradiction. ■

Lemma D.14 *Let U be any* `UPDATE` *executed in α. Then, U is linearized within its execution interval.*

Proof: Let U be an `UPDATE` on A_i 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_i^S of U_i^S is executed after C_1^S, the write to $pre[i]$ by U is executed between C_1^S and w_i^S, and U is linearized just before C_1^S. Obviously, the execution of U ends after C_1^S. Lemma D.13 implies that U begins its execution before C_1^S. 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 D.15 *Let S_1 and S_2 be two* `SCANS` *with $g_c^{S_1} \neq g_c^{S_2}$. Then, $\alpha(C_1^{S_1}, C_0^{S_1})$ and $\alpha(C_1^{S_2}, C_0^{S_2})$ do not intersect.*

Proof: Since $g_c^{S_1} \neq g_c^{S_2}$, $C_0^{S_1} \neq C_0^{S_2}$. Without loss of generality, assume that $C_0^{S_1}$ precedes $C_0^{S_2}$. Lemma D.2 implies that there is at least one successful CAS of type 1 betwen $C_0^{S_1}$ and $C_0^{S_2}$. Thus, $C_1^{S_1} \neq C_1^{S_2}$ and $C_1^{S_2}$ follows $C_0^{S_1}$. Since $C_1^{S_1}$ precedes $C_0^{S_1}$, it follows that $\alpha(C_1^{S_1}, C_0^{S_1})$ and $\alpha(C_1^{S_2}, C_0^{S_2})$ do not intersect, as needed. ■

Next lemma proves that the linearization order of the `UPDATES` on any component A_i respects the order in which these `UPDATES` perform their writes to $pre[i]$. Its proof is in its biggest part identical to the proof of Lemma A.5.

Lemma D.16 *Let U_1, U_2 be two* `UPDATES` *on some component A_i, $1 \leq i \leq m$. Denote by w_1 the write to $pre[i]$ by U_1 and by w_2 the write to $pre[i]$ 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 $pre[i]$, then the claim holds trivially. Therefore, assume that at least one of the U_1, U_2 is not linearized at its write to $pre[i]$. We consider the following cases.

1. U_2 is linearized at w_2. Lemma D.14 implies that U_1 is linearized within its execution interval, so U_1 is linearized the latest 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 `SCAN` S such that w_2 has been performed between C_1^S and w_i^S. Since w_1 precedes w_2, w_1 has been executed before w_i^S. In case w_1 follows C_1^S, both U_1 and U_2 are linearized just before C_1^S in the order that they perform their writes to $pre[i]$. Thus, U_1 is linearized before U_2, which is a contradiction. So, assume w_1 precedes C_1^S. Lemma D.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_1^S. Thus, U_1 is linearized before U_2. A contradiction.

3. None of U_1, U_2 is linearized at its write to $pre[i]$. By the way linearization points are assigned, there are SCANS S_1 and S_2 such that w_1 has been performed between $C_1^{S_1}$ and $w_i^{S_1}$, and w_2 has been performed between $C_1^{S_2}$ and $w_i^{S_2}$; moreover, U_1 is linearized just before $C_1^{S_1}$, and U_2 is linearized just before $C_1^{S_2}$. Lemma D.1 implies that $w_i^{S_1}$ precedes $C_0^{S_1}$, and $w_i^{S_2}$ precedes $C_0^{S_2}$.

   If $g_c^{S_1} = g_c^{S_2}$, then $C_0^{S_1} = C_0^{S_2}$, so $C_1^{S_1} = C_1^{S_2}$. Thus, both U_1 and U_2 are linearized just before $C_1^{S_1} = C_1^{S_2}$ in the order they perform their writes to $pre[i]$. So, U_1 is linearized before U_2, which is a contradiction.

   If $g_c^{S_1} \neq g_c^{S_2}$, Lemma D.15 implies that $\alpha(C_1^{S_1}, C_0^{S_1})$ and $\alpha(C_1^{S_2}, C_0^{S_2})$ do not intersect. If $C_1^{S_1}$ precedes $C_1^{S_2}$, the linearization point of U_1 which is placed just before $C_1^{S_1}$ precedes the linearization point of U_2 which is placed just before $C_1^{S_2}$, which is a contradiction.

   Assume finally that $C_1^{S_1}$ follows $C_1^{S_2}$. By Lemma D.1, $w_i^{S_1}$ precedes $C_0^{S_1}$. and $w_i^{S_2}$ precedes $C_0^{S_2}$. Thus, w_1 which is executed between $C_1^{S_1}$ and $w_i^{S_1}$ follows w_2 which is executed between $C_1^{S_2}$ and $w_i^{S_2}$, 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 D.17 *For each $i \in \{1, \ldots, m\}$, r_i^S follows C_1^S.*

Proof: Recall that r_i^S is the read of $pre[i]$ by g_0^S which executes C_0^S. By definition, C_1^S is the last sucessful CAS of type 1 that precedes C_0^S. Assume that C_1^S writes the value j to $seq.tm$. Then, Lemma D.3 implies that C_0^S also writes the value j to $seq.tm$. Thus, by the code, g_0^S reads j in seq (line 15). Corollary D.5 implies that at all points in time before C_1^S, the value of $seq.tm$ is at most $j-1$. Thus, g_0^S performs its read of seq (let it be r_{seq}) after C_1^S. Since r_i^S follows r_{seq}, it follows that r_i^S follows C_1^S, as needed. ■

Lemma D.18 *Assume that S is a* SCAN *in α such that V_i^S is well-defined, and let r_{pre} be the read of $pre[i]$ by V_i^S. Then, r_{pre} is executed after C_1^S.*

Proof: Assume, by the way of contradiction, that r_{pre} is executed before C_1^S.

