Adaptive Partial Snapshots with Logarithmic Step Complexity

Benyamin Bashari
benyamin.bashari@ucalgary.ca
University of Calgary
Calgary, Canada

Philipp Woelfel
woelfel@ucalgary.ca
University of Calgary
Calgary, Canada

ABSTRACT
The standard single-writer snapshot type allows processes to obtain a consistent snapshot of an array of $n$ memory locations, each of which can be updated by one of $n$ processes. In almost all algorithms, a Scan() operation returns a linearizable snapshot of the entire array. Under realistic assumptions, where hardware registers do not have the capacity to store many array entries, this inherently leads to a step complexity of $\Omega(n)$.

In this paper, we consider an alternative version of the snapshot type, where a Scan() operation stores a consistent snapshot of all $n$ memory locations, but does not return anything. Instead, a process can later observe the value of any component of that snapshot using a separate Observe() operation. This allows us to implement the type from fetch-and-increment and compare-and-swap objects, such that Scan() operations have constant step complexity and Observe() and Update() operations have step complexity $O(\log n)$.

KEYWORDS
Shared Memory; Concurrency; Snapshot; Predecessor Object; Asynchrony

ACM Reference Format:

1 INTRODUCTION
Taking linearizable snapshots of memory is one of the most fundamental and best studied problems in the area of concurrent shared memory algorithms. The problem can be described abstractly in terms of a snapshot type, which maintains an array of $m$ memory locations, $A[0 \ldots m-1]$. It supports the operations Update(i, x), which changes the value of the $A[i]$ to $x$, and Scan(), which returns the vector $(A[0], \ldots, A[m-1])$. The first linearizable snapshot implementations have been known since the 1990’s [1, 4].

Most research focuses on single-writer snapshot implementations, where $m$ is equal to the number of processes, $n$, and each array index is associated with a unique process, which is the only one that is allowed to update the corresponding array entry. Atiya, Herlihy, and Rachman [9] showed that implementing single-writer snapshots reduces to solving lattice agreement. They presented a lattice agreement algorithm from registers that then yields a snapshot algorithm with subquadratic (in $n$) worst-case step complexity. By devising a faster lattice agreement algorithm, Inoue and Chen [14] obtained a single-writer snapshot algorithm with linear step complexity. It follows from a proof technique by Jayanti, Tan, and Toueg [17] that this is optimal. Aspnes, Atiya, Censor-Hillel, and Ellen [6] proved that it is possible to break through this barrier as long as the number of operations on the object is polynomial in $n$. They implemented a deterministic algorithm with worst-case step complexity of $O(\log^3 n)$. Subsequently, Aspnes and Censor-Hillel, gave a randomized algorithm with poly-logarithmic expected step complexity [7], and Ahad Baig, Hendler, Milani, and Travers devised a deterministic algorithm with poly-logarithmic amortized worst-case step complexity [10].

Generally, snapshot implementations from registers are not practical: Almost all of them assume that a single hardware register can store the entire snapshot array. The only exceptions we are aware of have either exponential running time [4], or do not permit concurrent Scan() operations [18]. Therefore, snapshot implementations from stronger primitives have been considered: Using single-word compare-and-swap (CAS) and fetch-and-increment (FAI) objects, Riany, Shavit, and Touitou [19] devised a single-writer algorithm with step complexity $O(1)$ for Update() and $O(n)$ for Scan(). Jayanti [15] generalizes this to the multi-writer case, and achieves $O(m)$ step complexity for Scan() operations, even without relying on FAI.

It is not hard to see that the Scan() operation of any snapshot object implemented from single-word objects must have step complexity $\Omega(m)$. Therefore, the standard snapshot specification leads to inherently inefficient Scan() operations for large values of $m$, and in particular in the common case of single-writer snapshots, where $m = n$. The specification does not allow for a faster performance, even if a process is only interested in the value of some but not all components of the snapshot object.

1.1 Result
To improve upon this inherent complexity barrier, we implement a natural extension of the single-writer snapshot: A Scan() operation takes a snapshot of the array $A[0 \ldots n-1]$, but does not return anything. Instead, after taking a snapshot, a process can observe each individual component $A[i]$ of that snapshot by calling Observe(i). (Hence, the entire snapshot can be obtained using $n$ Observe() calls.) Using single-word registers, CAS, and FAI objects, we obtain an implementation, where Scan() has step-complexity $O(1)$, and Update() and Observe() have step complexity $O(\log n)$.

Since our algorithm uses sequence numbers, only up to $2^{W-1} - 1$ operations can be executed, if each base object can store $W$ bits. This is not a restriction that has any practical impact on current 64-, or even 32-bit architectures.

Note that our snapshot algorithm is more flexible than a standard snapshot: It can simulate “full” single-writer standard snapshots (with a logarithmic performance penalty over the best known algorithms [15, 19]), but can be much more efficient, if not all components of the snapshot need to be observed. Moreover, which components a process wants to observe can be decided adaptively.
after a process has taken a snapshot. For example, if a snapshot represents a data structure, the search path through the data structure can depend on the actual values found. The fact that Scan() operations have constant step complexity can be useful when a process would have to take a snapshot before it even knows for certain that it will need the snapshot later. This could, for example, be the case in applications that require backups or error recovery.

1.2 Related Solutions
Attiya, Guerraoui, and Ruppert [8] defined a partial snapshot type, where a process can choose to scan only some of the \( m \) array components. This can be considered an oblivious version of our specification, because processes must decide at the beginning of a Scan() operation, which memory locations they are interested in. The authors provide a multi-writer algorithm from registers, CAS, and FAI objects, in which scanning \( r \) array components takes \( O(r^2) \) steps in the worst-case. While Update() operations are not bounded wait-free, their amortized step complexity is bounded by the maximum interval contention, as well as the maximum number of components accessed by Scan() operations.

A more flexible multi-writer snapshot specification was proposed by Wei, Ben-David, Blelloch, Fatourou, Ruppert and Sun [21]. Their object allows a process to take a snapshot of multiple CAS objects, and returns a handle to that snapshot. Using that handle, a process can later determine the value of any of those CAS objects at the point in time the snapshot was taken. They gave an implementation of that type from CAS objects, where each CAS() and snapshot operation takes constant number of steps. However, the step complexity of reading a single snapshotted value of a memory location grows linearly with the number of updates that may have occurred on that location, since the corresponding snapshot was taken. Hence, their algorithm is not bounded wait-free.

The authors use a version list for each CAS object, that stores the complete history of updates performed on the object. Each update is associated with a global sequence number, which is also stored in the version list. The sequence number is incremented with each Scan() operation and is returned as the handle. It can then be used to identify the latest update that was applied to a CAS object, using the version list.

Instead of version lists, our algorithm uses single-writer predecessor objects, which we implement from sequential persistent red-black trees [12]. This allows processes to find the latest value that was written to a component of the snapshot object in logarithmic time, with respect to the total number of nodes stored in the red-black tree. We then devise a method for pruning the data structure from outdated values, in order to keep the size polynomial in \( n \).

The idea of associating data structure modifications with timestamps has also been used in algorithms for software transactional memory [20] and multi-version databases [22]. These algorithms are either not wait-free or allow operations to fail. Moreover, using an FAI object as a global clock to linearize operations at the time of FAI() operations is used in [5] to achieve a general method to add range queries to data structures.

As far as we know, our algorithm is the only solution to the partial snapshot problem, where snapshots of \( o(n/\log n) \) components can be obtained in \( o(n) \) time. A comparison of our implementation with the partial snapshot objects of [8, 21] is shown in Table 1.

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. We assume that each process’s ID is stored in a process-local variable \( myID \). Invocation and response of an operation \( op \) are denoted \( inv(op) \) and \( rsp(op) \), respectively.

