A Fully Concurrent Adaptive Snapshot Object for RMWable Shared-Memory

Benyamin Bashari
University of Calgary, Canada

David Yu Cheng Chan
University of Calgary, Canada

Philipp Woelfel
University of Calgary, Canada

Abstract

An adaptive RMWable snapshot object maintains an array $A[0..m-1]$ of m readable shared memory objects that support an arbitrary set of read-modify-write (RMW) operations, in addition to `Read()`. Each array entry $A[i]$ can be accessed by any process using an operation `Invoke(`i, *op*`)`, which simply applies a supported RMW operation *op* to $A[i]$ and returns the response of *op*. In addition, processes can record the state of the array by calling `Click()`. While `Click()` does not return anything, a process p can call `Observe(`i`)` to determine the value of $A[i]$ at the point of p's latest `Click()`.

Recently, Jayanti, Jayanti, and Jayanti [10] presented an RMWable adaptive snapshot object, where all operations have constant step complexity. Their algorithm is *single-scanner*, meaning that `Click()` operations cannot be executed concurrently. We present the first *fully concurrent* RMWable adaptive snapshot object, where all operations can be executed concurrently, assuming the the system provides atomic Fetch-And-Increment and Compare-And-Swap operations. `Click()` and `Invoke()` operations have constant step complexity, and `Observe()` has step complexity $O(\log n)$. The total number of base objects needed is $O(mn \log n)$.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms

Keywords and phrases Shared memory, snapshot, camera object, RMW, distributed computing

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.7

Funding We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC),under Discovery Grant RGPIN/2019-04852, and the Canada Research Chairs program.

1 Introduction

Linearizable snapshot objects are a fundamental building block for shared memory algorithms. A snapshot object maintains an array of m registers, $A[0 \dots m-1]$. The standard definition allows a process to write to an array entry, and to perform a `Scan()`, which returns the vector $(A[0], \dots, A[m-1])$. Most research considers single-writer snapshots, where m is equal to the number of processes, n, and process i can only write to $A[i]$.

Implementing deterministic linearizable single-writer snapshot objects from atomic registers (which support read and write operations) has been studied intensively (e.g., [6, 1, 2, 8]). Inoue and Chen [8] devised a linearizable snapshot, where each operation has at most linear step complexity, which is optimal at least for `Scan()` operations [12]. In order to circumvent this lower bound, researchers limited the number of operations [3] or employed randomization [4, 13].

Many snapshot algorithms assume that the size of a memory word is large enough to store the entire state of array A. This is an unrealistic assumption, unless large registers are simulated by smaller ones, which is inherently inefficient. Employing stronger primitives, such as compare-and-swap (CAS) and fetch-and-increment (FAI) objects, one can obtain snapshot objects, where it is sufficient for a memory location to store a single array entry [15]. However, then the complexity of a `Scan()` is inherently lower bounded by the size of the array.

To deal with this inherent inefficiency, some researchers studied snapshot types that allowed certain operations to return limited information about array A more efficiently. For example, Jayanti [9] proposed the *f-array* object, where a `Read()` operation returns the value of a function f applied to all components of array A. This function can be computed in a constant number of steps, but updating array A is more expensive: In Jayanti's original algorithm (which allows read-modify-write (RMW) operation to be applied to individual components of A) updating a single component of A has step complexity $\Theta(m)$, where m is the size of the array. Obryk [14] provided a version of this object, where components can only be updated with write operations, but in $O(\log^3 m)$ steps.

Attiya, Guerraoui, and Ruppert [5] followed a different approach: Their *partial snapshot object* allows processes to obtain a view of only some of the entries of A. The step complexity of such a *partial scan* is quadratic in the number of array entries the view contains, and the amortized step complexity of updates is bounded by the maximum interval contention, as well as the maximum number of components accessed by partial scan operations. Bashari and Woelfel [7] devised an *adaptive* single-writer snapshot object, where a snapshot is taken by a `Click()`[1] operation that does not return anything. Instead, a process can later determine the value of any array entry $A[i]$ at the point of its latest preceding `Click()`, by performing an `Observe(`i`)` operation. Contrary to the partial snapshots of Attiya, Guerraoui, and Ruppert [5], this semantics allows observed array entries to be chosen adaptively, based on previously observed values. The algorithm uses polynomially many single-word registers and CAS objects, as well as an unbounded FAI object. `Click()` has constant step-complexity, whereas updating or observing an array entry takes $O(\log n)$ steps.

Another shortcoming of many snapshot algorithms is that the entries of array A can only be updated with write operations. But modern shared memory systems critically support many types of read-modify-write (RMW) operations, which are much more powerful than reads and writes, and most non-trivial data structures rely on such RMW operations. Thus, conventional snapshot algorithms (where write is the only allowed update operation) cannot be used to obtain snapshots of most data structures. Jayanti's f-array object [9] addresses this issue, by allowing the components of array A to be of arbitrary types. But, as mentioned earlier, updates have step complexity of $\Omega(m)$.

Wei, Ben-David, Blelloch, Fatourou, Ruppert and Sun [16] also presented a snapshot object, where the array entries can be modified with `CAS()` operations. The algorithm supports a snapshot operation that returns a *handle*. The value of individual array entries at the point of when the handle was obtained, can then be inspected adaptively. The algorithm uses CAS objects, and the step complexity of observing the value of a single array entry grows linearly with the number of updates that may have occurred on that location, since the corresponding snapshot was taken. The authors also showed that their interface can be used to easily add snapshot operations to concurrent data structures (that are implemented from CAS objects), and presented experimental results, indicating a low overhead of this approach.

Very recently, Jayanti, Jayanti, and Jayanti [10] presented an *RMWable* adaptive snapshot object. Their algorithm generalizes the semantics of Bashari and Woelfel's adaptive snapshot object, by allowing array entries to be updated with any RMW operations [10] that are

[1] This operation was also called `Scan()` in [7]. Jayanti, Jayanti, and Jayanti [10] used the term `Click()`, which more clearly indicates that the semantics is different from a standard `Scan()`.

supported by the system. Their algorithm has optimal constant step complexity for `Click()`, but multiple `Click()` operations cannot be executed concurrently. We present a (completely different and independently devised) algorithm for the same sequential specification. Our algorithm achieves full concurrency (i.e., it allows concurrent `Click()` operations) for the price of `Observe()` operations having a step complexity $O(\log n)$ instead of constant.

Consider a set $\mathcal{O}$ of wait-free linarizable objects available to the system, such that each object supports a read operation (among others). Our adaptive RMWable snapshot object maintains an array A of m objects from $\mathcal{O}$, where m is an arbitrary positive integer. (The assumption that all array components are of the same type is made for ease of description only; in fact, each array entry can be of a different readable type.)

Each process p can execute `Invoke(`i, op`)` to apply any operation op (supported by the object represented by $A[i]$) to $A[i]$ and obtain the response of that operation. A process can take a snapshot of the array using a `Click()` operation, which returns nothing. Finally, p can at any point call `Observe(`i`)`, which returns the value of $A[i]$ at the point of p's latest `Click()` operation.

We assume that the system provides atomic FAI and CAS operations. In a system with n processes, `Click()` and `Invoke()` operations have constant step complexity, and `Observe()` has step complexity $O(\log n)$. The total number of base objects needed is $O(mn \log n)$.

The FAI object needs to perform approximately one increment per implemented operation, and the resulting values need to be stored in other objects. Thus, strictly speaking, our algorithm can only perform a bounded number of operations. However, in practice this bound will never be reached on 64-bit architectures.

In the following section we describe the system model and specify the object we are implementing. Then, in Section 3 we present the algorithm and its properties. Finally, in Section 4, we will proof correctness. The analysis of time and space complexity is omitted due to space restrictions.

2 Preliminaries

We consider the standard asynchronous shared memory model with n processes with IDs $0, \ldots, n-1$, which communicate using atomic (or linearizable) shared memory operations on *base objects*.

A register supports the standard `Read()` and `Write()` operations. An LL/SC object provides operations, `LL()` and `SC(`v`)`, where `LL()` returns the object's value, and `SC(`v`)` called by process p updates the value to v, if p has previously called `LL()` and no successful `SC()` operation has occurred since then. An `SC()` operation returns a Boolean value indicating if it successfully stored its parameter. A FAI object stores an integer, initially 1, and provides an operation `FAI()`, which increments the object's value by 1 and returns the value before the increment.

While `FAI()` is available on standard hardware, LL/SC is not. However, there are efficient implementations of LL/SC from registers and CAS objects, which are usually available. For example, by using unbounded sequence numbers, one can implement an LL/SC object from a single CAS object with constant step complexity. An algorithm by Jayanti and Petrovic [11] avoids unbounded sequence numbers, but needs $O(n)$ CAS objects to implement an LL/SC object with constant step complexity.

The Adaptive RMWable Snapshot Object

Let $\mathcal{O}$ denote a set of wait-free linearizable objects that are available in the system. Each object in that set must be readable, i.e., support an operation that returns the state of the object without changing it. For the ease of description, we assume that each operation on such an object takes at most a constant number of steps.

The adaptive RMWable snapshot object maintains an array of m components, each corresponding to an object in $\mathcal{O}$ in its initial state. For convenience, we assume w.l.o.g. that the initial state of each component is 0.

An adaptive RMWable snapshot object allows each process p to perform the following operations:

- `Invoke(`i, *op*`)` performs operation *op* (which must be one of the operations supported by $\mathcal{O}$) on the i-th component , and returns the corresponding response.
- `Click()` simply returns **done** (this is convenient for our proofs, but equivalently, one may assume that it returns nothing).
- `Observe(`i`)` returns the value of component i at the time of p's last `Scan()`; or the initial state 0 of component i if no such `Scan()` exists.

3 The Algorithm

Let n and m be positive integers, κ be a sufficiently large constant, $\Delta' = O(\log n)$, and $\Delta = \kappa n \log n / \Delta'$. In this section, we present an implementation of the adaptive RMWable snapshot object for n processes and m components such that:

- The space complexity of the implementation is $O(m\Delta)$.
- The time complexity of `Click()` operations is $O(1)$.
- The time complexity of `Invoke()` operations is $O(\Delta')$.
- The time complexity of `Observe()` operations is $O(\log \Delta)$.

Thus if we select $\Delta' = 1$, we have $\Delta = O(n \log n)$ and thus obtain:

- The space complexity of the implementation is $O(mn \log n)$.
- The time complexity of `Click()` operations is $O(1)$.
- The time complexity of `Invoke()` operations is $O(1)$.
- The time complexity of `Observe()` operations is $O(\log n)$.

