Abstract

We present a general transformation that takes any concurrent data structure written using CAS and adds wait-free linearizable query operations to it. These query operations may access arbitrary parts of the data structure, and do not interfere with the progress or running time of other operations. For example, our transformation can be used to add efficient and linearizable range queries, predecessor queries, and top-k queries to existing concurrent set data structures. We achieve this by presenting an efficient technique for taking lazy snapshots of CAS-based data structures.

1 Introduction

The widespread use of multiprocessor machines to handle large-scale computations has underscored the importance of efficient concurrent data structure implementations. Unsurprisingly, there has been a lot of work in recent years on designing practical lock-free and wait-free data structures to meet this demand and guarantee system-wide progress [4, 8, 9, 11, 20, 19, 21, 22]. Many applications that use concurrent data structures require querying large parts of the data structure. For example, one may want to know the size of the data structure, or filter all elements by a certain property, or perform range queries. However, supporting such query operations has been notoriously hard to do efficiently. Many concurrent data structures resort to locking large parts of the data structure, or not guaranteeing linearizability of these queries [1]. Other efforts have implemented specific queries (e.g., range queries) [22, 4, 12, 14], or constructed iterators for concurrent data structures, that, while relatively general, many interfere with other operations or get slowed down by them [38, 3, 21].

The problem boils down to achieving an efficient snapshot of the data structure. A snapshot captures the state of the data structure at a single point in time. Atomic snapshots are known to be extremely useful in many applications, since they may allow linearizable queries that can be run on the sequential state, even while updates run concurrently. Snapshots can also be used to view what the data structure contained at some point in the past. In database systems, snapshots can be used to recover state when the database is corrupted. Due to its usefulness, snapshotting has been used widely in a variety of settings, including database systems [42, 10, 36, 39, 15, 33, 45], persistent sequential data structures [43, 17, 18], and concurrent data structures [37, 6, 30, 9, 2]. Snapshots are often used for multi-versioning, or persistence. Atomic snapshots have been studied for decades in the concurrency theory community. Often, however, it is assumed that the state of the entire data structure...
must be returned immediately, and that registers in memory are unbounded in size, and can store a copy of the entire state.

In this paper, we describe a transformation to convert any shared data structure built from CAS objects and immutable data to support efficient lazy snapshots of the current state, and use this to implement arbitrary (computable) queries for a large class of concurrent data structures. A lazy snapshot returns a handle, which can later be used to read any given memory location and obtain the value of that location at the time the handle was created. We assume, as is the case in practical systems, that each memory location only contains one word. Using these snapshot handles, we show how to add queries to the data structure. Our queries’ running time can be bounded with respect to their running time in isolation (defined as solo in Section 3) plus the number of CAS operations concurrent with the query, and the running times of other operations are unaffected. The methodology supports standard reads and CAS on the current state of each object. It then supplies the ability to take lazy snapshots of the state of the system—i.e., the value of all CAS objects at the current point in the history. We call this a lazy snapshot because the operation does not explicitly return the values of all CAS objects. Instead, it returns a handle that can then be used to read the value of any CAS object at the time the snapshot was taken. Our transformation has the following properties:

- Taking a lazy snapshot of the current state (to get the associated handle) takes constant time.
- CAS and read operations on the current state take constant time. This implies that any asymptotic bounds on time for the original data structure are preserved. Therefore, lock-freedom and wait-freedom are preserved, and operations are never delayed by other processes reading a snapshot.
- Reading an old value of a CAS object using a handle takes time proportional to the number of successful CASes linearized on the object since the snapshot that produced the handle was taken. All reads are wait-free.
- The transformation uses single-word read and CAS. It does, however, require an unbounded timestamp counter.

Our transformation is based on using integer timestamps as handles, and keeping a version list \[42, 10, 43, 17, 18\] for each CAS object. A version list consists of one node per update (successful CAS) on the object, each consisting of the value of the update and the timestamp at the time of the update. The list is ordered by timestamp, most recent first. Taking a snapshot involves reading the current timestamp, and then ensuring it is incremented. An important aspect of the implementation is the method by which a CAS installs a timestamp in the newest node in the version list. The idea is to CAS a new node to the front of the version list, but to temporarily leave its timestamp as to-be-decided (TBD). Once the node has been added to the list, its timestamp is then set (also using a CAS) to the current value of the counter. Importantly, concurrent read or CAS instructions can help store the timestamp in this node. This technique is key for allowing queries and update operations not to interfere with one another. Reading a CAS object at timestamp \(t\) involves traversing the version list until a timestamp less than or equal to \(t\) is found, and returning the corresponding value.

**Queries on Linearizable Objects.** Although taking snapshots of a concurrent data structure is an important step for supporting efficient queries of the global state, it is not quite enough. Firstly, some queries do not need to access the entire state of the data structure, and obtaining a complete snapshot may be wasteful. To avoid this cost, we can use the handle to the snapshot conservatively, and only access parts of the state that are relevant to the query. To formalize this notion, we define a solo query to be one that is linearizable, provided no other processes take steps during the query. Solo queries are generally simpler to design than fully concurrent queries and any such solo query can be adapted to be fully linearizable and run efficiently on our transformed data structure. However, there is another challenge in supporting linearizable queries; not all data structures allow a solo query to be deterministically linearized. For some data structures, we may not know whether an update that is in progress has been linearized yet when looking at a snapshot, as its linearization may depend
on what happens later in the execution. Thus, we define a property that a concurrent data structure must satisfy for it to support solo queries. We consider the abstract data type that a concurrent data structure implements. If, in addition to being linearizable, there is an abstraction function $F$ from the concurrent state to an abstract state, then the abstract state is uniquely defined by a snapshot. The abstraction function must line up with the corresponding linearization points (e.g., at the linearization point of an insert(1) operation on a set data structure, the abstract state would transition from $\{2\}$ to $\{1, 2\}$). Having an abstraction function is similar in some ways to strong linearizability [25], but neither of these conditions implies the other (see Appendix C).

Many, if not most, concurrent data structures allow an abstraction function [3]. Having an abstraction function ensures we can answer any (computable) query on the abstract state, but to get a handle on performance and, equally importantly, to simplify writing queries, we suggest a methodology for designing solo queries based on mapping the concurrent state to an intermediate state that more closely corresponds to a sequential version of the data structure. Solo queries can then be designed similarly to the way they would run sequentially. We evaluate their running time and correctness by considering the mapping from the concurrent snapshot to the intermediate state, and from the intermediate state to the abstract one. We give examples of this design process for Ellen et al.’s non-blocking binary search tree [20] (which we use as a running example), Michael and Scott’s queue [31], and Harris’s linked list [26]. In all cases, the query time is proportional to the number of pointers traversed plus the sum of the write contention on each pointer. For example, finding the smallest key in the BST takes time proportional to the depth of that key, plus the number of successful concurrent updates along the path. However, our technique is applicable beyond these examples.