The following base objects are relevant for our work: A read-write register supports two operations, \( \text{Write}(v) \), which changes its value to \( v \) and returns nothing, and \( \text{Read()} \), which returns the value of the register. An LL/SC object provides two operations, \( \text{LL()} \) and \( \text{SC()} \). An LL() operation returns the object’s value, and an SC() operation 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 true if it succeeds to update the object’s value, and returns false otherwise. A FAI object stores an integer, initially 1, and provides an operation \( \text{FAI()} \), which increments the object’s value by 1 and returns the value before the increment. A CAS object provides an operation \( \text{CAS(old, new)} \). If the value of the object is \( \text{old} \), this operation updates the value to \( \text{new} \) and returns true, otherwise, the object remains unchanged and the operation returns false.

Let \( D_{\text{val}} \) be the domain of values stored in a snapshot component. We assume that the system provides atomic registers, FAI, and CAS objects of word size \( W \geq \max\{\log |D_{\text{val}}|, 3\log n + c\} \) bits for a sufficiently large constant \( c \). In order to avoid having to deal with ABAs, we will use LL/SC objects instead of CAS objects. It is trivial to replace those LL/SC objects with CAS objects and sequence numbers. Alternatively, one can use the efficient construction of LL/SC from single-word CAS by Jayanti and Petrovic [16], which has \( O(1) \) step complexity. Due to [11] we may assume w.l.o.g. that LL/SC objects have arbitrary large word size \( O(W) \).

Generally, we measure efficiency of wait-free shared memory algorithms in terms of the number of shared memory steps executed. Thus, step complexity ignores local computation time, which does not include shared memory steps. Time complexity, on the other hand, measures local and shared memory steps of processes (assuming a standard word-RAM model).

As a building block for our main algorithm, we use the destination array of Blelloch and Wei [11]. A destination array stores a sequence of \( n \) values, and supports the operations \( \text{Read()} \) and \( \text{Copy()} \). Operation \( \text{Read}(i) \) takes as argument an integer \( i \in \{1, \ldots, n\} \), and returns the value of the \( i \)-th component of the array. Operation \( \text{Copy}(R) \) takes as argument a reference \( R \) to a register, and if process \( p \) calls that operation, it changes the value of the \( p \)-th component of the array to the value of register \( R \). Blelloch and Wei [11] show that a linearizable and wait-free destination array can be implemented from \( O(n^2) \) single-word CAS objects and registers in such a way that each \( \text{Read()} \) and each \( \text{Copy()} \) operation can be executed in a constant number of steps.

Specification of the New Algorithms. In this paper we present linearizable single-writer predecessor and adaptive partial snapshots objects. The former is used as a building block for the latter, but
may be of independent interest. In the following we provide the sequential specifications of the underlying types.

The predecessor type maintains a set of pairs, each comprising a key and a value. The domain of keys must be totally ordered.

The predecessor type provides four operations. An Insert\( (k, v) \) operation inserts a pair with key \( k \) and value \( v \), provided that the data structure does not contain a pair with key \( k \). In that case, the Insert\( (\cdot) \) operation succeeds and returns true, otherwise, it fails and returns false. Operations Remove\( (k) \), Pred\( (k) \), and Succ\( (k) \) take as a single argument a key \( k \). Each of them fails and returns false if the data structure contains no pair with key \( k \). Otherwise, Remove\( (k) \) removes the pair with key \( k \) from the data structure, and Pred\( (k) \) and Succ\( (k) \) return the pairs with greatest key smaller than \( k \) and smallest key larger or equal than \( k \), respectively, if such pairs exist. If not, these operations fail and return false. We call operations Insert\( (\cdot) \) and Remove\( (\cdot) \) update operations, and all other operations query operations.

A predecessor object is single-writer, if there is only one dedicated process that is allowed to perform updates. We will consider single-writer predecessor objects with bounded capacity \( \Delta \), which informally means that at most \( \Delta \) elements can be stored in the data structure at any point. Since the object is single-writer, it is uniquely determined at the point of invocation of an update operation, whether that operation will be successful or not. (We call an incomplete update operation successful, if it must be successful in any extension of the execution in which it completes.) Bounded capacity \( \Delta \) formally means that at any point in time the number of invocations of successful Insert\( (\cdot) \) operations minus the number of responses of successful Remove\( (\cdot) \) operations is at most \( \Delta \).

The adaptive partial snapshots type stores an \( n \)-component array that supports three update operations Update\( (v) \), Scan\( (\cdot) \), and Observe\( (k) \). An Update\( (v) \) operation called by process \( p \) changes the value of the \( p \)-th component to \( v \), and returns nothing. Method Scan\( (\cdot) \) does not return anything, and its behaviour is only defined in terms of method Observe\( (k) \). A process is only allowed to call an Observe\( (k) \) operation after it has performed at least one Scan\( (\cdot) \) call, and an Observe\( (k) \) call by process \( p \) returns the value that the \( k \)-th component of the object had at the point of \( p \)’s latest preceding Scan\( (\cdot) \) operation.

---

3 SINGLE-WRITER PREDECESSOR ALGORITHM

In this section we present our linearizable and wait-free single-writer implementation of the predecessor type from registers and CAS objects. First, we will show that the sequential balanced red-black tree of [12] can be used in a concurrent system (i.e., is linearizable), as long as there is only one process that performs update operations. This concurrent red-black tree almost immediately yields a single-writer predecessor object: it natively supports the operations Insert\( (\cdot) \) and Remove\( (\cdot) \), and adding operations Pred\( (\cdot) \) and Succ\( (\cdot) \) is straightforward. However, in infinite executions, this algorithm may need an unbounded number of nodes. In Section 3.2 we will use memory reclamation to bound the space of our concurrent data structure.

3.1 The Basic Algorithm

Driscoll, Sarnak, Sleator, and Tarjan [12], present a technique called node-copying to make linked data structures persistent. We first describe a basic version of this technique that can be applied to any binary search tree (BST) implementations, where each node store only pointers to its children (i.e., there are no parent pointers).

A dedicated variable \( R \) stores a pointer to the root \( r \) of the tree. An update operation does not modify any nodes of the data structure. Instead, it adds copies of all nodes that need to be modified, as well as a new root \( r’ \), and finally changes the pointer \( R \) so that it points to \( r’ \) instead of \( r \). To be more precise, suppose the set of nodes reachable from \( r \) form a conventional BST \( T_1 \). Let \( T_2 \) be the BST obtained by applying a conventional update operation to \( T_1 \) (e.g., an insertion). Let \( S \) be the set of nodes in \( T_2 \), that are either added or modified by this update operation, and \( S’ \) the parent-closure of \( S \) (i.e., if \( u \) is in \( S’ \), then the parent of \( u \) is also in \( S’ \)). Instead of modifying the nodes in \( S \), we create a copy \( u’ \) of each node \( u \) in \( S’ \). Each field of \( u’ \) has the same value as the corresponding field in \( u \), except that a pointer to a node \( u \) in \( S’ \) is replaced with a pointer to the copy \( u’ \) of that node. Since \( S’ \) is parent-closed, the root \( r \) of \( S’ \) is also copied into a node \( r’ \). It is easy to see that the nodes reachable from \( r’ \) now form a red-black tree that is equivalent to \( T_2 \). Hence, to complete the update operation, it suffices to replace the value of \( R \) with a pointer to \( r’ \).

To perform a query operation on the persistent data structure, a process simply reads the pointer \( R \) to obtain a root \( r \), and then performs the same operations as it would in the conventional BST algorithm, using \( r \) as a root.

---

\(^1\)We believe that in [8] unbounded sequence numbers can be avoided by using LL/SC instead of CAS.
For the purpose of concreteness, we will now consider a red-black tree [13]. Driscoll, Sarnak, Sleator, and Tarjan have applied the node copying technique described above to that data structure to obtain a persistent red-black tree, where each of the operations Insert(), Remove() and Find() take O(\log m) steps [12], where \( m \) is the number of elements stored in the tree. It is straightforward to augment the data structure with query operations Pred() and Succ() so that all operations have time complexity \( O(\log m) \). Thus, we obtain an implementation of a persistent sequential predecessor type with the same asymptotic time complexity.