3.1 Bashari and Woelfel's Single-Writer Snapshot

The fundamental idea of our algorithm is based on Bashari and Woelfel's adaptive partial snapshot algorithm [7]. Their algorithm implements an adaptive snapshot object for n processes and $m = n$ components that each correspond to a single writer register. Hence, instead of `Invoke(`$i, -$`)`, it supports `Write(`i, *val*`)`, which only process i can execute in order to write some value *val* to the i-th component. Their algorithm employs a FAI object *clk*, and m single-writer multi-reader red-black trees. The i-th red-black tree can only be updated by process i, who uses it to record the past states of component i. On a high level, the algorithm works as follows:

- Each `Click()` operation takes a timestamp from the FAI object *clk*.
- Each `Write()`i, *val* operation takes a timestamp from the FAI object *clk*. Then it simply stores *val* along with its timestamp into the i-th red-black tree.
- Each `Observe(`i`)` operation by a process p searches the i-th red-black tree for the state with the largest timestamp that is smaller than the timestamp of the latest `Click()` operation by process p.
- The red-black trees are periodically pruned of recorded states that are no longer necessary, and thus inserts and searches take only $O(\log n)$ steps.

The i-th red-black tree serves as a predecessor data structure that can be queried by all processes but only updated by process i. As our algorithm allows updates on component i to be performed by any process, we need to replace each red-black tree with a multi-writer predecessor data structure. Moreover, adding the correct elements to the predecessor data structure is substantially more challenging, because multiple processes may perform `Invoke`$(i, -)$[2] concurrently.

3.2 Outline of our Algorithm

Algorithm 1 depicts our adaptive RMWable snapshot implementation. Similar to Bashari and Woelfel we use an FAI object *clk* to record timestamps. Consider some $i \in \{0, \ldots, m-1\}$. We use an object $O[i]$ with the same sequential specification as the i-th component object. To perform `Invoke`(i, op_i), a process p performs operation op_i on $O[i]$ and records the return value, which it will later use as its response. Before p's `Invoke`(i, op_i) can linearize, the resulting state of component i needs to be "recorded" in a predecessor data structure, together with a timestamp obtained from *clk*.

The predecessor data structure for the i-th component is implemented using a circularly sorted array $R[0 \ldots \Delta - 1][i]$. For now assume that at most Δ `Invoke`$(i, -)$ operations can be performed; this will ensure that $R[0 \ldots \Delta - 1][i]$ remains completely sorted.

First consider the simplified single-updater case, in which only one process p is allowed to call `Invoke`(i, op_i). In its j-th `Invoke`(i, op_i) operation, after performing op_i on $O[i]$, p can obtain a new timestamp k using a `FAI()` operation on *clk*, and then write k and the new value of $O[i]$ into $R[j][i]$. This way, $R[0 \ldots \Delta - 1][i]$ remains sorted (by timestamp values). A process q that performs a `Click()` also obtains a timestamp k' from *clk*. To observe component i, q can then simply return the value of $O[i]$ that was recorded in the array entry $R[j][i]$, $j \in \{0, \ldots, \Delta-1\}$, with the largest timestamp $k \leq k'$. That array entry can be found in $O(\log \Delta)$ steps using a binary search.

In order to support multiple concurrent `Invoke`$(i, -)$ operations, processes with pending such operations will agree on *some state* of $O[i]$, and add that agreed upon value to an appropriate array entry of $R[0 \ldots \Delta - 1][i]$, together with an appropriate timestamp k. This is done in a `HelpUpdate()` method, as follows: We use an LL/SC object $\mathit{lastUpdate}[i]$, which stores a triple (j, k, val) where j is a sequence number, k is either a timestamp or $\perp$, and val is either a state of $O[i]$ or $\perp$. Initially, $\mathit{lastUpdate}[i] = (0, \perp, \perp)$. In `HelpUpdate()`, a process q repeats the following several times: If $\mathit{lastUpdate}[i] = (j, \perp, \perp)$ then it reads the current value val from $O[i]$ and tries to change $\mathit{lastUpdate}[i]$ to $(j+1, \perp, \mathit{val})$ using an `SC()` operation. If $\mathit{lastUpdate}[i] = (j, \perp, \mathit{val})$ for $\mathit{val} \neq \perp$, then q obtains a timestamp k from *clk* and tries to change $\mathit{lastUpdate}[i]$ to (j, k, val). Once $\mathit{lastUpdate}[i] = (j, k, \mathit{val})$ for $k, \mathit{val} \neq \perp$, the pair (k, val) is the agreed upon pair that will be added to the predecessor object. Since (k, val) is the j-th agreement pair, it can simply be written to $R[j][i]$. Nothing changes for `Click()` and `Observe()` operations.

To see that this is linearizable, consider the following linearization points: A `Click()` linearizes when the calling process obtains a timestamp from *clk*, and an `Observe()` can linearize at any point during its execution interval. Now consider an `Invoke`(i, op_i) operation during which process p performs op_i on $O[i]$ at some point t. Let t' be the first point after t, at which the value of $O[i]$ is copied to $\mathit{lastUpdate}[i]$. Then p's `Invoke`(i, op_i) linearizes at

[2] Throughout this text we use a dash ("$-$") as the argument of a method call, to indicate that the statement applies to all arguments.

Algorithm 1 Adaptive RMWable Snapshot Implementation.

```
Shared:
    FAI clk, initially 1
    LL/SC lastScan[0 . . . n − 1][0 . . . m], each initially (0, 0, 0, 0, 0)
    LL/SC/Read lastUpdate[0 . . . m − 1], each initially (0, 0, ⊥)
    Object O[0 . . . m − 1], each initially fresh.
    LL/SC R[0 . . . Δ − 1][0 . . . m − 1], initially (0, 0, 0)
Code for each process p:
Function Click()
    (k, −, −, −, v) ← lastScan[p][m].LL()
    if v = 0 then lastScan[p][m].SC(k, 0, 0, 0, 1)
    HelpScan(p)
    return done
Function HelpScan(q)
    (−, −, −, −, v) ← lastScan[q][m].LL()
    if v = 1 then
        k ← clk.FAI()
        lastScan[q][m].SC(k, 0, 0, 0, 0)
Function Observe(i)
    HelpUpdate(i)
    repeat v ← HelpObserve(p, i) until v ≠ ⊥
    return v
Function HelpObserve(q, i)
    (k_i, maxKey, j_left, j_right, v) ← lastScan[q][i].LL()
    (k_m, −, −, −, −) ← lastScan[q][m].LL()
    if k_m > k_i then
        (j_u, k_u, v_u) ← lastUpdate[i].Read()
        if (k_u, v_u) ≠ (0, ⊥) then j_right ← j_u + Δ − 1 mod Δ
        else j_right ← j_u mod Δ
        j_left ← j_right + 1 mod Δ
        (−, maxKey, −) ← R[j_right][i].LL()
        (k_i, v) ← (k_m, ⊥)
        if maxKey < k_i then return ⊥
    else
        if v ≠ ⊥ then return v
        if j_left > j_right then j ← ⌈(j_left + j_right + Δ)/2⌉ mod Δ
        else j ← ⌈(j_left + j_right)/2⌉
        (−, k_r, −) ← R[j][i].LL()
        if k_r ≥ k_i and k_r ≤ maxKey then  j_right ← j + Δ − 1 mod Δ
        else j_left ← j
        if j_left = j_right then (−, −, v) ← R[j_left][i].LL()
    lastScan[q][i].SC(k_i, maxKey, j_left, j_right, v)
    return ⊥
Function Invoke(i, op_i)
    v_res ← O[i].op_i()
    HelpUpdate(i)
    return v_res
Function HelpUpdate(i)
    for a ∈ {0, . . . , 5} do
        (j_u, k_u, v_u) ← lastUpdate[i].LL()
        if v_u = ⊥ then
            v ← O[i].Read()
            lastUpdate[i].SC(j_u + 1, ⊥, v)
        else
            if k_u = ⊥ then
                k ← clk.FAI()
                lastUpdate[i].SC(j_u, k, v_u)
            else
                (j, −, −) ← R[j_u mod Δ][i].LL()
                if j < j_u then R[j_u mod Δ][i].SC(j_u, k_u, v_u)
                for a′ ∈ {0 . . . Δ′} do
                    HelpScan(j_u mod n)
                    HelpObserve(j_u mod n, i)
                lastUpdate[i].SC(j_u, 0, ⊥)
```

the first point when some process obtains a sequence number k from *clk*, such that the pair (k, val) gets stored in $lastUpdate[i]$. In other words, if (j, k_j, val_j) is the j-th triple stored in $lastUpdate[i]$ satisfying $val_j, k_j \neq \perp$, then all `Invoke(`$i, -$`)` operations whose operation on $O[i]$ is reflected in val_j but not val_{j-1} linearize at the point timestamp k_j is obtained. The essential steps of `HelpUpdate()` are repeated sufficiently many times to ensure that this happens before any of the linearized `Invoke(`$i, -$`)` methods respond.

In the above approach, `HelpUpdate()` allows processes to repeatedly agree on a value $O[i]$ and an associated timestamp. If (k_j, val_j) is the j-th agreed timestamp-value pair, then the triple (j, k_j, val_j) will be written to $R[j][i]$. As $R[0 \ldots \Delta - 1][i]$ has size Δ, this only works if the number of `Invoke(`$i, -$`)` operations is bounded by Δ. To support an unbounded number of `Invoke(`$i, -$`)` operations, the triple (j, k_j, val_j) will be written to $R[j \bmod \Delta][i]$, instead. While the array remains circularly sorted, and binary search is still possible, we now face the problem that old values in $R[0 \ldots \Delta][i]$ will eventually get overwritten.

We deal with that as follows: Following a `Click()` call by process p, for each $i \in \{0, \ldots, m-1\}$, the relevant value stored in $R[0 \ldots \Delta][i]$ (i.e., the one which p would have to return in a subsequent `Observe(`i`)` operation), will be copied to another LL/SC object, $lastScan[p][i]$. When some process q performs `HelpUpdate(`i`)`, it contributes $O(\Delta')$ of work to that, guaranteeing that all relevant array entries of $R[0 \ldots \Delta][i]$ are copied to $lastScan[p][i]$, before they get overwritten. It does so by calling `HelpObserve(`p, i`)`. In that method call, it contributes a constant number of steps to a binary search on $R[0 \ldots \Delta][i]$ for the relevant array entry. To facilitate multiple processes participating in this binary search, $lastScan[p][i]$ stores a 5-tuple $(k_i, maxKey, j_{left}, j_{right}, v)$, where k_i is the timestamp that p obtained during its `Click()`, $maxKey$ is essentially the largest key found in $lastUpdate[i]$, when the first process started the binary search, j_{left} and j_{right} are the current left and right borders found during the binary search, and v will eventually be set to the correct value (representing the state of $O[i]$) found in the binary search. Each process q contributes to the binary search by loading the value of $lastScan[p][i]$, computing the next value that needs to be written to $lastScan[p][i]$, and then attempting to write that value using an `SC()` operation. If some other process has already performed that next step of the binary search, then q's `SC()` will simply fail. The exact details of the binary search are described in Section 3.3.