**Optimizations.** While our transformation introduces only constant overhead for existing operations, and allows the implementation of wait-free queries, the overhead may be significant in practice. In particular, our construction introduces a level of indirection, since to access the value of a given versioned CAS object, one must first access a pointer to the head of the version list, which leads to the actual value. This may introduce an extra cache miss per access. We therefore consider various ways to optimize the implementation. Firstly, we can remove the versioning for objects that do not require it. This optimization hinges on the simple observation that some objects are only written to once, in which case there is no need to maintain versions. Furthermore, if some fields are never accessed by queries, there is no need to keep versions for them either. A second optimization avoids the level of indirection in some cases. We say that an object is recorded once in an execution, if a pointer to it is the new value of a successful CAS at most once (note this is distinct from being written into once, as discussed above). We show that if objects of a given type are always recorded once, then a CAS object that holds pointers to that type can avoid indirection. Instead of using a version list, the timestamp and pointer to the next older version can be stored directly in the objects themselves. Most tree-like structures can be implemented with this restriction and completely avoiding the level of indirection. We give an example for the trees of Ellen et al. [20].

**Memory Reclamation.** A legitimate worry when keeping snapshots is the extra memory needed to keep multiple versions. We show that epoch based memory reclamation [24] can be used with our versioning construction, despite the fact that retired nodes may remain connected to the data structure via links of the version lists. We show that the time and space bounds of epoch based memory reclamation hold for our data structures as well.

**Contributions of this paper.** In summary, the contributions of our paper are the following:

- We introduce a simple and efficient way of obtaining snapshot handles in concurrent data structures that are implemented from CAS objects.
- We show how to use these snapshot handles to implement complex queries on a large class of linearizable data structures, and provide a technique for coming up with efficient query implementations.

---

1. Herlihy and Wing [27] considered abstraction functions and showed that to allow all linearizable implementations requires a relation instead.
• We provide bounds on the running time of the queries with respect to their running time in isolation; the running time of other operations remains unchanged.

• We study cases where we can optimize our transformation by reducing or eliminating its space overhead.

• We show that the transformation allows for efficient memory reclamation.

2 Related Work

Taking a snapshot of an array is a classic problem in shared-memory computing with a long history. Fich surveyed some of this work [23]. A partial snapshot object allows operations that take a snapshot of selected entries of the array instead of the whole array [5]. An $f$-array [28] is another generalization of snapshot objects that allows a query operation that returns the value of a function $f$ applied to a snapshot of the array. We model queries in a similar way.

Recent work has shown how to support complex queries on specific data structures. Bronson et al. [11] gave a blocking implementation of AVL trees that supports a scan operation that returns a snapshot of the whole data structure. Prokopec et al. [41] gave a scan operation for a hash trie by making the trie persistent: updates copy the entire branch of nodes that they traverse. Scan operations have also been implemented for non-blocking queues [35, 40], deques [21], Kallimanis and Kanellou [29] gave a dynamic graph data structure that allows atomic dynamic traversals of a path. Dickerson [16] shows how to transform functional data structures into lock-free, concurrent data structures supporting snapshots using a lazy copy-on-write technique.

Range queries, which return a snapshot of all keys within a given range, have been studied for various implementations of ordered sets. Brown and Avni [12] gave an obstruction-free range query algorithm for $k$-ary search trees. Avni, Shavit and Suissa [7] described how to support range queries on skip lists. Basin et al. [8] described a concurrent implementation of a key-value map that supports range queries. Like our approach, it uses multi-versioning controlled by a global counter.

Fatourou, Papavasileiou and Ruppert [22] described a persistent implementation of a binary search tree that permits wait-free range queries. Whenever a child pointer in the tree is modified, the new child contains a prev pointer to the previous child. The implementation uses a global counter, and a copy of this counter’s value is stored in nodes when they are created. Following prev pointers allows one to reconstruct old versions of the tree. Our work borrows some of these ideas, but avoids the cumbersome handshaking and helping mechanism that were required in [22] to synchronize between scan and update operations. Unlike [22], our work leaves the linearization points of update operations unchanged. This more streamlined approach makes our approach easier to generalize to other data structures. Winblad, Sagonas and Jonsson [44] also gave a concurrent binary search tree that supports range queries.

Some researchers have also taken some steps towards the design of general techniques for supporting complex queries that can be applied to classes of data structures, rather than tailored solutions for individual data structures.

Petrank and Timnat [38] described how to add a non-blocking snapshot operation to non-blocking data structures such as linked lists and skip lists that implement a set abstract data type. Updates and scan operations must coordinate carefully using auxiliary snap collector objects. Agarwal et al. [3] discussed what properties a data structure must have in order for this technique to be applied. Chatterjee [14] adapted Petrank and Timnat’s algorithm to produce partial snapshots.

Arbel-Raviv and Brown [4] described how to implement range queries for concurrent set data structures that use epoch-based memory reclamation. They assume that one can design a traversal algorithm that is guaranteed to visit every item in the given range that is present in the data structure for the entire lifetime of the traversal. It is also assumed that updates are linearized at a write or CAS instruction, and that the location of this instruction is known in advance.
There has been a long history of having transactions see a snapshot of the state while other transactions make updates. This is often referred to as snapshot isolation or multiversioning \cite{42,10,36,37,39,15,30,33,45,9}. Indeed the idea of version lists for snapshots dates back to Reed’s thesis on transactions \cite{42}. This work is all applied to transactions and none of it has the theoretical guarantees described in this paper.

3 Preliminaries

We consider an asynchronous shared-memory system where processes communicate by accessing shared base objects. A base object \( V \) stores a value and supports two atomic primitives. A \texttt{read}(\( V \)) returns the value of \( V \). A \texttt{CAS}(\( V, old, new \)) compares the value of \( V \) to \( old \). If they are equal, it changes the value of \( V \) to \( new \) and returns \texttt{true}; otherwise, it returns \texttt{false} without changing \( V \)’s value.

A data structure is an implementation of an abstract data type (ADT). We use \textit{abstract state} to refer to the states of the ADT. A \textit{concurrent data structure} implements an ADT using base objects and for each process, provides an algorithm for each \textit{operation} that the ADT supports. A \textit{query} operation is one that does not modify the abstract state.

A configuration provides a global view of the system at some point in time. In an initial configuration, each process is in an initial state and each base object has an initial value. The \textit{concurrent state} for a configuration \( C \) is the state of the shared memory in \( C \), and thus, it does not include the states of the processes. A process takes a \textit{step} each time it applies a primitive on a base object; a step also involves the execution of any local computation that is performed before the application of the instruction. We consider each of the following as a step: 1) the invocation or the return instruction of a routine, 2) the invocation or the response of an operation, 3) the allocation of a new record/object by a process and its initialization, and 4) the de-allocation (free) of an object. An execution is an alternating sequence of configurations and steps. An execution \( \alpha \) of an implementation is \textit{valid} starting from a configuration \( C \), if the sequence of steps performed by each process follows the algorithm for that process starting from its state at \( C \), and for each base object, the responses to the primitives performed on the base object are compatible to its specification and to the value stored in the object at configuration \( C \). A configuration is \textit{reachable} if it is the final configuration of a valid execution starting from an initial configuration. A concurrent state is \textit{reachable} if it is the shared memory state in some reachable configuration. A \textit{history} is a sequence of steps. A history \( h \) is \textit{valid} from a configuration \( C \) if there is an execution \( \alpha \), valid from \( C \), whose sequence of steps is \( h \). In what follows, we consider executions that start from an initial configuration and histories that are valid.