We can now use that persistent predecessor implementation in a shared memory system, by storing \( R \) and each node of the data structure in an atomic register. We will allow only one process, \( p_w \), to perform update operations, and all processes are allowed to execute query operations. Observe that if at some point \( t \) pointer \( R \) points to a root \( r \) in the data structure, all nodes reachable from \( r \) form the BST that was obtained as a result of the update operation that wrote the pointer to \( r \) into \( R \). None of these nodes can change after point \( t \). Hence, if a query operation reads the pointer to \( r \) from \( R \), then it will visit exactly the same nodes that would be visited in the sequential case. Similarly, an update operation by process \( p_w \) does initially not make changes to any reachable nodes, and all tree modifications will become visible to other processes only when \( p_w \) changes the root pointer, \( R \), to point to the new root copy \( p_w \) created. Hence, it is easy to see that each update operation can linearize with the write to \( R \), and each query operation can linearize with the read of \( R \). It follows that this object is linearizable, provided that only one process can perform update operations (in fact, it is linearizable as long as no two update operations are concurrent).

### 3.2 Recycling Outdated Nodes

During each update operation on the concurrent red-black tree, described in the previous section, \( p_w \) makes copies of up to \( \Theta(\log m) \) nodes [12], and needs to allocate space for the registers storing them. Thus, in unbounded executions, an infinite number of registers is needed. In the following, we apply a memory reclamation technique to bound the space. We add a \texttt{Recycle()} method to the object, whose purpose is to remove the nodes from the tree that cannot be accessed anymore by any process. The registers storing these nodes’ information can then be reused for future nodes.

We will only consider single-writer predecessor objects with bounded capacity \( \Delta \). In our snapshot application, \( \Delta \leq 3n \).

For the ease of discussion and readability of the pseudocode (see Algorithm 1), we assume that \( p_w \) has access to a method \texttt{Allocate()} which allocates a new node and returns a reference to it, and a method \texttt{Deallocate(x)} which deallocates a node \( x \) that \( p_w \) previously allocated. We will show that at any point in time, there are at most \( \lambda = O(n\Delta \log \Delta) \) nodes that \( p_w \) has allocated but not deallocated. Thus, implementing methods \texttt{Allocate()} and \texttt{Deallocate()} with bounded memory is straightforward, by having \( p_w \) maintain a local pool of \( \lambda \) registers, one for each node. Our algorithm guarantees that when a process accesses a node \( v \), then at that point \( v \) has been allocated but not yet deallocated.

Consider an execution. Let \( r_0 \) be the initial root of the red-black tree pointed to by \( R \), and let \( r_i \) be the root pointed to by \( R \) after the \( i \)-th update operation. Let \texttt{reachable}(ri) denote the set of nodes reachable from \( r_i \). A node \( v \) is outdated at point \( t \), if at that point \( R \) points to a root \( r_k \), and there exists \( i \in \{0, \ldots, k-1\} \) such that \( v \in \text{reachable}(r_i) \setminus \text{reachable}(r_k) \). Outdated nodes are candidates for deallocation. However, it is possible that an outdated node \( v \) may still be accessed by a process \( q \), if \( v \) is reachable from a root \( r \), and \( R \) pointed to \( r \) when \( q \) read that pointer at the beginning of its query operation. To prevent \( v \) from being deallocated in such a situation, process \( q \) will initially protect a root \( r \), when it reads the pointer to \( r \) from \( R \) at the beginning of its query operation. It does so by storing the address of \( r \) in a register \texttt{protectedRoot}[q], where \texttt{protectedRoot} is an array with one entry for each process. We say a node is protected, if it is reachable from a root stored in some array entry \texttt{protectedRoot}[j], \( j \in \{0, \ldots, n-1\} \). The \texttt{Recycle()} method will then deallocate only outdated, but not protected nodes.

Using only registers, it is difficult to protect nodes this way: Process \( q \) may fall asleep immediately after reading the address of the current root \( r \) from \( R \), and then only wake up again when \( r \) has already been deallocated. It is then too late to protect \( r \) by writing it into \texttt{protectedRoot}[q]. Sophisticated techniques to deal with that in constant time and using only registers have been described in [2, 3]. An easier way is to use a destination array for \texttt{protectedRoot}. (The main motivation for the definition of the destination array in [11] has been to solve the same type of “protection” problem.) At the beginning of its query operation, process \( q \) calls \texttt{protectedRoot}[q].\texttt{Copy}(R). This copies the address of the root \( r \) pointed to by \( R \) into \texttt{protectedRoot}[q], and thus protects all nodes reachable from \( r \). As mentioned above, in a \texttt{Recycle()} call, \( p_w \) now only needs to find all outdated nodes (i.e., those that are not reachable from the current root pointed to by \( R \)), and remove those that are not protected.

It remains to show that this can be done without decreasing the worst-case time complexity of update and query operations. For query operations this is trivial, as only one \texttt{Copy()} operation is added.

We now show how to modify update operations, and how to implement the \texttt{Recycle()} method. The updater, process \( p_w \), maintains a (local) list \texttt{outdated} of outdated nodes that have not yet been deallocated. Initially, \texttt{outdated} is an empty list. Suppose that at the end of an update operation by process \( p_w \), the root pointed to by \( R \) changes from \( r \) to \( r' \). Then \( p_w \) computes \texttt{reachable}(\( r' \)) \setminus \texttt{reachable}(\( r \)) and adds all nodes in that set to \texttt{outdated}. Note that each node in \texttt{reachable}(\( r \)) \setminus \texttt{reachable}(\( r' \)) is in the parent closure of the set of nodes that get modified by a standard red-black tree update operation. Since that red-black tree contains at most \( \Delta \) nodes, it is easy to see that

\[
|\texttt{reachable}(\( r \)) \setminus \texttt{reachable}(\( r' \))| \leq c \cdot \log \Delta \tag{1}
\]

for some constant \( c \). Moreover, the set \texttt{reachable}(\( r \)) \setminus \texttt{reachable}(\( r' \)) can be computed in time \( O(\log \Delta) \).

Every \( n \Delta \) update operations, process \( p_w \) begins a new \texttt{Recycle()} operation. The total number of steps required to complete such a \texttt{Recycle()} call will be \( O(n\Delta \log \Delta) \), and with each update operation, \( p_w \) contributes \( O(\log \Delta) \) steps to an ongoing \texttt{Recycle()} call. In a \texttt{Recycle()} call, \( p_w \) first renames \texttt{outdated} to \texttt{outdated'}, and then sets \texttt{outdated'} to an empty list. (This way, future update operations will fill the set \texttt{outdated}, while the \texttt{Recycle()} call can process \texttt{outdated'}. And if, while that \texttt{Recycle()} call is ongoing,
a process \( p \) starts protecting a new node \( v \) by copying its root to \( \text{protectedRoot}(p) \), then \( v \) will not be in \( \text{outdated} \), and thus will not get deallocated. Then, \( p_w \) initializes a local empty list \( \text{protected} \), and reads all roots stored in \( \text{protectedRoot}(i) \) for each \( i \in \{0, \ldots, n-1\} \), and for each such root \( r \), adds \( \text{reachable}(r) \) to \( \text{protected} \). Finally, \( p_w \) deallocates all nodes in \( \text{outdated} \) that are not in \( \text{protected} \), and append list \( \text{protected} \) to \( \text{outdated} \), so that the protected nodes can be reconsidered in the next \( \text{Recycle()} \) operations.

Analysis. At any point in time the predecessor object stores a set of size at most \( \Delta \). Hence, from any root stored in \( \text{protectedRoot}(i) \), at most \( \Delta \) nodes can be reached, and thus

\[ |\text{protected}| \leq \Delta \cdot n, \quad (2) \]

and \( \text{protected} \) can be computed in time \( O(|\text{protected}|) = O(n\Delta) \).