We still need to deal with one other problem: Suppose process p obtains a timestamp k from *clk* in its `Click()` method, and immediately after that falls asleep, before it can write k anywhere. Then the relevant value of $R[0 \ldots \Delta][i]$ may get overwritten before any other process even learns about k. I.e., no process can help copying relevant values from $R[0 \ldots \Delta][i]$ to $lastScan[p][i]$, before it's too late. To deal with that, at the beginning of its `Click()`, process p announces that it has started a `Click()` operation by setting a bit in the last component of $lastScan[p][m]$. (Note the index m, which means the array entry is not used for values copied from R.) That bit indicates that other processes should help p with its `Click()` operation, specifically with obtaining and publishing a timestamp. They do so by calling a method `HelpScan(`p`)` before each `HelpObserve(`p`)` call during `HelpUpdate()` (the helped process, p is chosen in a round-robin fashion, based on the sequence number found in $lastUpdate[i]$). In such a `HelpScan(`p`)` call, process q checks if p wants help (as indicated by the last component of $lastScan[p][m]$), and if yes, q obtains a timestamp k from *clk*. Then, using an `SC()` operation, it tries to store that timestamp into the first component of $lastScan[p][m]$ while also resetting the last component to 0. The timestamp associated with p's `Click()` operation is then the first timestamp that gets written to $lastScan[p][m]$, and the `Click()` linearizes when that timestamp is obtained from *clk*.

3.3 Low Level Description

Our algorithm uses the following shared objects:

- *clk*: A FAI object that stores timestamps.
- $lastScan[0 \ldots n-1][0 \ldots m]$: An array of LL/SC objects that record the timestamps of the last `Click()` operation by each process and the states of each component object at the time when the timestamp was received from *clk*.

  For every process $p \in \{0, 1, \ldots, n-1\}$, $lastScan[p][m]$ stores a tuple $(k, 0, 0, 0, v)$, where k is the timestamp of the last `Click()` operation by process p, and $v = 1$ if p needs another timestamp for pending `Click()` operation; otherwise $v = 0$.

  For every process $p \in \{0, 1, \ldots, n-1\}$ and every integer $i \in \{0, 1, \ldots, m-1\}$, $lastScan[p][i]$ stores a tuple $(k, maxKey, j_{left}, j_{right}, v)$, where k is the last detected timestamp of the last `Click()` operation by process p, v is either the state of the i-th component object at the time when k was received from *clk* or $\perp$ if that is yet to be deduced, and $maxKey$, j_{left}, and j_{right} are integers that are used to help deduce that state.
- $O[0 \ldots m-1]$: For every integer $i \in \{0, 1, \ldots, m-1\}$, $O[i]$ is a wait-free linearizable readable base object with the same sequential specification as the i-th component object, and is used to determine the state of this i-th component object at various timestamps. Note that at any time, the state of the i-th component object is not necessarily the same as the state of $O[i]$.
- $lastUpdate[0 \ldots m-1]$: For every integer $i \in \{0, 1, \ldots, m-1\}$, $lastUpdate[i]$ is an LL/SC object that is intuitively used to repeatedly pair a timestamp k received from *clk* with a state v read from the base object $O[i]$, and thus intuitively set the state of the i-th component object to v at the time when timestamp k was received from *clk*.
- $R[0 \ldots \Delta-1][0 \ldots m-1]$: For every integer $i \in \{0, 1, \ldots, m-1\}$, $R[0 \ldots \Delta-1][i]$ is a circularly sorted array of LL/SC objects that records the previous states for the i-th component object (replacing the red-black trees of [7]).

  For every integer $i \in \{0, 1, \ldots, m-1\}$ and $j \in \{0, 1, \ldots, \Delta-1\}$, $R[j][i]$ stores a tuple (j_r, k_r, v_r), where roughly speaking, k_r is a timestamp, v_r was the state of the i-th component at the time when k_r was received from *clk*, and (j_r, k_r, v_r) was the value in $lastUpdate[i]$ at the time when $R[j][i]$ was last modified.

To achieve the desired time and space complexities, our algorithm heavily relies on various helping mechanisms, which we have divided into the auxiliary functions `HelpScan(`q`)`, `HelpObserve(`q, i`)`, and `HelpUpdate(`i`)` that intuitively help to complete `Click()`, `Observe()`, and `Invoke()` operations respectively.

In the following we describe which steps a process p performs during each of the indicated operations.

3.3.1 `HelpScan()`

During each `HelpScan(`q`)` operation, a process p performs the following steps:

1. It performs an `LL()` operation on $lastScan[q][m]$ to check whether process q needs a timestamp for a pending `Click()` operation (line 7).
2. If so, it takes a timestamp k from the FAI object *clk* (line 9), and attempts to give this timestamp to q's pending `Click()` operation by performing an `SC(`$k, 0, 0, 0, 0$`)` operation on $lastScan[q][m]$ (line 10).

3.3.2 `HelpObserve()`

During each `HelpObserve(`q, i`)` operation, a process p performs the following steps:

1. It performs an `LL()` operation on $lastScan[q][i]$ to get a tuple $(k_i, maxKey, j_{left}, j_{right}, v)$, which indicates the prior progress (if any) that has been made in helping a potential `Observe(`i`)` operation by process q after its last `Click()` operation (line 16).
2. It performs an `LL()` operation on $lastScan[q][m]$ to read the timestamp k_m of the last `Click()` operation by process q (line 17).
3. If the timestamp in $lastScan[q][i]$ is older than the timestamp k_m in $lastScan[q][m]$, then that indicates that no progress has been made in helping a potential `Observe(`i`)` operation by process q after its last `Click()` operation (line 18).
   In this case, the tuple $(k_i, maxKey, j_{left}, j_{right}, v)$ that was received from $lastScan[q][i]$ is outdated and p has to compute replacement values for them. So p performs the following steps:
   a. It reads $lastUpdate[i]$ (line 19) to determine the integer j_{right} that corresponds to the (first or second) most recent entry of $R[0 \dots \Delta - 1][i]$ to be modified (lines 20 to 21), and the integer j_{left} that is for the next entry after j_{right}.
   b. It then reads the timestamp $maxKey$ from the entry corresponding to j_{right} (line 23).
   c. It then sets k_i to k_m and v to $\perp$ (line 24).
   d. If the timestamp $maxKey$ is older than the timestamp $k_i = k_m$ that was received from $lastScan[q][m]$, then it is not safe to help any potential `Observe(`i`)` operation by q yet. Intuitively, this is because there could still be pending `Invoke(`$i, -$`)` operations that could potentially be linearized before the last `Click()` operation by process q. So in this case, p simply returns $\perp$ on line 25, indicating that future help may still be needed.
   e. Otherwise, p attempts to set $lastScan[q][i]$ to $(k_m, maxKey, j_{left}, j_{right}, \perp)$ (line 34), and returns $\perp$ on line 35, indicating that future help may still be needed.

   Otherwise, p performs the following steps:
   a. It checks whether v is a non-$\perp$ value. If so, then this non-$\perp$ value v is already the appropriate value for any potential `Observe(`i`)` operation by process q to return, and so there is no more need to help. Thus p simply returns this non-$\perp$ value v on line 27.
   b. It performs a single iteration of a binary search on the circularly sorted array $R[0 \dots \Delta - 1][i]$, checking the entry that is intuitively the mid-point of j_{left} and j_{right} to compare its timestamp to k_i, and then appropriately setting either j_{left} or j_{right} to the mid-point (lines 28 to 32).
   c. If $j_{left} = j_{right}$, then that indicates that the binary search has completed, and intuitively $R[j_{left}][i]$ should contain $(-, k_r, v_r)$ such that the timestamp k_r is just before the timestamp k_i, and v_r is the state of the i-th component object at the time that the timestamp k_r was received from clk. So in this case, p simply reads $(-, -, v)$ from $R[j_{left}][i]$ (line 33).
   d. Finally, p attempts to set $lastScan[q][i]$ to $(k_i, maxKey, j_{left}, j_{right}, v)$ (line 34), and returns $\perp$ on line 35, indicating that future help may still be needed.

3.3.3 `HelpUpdate()`

During each `HelpUpdate(`i`)` operation, a process p performs the following steps:

1. It performs an `LL()` operation on $lastUpdate[i]$ to receive a tuple (j_u, k_u, v_u) on line 42. If $v_u = \perp$, then $lastUpdate[i]$ currently contains neither a timestamp from clk nor a state from $O[i]$ (line 43). So p performs the following steps:

a. It reads a state v from $O[i]$ (line 44).
b. It performs an $\texttt{SC(}j_u + \mathit{1}, \bot, v\texttt{)}$ operation to store this state v into $\mathit{lastUpdate}[i]$ (line 45).

Otherwise, if $k_u = \bot$, then $\mathit{lastUpdate}[i]$ currently does not contain a timestamp from *clk* (line 47). So p performs the following steps:
a. It takes a timestamp k from *clk* via a FAI operation (line 48).
b. It performs an $\texttt{SC(}j_u, k, v_u\texttt{)}$ operation to store this timestamp k into $\mathit{lastUpdate}[i]$ (line 49).

Otherwise, $\mathit{lastUpdate}[i]$ currently contains both a timestamp k_u from *clk* and a state v_u from $O[i]$, indicating that the state of the i-th component object was v_u at the time when the timestamp k_u was received from *clk*. So p performs the following steps:
a. It performs an `LL()` operation on $R[j_u \bmod \Delta][i]$ (line 51), which intuitively should be the least recent entry of $R[0 \ldots \Delta - 1][i]$ to be modified, which makes it the safest to overwrite.
b. If this entry has not yet been modified by a concurrent $\texttt{HelpUpdate(}i\texttt{)}$ operation by any other process, then process p performs an $\texttt{SC(}j_u, k_u, v_u\texttt{)}$ operation on $R[j_u \bmod \Delta][i]$ (line 52) to now record that the state of the i-th component object was v_u at the time when the timestamp k_u was received from *clk*.
c. It performs $\Delta' + 1$ alternating $\texttt{HelpScan(}j_u \bmod n\texttt{)}$ and $\texttt{HelpObserve(}j_u \bmod n, i\texttt{)}$ operations (lines 53 to 55). Intuitively, this ensures that enough help is given to `Click()` and `Observe()` operations such that the next least recent entries of $R[0 \ldots \Delta - 1][i]$ are no longer needed and can be safely overwritten.
d. It performs an $\texttt{SC(}j_u, \mathit{0}, \bot\texttt{)}$ operation on $\mathit{lastUpdate}[i]$ to indicate that it is now ready for a new timestamp and state pair (line 56).

2. It repeats from the start another 5 times, which intuitively ensures that enough help is given to `Invoke()` operations such that the resulting state of the i-th component object and a corresponding timestamp is now recorded.