Suppose that $C_1^S = C_1^j$, for some j, $1 \leq j \leq k_1$. Then, Corollary D.5 implies that C_1^S writes $j+1$ to $seq.tm$. By definition of V_i^S, the CAS C on $post[i]$ by V_i^S is a successful CAS, and the value written in $post[i].value$ by C is read by $\tilde{r}_i^S$. By Lemma D.17, r_i^S follows C_1^S. Since $\tilde{r}_i^S$ follows r_i^S, $\tilde{r}_i^S$ follows C_1^S. By Lemma D.8, $post[i].tm = \perp$ immediately after C_1^S. By definition of V_i^S, the value written to $post[i]$ by C is read by $\tilde{r}_i^S$. Since $\tilde{r}_i^S$ follows C_1^S, it must be that C is executed after C_1^S. By Lemma D.8, it follows that $post[i].tm \geq j$ at all points in time after C_1^S. Thus, $post[i].tm \geq j$ when C is executed.

Since the read r of seq by V_i^S precedes r_{pre} and we have assumed that r_{pre} precedes C_1^S, r precedes C_1^S. Corollary D.5 implies that $seq.tm \leq j$ at all points in time before C_1^S. Thus, r reads a value for $seq.tm$ smaller than or equal to j. By the code, it follows that $lpost.tm < j$ when C is executed. Since $post[i].tm \geq j$ and $lpost.tm < j$ 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 A.6.

Lemma D.19 *Let α be an execution of* T-Opt. *Any* SCAN *executed in α returns a consistent vector.*

Proof: Assume that S returns the vector $\mathbf{v} = \langle v_1, ..., v_m \rangle$. By definition of C_0^S and by Lemma D.2, it follows that S returns the vector written in seq by C_0^S. Thus, the vector calculated by g_0^S is $\mathbf{v}$. So, either g_0^S reads $\perp$ in $post[i].value$ (line 8) and v_i in $pre[i]$, $1 \leq i \leq m$ or g_0^S reads v_i in $post[i].value$. In either case, by definition of U_i^S, U_i^S must write v_i to $pre[i]$. Thus, U_i^S uses v_i as a parameter. In case w_i^S precedes C_1^S, Lemma D.14 implies that U_i^S is linearized before C_1^S, where S is linearized. In case w_i^S follows C_1^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 $pre[i]$, and let w be the write to $pre[i]$ by U. We consider the following cases.

1. Assume that g_0^S reads $\perp$ in $post[i]$ (line 8) and v_i in $pre[i]$ (line 7). In case w precedes w_i^S, Lemma D.16 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 $pre[i]$. Therefore, w must follow r_i^S. By Lemma D.17, r_i^S follows C_1^S. So, w follows C_1^S. Since U is linearized before S, and S is linearized at C_1^S, U cannot be linearized at w. Therefore, there is a SCAN S' such that w is performed between $C_1^{S'}$ and $w_i^{S'}$, and U is linearized just before $C_1^{S'}$. Since w follows w_i^S, $g_0^S \neq g_c^{S'}$. Lemma D.15 implies that $\alpha(C_1^S, C_0^S)$ and $\alpha(C_1^{S'}, C_0^{S'})$ do not intersect. If C_1^S precedes $C_1^{S'}$, the linearization point of U which is placed at $C_1^{S'}$ follows the linearization point of S which is placed at C_1^S, which is a contradiction. Assume finally that C_1^S follows $C_1^{S'}$. Lemma D.1 implies that $w_i^{S'}$ precedes $C_0^{S'}$. Thus, w which is performed between $C_1^{S'}$ and $w_i^{S'}$, precedes C_1^S, which is a contradiction.

2. Assume now that g_0^S reads v_i in $post[i]$ (line 8). Then, V_i^S is well-defined and let r_{pre} be the read of $pre[i]$ by V_i^S (line 7). Recall that S returns the value that U_i^S uses as a parameter and therefore S returns the value that has been written to $pre[i]$ by U_i^S. In case w precedes w_i^S, Lemma D.16 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_{pre} since (by the definition of U_i^S) V_i^S reads the value that w_i^S writes. By Lemma D.18, r_{pre} follows C_1^S. Therefore, w follows C_1^S. Since U is linearized before S, and S is linearized at C_1^S, U cannot be linearized at w. Thus, there is a SCAN S' such that $w_i^{S'}$ follows $C_1^{S'}$, w is performed between $C_1^{S'}$ and $w_i^{S'}$, and U is linearized just before $C_1^{S'}$. Since w is performed after w_i^S, $g_c^{S'} \neq g_0^S$.

   Lemma D.15 implies that $\alpha(C_1^S, C_0^S)$ and $\alpha(C_1^{S'}, C_0^{S'}$ do not intersect. If C_1^S precedes $C_1^{S'}$, the linearization point of U which is placed at $C_1^{S'}$ follows the linearization point of S which is placed at C_1^S, which is a contradiction. Assume finally that C_1^S follows $C_1^{S'}$. Lemma D.1 implies that $w_i^{S'}$ precedes $C_0^{S'}$. Thus, w which is performed between $C_1^{S'}$ and $w_i^{S'}$, precedes C_1^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. ■