We will now argue that

\[ |\text{outdated}| \leq (c \log \Delta + 1)n\Delta \quad (3) \]

at all times, where \( c \) is the constant from (1). Clearly, this is true initially and at the beginning of each \( \text{Recycle()} \) call, when \( \text{outdated} \) is set to an empty list. In the interval during which a \( \text{Recycle()} \) call completes (or until the first \( \text{Recycle()} \) call is invoked), \( n\Delta \) update operations are executed. Hence, by (1), at most \( cn\Delta \log \Delta \) nodes become outdated. In addition, by (2), at the end of the \( \text{Recycle()} \) call at most \( \Delta n \) nodes are added from \( \text{protected} \) to \( \text{outdated} \). Thus, once the \( \text{Recycle()} \) call terminates, \( |\text{outdated}| \leq \Delta n + cn\Delta \log \Delta \), and thus (3) is true.

By definition, each node in the data structure is either outdated or reachable from the root pointed to by \( R \). Since at most \( \Delta \) nodes are reachable from that node, it follows from (3) that at any point \( p_w \) needs to have only \( \lambda = O(n\Delta \log \Delta) \) nodes allocated. Thus, a pool \( P_\lambda \) of \( \lambda \) registers suffices to store all nodes.

Since \( p_w \) needs only \( \lambda \) nodes in its entire pool (of unallocated and allocated nodes), it is easy to see that \( p_w \) can compute the set difference \( S_1 \setminus S_2 \) of two sets \( S_1, S_2 \subseteq P_\lambda \) (given as linked lists) in time \( O(\lambda) = O(n\Delta \log \Delta) \) using the standard lookup-table technique.

In particular, at the end of its \( \text{Recycle()} \) method, process \( p_w \) can compute all nodes that are in \( \text{outdated} \) but not in \( \text{protected} \) in time \( O(n\Delta \log \Delta) \).

To summarize, a complete execution of the \( \text{Recycle()} \) method takes \( O(n\Delta \log \Delta) \) time. In each update operation, process \( p_w \) computes the set \( \text{of}(\log \Delta) \) nodes that have become unreachable (see (1)) in \( O(\log \Delta) \) time. Then it contributes sufficiently many steps towards a (new or ongoing) \( \text{Recycle()} \) operation, so that the \( \text{Recycle()} \) method completes during \( n\Delta \) update operations by \( p_w \). Since the \( \text{Recycle()} \) method takes \( O(n\Delta \log \Delta) \) time, there is a constant \( \kappa \), such that \( p_w \) needs to contribute at most \( \kappa \Delta \log \Delta \) steps to the \( \text{Recycle()} \) method during each update. Hence, each update operation takes time \( O(\log \Delta) \), and thus it also comprises only \( O(\log \Delta) \) shared memory steps.

As discussed, the algorithm needs to store \( \lambda = O(n\Delta \log \Delta) \) nodes in registers. In addition it requires a destination array of size \( n \), which can be implemented from \( O(n^2) \) registers and CAS objects (see Section 2).

\textbf{Theorem 1.} A wait-free linearizable single-writer predecessor object with bounded capacity \( \Delta \) can be implemented from \( O(n\Delta \log \Delta + n^2) \) single-word CAS objects and registers, such that each update and query operation has time and step complexity \( O(\log \Delta) \).

A full proof is omitted due to space constraints.

### Algorithm 1: Single-Writer Predecessor Algorithm

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Update()}</td>
<td>Let ( r ) be the root pointed to by ( R ). Perform the corresponding update operation on ( R ) as in the sequential implementation in [12], allocating new nodes using method Allocate().</td>
</tr>
<tr>
<td>\text{Query()}</td>
<td>Let ( r' ) be the new root pointed to by ( R ). Every ( n\Delta ) update calls, start a new ( \text{Recycle()} ) method. Contribute ( \kappa \Delta ) steps towards an ongoing ( \text{Recycle()} ) call, where ( \kappa ) is a sufficiently large constant.</td>
</tr>
<tr>
<td>\text{Recycle()}</td>
<td>Rename outdated to ( \text{outdated} ). Let ( \text{outdated} ) and ( \text{protected} ) be new empty lists. For each ( i \in {0, \ldots, n-1} ) do ( r \leftarrow \text{protectedRoot}.\text{Read}(i) ). If ( r \neq \bot ) then ( \text{protected} \leftarrow \text{protected} \cup \text{reachable}(r) ). For each node ( x \in \text{outdated} ), such that ( x \notin \text{protected} ), perform ( \text{Deallocate}(x) ). Append protected to ( \text{outdated} ).</td>
</tr>
</tbody>
</table>

### ADAPTIVE PARTIAL SNAPSHOT ALGORITHM

#### 4.1 The Basic Algorithm

In this section we present a basic linearizable implementation of the adaptive partial snapshot type, see Algorithm 2. The algorithm uses, for each process \( p \), a predecessor object \( \text{versions}[p] \). Each \( \text{Update()} \) by process \( p \) adds an element to \( \text{versions}[p] \), and the time complexity of \( \text{Update()} \) and \( \text{Observe()} \) operations is dominated by operations on the predecessor objects. This basic algorithm never removes elements from the predecessor objects. In order to obtain our desired space and step complexity bounds, we will later show how we can prune outdated elements from the predecessor object. This will allow us to use a predecessor object of bounded capacity \( \Delta = 3n \).

Algorithm 2 uses a shared FAI object \( \text{clk} \) in addition to the predecessor objects \( \text{versions}[p] \) for each predecessor. The value of \( \text{clk} \) is incremented at least once for each \( \text{Update()} \) and \( \text{Scan()} \) operation. Each \( \text{Update()} \) and \( \text{Scan()} \) operation is associated with exactly one such increment (even though multiple ones can happen during
such an operation execution), and the operation linearizes at the point of that increment. I.e., each \texttt{Update()} and \texttt{Scan()} operation is associated with a unique value \( x \), where \( clk \) is incremented from \( x \) to \( x + 1 \) at the linearization point of that operation.

We will first discuss a very simple high-level idea, which leads to an incorrect algorithm, and then show how to fix it in order to arrive at Algorithm 2. In an \texttt{Update}(val) operation, the calling process, \( p \), increments the FAI object \( clk \) from \( x \) to \( x + 1 \), and then inserts the pair \((x, \text{val})\) into \texttt{versions}[p]. In a \texttt{Scan()} operation, a process \( q \) determines the current value of \( clk \) (also by executing an \texttt{FAI}()), and stores it in \texttt{lastScan}[q]. (For this basic algorithm, \texttt{lastScan}[q] can be a local variable. But for the advanced algorithm we will need it to be an LL/SC object.) In a later \texttt{Observe()} operation, \( q \) can then determine the last value \( \text{val} \) that was written to component \( p \) of the array, by calling \texttt{versions}[p].\texttt{Pred}(sTime), where \( sTime \) is the value of \texttt{lastScan}[q]. This call returns the pair \((x, \text{val})\) stored in the predecessor object, such that \( x \) is the largest key less than or equal to \( sTime \).

However, this simple approach is not linearizable: Suppose that during its \texttt{Update}(val), process \( p \) executes \texttt{FAI}(), but then before it inserts the pair \((x, \text{val})\) into its predecessor object \texttt{versions}[p], some process \( q \) performs a \texttt{Scan()} operation, during which it fetches \( x' \geq x \) from \( clk \). After that, \( q \) calls \texttt{Observe}(p), and determines the predecessor of \( x' \) in \texttt{versions}[p]. Whether or not \( q \) observes the value \text{val} depends on when \( p \) inserts the pair \((x, \text{val})\) into \texttt{versions}[p]. Thus, the relative order of \( p \)'s \texttt{Update}(\text{val}) and \( q \)'s \texttt{Scan()} would now have to be determined by the execution of \( q \)'s \texttt{Observe}(p) call relative to \( p \)'s insert of \((x, \text{val})\) into the predecessor object. In particular, \( q \) may perform two subsequent \texttt{Observe}(p) calls that return different values, which is incorrect.