3.3.4 `Click()`

Each process p performs the following steps to perform a `Click()` operation:
1. It changes the last field of $\mathit{lastScan}[p][m]$ to 1 (line 2), to indicate to all other processes that process p needs a timestamp for this pending `Click()` operation.
2. It calls $\texttt{HelpScan(}p\texttt{)}$ (line 4) to help itself complete this `Click()` operation, then returns **done** (line 5).

3.3.5 `Observe()`

Each process p performs the following steps to perform an $\texttt{Observe(}i\texttt{)}$ operation:
1. It calls $\texttt{HelpUpdate(}i\texttt{)}$ (line 12) to help complete any pending $\texttt{Invoke(}i, -\texttt{)}$ operations that could interfere with this $\texttt{Observe(}i\texttt{)}$ operation.
2. It repeatedly calls $\texttt{HelpObserve(}p, i\texttt{)}$ to help this $\texttt{Observe(}i\texttt{)}$ operation until it receives a non-$\bot$ value v (line 12), which it then returns (line 14).

3.3.6 `Invoke()`

Each process p performs the following steps to perform an $\texttt{Invoke(}i, \mathit{op}_i\texttt{)}$ operation:
1. It performs the operation op_i on $O[i]$ (line 37), changing the state of $O[i]$ and receiving an appropriate response value v_{res} for this $\texttt{Invoke(}i, \mathit{op}_i\texttt{)}$ operation.
2. It calls $\texttt{HelpUpdate(}i\texttt{)}$ (line 38) to help to record down a state of the i-th component and a timestamp into $R[0 \ldots \Delta - 1][i]$, then returns v_{res} (line 39).

4 Proof of Correctness

In this section we prove that our algorithm is linearizable. Let H be any history of the adaptive RMWable snapshot object.

▶ **Observation 1.** *From the algorithm, it is clear that for every integer $i \in [0 \ldots m-1]$:*

- *Whenever $lastUpdate[i] = (-, k, \perp)$, $k = 0$.*
- *Every successful SC operation on $lastUpdate[i]$ on line 45 changes $lastUpdate[i]$ from $(j, 0, \perp)$ to $(j+1, \perp, v)$ for some integer j and some non-$\perp$ value v such that between the matching LL operation on $lastUpdate[i]$ on line 42 and this successful SC operation on $lastUpdate[i]$, v is received from a* `Read()` *operation on $O[i]$ on line 44.*
- *Every successful SC operation on $lastUpdate[i]$ on line 49 changes $lastUpdate[i]$ from $(j, \perp, v)$ to (j, k, v) for some integer j, some positive integer k, and some non-$\perp$ value v such that between the matching LL operation on $lastUpdate[i]$ on line 42 and this successful SC operation on $lastUpdate[i]$, k is received from a* `FAI()` *operation on clk on line 48.*
- *Every successful SC operation on $lastUpdate[i]$ on line 49 sets $lastUpdate[i]$ to $(-, k, -)$ for some positive integer k that is greater than any previous successful SC operation on $lastUpdate[i]$ on line 49.*
- *Every successful SC operation on $lastUpdate[i]$ on line 56 changes $lastUpdate[i]$ from (j, k, v) to $(j, 0, \perp)$ for some integer j, some positive integer k, and some non-$\perp$ value v.*

We now assign every operation on $O[0 \ldots m-1]$ a *timestamp* that roughly approximates the order in which they occur:

▶ **Definition 2.** *For every integer $i \in [0 \ldots m-1]$, we assign every operation on $O[i]$ a timestamp as follows:*

- *For each* `Read()` *operation op_i on $O[i]$, let p be the process that performs op_i and v be the return value of op_i. If (i) op_i is performed when p executes line 44, (ii) p successfully performs an* `SC`$(-, \perp, v)$ *operation on $lastUpdate[i]$ when it next executes line 45, and (iii) the next successful SC operation on $lastUpdate[i]$ changes it to $(-, k, v)$ for some positive integer k, then the timestamp of op_i is this positive integer k.*
- *For each remaining operation op_i on $O[i]$, let op_i' be the earliest operation such that op_i' has a timestamp and op_i precedes op_i'. If op_i' exists, then the timestamp of op_i is the timestamp of op_i'; otherwise the timestamp of op_i is ∞.*

Thus by Observation 1 and Definition 2:

▶ **Observation 3.** *For every operation op_i on $O[i]$:*

- *If op_i precedes another operation op_i' on $O[i]$, then the timestamp of op_i cannot be greater than the timestamp of op_i'.*
- *If the timestamp of op_i is a positive integer k, then k is received from a* `FAI()` *operation on clk after op_i is performed on $O[i]$.*

We now define a completion H' of H, for which we will find a linearization.

▶ **Definition 4.** *Let H' be a completion of H such that:*

- *For each incomplete* `Invoke`$(i, \texttt{op}_i)$ *operation op that has performed op_i on $O[i]$ on line 37 such that the timestamp of op_i is a positive integer (not ∞), op is completed with the same return value as op_i.*
- *All other incomplete operations are removed.*

The following lemma will help us prove that each $\texttt{Invoke}(i, op_i)$ operation can be linearized between its invocation and response, because the timestamp of op_i is received during that interval. We will need it later in Definition 8, where we associate that timestamp with the $\texttt{Invoke}(i, op_i)$ operation.

▶ **Lemma 5.** *For each* $\texttt{Invoke}(i, \texttt{op}_i)$ *operation op in* H'*, op has performed* op_i *on* $O[i]$ *and the timestamp of* op_i *is a positive integer* k *that is received from a* `FAI()` *operation on clk between* $(inv(op), rsp(op))$ *(or simply after* $inv(op)$ *if op is incomplete in* H*).*

Observation 6 below describes some important structural properties of the timestamps and the last bit stored in $lastScan[p][m]$.

▶ **Observation 6.** *From the algorithm, it is clear that for every process* $p \in [0 \ldots n-1]$*:*

- *At any time* t*, there is a non-negative integer* k *and a value* $v \in \{0, 1\}$ *such that* $lastScan[p][m] = (k, 0, 0, 0, v)$*.*
- *Every successful SC operation on* $lastScan[p][m]$ *on line 3 changes* $lastScan[p][m]$ *from* $(k, 0, 0, 0, 0)$ *to* $(k, 0, 0, 0, 1)$*, for some non-negative integer* k*.*
- *Every successful SC operation on* $lastScan[p][m]$ *on line 10 changes* $lastScan[p][m]$ *from* $(k, 0, 0, 0, 1)$ *to* $(k', 0, 0, 0, 0)$ *for some positive integer* $k' > k$ *such that* k' *was previously received from a* `FAI()` *operation on clk on line 9.*
- *Only process* p *can set* $lastScan[p][m]$ *to* $(-, 0, 0, 0, 1)$*, and only on line 3.*

The following lemma will help us associate each `Click()` operation with a timestamp (see also Definition 8 below), which will then help us determine the linearization order.

▶ **Lemma 7.** *For every process* $p \in [0 \ldots n-1]$*:*

1. *For every complete* $\texttt{HelpScan}(p)$ *operation hs, there is a time* t *between* $(inv(hs), rsp(hs))$ *such that* $lastScan[p][m]$ *contains* $(-, 0, 0, 0, 0)$ *at time* t*.*
2. *For each* `Click()` *operation op in* H' *invoked by process* p*, there is a non-negative integer* k *such that op finds that* $lastScan[p][m] = (k, 0, 0, 0, 0)$ *on line 2 and then successfully changes* $lastScan[p][m]$ *to* $(k, 0, 0, 0, 1)$ *on line 3.*
3. *For each* `Click()` *operation op in* H' *invoked by process* p*, there is a positive integer* k_{op} *such that* $lastScan[p][m] = (k_{op}, 0, 0, 0, 0)$ *at time* $rsp(op)$ *and at some time* $t \in (inv(op), rsp(op))$*, some process performs a* `FAI()` *operation on clk on line 9 that returns* k_{op}*.*

We now assign every operation in H' an integer called its *timestamp*. These timestamps roughly approximate the order in which the operations occur, and so they are useful for constructing a linearization of H'.

▶ **Definition 8.** *We assign every operation in* H' *an integer* timestamp *as follows:*

- *For each* $\texttt{Invoke}(i, \texttt{op}_i)$ *operation op in* H'*, the timestamp of op in* H' *is the timestamp of* op_i *on* $O[i]$*. Note that by Lemma 5, this timestamp is a positive integer* k *such that* k *is received from a* `FAI()` *operation on clk between* $(inv(op), rsp(op))$ *(or simply after* $inv(op)$ *if op is incomplete in* H*).*
- *If op is a* `Click()` *operation by a process* p*, then by Lemma 7, there is a positive integer* k_{op} *such that at time* $rsp(op)$*,* $lastScan[p][m] = (k_{op}, 0, 0, 0, 0)$*. The timestamp of op is this positive integer* k_{op}*.*
- *If op is an* $\texttt{Observe}(i)$ *operation by a process* p*, then the timestamp of op is the same as the timestamp of the last* `Click()` *operation by process* p *that precedes op; or* 0 *if no such* `Click()` *operation exists.*

Next, we define a linearization L of H'.

▶ **Definition 9.** *Let L be a linearization of H' such that:*

- *Each* `Invoke(`i, $\textit{op}_i$`)` *operation with timestamp k is linearized ϵ infinitesimals before the time when a* `FAI()` *operation that returns k is applied on clk, where ϵ is the number of operations on $O[i]$ between op_i and the last operation on $O[i]$ with timestamp k.*
- *Each* `Click()` *operation with timestamp k is linearized at the time when a* `FAI()` *operation that returns k is applied on clk.*
- *Each* `Observe(`i`)` *operation is linearized at the end of its interval.*

▶ **Lemma 10.** *Each operation in H' has a unique, well-defined linearization point in L that is within its execution interval.*

The next lemma shows that the order of `Invoke(`i, $\textit{op}_i$`)` operations in the linearization, L, is consistent with the order of op_i operations on $O[i]$.

▶ **Lemma 11.** *Let op be an* `Invoke(`i, $\textit{op}_i$`)` *operation in H', and op' be an* `Invoke(`i, $\textit{op}'_i$`)` *operation in H'. Then op precedes op' in L if and only if op_i precedes op'_i on $O[i]$.*

▶ **Observation 12.** *For every integer $i \in [0 \dots m-1]$ and every integer $j \in [0 \dots \Delta - 1]$, if some process p changes $R[j][i]$ from some value $(j', -, -)$ to some value (j'', k, v) at some time t, then:*

- *p does so on line 52.*
- *p found that $lastUpdate[i] = (j'', k, v)$ when it last executed line 42.*
- *$j' < j''$ and $j'' \bmod \Delta = j$.*
- *j'' is a positive integer, k is a positive integer and v is a non-$\perp$ value.*

The following two lemmas describe some structural properties of arrays R and $lastUpdate$, which will be useful for the linearization proof.