For every operation \( \texttt{op} \) invoked by some process \( q \) in an execution \( \alpha \), the execution interval of \( \texttt{op} \) is the subsequence of \( \alpha \) starting with the invocation of \( \texttt{op} \) and finishing with its response, or is the suffix of \( \alpha \) starting with \( \texttt{op} \)’s invocation if \( \texttt{op} \) has no response in \( \alpha \). If \( \alpha \) contains a response for \( \texttt{op} \), \( \texttt{op} \) is \textit{complete}. We say that \( \texttt{op} \) is executed \textit{solo} in \( \alpha \) if there are no steps by processes other than \( p \) between \( \texttt{op} \)’s invocation and \( \texttt{op} \)’s response (similar definitions hold for histories).

An execution \( \alpha \) is \textit{linearizable} if for every complete operation \( \texttt{op} \) in \( \alpha \) (as well as for some of the uncompleted operations), we can assign it a \textit{linearization point} within its execution interval, so that in the sequential execution defined by the linearization points, each operation has the same response as in \( \alpha \). The annotation of \( \alpha \) that also contains these linearization points is called \textit{linearization}. An implementation is \textit{linearizable} if all its executions are linearizable.

An implementation is \textit{wait-free} if every process completes each operation it invokes within a finite number of steps. A subset of the operations are wait-free in an implementation if they have this property. The \textit{step complexity} of an operation instance \( \texttt{op} \) invoked by a process \( q \) in an execution is the number of steps \( q \) performs from \( \texttt{op} \)’s invocation to is response. The \textit{step complexity} of an operation is the maximum step complexity of any instance of the operation in any execution.

\footnote{In what follows, we use the term operation to refer to the operation itself or to an instance of it (with clear meaning from the context).}
Running Example: EFRB Binary Search Tree. We now outline the implementation of a concurrent binary search tree (BST) by Ellen et al. [20], referred to as EFRB. Throughout the paper, we use EFRB as a running example of how to apply our approaches to concurrent data structures. EFRB implements a leaf-oriented tree representing an ordered set with elements as keys stored in the leaf nodes. Each internal node has exactly two children. EFRB supports three operations, Insert\((k)\), Delete\((k)\), and Find\((k)\), for a key \(k\), each returning a boolean indicating success. All three operations start by calling Search\((k)\), which searches for \(k\) using the standard BST searching algorithm, and returns a pointer to the leaf \(l\), its parent \(p\), and its grandparent \(gp\). Find checks whether \(l\) stores key \(k\) and returns the result. The Insert and Delete operations use \(l, p,\) and \(gp\) to change a single child pointer in that neighbourhood of the tree, thereby adding or removing the key.

Processes flag a node before changing its child pointer and mark a node before deleting it, to avoid making two inconsistent changes concurrently in the same neighbourhood. These flags and marks contain information about the operation to be done, so that processes can help one another complete their operations, ensuring lock-freedom. To avoid helpers applying a change twice, EFRB uses a CAS to apply a change to a child pointer of a node (called a child CAS).

Search\((k)\) is linearized at a point when the leaf it returns was on the search path for \(k\). An unsuccessful insert and delete are linearized at the same point as the Search they perform. A successful insert or delete is linearized at the unique successful child CAS of the operation. More details about the EFRB are in Appendix D.

4 Versioned CAS object

We begin by defining two new objects, a versioned CAS object and a camera object. The versioned CAS object behaves similarly to a regular CAS object, but “saves” previous versions to support taking snapshots. Each versioned CAS object is associated with one camera object. A camera object is a shared object that provides an interface for taking snapshots of the current state of all associated versioned CAS objects. The versioned CAS object supports three operations, vRead, vCAS and readSnapshot. Just as in a regular CAS object, vRead returns the current value of the object and vCAS\((oldV, newV)\) changes the object’s value to \(newV\) provided that the current value is equal to \(oldV\). The camera object supports an operation takeSnapshot which returns a handle to a snapshot. This handle serves as an identifier that can be used by later readSnapshot operations to access the value a versioned CAS object had at the linearization point of the takeSnapshot. Thus, by calling takeSnapshot, one clicks the “shutter” to take a snapshot of the current state of the versioned CAS objects immediately, but the readSnapshot operation “develops the film” of that snapshot of the versioned CAS objects lazily later on. The full state of the snapshot is retrievable solely based on the handle returned. In this section, we provide a linearizable implementation of versioned CAS and camera objects, where vCAS, vRead and takeSnapshot can all be supported in a constant number of steps. A readSnapshot operation is wait-free, and the time it takes is proportional to the number of times the versioned CAS object changed between the takeSnapshot and readSnapshot. We start with formally defining the sequential specifications of camera and versioned CAS objects.

Definition 1 (Camera and Versioned CAS Objects). A versioned CAS object stores a value and supports three operations, vRead, vCAS, and readSnapshot. A camera object supports a single operation takeSnapshot. Each versioned CAS object is associated with a single camera object and multiple versioned CAS objects can share the associated camera object. Let \(O\) be a versioned CAS object that is associated with a camera object \(S\). Consider a sequential history of operations on both \(O\) and \(S\). The behavior of \(O.vRead, O.vCAS, O.readSnapshot,\) and \(S.takeSnapshot\) is specified as follows.

- A \(O.vCAS(oldV, newV)\) attempts to update the value of \(O\) to \(newV\) and this update takes place if and only if the current value of \(O\) is \(oldV\). If the update is performed, the vCAS operation returns \(true\) and is successful. Otherwise, the vCAS returns \(false\) and is unsuccessful.
• A $O.v\text{Read}()$ returns the current value of $O$.
• The behavior of $O.\text{readSnapshot}$ and $S.\text{takeSnapshot}$ are specified simultaneously. A precondition of calling $O.\text{readSnapshot}(t)$ is that there must have been an earlier $S.\text{takeSnapshot}$ operation that returned the handle $t$. For any $S.\text{takeSnapshot}$ operation $T$ that returns $t$ and any $O.\text{readSnapshot}(t)$ operation $R$, $R$ must return the value of $O$ when $T$ occurred.

Multiple $\text{takeSnapshot}$ operations on a snapshot object $S$ may return the same handle, but Definition 1 implies that two $\text{takeSnapshot}$ operations can return the same handle $t$ only if each associated versioned CAS object has the same value when these two $\text{takeSnapshot}$ operations occurred. This is because a subsequent invocation of $\text{readSnapshot}(t)$ on any versioned CAS object $O$ associated with the snapshot object must return the value that $O$ had at the time of each of the two $\text{takeSnapshot}$ operations.

4.1 A Linearizable Implementation of Camera and Versioned CAS Objects

We implement the camera object as a counter that stores an integer value, referred to as the timestamp, which is used as the returned handle for snapshots. A $\text{takeSnapshot}$ simply returns the current value of the counter as the handle and attempts to increment the counter using a CAS. Each versioned CAS object is implemented as a linked list (a version list) that preserves all earlier values committed by $v\text{CAS}$ operations, where each version is labelled by a timestamp read from the camera’s counter during the $v\text{CAS}$. The list is ordered with more recent versions closer to the head of the list. A regular $v\text{Read}$ operation just returns the version at the head of the list. A successful $v\text{CAS}$ adds a node to the head of the list. After the node has been added to the list, the value of the snapshot object’s counter is recorded as the node’s timestamp. A $\text{readSnapshot}(t)$ traverses the version list and returns the value in the first node with timestamp at most $t$. We now describe the implementation in more detail.