To deal with that problem, we let any process calling \texttt{Observe}(p) help process \( p \) linearize a possible pending \texttt{Update()} operation. This is facilitated by method \texttt{HelpUpdate(\text{pID})} and an auxiliary LL/SC object \texttt{lastUpdate}[p] for each process \( p \). To indicate that process \( p \) has started performing \texttt{Update}(\text{val}), it writes the pair \((\bot, \text{val})\) into that variable (using a pair of \texttt{LL()} and \texttt{SC()} operations). A process \( q \) that wants to observe the value of \( p \)'s array component following a \texttt{Scan()} can retrieve that value either from the predecessor object \texttt{versions}[p], or, if \texttt{versions}[p] is outdated, from \texttt{lastUpdate}[p]. (How a process determines if the value is outdated, will be explained in the low level description.)

But \( p \)'s \texttt{Update}(\text{val}) may not yet have been linearized, if \( p \) has not yet incremented \( clk \). Therefore, during its \texttt{Observe()} operation, \( q \) increments \( clk \) itself from \( x \) to \( x + 1 \) and tries to communicate that to other processes by trying to store the pair \((x, \text{val})\) into \texttt{lastUpdate}[p], using an \texttt{SC()} operation. If successful, then \( p \)'s \texttt{Update}(\text{val}) operation can linearize with \( q \)'s increment of \( clk \). And otherwise, some other process has already helped \( p \)'s \texttt{Update()} linearize.

4.2 Pruning Predecessor Objects
In Algorithm 2, the number of pairs stored in the predecessor objects increases with the number of \texttt{Update()} operations, and thus needs predecessor objects with unbounded capacity.

In order to be able to use a predecessor objects with bounded capacity, we remove unnecessary elements. The corresponding pseudo-code is shown in Algorithm 3. We will need an auxiliary method \texttt{HelpScan(q)}. Its implementation guarantees that if at the invocation of a \texttt{HelpScan(q)} call a \texttt{Scan()} call by some process \( q \) has linearized, then by the time the \texttt{HelpScan(q)} call responds, the value of \( clk \) that is associated with the \texttt{Scan()} operation will have been written to \texttt{lastScan}[q]. This helping mechanism is implemented in essentially the same way as that of \texttt{HelpUpdate(\text{pID})}.

We now describe how a process \( p \) can remove pairs from its predecessor object, \texttt{versions}[p], which are not needed anymore. To that end, \( p \) computes a set of \textit{required keys}, which will not remove from \texttt{versions}[p]. Consider an arbitrary point \( t \) in the execution. A pair \((k, \text{v})\) in \texttt{versions}[p] can be safely removed after point \( t \), if no \texttt{Observe}(p) operation that responds after \( t \) needs to return \((k, \text{v})\). Conversely, a key is required, if it may at any point after \( t \) be the predecessor of some value that is then stored in \texttt{lastScan}[q] of some process \( q \).

Process \( p \) can compute the set of required keys as follows: First, at some point \( t_0 \) it determines the largest key, \texttt{maxKey}, stored in \texttt{versions}[p]. \texttt{Observe} that then \( clk > \texttt{maxKey} \) at that point, and also at any later point. Next, process \( p \) calls \texttt{HelpScan(q)} for each process \( q \), and then reads a value \texttt{vq} from \texttt{lastScan}[q]. Let \( V \) be the set of values \texttt{vq} ≠ \( \bot \), for each process \( q \). It is not hard to see that at any point after \( p \) has read \texttt{vq} from \texttt{lastScan}[q], the value of that object is either still \texttt{vq}, or \( \bot \) or an integer of value at least \( clk \).
Adaptive Partial Snapshots with Logarithmic Step Complexity

Function \texttt{Update}(v)
- See Algorithm 2
- Perform $O(\log n)$ steps of the \texttt{Prune()} method.

Function \texttt{HelpUpdate}(pID)
- See Algorithm 2
- \texttt{Observe(pID)}

Function \texttt{Scan()}
- \texttt{lastScan[myID].LL()} (if $i \in \{0,\ldots,n-1\}$ do)

Function \texttt{HelpScan(pID)}
- \texttt{stime ← lastScan[pID].LL()}
- \texttt{nlTime ← clk.FAI()}
- \texttt{lastScan[pID].SC(nlTime)}

Function \texttt{Prune()}
- $S = \emptyset$
- $maxKey ← versions[myID].Pred(\infty).key$
- for $i \in \{0,\ldots,n-1\}$ do
- \texttt{HelpScan()}
- \texttt{v ← lastScan[i].LL()}
- if $v = \infty$ then
- \texttt{S ← S ∪ versions[myID].Pred(v)}
- \texttt{cur ← versions[myID].Succ(−∞)}
- while \texttt{cur.key < maxKey do}
- \texttt{nxt ← versions[myID].Succ(cur.key)}
- if \texttt{cur.key ≠ S then}
- \texttt{versions[myID].Remove(cur.key)}
- \texttt{cur ← nxt}

Algorithm 3: Modifications for Pruning

Since $clk > maxKey$ at any point after $t_0$, it suffices if the set of required keys contains at least all predecessors of values larger than $maxKey$, and of the keys in $V$. Let $S$ be the set of all predecessors of values in $V$ that are in $versions[p]$. Since $maxKey$ is in $versions[p]$, the predecessors of values larger than $maxKey$ have a value of at least $maxKey$. Hence, $p$ determines as the set of required keys as the set of keys $k$, where $k \in S$ or $k ≥ maxKey$.

To prune its predecessor object, a process $p$ can call method \texttt{Prune()}. In lines 62-67 of that method, $p$ computes the value $maxKey$ and the set $S$ exactly as described above. Then, as indicated in the last line of the method, $p$ removes all keys from $versions[p]$ that are not in $S$ and that are smaller than $maxKey$.

Clearly, the step complexity of method \texttt{Prune()} is super-linear in $n$. But we can distribute the total work of a \texttt{Prune()} call over $n \texttt{Update()}$ calls. We maintain the invariant that $p$’s predecessor object contains at most $3n$ pairs at any point in time, and thus we can use a predecessor object with bounded capacity $\Delta = 3n$. Once the predecessor object contains $2n$ elements, process $p$ begins distributing the total work of a single \texttt{Prune()} call over its next $n \texttt{Update()}$ operations. This way, $p$ can remove at least $n$ elements from $versions[p]$, while $n$ new elements are added.

We will now argue that the total amount of work of each \texttt{Prune()} call is $O(n \log n)$. Thus, to preserve the $O(\log n)$ step complexity for all snapshot operations, it suffices if during each \texttt{Update()}, $p$ devotes $O(\log n)$ steps to this recycling method.

Let $K$ be the set of keys with value at most $maxKey$ that are in $versions[p]$ at any point after $t_0$, which is when $p$ determines $maxKey$. Clearly, no such keys are added to this set after $t_0$, so $|K| = O(n)$. Process $p$ can find all keys in $K$ by first determining the successor of $-1$, which is the smallest such key, and then following the chain of successors, until $maxKey$ is reached. Since the step complexity of computing a successor in $versions[p]$ is $O(\log n)$, the total work complexity of $O(n \log n)$ for \texttt{Prune()} follows.

4.3 Linearizability Proof

In the following we will prove that any execution $E$ is linearizable. Since our algorithm is wait-free, we assume w.l.o.g. that in $E$ all operations complete. (If they don’t we can construct a completion of the execution, by letting processes with pending operations take steps in an arbitrary order, until their operations have completed.)

In Section 4.3.1, we define a mapping $lin$ that maps each operation $op$ in $E$ to a point in time. Then, in Section 4.3.2, we show that for every \texttt{Update()}, \texttt{Scan()} and \texttt{Observe()} operation, $inv_{nTime} ≤ lin_{郫} ≤ rsp_{郫}$. In Sections 4.3.3 and 4.3.4 we prove that if we order all operations by $lin$, the resulting sequential history is valid (w.r.t. the specification of the adaptive partial snapshot type).