▶ **Lemma 13.** *For every integer $i \in [0 \dots m-1]$, every positive integer j, every positive integer k, and every non-$\perp$ value v, if $lastUpdate[i]$ is set to (j, k, v) at some time t, let t' be the earliest time when a process executes line 52 after finding that $lastUpdate[i]$ contains (j, k, v) on line 42. Then at any time t_R, $R[j \bmod \Delta][i]$ is changed to (j, k, v) if and only if t' exists and $t' = t_R$.*

▶ **Lemma 14.** *For every integer $i \in [0 \dots m-1]$, every positive integer k_r, every integer $j \in [0 \dots \Delta - 1]$, every value j_r, and every value v_r, if $R[j][i]$ is set to (j_r, k_r, v_r) at some time t then j_r is a positive integer, $j_r \bmod \Delta = j$, $v_r \neq \perp$, and $lastUpdate[i] = (j_r, k_r, v_r)$ at time t.*

The next observation will describe how array $lastScan[0 \dots n-1]$ can change.

▶ **Observation 15.** *From the algorithm, it is clear that for every process $p \in [0 \dots n-1]$ and every integer $i \in [0 \dots m-1]$:*

1. *$lastScan[p][i]$ can only be modified on line 34.*
2. *Let k be a positive integer, and t be the earliest time when $lastScan[p][i]$ contains $(k, -, -, -, -)$. Then at time t, $lastScan[p][i]$ contains $(k, maxKey, j_{left}, j_{right}, \perp)$ such that $maxKey \geq k$ and $dist(j_{left}, j_{right}) = \Delta - 1$. Furthermore, before time t, (i) there is a time when $lastScan[p][m]$ is set to $(k, 0, 0, 0, -)$, (ii) there is a time when $R[j_{right}][i]$ is set to $(-, maxKey, -)$, and (iii) there is no time when $lastScan[p][i]$ contains $(k', -, -, -, -)$ such that $k' \geq k$.*

3. *Let k be a non-negative integer and v be a non-$\perp$ value. Then $lastScan[p][i]$ cannot be changed from $(k, -, -, -, v)$ to $(k', -, -, -, -)$ for any non-negative integer $k' \le k$.*
4. *At any time t, if $lastScan[p][i]$ is changed from $(k, maxKey, j_{left}, j_{right}, \perp)$ to $(k, maxKey', j'_{left}, j'_{right}, v')$, then (i) $maxKey' = maxKey$, (ii) $dist(j'_{left}, j'_{right}) = \lceil (dist(j_{left}, j_{right}) + 1)/2 \rceil - 1$, and (iii) $v' = \perp$ if and only if $dist(j'_{left}, j'_{right}) \ne 0$.*

We will now define successful `HelpObserve()` operations, which manage to update $lastScan[0 \dots n-1]$.

▶ **Definition 16.** *For each* `HelpObserve(`q, i`)` *operation ho, we say that ho is successful if and only if ho performs a successful SC operation on $lastScan[q][i]$ on line 34.*

The next lemma shows that each successful `HelpObserve()` operation corresponds to a `Click()` operation after which the helped `Observe()` operation can linearize.

▶ **Lemma 17.** *For every process $p \in [0 \dots n-1]$ and every integer $i \in [0 \dots m-1]$, if there is a successful* `HelpObserve(`p, i`)` *operation ho, then p invokes a* `Click()` *operation op in H such that ho executes line 34 after $inv(op)$.*

▶ **Corollary 18.** *For every process $p \in [0 \dots n-1]$ and every integer $i \in [0 \dots m-1]$, $lastScan[p][i]$ can only be changed from its initial value $(0, 0, 0, 0, 0)$ after the invocation of a* `Click()` *operation by p.*

▶ **Lemma 19.** *For every process $p \in [0 \dots n-1]$, every integer $i \in [0 \dots m-1]$, every non-negative integer k, every non-$\perp$ value v, every non-negative integer j_{left} and every non-negative integer j_{right}, if $lastScan[p][m] = (k, 0, 0, 0, -)$ and $lastScan[p][i]$ is set to $(k, -, j_{left}, j_{right}, v)$ at some time t, then $R[j_{left}][i] = (-, k^*, v)$ at some time $t^* \le t$, where k^* is the largest integer such that $k^* \le k$ and there is a time when some entry of $R[0 \dots \Delta - 1][i]$ contains $(-, k^*, -)$.*

The following technical lemma is critical for the linearizability proof; it helps us determine that `Observe()` operations follow the corresponding `Click()` operation.

▶ **Lemma 20.** *For every process $p \in [0 \dots n-1]$ and every integer $i \in [0 \dots m-1]$, if op is an* `Observe(`i`)` *operation with response v by p in H' and a positive integer k is the timestamp of op, then (i) $lastScan[p][i] = (k, -, -, -, v)$ at some time before $rsp(op)$, and (ii) some entry of $R[0 \dots \Delta - 1][i]$ contains $(-, k', -)$ for some integer $k' \ge k$ at some time before $rsp(op)$.*

▶ **Lemma 21.** *Linearization L of H' respects the specification of the adaptive RMWable snapshot object.*

Proof. Suppose, for contradiction, that the linearization L of H' does not respect the specification of the adaptive RMWable snapshot object. Let *op* be the operation with the earliest linearization in L such that *op* violates the specification of the adaptive RMWable snapshot object, i.e., the return value of *op* differs from what *op* would have returned in a sequential history corresponding to L.

First assume that *op* is an `Invoke(`i, op_i`)` operation. From the algorithm, for each operation op'_i performed on $O[i]$, either op'_i is a `Read()` operation, or op'_i is performed on $O[i]$ by an `Invoke(`i, op'_i`)` operation in H on line 37. By Definition 2 and Definition 4, every `Invoke(`$i, -$`)` operation that is in H but not in H' does not perform any operation on $O[i]$ that precedes op_i. Furthermore, by Lemma 11, for each `Invoke(`i, op'_i`)` operation op' in H', op' precedes *op* in L if and only if op'_i precedes op_i on $O[i]$. Consequently, since `Read()` operations cannot change the state of $O[i]$, the return value of *op* cannot violate the specification of the adaptive RMWable snapshot object. This is a contradiction.

Now assume that op is an `Observe(`i`)` operation invoked by some process p. Let k_{op} be the timestamp of op.

First, consider the case where $k_{op} = 0$. Then by Definition 8, p does not invoke any `Click()` operation before op. Then according to the specification of the adaptive RMWable snapshot object, the `Observe(`i`)` operation op by p should return 0, the initial state of component i. Furthermore, by Corollary 18, $lastScan[p][i]$ always contains its initial value $(0, 0, 0, 0, 0)$ before $rsp(op)$.

So consider the first `HelpObserve(`p, i`)` operation ho called by op on line 13. Since $lastScan[p][i]$ always contains its initial value $(0, 0, 0, 0, 0)$ before $rsp(op)$, ho finds that $lastScan[p][i] = (0, 0, 0, 0, 0)$ on line 16. Then since there is no `Click()` operation by p before $rsp(op)$, by Observation 6 ho finds that $lastScan[p][m] = (0, 0, 0, 0, 0)$. So ho evaluates the conditional on line 18 as **false**, then evaluates the conditional on line 27 as **true**. Thus ho returns $v = 0 \neq \perp$ to op, and so op returns 0 on line 14 – contradicting that op violates the specification of the adaptive RMWable snapshot object.

So it remains to consider the case where $k_{op} > 0$. Then by Definition 8, p invokes `Click()` operation(s) before op, and the last `Click()` operation op' by p before op also has timestamp k_{op}. Thus by the definition of L, op' is linearized at the time that the `FAI()` operation that returns k_{op} is applied on clk.

Let v_{op} be the response value of op. Then by Lemma 20, (i) $lastScan[p][i] = (k_{op}, -, j_{left}, j_{right}, v_{op})$ at some time before $rsp(op)$, for some values j_{left} and j_{right}, and (ii) some entry of $R[0 \ldots \Delta-1][i]$ contains $(-, k', -)$ for some integer $k' \geq k_{op}$ at some time before $rsp(op)$. Let $t < rsp(op)$ be the time when $lastScan[p][i]$ is set to $(k_{op}, -, j_{left}, j_{right}, v_{op})$. Then by Observation 15(2), $lastScan[p][m]$ is set to $(k_{op}, 0, 0, 0, -)$ at some time $t_m < t$.

By Definition 8, since k_{op} is the timestamp of the `Click()` operation op' by p, $lastScan[p][m] = (k_{op}, 0, 0, 0, 0)$ at time $rsp(op')$. So by Observation 6, since op' is the last `Click()` operation by p before the `Observe(`i`)` operation op, $lastScan[p][m] = (k_{op}, 0, 0, 0, 0)$ between $(rsp(op'), rsp(op))$. Thus $t_m < rsp(op')$, and by Observation 6, $lastScan[p][m] = (k_{op}, 0, 0, 0, 0)$ between $(t_m, rsp(op))$. So since $t_m < t < rsp(op)$, $lastScan[p][m] = (k_{op}, 0, 0, 0, 0)$ at time t. Thus by Lemma 19 $R[j_{left}][i] = (-, k^*, v_{op})$ at some time $t^* \leq t$, where k^* is the largest integer such that $k^* \leq k_{op}$ and there is a time when some entry of $R[0 \ldots \Delta-1][i]$ contains $(-, k^*, -)$.

Therefore, there is no integer $\hat{k}$ such that $k^* < \hat{k} \leq k_{op}$ and there is a time when some entry of $R[0 \ldots \Delta-1][i]$ contains $(-, \hat{k}, -)$. Now recall that some entry of $R[0 \ldots \Delta-1][i]$ contains $(-, k', -)$ for some integer $k' \geq k_{op}$ at some time before $rsp(op)$. So by Observation 1 and Lemma 13, from the algorithm it is clear that there is no integer $\hat{k}$ such that $k^* < \hat{k} \leq k_{op}$ and there is a time when $lastUpdate[i]$ contains $(-, \hat{k}, -)$. Thus by Definition 2 and Definition 8, there is no integer $\hat{k}$ such that $k^* < \hat{k} \leq k_{op}$ and some `Invoke(`$i, -$`)` operation in H' has timestamp $\hat{k}$. Now there are two cases: either $k^* = 0$, or $k^* > 0$.

First, consider the case where $k^* = 0$. Then, by the definition of L there are no `Invoke(`$i, -$`)` operations in H' linearized before the `Click()` operation op' by p. Thus according to the specification of the adaptive RMWable snapshot object, the `Observe(`i`)` operation op by p should return 0, the initial state of component i. Furthermore, since $k^* = 0$, by Observation 12 $R[j_{left}][i]$ still contains its initial value $(0, 0, 0)$ at time t^*, and so the `Observe(`i`)` operation op returns $v_{op} = 0$ – contradicting that op violates the specification of the adaptive RMWable snapshot object.