The Camera Object. The camera object behaves like a global clock for all versioned CAS objects. The pseudo-code for a camera object is shown in Fig. 1. The variable timestamp is the counter that stores the current timestamp. A $\text{takeSnapshot}$ operation simply reads the current value $t$ of the timestamp, and attempts to change the value of timestamp from $t$ to $t+1$ by using a CAS. If this CAS does not succeed, it means that another concurrent $\text{takeSnapshot}$ has incremented the counter, so there is no need to try again. Finally, the $\text{takeSnapshot}$ returns the old timestamp $t$ to be used as a handle. This handle can be used by future $\text{readSnapshot}$ operations to find the version of any versioned CAS object that existed when the counter was incremented from $t$ to $t+1$.

The Versioned CAS Object. Fig. 1 also presents a linearizable implementation of the versioned CAS object. The versioned CAS object stores a pointer VHead to the last node added to the front of the object’s version list. Each node in this list is of type $\text{VNode}$ and stores
• a value val, which is immutable once initialized,
• a timestamp ts, which is the timestamp of the successful $v\text{CAS}$ that stored val into the object, and
• a pointer nextv to the next $\text{VNode}$ of the list, which contains the next (older) version of the object.

The version list essentially stores the history of the object. We use a special timestamp TBD (to-be-decided) as the default timestamp for any newly-created $\text{VNode}$. We note that TBD is not a valid timestamp and must be substituted by a concrete value later, once the $\text{VNode}$ has been added to the version list. We next describe the algorithms in Fig. 1 in detail.

Initializing timestamps. We use $\text{initTS}$ as a separate subroutine (Line 23-25) to initialize the timestamp of a $\text{VNode}$ if it has not been assigned one yet (i.e., if its current timestamp is TBD). For any $\text{VNode}$ that is newly added to the version list, we call $\text{initTS}$ to assign it a timestamp by using the timestamp from the global camera $S$. Once a $\text{VNode}$’s timestamp changes from TBD to something valid, it will never change again, because the CAS on Line 25 succeeds only if the current value is TBD. This $\text{initTS}$ function can be performed either by the process that added the new $\text{VNode}$ to the list, or by another process that is trying to help (see paragraph Helping below for more details).
```java
class Camera {
    int timestamp;
    Camera() { timestamp = 0; }
    int takeSnapshot() {
        int ts = timestamp;
        CAS(&timestamp, ts, ts+1);
        return ts; }
}

Camera S; // global camera object

class VNode {
    Value val; VNode* nextv; int ts;
    VNode(Value v, VNode* n) {
        val = v; ts = TBD; nextv = n; }
}

class VersionedCAS {
    VNode* VHead;
    // constructor
    VersionedCAS(Value v) {
        VHead = new VNode(v, NULL);
        initTS(VHead); }
    void initTS(VNode* n) {
        if (n->ts == TBD) {
            int curTS = S.timestamp;
            CAS(&n->ts, TBD, curTS); }
    }
    Value readSnapshot(int ts) {
        VNode* n = VHead;
        initTS(n);
        while (n->ts > ts) n = n->nextv;
        return n->val; }
    Value vRead() {
        VNode* head = VHead;
        initTS(head);
        return head->val; }
    bool vCAS(Value oldV, Value newV) {
        VNode* head = VHead;
        initTS(head);
        if (head->val != oldV) return false;
        if (newV == oldV) return true;
        VNode* newN = new VNode(newV, head);
        if (CAS(&VHead, head, newN)) {
            initTS(newN);
            return true; }
        else {
            delete newN;
            initTS(VHead);
            return false; }
    }
}
```

Algorithm 1: Linearizable implementation of a camera object and a versioned CAS object.

Implementing `readSnapshot(ts)` and `vRead()`. The `readSnapshot` function returns the latest version of the versioned CAS object with timestamp at most `ts`. It first reads `VHead` and helps set the timestamp of the `VNode` that `VHead` points to by calling `initTS`. The `readSnapshot` then traverses the version list by following `nextv` pointers until it finds a version with timestamp smaller than or equal to `ts`, and returns the value in this `VNode`. The `vRead` function is a special case of `readSnapshot` which only looks at `VHead`, helps set the timestamp of the `VNode` that `VHead` points to, and returns the value in that `VNode`.

Implementing `vCAS(oldV, newV)`. This operation begins by reading `VHead` into a local variable `head`. Then it calls `initTS` on `head` to ensure its timestamp is valid. If the value in the `VNode` that `head` points to is not `oldV`, then the `vCAS` operation fails and returns `false` (Line 45). Otherwise, if `oldV` equals `newV`, the `vCAS` returns `true` because nothing needs to be updated. (This is not just an optimization that avoids creating another `VNode` unnecessarily; it is also required for correctness.) If `oldV` and `newV` are different, and the `VNode` that `head` points to contains the value `oldV`, the algorithm attempts to add a new `VNode` containing the value `newV` to the version list. It first allocates a new `VNode` `newN` (Line 47) to store `newV` and lets it point to `head` as its next older version. It then attempts to add `newN` to the beginning of the list by swinging the pointer `VHead` from `head` to `newN` using a CAS (Line 48). If this CAS is successful, it then calls `initTS` on the new `VNode` to make sure its timestamp is valid, and returns `true` to indicate that the `vCAS` succeeded. Before this call to `initTS` terminates, a valid timestamp will have been recorded in the new `VNode`, either by this `initTS` or by another operation helping the `vCAS`.

If the `CAS` on Line 48 fails, then `VHead` must have changed during the versioned CAS operation. In this case, the new `VNode` is not appended to the version list. The versioned CAS algorithm deallocates the new `VNode` (Line 52) and returns `false`. An unsuccessful `vCAS` is also responsible for helping the first `VNode` in the version list acquire a valid timestamp, since other processes may have appended a new `VNode` to the list without assigning it a timestamp.
Helping. As mentioned, a vRead, readSnapshot and an unsuccessful vCAS all help to set the timestamp of the VNode pointed to by VHead (by calling initTS) before they return. Besides ensuring the timestamp of this VNode is valid, this allows us to linearize a successful vCAS operations as follows. If p performs a successful vCAS that adds a node x to the list, the vCAS is linearized when a process reads the timestamp of x from the camera object S (on Line 24 of initTS) before successfully storing it in x. This ensures that the timestamp is indeed the timestamp in S when the vCAS is linearized. It may be a process helping p that performs the step where p’s vCAS is linearized. Thus, vRead and readSnapshot operations that return the value written by this vCAS, or unsuccessful vCAS operations that fail because of this vCAS, must help in order to ensure that the vCAS is linearized before them.

Initialization. We provide constructors to create new camera and versioned CAS objects. We assume that the constructor for the camera completes before the constructor for any associated versioned CAS object is invoked. (In practice, one will often have just one global camera object for all versioned CAS objects used in a data structure.)