Let $p$ be a process, $up$ an \texttt{Update()} operation by $p$ and $sc$ a \texttt{Scan()} operation by $p$. Let $t(up)$ be the point in time at which process $p$ performs $lastUpdate[p].SC$ in line 22 of $up$. Similarly, let $t(sc)$ be the point in time at which process $p$ performs $lastScan[p].SC$ in line 53 of $sc$. If $lastUpdate[p]$ changes in $(t(up), \infty)$, then we let $t'(up)$ be the first point when this happens in that interval, and otherwise we let $t'(up) = \infty$. Similarly, if $lastScan[p]$ changes in $(t(sc), \infty)$, $t'(sc)$ denotes the first point when this happens in that interval, and otherwise $t'(sc) = \infty$.

4.3.1 Linearization Points. Below we will prove two lemmas that facilitate the definition of mapping $lin$. Lemma 2 shows that during each \texttt{Update()} operation by process $p$, $lastUpdate[p]$ changes exactly once from $\perp$ to a positive integer fetched from $clk$. (It is possible that $p$ changes $lastUpdate[p]$ when it calls \texttt{HelpUpdate()} during its own \texttt{Update()}, or some other process may do that during a \texttt{HelpUpdate()} call.) Lemma 3 proves a symmetric statement for $lastScan[p]$. Due to space constraints, proofs of these two lemmas are omitted in this extended abstract.

Lemma 2. Let $p$ be a process that executes first an \texttt{Update}(y) operation up and later an \texttt{Update}(y') operation up', and assume $p$ executes no \texttt{Update()} in-between up and up'. Then

(a) at point $t(up)$ process $p$ changes $lastUpdate[p].key$ to $\perp$, (b) $t'(up)$ is before the point at which process $p$ reads $lastUpdate[p]$ in line 24 of up, (c) at point $t'(up)$ some process performs a successful $lastUpdate[p].SC(x,y)$ operation in line 30 of a \texttt{HelpUpdate}(p) call, for some $x \in \mathbb{N}$, and (d) $lastUpdate[p]$ remains unchanged throughout $(t'(up), t(up'))$.

Lemma 3. Let $p$ be a process that executes first a \texttt{Scan()} operation $sc$ and later a \texttt{Scan()} operation $sc'$, and no \texttt{Scan()} operation between $sc$ and $sc'$.

(a) Process $p$ changes $lastScan[p]$ to $\perp$ at $t(sc)$,
when

Then, $\text{help}(\text{update}(p))$ changes due to a shared memory operation executed at some point $t'(\text{update}(p))$ in a $\text{HelpUpdate}(p)$ operation. If $p$ executes a $\text{Scan}(\cdot)$ operation, then by Lemma 3(c), $\text{lastScan}[p]$ changes at $t'(\text{scan}(p))$ during a $\text{HelpUpdate}(p)$. We call these $\text{HelpUpdate}(p)$ and $\text{HelpScan}(p)$ operations $\text{help}(\text{up})$ and $\text{help}(\text{sc})$, respectively.

Let $op$ be an $\text{Observe}(\cdot)$, $\text{Scan}(\cdot)$ or $\text{Update}(\cdot)$ operation. Recall that we consider only complete executions. If $op$ is an $\text{Observe}(\cdot)$ operation, then we define $\text{lin}(op) = \text{rsp}(op)$. If it is an $\text{Update}(\cdot)$ or $\text{Scan}(\cdot)$, then $\text{lin}(op)$ is the point at which $\text{clk}$ is increased. $\text{Claim 6}$ below states that the linearization order of $\text{update}(\cdot)$ and $\text{scan}(\cdot)$ operations $op$ matches the order of integers $\text{val}(op)$.

**Claim 6.** Let each of $\text{op}_1$ and $\text{op}_2$ be an $\text{Update}(\cdot)$ or $\text{Scan}(\cdot)$ operation. Then, $\text{lin}(\text{op}_1) < \text{lin}(\text{op}_2)$ if and only if $\text{val}(\text{op}_1) < \text{val}(\text{op}_2)$.

We now show that during an $\text{Update}(x)$ operation $op$ by process $p$, the pair $(\text{val}(op), x)$ gets inserted into $\text{versions}[p]$.

**Claim 7.** If process $p$ completes an $\text{Update}(x)$ operation $op$, then in line 25 of $op$ it inserts $(\text{val}(up), x)$ into $\text{versions}[p]$.

**Proof.** Let $t_1$ be the point in time at which process $p$ reads $\text{lastUpdate}[p]$ in line 24 of $up$. By Lemma 2(b), $t'(up) < t_1$. Also, by Lemma 2(b), $\text{lastUpdate}[p].\text{key}$ does not change throughout $(t'(up), \text{rsp}(up))$. Therefore, the value assigned to $u\text{Time}$ in line 24 of $up$ is $\text{val}(up)$. So process $p$ inserts $(\text{val}(up), x)$ into $\text{versions}[p]$.

The next claim shows that if a $\text{Scan}(\cdot)$ operation $sc$ by process $p$ linearizes before the invocation of a $\text{HelpScan}(p)$ operation $hs$ by process $q$, then, $\text{val}(hs)$ is written to $\text{lastScan}[p]$ before $\text{rsp}(hs)$. Therefore, if $q$ reads $\text{lastScan}[p]$ after it performed $hs$, then it either reads $\text{val}(sc)$ or a value that is written to $\text{lastScan}[p]$ by $p$ during a $\text{Scan}(\cdot)$ operation that linearizes after $\text{inv}(hs)$. This shows that the $\text{HelpScan}(\cdot)$ method prevents processes from reading outdated values from $\text{lastScan}[\ldots n - 1]$.

**Claim 8.** Suppose process $p$ executes a $\text{Scan}(\cdot)$ operation $sc$, and process $q$ executes a $\text{HelpScan}(p)$ operation $hs$, such that $\text{lin}(sc) < \text{inv}(hs)$. Then, $t'(sc) < \text{rsp}(hs)$.

**Proof.** For the purpose of a contradiction, assume $t'(sc) \geq \text{rsp}(hs)$. By definition and Lemma 3(a), $\text{lastScan}[p] = \bot$ throughout $(t(sc), t'(sc))$. Also, by Claim 4, $t(sc) < \text{lin}(hs) < t'(sc)$. Therefore, $\text{lastScan}[p] = \bot$ throughout the execution of $hs$. So process $p$ reads $\bot$ from $\text{lastScan}[p]$ in line 56 of $hs$, and changes $\text{lastScan}[p]$ in line 59 of $hs$ to a positive integer fetched from $\text{clk}$. This contradicts the fact that $\text{lastScan}[p] = \bot$ throughout the execution of $hs$.

Finally, we use the above to prove that elements added to $\text{versions}[\ldots n - 1]$ are not removed prematurely, i.e., our memory reclamation scheme works correctly. Suppose a process $p$ adds a pair $(\text{key}, v)$ to $\text{versions}[p]$ during an $\text{Update}(\cdot)$ operation. The memory reclamation scheme ensures that the pair remains in the predecessor object until every $\text{Observe}(\cdot)$ operation that may need to return $v$ has completed. Note that an $\text{Observe}(\cdot)$ operation by process $q$
Adaptive Partial Snapshots with Logarithmic Step Complexity

needs to return $v$ if $q$'s latest preceding $\text{Scan}()$ linearizes after $p$'s $\text{Update}(v)$ and no $\text{Update}()$ by $p$ linearizes in between.

**Observation 9.** Suppose process $p$ executes an $\text{Update}()$ operation up during which it inserts a pair $(key, v)$ into versions of $p$. Let $sc$ be a $\text{Scan}()$ operation by some process $q$, such that $\text{lin}(up) < \text{lin}(sc)$, and no $\text{Update}()$ operation by $p$ linearizes between $\text{lin}(up)$ and $\text{lin}(sc)$. If at point $t^*$ > $\text{rsp}(up)$ the pair $(key, v)$ is in versions of $p$, and some process executes versions of $p$. $\text{Pred}(\text{val}(sc))$, then that operation returns $(key, v)$.