Now it remains to consider the case where $k^* > 0$. Let $\hat{t}^* \leq t^*$ be the time when $R[j_{left}][i]$ is set to $(-, k^*, v_{op})$. Then let j^* be an integer such that at time $\hat{t}^*$, $R[j_{left}][i]$ is set to (j^*, k^*, v_{op}). Then by Lemma 14, $lastUpdate[i] = (j^*, k^*, v_{op})$ at time $\hat{t}^*$. Thus by Observation 1, there is a process q such that (i) q performs a `Read()` operation op_i on $O[i]$ that returns v_{op} on line 44, (ii) q successfully performs an `SC(`$j^*, \perp, v_{op}$`)` operation on $lastUpdate[i]$ when it next executes line 45, and (iii) the next successful SC operation on $lastUpdate[i]$ changes it to (j^*, k^*, v_{op}). So by Definition 2, this `Read()` operation op_i on $O[i]$ has timestamp k^*.

Now recall that there is no integer $\hat{k}$ such that $k^* < \hat{k} \leq k_{op}$ and some `Invoke(`$i, -$`)` operation in H' has timestamp $\hat{k}$. So by the definition of L, every `Invoke(`$i, -$`)` operation in H' is linearized before the `Observe(`i`)` operation op if and only if its timestamp is at most k^*. Thus by Definition 2 and Definition 8, every `Invoke(`$i, -$`)` operation in H' that is linearized before op executes line 37 before the `Read()` operation op_i on $O[i]$ with timestamp k^*.

By Definition 2 and Definition 4, every `Invoke(`$i, -$`)` operation that is in H but not in H' does not perform any operation on $O[i]$ that precedes op_i. Furthermore, by Lemma 11, all `Invoke(`$i, -$`)` operations in H' are linearized by the order in which they execute line 37. Consequently, according to the specification of the adaptive RMWable snapshot object, the `Observe(`i`)` operation op by p should have the same response value as the `Read()` operation op_i on $O[i]$. Finally, recall that the response value of the `Read()` operation op_i on $O[i]$ is v_{op}, the response value of op – contradicting that op violates the specification of the adaptive RMWable snapshot object. Thus, we have shown that op is not an `Observe()` operation.

Since op is neither an `Invoke()` nor an `Observe()` operation, it must be a `Click()` operation. Thus the `Click()` operation op returns **done** on line 5 – contradicting that op violates the specification of the adaptive RMWable snapshot object. ◀

Consequently, this algorithm implements a linearizable adaptive RMWable snapshot object.

References

1. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. of the ACM*, 40(4):873–890, 1993. `doi:10.1145/153724.153741`.
2. Thomas Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1:6–16, 1990. `doi:10.1109/71.80120`.
3. James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. Limited-use atomic snapshots with polylogarithmic step complexity. *J. of the ACM*, 62(1):1–22, 2015. `doi:10.1145/2732263`.
4. James Aspnes and Keren Censor-Hillel. Atomic snapshots in $O(\log^3 n)$ steps using randomized helping. In *Proc. of 27th DISC*, pages 254–268, 2013. `doi:10.1007/978-3-642-41527-2_18`.
5. Hagit Attiya, Rachid Guerraoui, and Eric Ruppert. Partial snapshot objects. In Friedhelm Meyer auf der Heide and Nir Shavit, editors, *Proc. of 20th SPAA*, pages 336–343, 2008. `doi:10.1145/1378533.1378591`.
6. Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Efficient atomic snapshots using lattice agreement. In *Proc. of 6th WDAG*, pages 35–53, 1992.
7. Benyamin Bashari and Philipp Woelfel. An efficient adaptive partial snapshot implementation. In *Proc. of the 2021 ACM PODC*, pages 545–555, 2021. `doi:10.1145/3465084.3467939`.
8. Michiko Inoue and Wei Chen. Linear-time snapshot using multi-writer multi-reader registers. In *Proc. of the 8th WDAG*, pages 130–140, 1994. `doi:10.1007/BFb0020429`.
9. Prasad Jayanti. f-arrays: Implementation and applications. In *Proc. of 21st PODC*, pages 270–279, 2002. `doi:10.1145/571825.571875`.
10. Prasad Jayanti, Siddhartha Jayanti, and Sucharita Jayanti. MemSnap: A fast adaptive snapshot algorithm for RMWable shared-memory. In *Proc. of 43rd PODC*, pages 25–35, 2024. `doi:10.1145/3662158.3662820`.

11 Prasad Jayanti and Srdjan Petrovic. Efficient and practical constructions of ll/sc variables. In *Proc. of 22nd PODC*, pages 285–294, 2003. `doi:10.1145/872035.872078`.

12 Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. on Comp.*, 30(2):438–456, 2000. `doi:10.1137/S0097539797317299`.

13 MirzaBaig, Danny Hendler, Alessia Milani, and Corentin Travers. Long-lived snapshots with polylogarithmic amortized step complexity. In *Proc. of the 2020 ACM PODC*, pages 31–40, 2020. `doi:10.1145/3382734.3406005`.

14 Robert Obryk. Write-and-f-array: implementation and an application. Master's thesis, Jagiellonian University, 2013.

15 Yaron Riany, Nir Shavit, and Dan Touitou. Towards a practical snapshot algorithm. *Theor. Comp. Sci.*, 269(1-2):163–201, 2001. `doi:10.1016/S0304-3975(00)00412-6`.

16 Yuanhao Wei, Naama Ben-David, Guy Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *Proc. of 26th PPOPP*, pages 31–46, 2021. `doi:10.1145/3437801.3441602`.

A Additional Proofs

This appendix contains some of the proofs omitted from Section 4.

In order to prove Lemma 5, we use the following statement, which describes how $\mathit{lastUpdate}[i]$ is affected by a complete `HelpUpdate(`i`)` operation.

▶ **Lemma 22.** *For each complete* `HelpUpdate(`i`)` *operation hu, there is a positive integer j such that:*

- *There are at least 6 successful SC operations on $\mathit{lastUpdate}[i]$ that occur between $(\mathit{inv}(\mathit{hu}), \mathit{rsp}(\mathit{hu}))$.*
- *Some process reads a non-$\perp$ value v from $O[i]$ on line 44 at some time $t_0 \in (\mathit{inv}(\mathit{hu}), \mathit{rsp}(\mathit{hu}))$, then successfully performs an* `SC(`$j, \perp, v$`)` *operation on $\mathit{lastUpdate}[i]$ when it next executes line 45 at some time $t_1 \in (t_0, \mathit{rsp}(\mathit{hu}))$.*
- *Some process (not necessarily distinct from the first) receives a positive integer k from a* `FAI()` *operation on clk on line 48 at some time $t_2 \in (t_1, \mathit{rsp}(\mathit{hu}))$, then performs the next successful SC operation on $\mathit{lastUpdate}[i]$ when it next executes line 49 at some time $t_3 \in (t_2, \mathit{rsp}(\mathit{hu}))$, which changes $\mathit{lastUpdate}[i]$ from $(j, \perp, v)$ to (j, k, v).*
- *Some process (not necessarily distinct from the first two) performs the next successful SC operation on $\mathit{lastUpdate}[i]$ on line 56 at some time $t_4 \in (t_3, \mathit{rsp}(\mathit{hu}))$, which changes $\mathit{lastUpdate}[i]$ from (j, k, v) to $(j, 0, \perp)$.*

Proof. From the algorithm, it is clear that in every outermost loop iteration of *hu*, *hu* performs an LL operation on $\mathit{lastUpdate}[i]$ on line 42, then performs an SC operation on $\mathit{lastUpdate}[i]$ (line 45, 49, or 56). So a successful SC operation on $\mathit{lastUpdate}[i]$ occurs within each loop iteration. Thus there are at least 6 successful SC operations on $\mathit{lastUpdate}[i]$ that occur between $(\mathit{inv}(\mathit{hu}), \mathit{rsp}(\mathit{hu}))$. Consequently, by Observation 1:

- There is a positive integer j and non-$\perp$ value v such that the second, third, or fourth successful SC operation on $\mathit{lastUpdate}[i]$ within $(\mathit{inv}(\mathit{hu}), \mathit{rsp}(\mathit{hu}))$ changes $\mathit{lastUpdate}[i]$ from $(j-1, 0, \perp)$ to $(j, \perp, v)$ at some time $t_1 \in (\mathit{inv}(\mathit{hu}), \mathit{rsp}(\mathit{hu}))$.
- The process that does this successful SC operation on $\mathit{lastUpdate}[i]$ at time t_1 reads v from $O[i]$ on line 44 at some time $t_0 \in (\mathit{inv}(\mathit{hu}), t_1)$.
- There is a positive integer k such that the next successful SC operation on $\mathit{lastUpdate}[i]$ within $(\mathit{inv}(\mathit{hu}), \mathit{rsp}(\mathit{hu}))$ changes $\mathit{lastUpdate}[i]$ from $(j, \perp, v)$ to (j, k, v) at some time $t_3 \in (t_1, \mathit{rsp}(\mathit{hu}))$.

- The process that does this successful SC operation on $lastUpdate[i]$ at time t_3 receives this positive integer k from a `FAI()` operation on clk on line 48 at some time $t_2 \in (t_1, t_3)$.
- The next successful SC operation on $lastUpdate[i]$ within $(inv(hu), rsp(hu))$ changes $lastUpdate[i]$ from (j, k, v) to $(j, 0, \perp)$ at some time $t_4 \in (t_3, rsp(hu))$.

Thus, since $inv(hu) < t_0 < t_1 < t_2 < t_3 < t_4 < rsp(hu)$, the lemma holds. ◀

Proof of Lemma 5. First, consider the case where op is incomplete in H. Then by Definition 4, op has performed op_i on $O[i]$ and the timestamp of op_i is a positive integer k. So by Definition 2, there exists a `Read()` operation op'_i on $O[i]$, a process p that performs op'_i, and a value v returned by op'_i such that (i) op'_i is performed when p executes line 44, (ii) p successfully performs an $\texttt{SC}(-, \perp, v)$ operation on $lastUpdate[i]$ when it next executes line 45, (iii) the next successful `SC()` operation on $lastUpdate[i]$ changes it to $(-, k, v)$, and (iv) op_i precedes op'_i on $O[i]$. Thus by Observation 1, this positive integer k is received from a `FAI()` operation on clk after op performs op_i on $O[i]$, which is clearly after $inv(op)$.