We require, as a precondition of any readSnapshot(ts) operation on a versioned CAS object O, that O was created before the takeSnapshot operation that returned the handle ts was invoked. In other words, one should not try to read the version of O in a snapshot that was taken before O existed. When we use versioned CAS objects to implement a pointer-based data structure (like a tree or linked list), this constraint will be satisfied naturally: if we take a snapshot of the data structure, and then try to traverse a sequence of pointers in it using readSnapshot instructions, we will never find a pointer to O if O did not exist when the snapshot was taken.

4.2 Correctness of the Implementation

The following theorem says that our implementation is correct and provides good time bounds.

Theorem 2. The implementation in Fig. 7 is a linearizable implementation of versioned CAS and camera objects. The number of steps taken by read, vCAS, and takeSnapshot are constant. The number of steps taken by readSnapshot(ts) is proportional to the number of successful vCAS operations that have been assigned timestamps larger than ts (this number is measured at the time the readSnapshot(ts) operation reads VHead).

Given a camera object S and a versioned CAS object O associated with it, in this section, we describe how their operations are linearized, but we defer the detailed proof of Theorem 2 to Appendix A.

To state the linearization points, we first introduce some useful terminology. When referring to the variables O,VHead and S.timestamp, we often abbreviate them to VHead and timestamp. We say that a VNode has a valid timestamp at some configuration C if the value of its ts field is not TBD at C. Otherwise, the timestamp of the node is called invalid. We use the term version list to refer to the list that results from starting at the VNode pointed to by VHead and following the nextv pointers. The head of the version list is the VNode pointed to by VHead.

The only way to modify the version list is the CAS at Line 48, which swings the VHead pointer to a new VNode whose nextv pointer points to the previous head of the version list. This has the effect of adding the new VNode to the beginning of the version list. Before this can happen, initTS is called to install a valid timestamp in the old head of the version list. This ensures that the only VNode in the version list with an invalid timestamp is the first one. At the time a VNode’s timestamp becomes valid, it is therefore still at the head of the version list.

The correctness of readSnapshot operations depends on ensuring that the timestamp associated with a value is current (i.e., in S.timestamp) at the linearization point of the vCAS that stored the value in O. So, we linearize a vCAS that adds a VNode to the version list at the time that the timestamp eventually written into that VNode was read from S.timestamp. This means that there may be a VNode at the head of the version list before the vCAS that created that VNode is linearized. This is why any other operation that finds a VNode
with an invalid timestamp at the head of the version list calls initTS to help install a valid timestamp in it before proceeding. This helping mechanism is crucial in the argument (given in detail in the appendix) that all of the following linearization points are well-defined and within the intervals of their respective operations.

- A vCAS operation is linearized depending on how it executes.
  - If the vCAS performs a successful CAS on Line 48 that adds a node n to the version list, and n’s timestamp eventually becomes valid, then the vCAS is linearized on Line 24 of the initTS method that makes n’s timestamp valid. Note that this case includes all vCAS operations that return true at Line 50.
  - Let h be the value of VHead at Line 43 of a vCAS operation. If the vCAS operation returns on Line 45 or 46, then it is linearized either at Line 43 if h’s timestamp is valid at that time, or the first step afterwards that makes h’s timestamp valid.
  - Finally, consider a vCAS(oldV, newV) operation V that returns false on Line 54. This is the most subtle case. The return on Line 54 is only reached when V fails its CAS on Line 48 because some other vCAS operation changed VHead after V read it at Line 43. We linearize the vCAS immediately after the linearization point of the vCAS operation V’ that made the first such change. (If several vCAS operations that return on Line 54 are linearized immediately after V’, they can be ordered arbitrarily.)

- For a vRead operation that terminates, let h be the VNode read from VHead at Line 38. The vRead is linearized at Line 38 if h’s timestamp is valid at that time, or at the first step afterwards that makes h’s timestamp valid.

- A readSnapshot operation that terminates is linearized at its last step.

- For takeSnapshot operations, let t be the value read from timestamp on line 5. A takeSnapshot operation that terminates is linearized when the value of timestamp changes from t to t + 1.

5 Adding Linearizable Queries to Concurrent Data Structures

In this section, we show how to use versioned CAS objects to extend a large class of concurrent data structures that are implemented using reads and CAS primitives to support linearizable wait-free queries. Throughout the section we also use EFRB as an example to show how our approach works. The idea of this construction is to replace CAS objects with their versioned counterparts, and to use this to obtain snapshots of the concurrent data structure. We can then run queries on the obtained snapshot, without worrying about concurrent updates to the data structure.

The techniques in this section are general. For many data structures, they allow translating any read-only operation on a sequential data structure into a linearizable query on the corresponding concurrent data structure. To achieve this generality, the techniques go through multiple layers of abstraction. To make it more concrete, we show examples of how to add specific linearizable queries to the Michael and Scott queue [31] and EFRB tree [20] in Appendix D.

We present this construction in two parts. First, we define the concept of solo linearizable queries. A query operation is an operation that does not modify the shared state (i.e., it is read-only). A solo linearizable query (or solo query) is a query operation that is only guaranteed to be correct if it is run solo. Intuitively, a solo query is one that can run on a “snapshot” of a concurrent data structure, and it never changes the current state of the data structure. They may be invoked while other operations are pending, but once invoked, they need to run to completion without any other process taking steps during their interval. Section 5.1 describes how to transform a concurrent data structure that supports solo queries (which we refer to as a solo linearizable data structure)
into a fully linearizable one using our versioned CAS objects. However, most concurrent data structures in the literature do not come with solo queries. In Section 5.2 we discuss how to add solo query operations to a given linearizable data structure.

**Definition 3.** We denote by $\mathcal{H}(D, Q)$ the set of histories of concurrent data structure $D$ in which every operation instance from some set of query operations $Q$ is run solo.

**Definition 4.** A concurrent data structure $D$ is linearizable with solo queries $Q$ if every history $H \in \mathcal{H}(D, Q)$ is linearizable. A query $q \in Q$ is called a solo linearizable query on $D$. With clear context of solo queries $Q$, we call $D$ a solo linearizable data structure.

The running time of a solo query may depend on the concurrent state at which it is run. We denote by $T(q, C)$ the running time (number of steps) of a solo query $q$ at concurrent state $C$ of the data structure.

### 5.1 Making Solo Queries Fully Linearizable

We now show how to transform a solo linearizable data structure $D$, implemented with CAS objects, that has a set of solo queries $Q_D$, into a fully linearizable data structure $D_\ell$. Let $L_D$ be the operations of $D$ that are not in $Q_D$. The transformation uses our versioned CAS objects in place of the regular CAS objects of $D$. It preserves all existing correctness guarantees (e.g., linearizability, strong linearizability, sequential consistency) and progress guarantees (e.g., wait-freedom, lock-freedom) of the operations in $L_D$. Furthermore, it preserves the running time of operations of $L_D$ up to constant factors. The time complexity of a linearizable query $q \in Q_D$ in $D_\ell$ is bounded by $q$’s time complexity in $D$, plus a contention term.

**Construction 5.** To obtain $D_\ell$ we replace every CAS object with a versioned CAS object, initialized with the same value. All versioned CAS objects are associated with a single camera object. Each CAS or read by an operation in $L_D$ on a CAS object is replaced by a $v$CAS or $v$Read (respectively) on the corresponding versioned CAS object. To perform a solo query operation $q \in Q_D$ in $D_\ell$, a process $p$ first executes $\text{takeSnapshot}$ on the camera object, to obtain a handle $h$. Then, for any CAS object in $D$ that $q$ would have accessed, $p$ performs $\text{readSnapshot}(h)$ on the corresponding versioned CAS object. Recall that all operations in $Q_D$ are read-only, and thus never perform a CAS.