**Proof.** By Claim 7, $key = \text{val}(up)$. Therefore, by the assumption of this observation, $(\text{val}(up), v)$ is in versions of $p$ at $t^*$. Also, by Claim 6, since $\text{lin}(up) < \text{lin}(sc)$, we have $\text{val}(up) < \text{val}(sc)$. Hence, if there is no pair $E$ in versions of $p$ at $t^*$, such that $\text{val}(up) < E.\text{key} < \text{val}(sc)$, then, versions of $p.\text{Pred}(\text{val}(sc))$ operation returns $(key, v)$.

We will prove by contradiction that no such pair exists. Suppose at point $t^*$ there is a pair $E$ in versions of $p$, such that $\text{val}(up) < E.\text{key} < \text{val}(sc)$. By Claim 7, this pair is inserted into versions of $p$ during an $\text{Update}()$ operation up by $p$, such that $E'.\text{key} = \text{val}(up)$. Therefore, $\text{val}(up) < \text{val}(up') < \text{val}(sc)$. By Claim 6, $\text{lin}(up') < \text{lin}(sc)$. This contradicts the fact that no $\text{Update}()$ operation by $p$ linearizes between $\text{lin}(up)$ and $\text{lin}(sc)$. □

**Lemma 10.** Suppose process $p$ executes an $\text{Update}()$ operation up and inserts a pair $(key, v)$ into versions of $p$ in line 25. Let $q$ be a process and $sc$ a $\text{Scan}()$ operation by $q$, such that $\text{lin}(up) < \text{lin}(sc)$, and no $\text{Update}()$ operation by $p$ linearizes between $\text{lin}(up)$ and $\text{lin}(sc)$. Let $t^* > \max(\text{lin}(sc), \text{rsp}(up))$, such that $q$ invokes no $\text{Scan}()$ in $(\text{rsp}(sc), t^*)$. Then $(key, v)$ is in versions of $p$ at $t^*$.

**Proof.** Let $t_1$ be the point in time at which $q$ inserts $(key, v)$ into versions of $q$. We need to prove that this pair is not removed from versions of $q$ throughout $(t_1, t^*)$. For the purpose of contradiction, assume $p$ removes $(key, v)$ from versions of $p$ in line 72 a $\text{Prune}()$ operation $pr$ at $t_{\text{rmv}} < t^*$.

Let $t_3$ be the point when $p$ executes its versions of $p.\text{Pred}(\infty)$ operation in line 62 of its $\text{Prune}()$ call $pr$, and let $key'$ be the key that this operation returns, and which it assigns to $\text{maxKey}$. Process $p$ removes pair $(key, v)$ from versions of $p$ in line 72, and thus $key < key'$.

Let $t_2 < t_3$ be the point when $p$ inserts the pair with key $key'$ into versions of $p$. Then $p$ executes an versions of $p.\text{Pred}(\infty)$ operation in line 25 at point $t_2$, during an $\text{Update}()$ operation up. Since $key' < key'$ it follows from Claims 6 and 7 that $p$ executes up before up'. By Lemma 2(b), $\text{inv}(up') < \text{lin}(up') < t_2 < t_3$. Since up is $p$'s last update that linearizes before $sc$, we conclude $\text{lin}(sc) < \text{lin}(up') < t_3$.

Let $hs$ be the $\text{HelpScan}(q)$ operation that $p$ executes in line 64 of $pr$. Recall that $t_3$ is the point when $p$ executes line 62 of $pr$, and thus $t_3 < \text{inv}(hs)$. Since $\text{lin}(sc) < t_3$, we obtain $\text{lin}(sc) < \text{inv}(hs)$. Therefore, by Claim 8, $t'(sc) < \text{rsp}(hs)$. By definition, $\text{lastScan}[q] = \text{val}(sc)$ at point $t'(sc)$. Since there is no $\text{Scan}()$ by $q$ throughout $(\text{rsp}(sc), t')$, by Lemma 3(d), $\text{lastScan}[q] = \text{val}(sc)$ throughout $(t'(sc), t^*)$. By the assumption that $t_{\text{rmv}} < t'$, and, as shown above, $t'(sc) < \text{rsp}(hs)$, $\text{lastScan}[q] = \text{val}(sc)$ throughout $(\text{rsp}(hs), t_{\text{rmv}})$. Note that $p$ calls $hs$ in line 64 of $pr$, and at point $t_{\text{rmv}}$ it executes line 72 of $pr$ (where it removes pair $(key, v)$ from versions of $p$). Hence, when $p$ executes line 65 during $(\text{rsp}(hs), t_{\text{rmv}})$, it reads $\text{val}(sc)$ from $\text{lastScan}[q]$, and in line 67 it determines a key $y = \text{versions}[p].\text{Pred}(\text{val}(sc))$, which it adds to $S$. Since $(key, y)$ is stored in versions of $p$ throughout $(t_1, t_{\text{rmv}})$, and up is $p$'s latest $\text{Update}()$ preceding $sc$, by Observation 9, $y = key$. Thus, when $p$ executes line 72 of $pr$, $key < S$, and so $p$ does not remove key from versions of $p$ in that line. This is a contradiction. □

4.3.4 Validity. We will now show that the return values of $\text{Observe}(q)$ operations are correct. Since no other operations return anything, this is sufficient for validity. The observation below follows from Lemma 3(c) and (d).

**Observation 11.** Let $ob$ be an $\text{Observe}()$ operation executed by process $p$, and let $sc$ be $p$'s latest $\text{Scan}()$ operation preceding ob. Then, $\text{lastScan}[p] = \text{val}(sc)$ throughout $(\text{inv}(ob), \text{rsp}(ob))$.

**Claim 12.** Suppose process $p$ executes a $\text{Scan}()$ operation $sc$ and later an $\text{Observe}(q)$ operation $ob$, and no other $\text{Scan}()$ between $sc$ and $ob$. Let $t_1$ be the point at which $p$ performs $\text{lastUpdate}[q].\text{ll}(\bot)$ in line 33 during $ob$. Let $up$ be the last $\text{Update}()$ operation by $q$, such that $\text{lin}(up) < t_1$. If $\text{lin}(sc) < \text{lin}(up)$, then $p$ does not execute line 36 throughout $ob$.

**Proof.** In line 33 of $ob$, process $p$ reads a value $x$ from $\text{lastScan}[q].\text{key}$ at $t_1$, and in line 34 it reads a value $y$ from $\text{lastScan}[p]$. For the purpose of contradiction, assume $p$ executes line 36 of $ob$. Therefore, the if-statement in line 35 evaluates to true. Then $\bot \leq x < y$. By Observation 11, $y = \text{val}(sc)$. We will now show that $x = \text{val}(up)$, and then arrive at a contradiction.

By Lemma 2, the value of $\text{lastUpdate}[q].\text{key}$ is $\bot$ throughout $(t(up), t'(up))$, and changes to $\text{val}(up)$ at $t'(up)$.

Because $t'(up) < t_1$, it follows that $t'(up) < t_1$. Also, by Lemma 2(d), $\text{lastUpdate}[q].\text{key}$ remains $\text{val}(up)$ until process $q$ performs $\text{lastUpdate}[q].\text{sc}(\bot)$ in line 22 of a later $\text{Update}()$ operation up' at point $t'(up')$. Then, because up is the last $\text{Update}()$ by $q$ satisfying $t(up) < t_1$, we obtain $x = \text{val}(up)$. Since $\text{lin}(sc) < \text{lin}(up)$ it follows from Claim 6 that $y = \text{val}(sc) < \text{val}(up) = x$. This is a contradiction. □

For the proof of linearizability, we will use the following claim. The proof is omitted due to space restrictions.

**Claim 13.** Suppose process $p$ executes a $\text{Scan}()$ operation $sc$ and later an $\text{Observe}(q)$ operation $ob$, and no other $\text{Scan}()$ between $sc$ and $ob$. Let $t_1$ be the point at which $p$ performs $\text{lastUpdate}[q].\text{ll}(\bot)$ in line 33 during $ob$. Let $up$ be the last $\text{Update}()$ operation by $q$, such that $\text{lin}(up) < \text{lin}(sc)$. Then, $t'(up) < t_1$.