It now remains to the consider the case where op is complete in H. Thus op performs op_i on $O[i]$ on line 37, then calls $\texttt{HelpUpdate}(i)$ on line 38. Since op is complete in H, this $\texttt{HelpUpdate}(i)$ call completes before $rsp(op)$. So by Lemma 22, during this complete $\texttt{HelpUpdate}(i)$ call:

- There are at least 6 successful SC operations on $lastUpdate[i]$
- Some process q receives non-$\perp$ value v from a `Read()` operation op'_i on $O[i]$ on line 44, then performs the second, third, or fourth successful SC operation on $lastUpdate[i]$ when it next executes line 45, changing it to $(-, \perp, v)$.
- The next successful SC operation on $lastUpdate[i]$ changes it to $(-, k', v)$ for some positive integer k' that was received from a `FAI()` operation on clk after the successful $\texttt{SC}(-, \perp, v)$ operation on $lastUpdate[i]$ by q.

Thus by Definition 2, the timestamp of op'_i is this positive integer k'. As op_i precedes op'_i, by Definition 2 and Observation 3, the timestamp of op_i is a positive integer $k \leq k'$, which is returned from a `FAI()` operation on clk after op performs op_i on $O[i]$ on line 37. Hence, as $k' \geq k$ is received from a `FAI()` operation on clk during the complete $\texttt{HelpUpdate}(i)$ call of op, k is received from a `FAI()` operation on clk during $(inv(op), rsp(op))$. ◀

Proof of Lemma 7. (1): Let hs be a complete $\texttt{HelpScan}(p)$ operation. By Observation 6, there is a value $v \in \{0, 1\}$ such that hs finds that $lastScan[p][m]$ contains $(-, 0, 0, 0, v)$ on line 7. If $v = 0$, we are done. So suppose $v = 1$. Then hs evaluates the conditional on line 8 as **true**, gets a positive integer k from a `FAI()` operation on clk on line 9, and then performs an $\texttt{SC}(k, \mathit{0}, \mathit{0}, \mathit{0}, \mathit{0})$ operation on $lastScan[p][m]$ on line 10.

Let t_0 and t_1 be the times when hs executes lines 7 and 10 respectively. Note that $inv(hs) < t_0 < t_1 < rsp(hs)$. Then, since hs performs an LL operation on $lastScan[p][m]$ at time $t_0 > inv(hs)$, and an SC operation on $lastScan[p][m]$ at time $t_1 < rsp(hs)$, there must exist a successful SC operation on $lastScan[p][m]$ between $(inv(hs), rsp(hs))$. By Observation 6, every successful SC operation on $lastScan[p][m]$ changes it either from $(-, 0, 0, 0, 0)$ to $(-, 0, 0, 0, 1)$ or from $(-, 0, 0, 0, 1)$ to $(-, 0, 0, 0, 0)$. So there is a time t between $(inv(hs), rsp(hs))$ such that $lastScan[p]$ contains $(-, 0, 0, 0, 0)$ at time t. Thus we have proven (1).

(2): Initially, $lastScan[p][m] = (0, 0, 0, 0, 0)$. By Observation 6:

- Only process p can set $lastScan[p][m]$ to $(-, 0, 0, 0, 1)$, and only on line 3.
- Every successful SC operation on $lastScan[p][m]$ changes it either from $(-, 0, 0, 0, 0)$ to $(-, 0, 0, 0, 1)$ or from $(-, 0, 0, 0, 1)$ to $(-, 0, 0, 0, 0)$.

Thus only a `Click()` operation by process p can change $lastScan[p][m]$ from $(-, 0, 0, 0, 0)$, and only on line 3. Consequently, every complete `Click()` operation by process p in H:

- Finds that $\mathit{lastScan}[p][m]$ contains $(k, 0, 0, 0, 0)$ on line 2 for some non-negative integer k (Observation 6).
- Successfully changes $\mathit{lastScan}[p][m]$ to $(k, 0, 0, 0, 1)$ on line 3.
- Finishes with a `HelpScan(`p`)` call on line 4, which, since we have already proven (1), ensures that $\mathit{lastScan}[p][m]$ is changed to $(-, 0, 0, 0, 0)$ for the next `Click()` operation by p.

Finally, by Definition 4, every `Click()` operation in H' is complete in H. Thus we have proven (2).

(3): Let op be a `Click()` operation in H' that is invoked by process p. By Definition 4, every `Click()` operation in H' is complete in H, so op is complete in H. Since we have already proven (2), p successfully changes $\mathit{lastScan}[p][m]$ to $(-, 0, 0, 0, 1)$ on line 3 at some time $t_1 > \mathit{inv}(op)$. Then p calls `HelpScan(`p`)` on line 4, which, since we have already proven (1), ensures that some process q (not necessarily distinct from p) sets $\mathit{lastScan}[p][m]$ to $(k_{op}, 0, 0, 0, 0)$ for some value k_{op} at some time $t' < \mathit{rsp}(op)$.

By Observation 6, k_{op} is a positive integer, and at time t', q performs a successful `SC(`$k_{op}, 0, 0, 0, 0$`)` operation on $\mathit{lastScan}[p][m]$ on line 10 within a `HelpScan()` operation. Furthermore, since q performs a successful SC operation on $\mathit{lastScan}[p][m]$ on line 10, q must have performed the matching LL operation on $\mathit{lastScan}[p][m]$ on line 7 after the successful `SC(`$-, 0, 0, 0, 1$`)` on $\mathit{lastScan}[p][m]$ by process p at time t_1. Thus q received k_{op} from a `FAI()` operation on clk on line 9 at some time $t \in (t_1, t')$. Then, since $t_1 > \mathit{inv}(op)$ and $t' < \mathit{rsp}(op)$, $t \in (\mathit{inv}(op), \mathit{rsp}(op))$.

Finally, from the algorithm it is clear that p does not execute line 3 after calling `HelpScan(`p`)` on line 4. So by Observation 6, $\mathit{lastScan}[p][m]$ cannot be changed again before $\mathit{rsp}(op)$, so $\mathit{lastScan}[p][m]$ still contains $(k_{op}, 0, 0, 0, 0)$ at time $\mathit{rsp}(op)$. Thus we have proven (3). ◀

Proof of Lemma 10. This is clearly true for all `Observe(`i`)` operations in H'.

By Lemma 7, for each `Click()` operation op in H' invoked by a process $p \in [0 \ldots n-1]$, there is a positive integer k_{op} such that $\mathit{lastScan}[p][m] = (k_{op}, 0, 0, 0, 0)$ at time $\mathit{rsp}(op)$ and at some time $t \in (\mathit{inv}(op), \mathit{rsp}(op))$, some process performs a `FAI()` operation on clk on line 9 that returns k_{op}. So by Definition 8, this positive integer k_{op} is the timestamp of op. Thus by Definition 9, op is linearized at the time $t \in (\mathit{inv}(op), \mathit{rsp}(op))$ when some process performs a `FAI()` operation on clk on line 9 that returns k_{op}. Consequently, every `Click()` operation in H' has a unique, well-defined linearization point in L that is within its execution interval.

Thus it remains to consider the `Invoke()` operations in H'. Let op be an `Invoke(`i, op_i`)` operation in H', and k be the timestamp of op in H'. Then by Definition 8, k is also the timestamp of op_i on $O[i]$. Then by Lemma 5, k is received from a `FAI()` operation on clk between $(\mathit{inv}(op), \mathit{rsp}(op))$. Then by Definition 9, there is a finite integer ϵ such that op is linearized ϵ infinitesimals before this `FAI()` operation on clk. Consequently, every `Invoke()` operation in H' has a unique, well-defined linearization point in L that is within its execution interval. ◀

Proof of Lemma 11. Suppose op_i precedes op'_i on $O[i]$. Then by Observation 3, the timestamp of op_i on $O[i]$ cannot be greater than the timestamp of op'_i on $O[i]$. So by Definition 9, the `Invoke(`i, op_i`)` operation op precedes the `Invoke(`i, op'_i`)` operation op' in L.

Thus if op_i precedes op'_i on $O[i]$, then op precedes op' in L. By symmetric arguments, if op'_i precedes op_i on $O[i]$, then op' precedes op in L. Consequently, op precedes op' in L if and only if op_i precedes op'_i on $O[i]$. ◀

Proof of Lemma 13. Suppose, for contradiction, that the lemma does not hold. Then let j be the smallest positive integer for which the lemma does not hold.

First, consider the case where t' does not exist, i.e., no process executes line 52 after finding that $\mathit{lastUpdate}[i]$ contains (j, k, v) on line 42. Then by Observation 12, no process ever sets $R[j \bmod \Delta][i]$ to (j, k, v) – contradicting that the lemma does not hold for j.

So it remains to consider the case where t' exists. Then t' is the earliest time when a process p executes line 52 after finding that $\mathit{lastUpdate}[i]$ contains (j, k, v) on line 42. By Observation 1:

- $\mathit{lastUpdate}[i]$ is never set to (j, k', v') for some non-$\perp$ values k' and v' such that $(k', v') \neq (k, v)$.
- $\mathit{lastUpdate}[i]$ is only set to (j, k, v) at time t (and so $t < t'$).
- Before $\mathit{lastUpdate}[i]$ is set to (j, k, v) at time t, $\mathit{lastUpdate}[i]$ never contains $(j', -, -)$ such that $j' > j$.

Thus by Observation 12, before time t', no process ever sets $R[j \bmod \Delta][i]$ to $(j', -, -)$ such that $j' \geq j$. So p finds that $R[j \bmod \Delta][i]$ contains $(j_{LL}, -, -)$ for some integer $j_{LL} < j$ on line 51. Thus at time t', p evaluates the conditional on line 52 as **true**, and performs an `SC(`j, k, v`)` operation on $R[j \bmod \Delta][i]$.

Since the lemma does not hold, this p must fail this `SC(`j, k, v`)` operation on $R[j \bmod \Delta][i]$ at time t'. Thus for some integer $\hat{j}$, at some time $\hat{t}$ that is between the time when p performs the matching LL operation on $R[j \bmod \Delta][i]$ on line 51 and time t', there is a successful `SC(`$\hat{j}, -, -$`)` operation on $R[j \bmod \Delta][i]$. So by Observation 12, $\hat{j}$ is a positive integer. Furthermore, recall that before time t', no process ever sets $R[j \bmod \Delta][i]$ to $(j', -, -)$ such that $j' \geq j$. Thus $\hat{j} < j$.

Now recall that j is the smallest positive integer for which the lemma does not hold. Thus the lemma holds for the positive integer $\hat{j} < j$. So $\hat{t}$ is the earliest time when a process executes line 52 after finding that $\mathit{lastUpdate}[i]$ contains $(\hat{j}, \hat{k}, \hat{v})$ on line 42 for some positive integer $\hat{k}$ and some non-$\perp$ value $\hat{v}$.