For this construction to be legal, we must show that the precondition for $\text{readSnapshot}(h)$ holds. Namely, we need to show that $\text{readSnapshot}(h)$ is never called on a versioned CAS object that was created after the handle $h$ was produced. Intuitively, this is satisfied since no versioned CAS object that was created after $h$ can be reachable in the data structure through version nodes with timestamp $h$ or earlier, which are the only version nodes that a query reads. The following claim makes the argument more formal.

**Claim 6.** In Construction 5 no versioned CAS object $O$ is ever accessed using a $\text{readSnapshot}(h)$ operation where $h$ was produced before $C$ was created.

**Proof.** Consider a query operation $q \in Q_D$ that uses handle $h$ to run on a data structure $D_\ell$ as prescribed by Construction 5. We say that a versioned CAS object is new if it was created after $h$ was produced, and old otherwise. Assume by contradiction that $q$ accesses a new $v$CAS object $O_{new}$. $O_{new}$ must be reachable from the root of $D_\ell$ for $q$ to access it. Note that by the way $D_\ell$ is initialized, the root must be an old versioned CAS object. Without loss of generality, assume $O_{new}$ is the first new object that $q$ accesses in its execution. $O_{new}$ must be pointed to by some old versioned CAS object $O_{old}$, through which $q$ accessed $O_{new}$. Since the only updates to versioned CAS objects are via $v$CAS operations, $O_{old}$ must have been updated with a $v$CAS to point to $O_{new}$, thereby creating a new version of $O_{old}$. Note that since $O_{new}$ was created after $h$, this update must have also happened after $h$ was produced, and therefore the version of $O_{old}$ that points to $O_{new}$ has a timestamp larger than $h$. So, $q$ executing $\text{readSnapshot}(h)$ on $O_{old}$ would not access the version pointing to $O_{new}$, but some older version instead. This contradicts the fact that $q$ reaches $O_{new}$. □
Using Construction 5 we can make solo queries linearizable with the bounds specified in the following theorem.

**Theorem 7.** Given a concurrent data structure \( D \) with a set of linearizable operations \( L_D \) and a set of solo query operations \( Q_D \). Construction 5 produces a linearizable data structure \( D_L \) that supports operations from both \( L_D \) and \( Q_D \). This construction maintains the following properties:

- Operations from \( L_D \) have the same progress properties in \( D_L \) as in \( D \), and their runtimes are increased by only a constant factor.
- Each operation \( q \in Q_D \) costs \( O(T(q, C_\ell) + W \cdot A) \) where \( C_\ell \) is the concurrent state at which \( q \) executes the `takeSnapshot` operation, \( W \) is the number of `vCAS` operations concurrent with \( q \) on memory locations accessed by \( q \) in the execution, and \( A \) is the maximum number of repeated accesses to the same object by the query.

The proof is in Appendix B. We note that in most cases, the number of accesses that a query executes to the same object is 1 (or a small constant). If not, this bound can be improved by caching the values read from the data structure locally to avoid the extra overhead of reading it repeatedly from the concurrent data structure.

### 5.2 Adding Solo Queries to Linearizable Data Structures

Concurrent data structures in the literature are usually designed to support a set of operations that are all linearizable. Thus, the question of whether solo linearizable operations can be easily incorporated is generally not considered when designing these data structures. Is it always possible to run queries in a linearizable manner on a snapshot of any given data structure? How efficient can such queries be? In this section, we address these questions.

While designing queries to run solo is certainly much simpler than designing them to be linearizable in the concurrent setting, it is still not as easy as designing queries for a sequential data structure. This is because, in some cases, linearization points cannot be uniquely determined from the state of shared memory; instead, the linearization points may only be determined at the end of the execution, since they can depend on future events. If this is the case, a query that is run solo cannot determine whether a pending update operation has linearized or not, and, since the query may not change the state, it cannot enforce a placement of the linearization point. Herlihy and Wing [27] describe a queue implementation in which the linearization order of the enqueue operations depends on future dequeue operations. For that algorithm, no solo query is possible. Herlihy and Wing [27] point out that the difficulty in this scenario is the absence of an abstraction function from states of the implementation to states of the abstract data type being implemented. We therefore define the notion of direct linearizability, which intuitively means that there is always such a mapping from every concurrent state in an execution of the concurrent data structure to the abstract state of the abstract data type being implemented.

**Definition 8.** An abstraction function of a solo linearizable data structure \( D \) with solo queries \( Q \) that implements an abstract data type \( A \), is a function \( F : C_D \rightarrow C_A \) from concurrent states of \( D \) to abstract states of \( A \) such that for every history \( H \in \mathcal{H}(D, Q) \), there exists a linearization of \( E \) such that:

1. \( F \) maps the initial state of \( D \) to the initial state of \( A \) (i.e., \( F(C_D^{init}) = C_A^{init} \)).
2. If a concurrent state of \( D, C_{D,2} \) can be obtained from another concurrent state \( C_{D,1} \) in \( H \) without the linearization of any operation between \( C_{D,1} \) and \( C_{D,2} \), then they map to the same abstract state (i.e., \( F(C_{D,1}) = F(C_{D,2}) \)).
3. If a concurrent state of \( D, C_{D,2} \) can be obtained from another concurrent state \( C_{D,1} \) in \( H \) where operations, \( op_1, \ldots, op_k \), linearized between \( C_{D,1} \) and \( C_{D,2} \) in this order, then \( F(C_{D,2}) \) is the state of \( A \) that is obtained from applying \( op_1 \ldots op_k \) in this order to \( F(C_{D,1}) \).
Intuitively, the abstraction function respects the linearization points in the execution of \( D \). At first glance, it seems like the abstraction function’s behavior is determined solely by the update operations from \( D \). However, query operations do have an indirect impact because they can affect the linearization points of the update operations, which affects the behavior of the abstraction function. When the definition is applied to fully linearizable data structures, \( Q = \emptyset \), so \( \mathcal{H}(D, Q) \) is the set of all histories of \( D \).

**Definition 9.** A linearizable data structure is said to be **directly linearizable** if it has an abstraction function.

Direct linearizability is reminiscent of **strong linearizability** [25]. Strong linearizability requires that the linearizations can be chosen for histories in a *prefix-preserving* way: for a prefix \( H_p \) of a history \( H \), the linearization of \( H_p \) must be a prefix of the linearization of \( H \). Thus, future events cannot determine whether a given step in the execution was a linearization point or not. Intuitively, direct linearizability requires that update operations be strongly linearizable, but does not require the same behavior from query operations (that do not change the high-level state). Furthermore, while strong linearizability only requires this “prefix preserving” behavior for parts of the state that can be observed by operations of the data structure, direct linearizability imposes this behavior on the entire shared state, regardless of the interface through which operations of the data structure can access it. Appendix[C] shows that strong and direct linearizability are incomparable. However, all strongly linearizable data structures that we are aware of are also directly linearizable.