The following lemma proves that the return values of $\text{Observe}()$ operations are correct, and thus the execution is linearizable.

**Lemma 14.** Suppose process $p$ performs a $\text{Scan}()$ operation $sc$, and later an $\text{Observe}(q)$ operation $ob$, and no other $\text{Scan}()$ between $sc$ and $ob$.

(a) If process $q$ performs no $\text{Update}()$ operation that linearizes before $\text{lin}(sc)$, then $ob$ returns $\bot$.

(b) Otherwise, if process $q$’s latest $\text{Update}()$ that linearizes before $\text{lin}(sc)$ uses argument $v$, then $ob$ returns $v$.

**Proof.** We will only prove part (b). The proof of part (a) is very similar, and omitted due to space restrictions. Let $up$ be process
q’s Update(v) operation that linearizes before lin(sc), and that no other Update() by q linearizes before lin(sc).

Let t₁ be the point in time at which process p performs lastUpdate[q].LL() in line 33 of ob. Process p reads a value (x, v’) from lastUpdate[q] at t₁ in line 33, and then reads a value y from lastScan[p] in line 34. By Claim 13, t₁ < t₂. Also, by Observation 11, y = val(sc). We consider two cases.

Case 1: lastUpdate[q] does not change throughout (t₁(t₂), t₁].
Then (x, v’) = (val(up), v). Therefore, since lin(up) < lin(sc) it follows from Claim 1 that x < y. So the if-statement in line 36 of ob evaluates to true and process p returns v.

Case 2: lastUpdate[q] changes during (t₁(t₂), t₁].
Let up* be the last Update() operation by q, such that t(up*) < t₁. Operation up ≠ up* because otherwise by Lemma 2(d), lastUpdate[q] could not change throughout (t₁(t₂), t₁]. Therefore, up happens before up*. Since up is the last Update() operation by q, such that lin(up) < lin(sc), it follows that lin(sc) < lin(up*). So by Lemma 12, process p does not execute line 36 of ob, and hence executes line 38 of ob. Since process q performs lastUpdate[q].SC() in line 22 of up* before t₁, we have inv(up*) < t₁. Hence, rsvp(up*) < t₁. Therefore, by Observation 9 and Lemma 10, process p returns v.

Since Scan() and Update() operations don’t return anything, linearizability of our algorithm follows from Lemma 14.

4.3.5 Analysis. Our algorithm uses the single-writer predecessor objects with bounded capacity Δ, implemented in Section 3. First, we show that a capacity bound of Δ = 3n is sufficient.

LEMMA 15. For any process p, at any point in time, the number of elements stored in versions[p] does not exceed 3n.

Proof. Let pr be a Prune() operation by process p. We will show that at rsvp(pr) at most 2n elements are stored in versions[p]. Since each Prune() operation is distributed over n Update() operations, and during each update exactly one element is inserted into versions[p], the number of elements inserted into versions[p] between the responses of two consecutive Prune() operations is at most n. Therefore, at any point in time, the number of elements in versions[p] is less than 3n.

Consider the local set S that is initialized to ∅ in line 61 of pr. During each iteration of the for-loop starting in line 63, at most one element is inserted into S. Since there are n iterations, |S| ≤ n. Let t* be the point in time at which p reads maxKey in line 62 of pr. By Claim 6 and Claim 7, the pairs that are inserted into versions[p] after t* have keys greater than maxKey. Let U be the set of pairs in versions[p] at t*. Then in line 72, all pairs in U \ S, except for the one with largest key, are removed from versions[p]. Since |S| ≤ n, at most n elements from U are not removed. Moreover, at most n elements are inserted into versions[p] throughout pr. Hence, the number of elements in versions[p] at rsvp(pr) is at most 2n.

It follows that we can use our single-writer predecessor object from Section 3 with bounded capacity Δ = 3n.

Our algorithm also uses multi-word LL/SC objects. We will use the algorithm of Blelloch and Wei [11], which implements a collection of M L-word LL/SC objects from O(M + n³L) registers and CAS objects with step complexity of O(L).

LEMMA 16. The time complexity of a Prune() operation is O(n log n).

Proof. Let p be a process and pr a Prune() operation by p. By Lemma 15, the number of elements in versions[p] is at most Δ = 3n at any point in time. Therefore, by Theorem 1, performing a Préd() operation on versions[p] takes time O(log n).

We use a sequential set, implemented by a balanced binary search tree for the local set S that is initialized in line 61 of pr. During each iteration of the for-loop starting in line 63, at most one element is inserted into S, so |S| ≤ n. Therefore, inserting and finding a pair in S takes time O(log n).

Each He1pScan() operation in line 64 and each LL() operation in line 65 takes constant time. The Préd() operation on versions[p] in line 67 takes O(log n) time, and so does adding the predecessor to S. Hence, the for-loop comprising lines 63-67, takes O(n log n) time.

By Claim 6 and Claim 7, all pairs that are inserted into versions[p] after p reads maxKey in line 62 of pr have keys greater than maxKey. Moreover, process p can remove all outdated pairs from versions[p] in the while-loop in line 69 of pr by traversing through all possible pairs with key less than maxKey. By Lemma 15, the number of pairs in versions[p] is less than maxKey at most 3n. Checking if a key is in S takes O(log n) time complexity, and traversing through elements can be done by calling successor of elements starting from the element with the smallest key. (versions[p].Succ(_∞))

Therefore, line 72 of pr takes O(n log n) steps. So the total amount of work throughout the execution of pr is O(n log n).

THEOREM 17. A wait-free linearizable single-writer adaptive partial snapshot object can be implemented from O(n³ log n) single-word CAS objects, FAI objects, and registers, such that Scan() has constant time (and step) complexity, and Update() and Observe() have O(log n) time (and step) complexity.

Proof of Theorem 17. Linearizability follows immediately from Lemma 14.

By Lemma 16, the total amount of work devoted to each Prune() operation is O(n log n). Moreover, by Lemma 15 and Theorem 1, each operation on versions[p] takes time O(log n). The time complexity of a Scan() operation is O(1). The time complexity of Update() and Observe() is dominated by the operations on a predecessor object, and thus is O(log n).

Our algorithm uses n single-writer predecessor objects of bounded capacity Δ = 3n. By Theorem 1, these can be obtained from a total of O(n³ log n) base objects. In addition, we use 2n O(1)-word LL/SC objects, which can be implemented from O(n³) single-word CAS objects and registers [11]. Finally, we need a single FAI object. Thus, in total O(n³ log n) base objects suffice.

5 CONCLUSION

In this paper we presented a powerful variant of the single-writer snapshot type, which allows a process to adaptively read a consistent view of k memory components in O(k log n) steps. Contrary to most fast snapshot solutions, our algorithm does not make unrealistic assumptions about the size of base objects. Instead, it employs powerful synchronization primitives, which are readily available in most common hardware architectures. To achieve our result we
implement as a building block a bounded memory single-writer predecessor object, which may be of independent interest.

Our algorithm uses unbounded sequence numbers, which has no practical impact on modern 64-bit architectures. However, from a theoretical point of view, this is not satisfying. But we believe that our unbounded sequence numbers can be replaced with a bounded timestamp system. Unfortunately, no practical bounded timestamp systems are known that could replace the unbounded FAI in our algorithm, without significantly reducing efficiency.

An important open problem is to generalize our algorithm to a multi-writer version, or to allow snapshots of stronger primitives, such as CAS objects (similar to [21]).

ACKNOWLEDGMENTS
We thank Trevor Brown and Wojciech Golab for their insightful discussions, and the anonymous PODC reviewers for many useful comments that helped improve our paper.

Support is gratefully acknowledged from the Natural Science and Engineering Research Council of Canada (NSERC) under Discovery Grant RGPIN/2019-04852, and the Canada Research Chairs program.

REFERENCES