By Observation 1, $\mathit{lastUpdate}[i]$ can only be changed from $(\hat{j}, \hat{k}, \hat{v})$ on line 56. Thus from the algorithm, it is clear that at time $\hat{t}$, $\mathit{lastUpdate}[i]$ still contains $(\hat{j}, \hat{k}, \hat{v})$. Consequently, $\mathit{lastUpdate}[i]$ contains $(j, -, -)$ at time t and $(\hat{j}, -, -)$ at time $\hat{t}$ such that $t < \hat{t}$ and $j > \hat{j}$ – contradicting Observation 1. ◀

Proof of Lemma 14. Let q be the process that sets $R[j][i]$ to (j_r, k_r, v_r) at time t. By Observation 12, j_r is a positive integer, $j_r \bmod \Delta = j$, $v_r \neq \perp$, and q does so on line 52, after finding that $\mathit{lastUpdate}[i]$ contains (j_r, k_r, v_r) on line 42. Then, by Lemma 13, t is the earliest time when a process (namely q) executes line 52 after finding that $\mathit{lastUpdate}[i]$ contains (j_r, k_r, v_r) on line 42.

By Observation 1, $\mathit{lastUpdate}[i]$ can only be changed from (j_r, k_r, v_r) on line 56. Thus from the algorithm, it is clear that at time t, $\mathit{lastUpdate}[i]$ still contains (j_r, k_r, v_r). ◀

Proof of Lemma 17. Let ho be the successful `HelpObserve(`p, i`)` operation that executes line 34 earliest. Then let t be the time when ho executes line 34. By Observation 15(1) and Definition 16, $\mathit{lastScan}[p][i]$ can only be changed on line 34, within a successful `HelpObserve(`p, i`)` operation. So at time t, $\mathit{lastScan}[p][i]$ is changed from its initial value $(0, 0, 0, 0, 0)$. Thus by Observation 15(3), there is a positive integer $k > 0$ such that at time t, $\mathit{lastScan}[p][i]$ is changed to $(k, -, -, -, -)$. So by Observation 15(2), before time t, there is a time when $\mathit{lastScan}[p][m]$ is set to $(k, 0, 0, 0, -)$. By Observation 6, $\mathit{lastScan}[p][m]$ can only be changed from its initial value $(0, 0, 0, 0, 0)$ on line 3, within a `Click()` operation by process p. Thus p invokes a `Click()` operation op in H such that ho executes line 34 after $\mathit{inv}(\mathit{op})$. ◀

▶ **Lemma 23.** *For every process $p \in [0 \dots n-1]$, every integer $i \in [0 \dots m-1]$, and every positive integer k, if lastScan$[p][i]$ is first set to $(k, -, -, -, -)$ at some time t_i, then clk returns k to a* `FAI()` *operation at some time $t < t_i$ and between (t, t_i), $R[0 \dots \Delta-1][i]$ is modified at most $n+2$ times.*

The proof is omitted due to space restrictions.

Given any two integers j and j' in $[0 \dots \Delta-1]$, we define $dist(j, j')$ to be $j'-j$, if $j' \geq j$, and $j'-j+\Delta$, otherwise. Note that if $j \neq j'$, then $dist(j, j') = \Delta - dist(j', j)$.

Proof of Lemma 19. First, consider the case where $k = 0$. Then by Observation 15(3), $lastScan[p][i]$ still contains its initial value $(0,0,0,0,0)$. Then, since $R[0][i]$ initially contains $(0,0,0)$, it is clear that the lemma holds.

So it remains to consider the case where k is a positive integer. Let t_i be the earliest time when when $lastScan[p][i] = (k, -, -, -, -)$. Since $lastScan[p][i]$ initially contains $(0,0,0,0,0)$, t_i exists and $t_i < t$. By Observation 15(2), at time t_i, $lastScan[p][i] = (k, maxKey, j'_{left}, j'_{right}, \perp)$, such that $maxKey \geq k$, $dist(j'_{left}, j'_{right}) = \Delta - 1$, and before time t_i, there is a time when $lastScan[p][m] = (k, 0, 0, 0, -)$ and a time when $R[j'_{right}][i] = (-, maxKey, -)$.

▶ **Subclaim 23.1.** *For each complete* `HelpObserve`(p,i) *operation ho such that $t_i < inv(ho) < rsp(ho) < t$, there is a successful* `HelpObserve`(p,i) *operation (not necessarily distinct from ho) that executes line 34 between $(inv(ho), rsp(ho))$.*

Proof. Consider *ho*:

- Since $t_i < inv(ho) < rsp(ho) < t$, by Observation 15 *ho* finds that $lastScan[p][i] = (k, -, -, -, \perp)$ on line 16.
- Since $lastScan[p][m] = (k, 0, 0, 0, -)$ at some time before time t_i and $lastScan[p][m] = (k, 0, 0, 0, -)$ at time $t > t_i$, by Observation 6, $lastScan[p][m]$ always contains $(k, 0, 0, 0, -)$ between (t_i, t). Thus since $t_i < inv(ho) < rsp(ho) < t$, *ho* finds that $lastScan[p][m] = (k, 0, 0, 0, -)$ on line 17.
- So *ho* evaluates the conditionals on lines 18 and 27 as **false**.
- Thus *ho* performs an SC operation on $lastScan[p][i]$ on line 34.

Consequently, by Definition 16 there exists a successful `HelpObserve`(p,i) operation (not necessarily distinct from *ho*) that executes line 34 between $(inv(ho), rsp(ho))$. ◀

Let t_k be the time when a `FAI()` operation on *clk* returns k. By Lemma 23, t_k exists and $t_k < t_i < t$.

The proofs of the following twwo claims are omitted due to space restrictions.

▶ **Subclaim 23.2.** *$R[0 \dots \Delta-1][i]$ is modified $O(n \log \Delta/\Delta')$ times between (t_k, t).*

▶ **Subclaim 23.3.** *There is an integer $j^* \in [0 \dots \Delta-1]$ such that:*

1. *$R[j'_{right}][i]$ always contains $(-, maxKey, -)$ between (t_i, t).*
2. *$R[j^*][i]$ always contains $(-, k^*, -)$ between (t_i, t).*
3. *For every integer $j \in [0 \dots \Delta-1]$ such that $dist(j'_{left}, j) \leq dist(j'_{left}, j^*)$, at any time $\hat{t}$ such that $t_i \leq \hat{t} \leq t$, $R[j][i] = (-, \hat{k}, -)$ for some integer $\hat{k}$ such that either $\hat{k} < k$ or $\hat{k} > maxKey$.*
4. *For every integer $j \in [0 \dots \Delta-1]$ such that $dist(j'_{left}, j) > dist(j'_{left}, j^*)$, at any time $\hat{t}$ such that $t_i \leq \hat{t} \leq t$, $R[j][i] = (-, \hat{k}, -)$ for some integer $\hat{k}$ such that $\hat{k} > k$ and $\hat{k} \leq maxKey$.*

Now consider each successful `HelpObserve`(p,i) operation ho' that sets $lastScan[p][i]$ to $(k, -, -, -, -)$ on line 34 after time t_i. Recall that t_i is the earliest time when $lastScan[p][i]$ contains $(k, -, -, -, -)$, t is the time when $lastScan[p][i]$ is set to $(k, -, j_{left}, j_{right}, v)$, and v is a non-$\perp$ value. So by Observation 15, from the algorithm it is clear that ho' executes line 16 after time t_i, and executes line 34 before or at time t. Furthermore, by Observation 15, there are integers j_1, j_2, j_3, and j_4 such that ho' changes $lastScan[p][i]$ from $(k, -, j_1, j_2, -)$ to $(k, -, j_3, j_4, -)$ on line 34. Thus from the algorithm it is clear that ho' evaluates the conditionals on lines 18 and 27 as **false**. Therefore ho':

- Finds that $lastScan[p][i] = (k, -, j_1, j_2, -)$ on line 16.
- Evaluates the conditional on line 18 as **false**.
- Finds that $R[j][i] = (-, k_r, -)$ on line 30, where j is an integer such that $dist(j_1, j) \leq dist(j_1, j_2)$.
- By Subclaim 23.3, evaluates the conditional on line 31 as **true** if and only if $dist(j'_{left}, j) > dist(j'_{left}, j^*)$.

Consequently $dist(j'_{left}, j_1) \leq dist(j'_{left}, j_3) \leq dist(j'_{left}, j^*) \leq dist(j'_{left}, j_4) \leq dist(j'_{left}, j_2) \leq dist(j'_{left}, j'_{right})$.

Finally, by Observation 15(1) and Definition 16, some successful `HelpObserve(`p, i`)` operation ho_t sets $lastScan[p][i]$ to $(k, -, j_{left}, j_{right}, v)$ on line 34 at time $t > t_i$. So, since $v \neq \bot$, by Observation 15(4), $j_{left} = j_{right} = j^*$. Thus ho_t finds that $R[j^*][i] = (-, -, v)$ on line 33 at some time between (t_i, t). Therefore by Subclaim 23.3, $R[j_{left}][i] = (-, k^*, v)$ at some time $t^* \leq t$, where k^* is the largest integer such that $k^* \leq k$ and there is a time when some entry of $R[0 \ldots \Delta - 1][i]$ contains $(-, k^*, -)$. ◀

Proof of Lemma 20. By Definition 8, k is also the timestamp of the last `Click()` operation op' by p that precedes op in H', and at $rsp(op')$, $lastScan[p][m] = (k, -, -, -, -)$. Note that by Observation 6, $lastScan[p][m]$ always contains $(k, -, -, -, -)$ between $(rsp(op'), rsp(op))$. Since the `Observe(`i`)` operation op returns v, op calls a `HelpObserve(`p, i`)` operation ho that returns v on line 13, and $v \neq \bot$.

Consider this `HelpObserve(`p, i`)` operation ho. On line 16, ho finds that $lastScan[p][i] = (k_i, -, -, -, v')$. Since $lastScan[p][m]$ always contains $(k, -, -, -, -)$ between $(rsp(op'), rsp(op))$, ho finds that $lastScan[p][m] = (k, -, -, -, -)$ on line 17. Then since ho returns $v \neq \bot$, from the algorithm it is clear that ho evaluates the conditional on line 18 as **false**, and so $k \leq k_i$. So by Observation 15(2), $k = k_i$. Finally, since ho does not return $\bot$, ho returns v' on line 27. Thus $v' = v$, and so ho found that $lastScan[p][i] = (k, -, -, -, v)$ on line 16.

Next, let k' be a value such that ho found that $lastScan[p][i] = (k, k', -, -, v)$ on line 16. Furthermore, let t be the earliest time when $lastScan[p][i]$ contains $(k, -, -, -, -)$. Then by Observation 15(4), $lastScan[p][i]$ contains $(k, k', -, -, -)$ at time t. Consequently, by Observation 15(2), $k' \geq k$ and before time t, there is a time when some entry of $R[0 \ldots \Delta - 1][i]$ is set to $(-, k', -)$. ◀