Consider the EFRB binary search tree [20] outlined in Section[3]. Recall that EFRB implements the **ordered set abstract data type**, with keys as elements. To avoid special cases, the tree includes two leaves containing dummy keys.

**Proposition 10.** Consider the function \( F : C_E \rightarrow C_A \) that maps concurrent states of the EFRB BST to states of the ordered set abstract data type as follows. Given a concurrent state \( C_E \) of EFRB, \( F(C_E) \) is the set of keys in leaf nodes reachable from the root in \( C_E \) except for the two dummy keys. \( F \) is an abstraction function of EFRB.

**Proof.** This theorem is proved as Lemmas 29 and 30 in the technical report [20], so we just sketch it here. Initially, the tree has only the two leaves containing the dummy keys, which \( F \) maps to the empty set, as desired. Each \( \text{Insert}(k) \) that modifies the tree is linearized at the child CAS that adds a leaf containing \( k \) to the tree (and it is shown that \( k \) was not present in the tree before this change). Similarly, each \( \text{Delete}(k) \) that modifies the tree is linearized at the child CAS that removes a leaf containing \( k \) from the tree. Each \( \text{Insert}(k) \) that returns \( \text{false} \) is linearized when \( k \) is in a leaf of the tree, and each \( \text{Delete}(k) \) that returns \( \text{false} \) is linearized when there is no leaf containing \( k \), so these operations have no effect on the tree or on the abstract state of the set. Each terminating \( \text{Find}(k) \) returns \( \text{true} \) if and only if \( k \) appears in some leaf at the linearization point of the \( \text{Find} \). It follows that each operation is linearized so that its effect on the set of keys stored in leaves exactly matches its effect on the abstract state of the ordered set that the tree implements.

Abstraction functions can help us both design solo queries and prove their correctness. It is often helpful to reason about solo queries based on how they behave on each concurrent state. For this purpose, we present the following definition.

**Definition 11.** Let \( op \) be an operation from a concurrent data structure \( D \) and let \( C \) be a reachable concurrent state, we define \( op(C) \) to be the response value of \( op \) when run solo on concurrent state \( C \).

Now we present a proof technique for showing that a read-only operation \( q \) is a solo query. Consider a concurrent data structure \( D \) that implements an abstract data type \( A \) and is linearizable with solo queries \( Q \). Suppose \( D \) has an abstraction function \( F \). We add a query operation \( q_A \) to \( A \) to get the ADT \( A' \) and we add \( q \) to \( D \) to get \( D' \). Our goal is to show that \( D' \) is an implementation of \( A' \) that is linearizable with solo queries \( Q \cup \{q_A\} \). The following observation says that it suffices to show \( q(C) = q_A(F(C)) \) for all reachable concurrent states \( C \).
Observation 12. If $q(C) = q_A(F(C))$ for any reachable concurrent state $C$, then $D'$ is an implementation of $A'$ that is linearizable with solo queries $Q \cup \{q_A\}$. Furthermore, in this case, $F$ is still an abstraction function for $D'$.

Note that the set of reachable concurrent states for $D$ does not change when we add a read-only operation $q_A$ to $D$. The fact that $F$ is still an abstraction function for $D'$ is important because it allows us to add solo queries one at a time. This is summarized by the following observation.

Observation 13. Suppose two data structures have the exact same linearizable operations $L$, but different solo queries $Q_1$ and $Q_2$. If the same abstraction function works for both queries, then adding $Q_2$ to the first data structure results in a new data structure that is linearizable with solo queries $Q_1 \cup Q_2$.

Next, we show how to use the abstraction function as a guide for designing solo queries. If the abstraction function is computable and there is some way of viewing/traversing the state of shared memory, then an easy, but not necessarily efficient, method would be to first traverse the state of $D$, then use the abstraction function to arrive at an abstract state, and finally compute the query on the abstract state. This query literally computes $q_A(F(C))$, so we can apply Observation 12. This is inefficient, since traversing the entire concurrent state often takes much longer than executing the query. We show examples of how to compute queries designed for a sequential version of the data structure on a concurrent state.

5.2.1 Solo queries for EFRB

Consider the EFRB. The concurrent state of the EFRB includes a lot of information used to coordinate concurrent updates. By removing everything except the root pointer, the key, left, right fields of each Internal node, and the key fields of each Leaf node, we end up with a standard leaf-oriented BST (with child pointers, but no parent pointers). This means that sequential read-only queries that work on a leaf-oriented BST, such as predecessor or range queries, can be run on the EFRB as is, because they only access fields that we keep. In the following theorem, we show that these read-only queries can be added to the EFRB as solo queries without any modification.

Theorem 14. Let $Q$ be a set of read-only, sequential operations on a leaf-oriented BST implementing a set of abstract queries $Q_A$. Let $A'$ be an ADT that supports ordered set operations as well as queries from $Q_A$. Adding the operations in $Q$, without modification, to the EFRB yields a concurrent implementation of $A'$ that is linearizable with solo queries $Q$.

Proof. Let $F$ be the abstraction function from Proposition 10 for EFRB. Pick any $q \in Q$ and let $q_A \in Q_A$ be the abstract operation that it implements. Our goal is to show that $q(C) = q_A(F(C))$ for all reachable concurrent states $C$. Then we can apply Observations 12 and 13 to complete the proof.

We begin by defining a mapping $F_A$ from states of leaf-oriented BSTs to states of $A$ and a mapping $F_I$ from concurrent states to states of a leaf-oriented BST. (The $I$ in $F_I$ stands for intermediate state because it is in between the abstract state and the concurrent state.) For $F_A$ we use the textbook mapping which maps an leaf-oriented BST to the set of keys that appear in its leaves. Given an leaf-oriented BST state $S$, return value of $q$ on state $S$ (denoted $q(S)$) equals $q_A(F_A(S))$. To compute the mapping $F_I$, we start with a concurrent state and remove everything except the root pointer, the key, left, right fields of each Internal node, and the key fields of each Leaf node. Since $q$ only accesses the fields that we keep, it cannot tell the difference between running on a concurrent state $C$ and running on $F_I(C)$. Therefore $q(C) = q(F_I(C))$. It is easy to verify that $F = F_A \circ F_I$ and this completes the proof because $q(C) = q(F_I(C)) = q_A(F_A(F_I(C))) = q_A(F(C))$. 

We apply Construction 8 on top of Theorem 14 to get a data structure $D_I$ that supports insert, delete, and find, as well as linearizable implementations of any query for which there is a read-only sequential algorithm. By Theorem 7 we maintain the efficiency of insert, delete, and find up to constant factors.
and for each new query operations in $D_\ell$, it is wait-free and its runtime is proportional to the sequential cost of the query plus $W \cdot A$, where $W$ is the number of $vCAS$ operations that occur during the query and that operate on objects accessed by the query, and $A$ is the maximum number of repeated accesses to the same object by the query. Most read-only, sequential operations on a leaf-oriented BST, such as $\text{predecessor}$ and $\text{range_query}$, can be written so that each query accesses a memory location no more than a constant number of times. For such operations, the added cost is just $O(W)$. For example, consider a $\text{range_query}$ operation that computes the list of keys within a certain range. If we start with a sequential implementation that takes $O(h + k)$ time, where $h$ is the height of the BST and $k$ is the number of keys within the specified range, then the corresponding concurrent query in $D_\ell$ would take $O(h' + k + W)$ time, where $h'$ is the height of the concurrent tree at the linearization point of the operation.

The EFRB is an easy example because the function $F_I$ from the proof of Theorem 14 is essentially an identity function. We show a more complicated example with Harris’s linked list in Appendix E. With an appropriately defined $F_I$, the proof structure we used for Theorem 14 works for Michael and Scott’s queue [31], Harris’s linked list [26], Natarajan and Mittal’s BST [32], and chromatic BSTs [13]. For these algorithms, we need to slightly modify the sequential, read-only operations to make them solo queries. The mapping $F_A$ is always defined to be the standard mapping from sequential to abstract state and the key property to prove is that $F = F_A \circ F_I$.

### 6 Optimizations

Consider a concurrent implementation $D$ that is directly linearizable and implements an abstract data type $A$. Suppose we add a set $Q$ of $k$ query operations to $A$ to get the ADT $A'$. Let $D'$ be the directly linearizable implementation of $A'$ using versioned CAS objects (as described in Section 5).

#### Reducing the Number of Versioned CAS Objects

The first optimization aims at examining cases where the creation of version lists can be avoided. This is accomplished by leaving some of the CAS objects of $D$ unversioned, i.e., by not replacing them with $vCAS$ objects. We can do this for CAS objects that are never accessed by any query $q \in Q$, or are never updated after being initialized. For example, the update fields of Nodes in EFRB (i.e. flags and marks) are objects of this type. So, we do not have to replace them with $vCAS$ objects.

#### Avoiding Indirection

The second optimization applies to $vCAS$ objects used for fields of nodes whose values are pointers to other nodes. We assume that in $D$ (and therefore also in $D_\ell$), every operation accesses nodes of the data structure through one (or more) immutable entry points (e.g. the pointer to the root in a BST).

Consider any execution $\alpha$ of $D_\ell$. We say that a node is recorded-once in $\alpha$, if a pointer to it is the newV parameter of a successful $vCAS$ (that could be applied on any versioned CAS object allocated in $\alpha$) at most once. We say that $D_\ell$ is recorded-once if for every execution of $D_\ell$, every node used in the execution is recorded-once.

The optimization requires that $D_\ell$ is a recorded-once implementation and works as follows. For each versioned CAS object $O$ that stores a pointer to a node in $D_\ell$, instead of creating a new $VNode$ to store the version pointer and the timestamp, the optimization stores this information directly in the node pointed to by $O$, thus avoiding the level of indirection introduced by $VNodes$. This requires expanding each node object with two extra fields. In Appendix G, we show pseudocode for the new version of a node and provide pseudocode for $\text{readSnapshot}$, $\text{vRead}$, and $\text{vCAS}$ after the optimization has been applied. We call the optimized implementation of a versioned CAS object, $OptvCAS$ (and refer to the algorithm provided in Section 4 as $VCAS$). We call the resulting implementation $Opt_2$. 
Correctness of OptVerCAS  We say that $O$ is a distinct-values versioned CAS object if each value written into $O$ is distinct. We first focus on a single distinct-values versioned CAS object $O$ and show that OptVerCAS is linearizable when used to implement $O$. Let $\delta$ be any execution of OptVerCAS on $O$. When $O$ is created, the constructor of $O$ calls initTS on the node $nd$ pointed to by the initial value of $O$. This implies that the timestamp of $nd$ has been set before any operation is performed on $O$. To define the version list of $O$, we find it useful to assign $O$ a timestamp. Specifically, the timestamp of $nd$ (i.e., the timestamp of the node that serves as the initial value of $O$) serves also as the timestamp of $O$. The version list of $O$ at a configuration $C$ of $\delta$ is the list of nodes we get if we start from the node pointed to by $O$.Head at $C$ and follow nextv pointers until we reach a node whose timestamp is less than or equal to the timestamp of $O$.

By inspection of the pseudocode for VerCAS and OptVerCAS, we observe that there is a straightforward analogy between the code executed on each line of VerCAS and the code executed on the corresponding line of OptVerCAS (see Lines 19–55 of the algorithms). This similarity makes it easy to prove that OptVerCAS is linearizable, by following the same proof technique as for VerCAS.

Correctness of $Opt_2$  Since $D_\ell$ is a recorded-once concurrent data structure, $Opt_2$ is also a recorded-once implementation. Consider any execution $\alpha$ of $Opt_2$. Consider two versioned CAS objects $O_1$ and $O_2$ in $\alpha$. Since all nodes are recorded-once, every OptvCAS operation on any versioned CAS object stores a distinct value. This means that every time an OptvCAS is executed (on any versioned CAS object), it writes a pointer to a newly allocated node. Thus, the only way for a node $nd$ other than the last in the version list of $O_1$, to appear in the version list of $O_2$, is if a pointer to $nd$ were used as the initial value of $O_2$. Therefore $nd$ is the last node in $O_2$’s version list. We also argue that no invocation of OptreadSnapshot on a versioned CAS object $O$ traverses the nextv pointer of the last node in the version list of $O$. These imply that the version lists of versioned CAS objects behave as if they are disjoint. In particular, we never have to store nextv pointers for two different version lists in the same node.

References

[1] Java platform standard edition 7 documentation. [http://docs.oracle.com/javase/7/docs/index.html](http://docs.oracle.com/javase/7/docs/index.html).

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

[3] A. Agarwal, Z. Liu, E. Rosenthal, and V. Saraph. Linearizable iterators for concurrent data structures. *CoRR*, abs/1705.08885, 2017.

[4] M. Arbel-Raviv and T. Brown. Harnessing epoch-based reclamation for efficient range queries. In *Proc. 23rd ACM Symposium on Principles and Practice of Parallel Programming*, pages 14–27, 2018.

[5] H. Attiya, R. Guerraoui, and E. Ruppert. Partial snapshot objects. In *Proc. 20th Symposium on Parallelism in Algorithms and Architectures*, pages 336–343, 2008.

[6] H. Attiya and E. Hillel. Single-version STMs can be multi-version permissive (extended abstract). In *Distributed Computing and Networking*, 2011.

[7] H. Avni, N. Shavit, and A. Suissa. Leaplist: Lessons learned in designing TM-supported range queries. In *Proc. 2013 ACM Symposium on Principles of Distributed Computing*, pages 299–308, 2013.

[8] D. Basin, E. Bortnikov, A. Braginsky, G. Golan-Gueta, E. Hillel, I. Keidar, and M. Sulamy. KiWi: A key-value map for scalable real-time analytics. In *Proc. 22nd ACM Symposium on Principles and Practice of Parallel Programming*, pages 357–369, 2017.
[9] N. Ben-David, G. E. Blelloch, Y. Sun, and Y. Wei. Multiversion concurrency with bounded delay and precise garbage collection. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019.

[10] P. A. Bernstein and N. Goodman. Multiversion concurrency control - theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, Dec. 1983.

[11] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proc. 15th ACM Symposium on Principles and Practice of Parallel Programming*, pages 257–268, 2010.

[12] T. Brown and H. Avni. Range queries in non-blocking \(k\)-ary search trees. In *Proc. 16th International Conference on Principles of Distributed Systems*, volume 7702 of LNCS, pages 31–45, 2012.

[13] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 329–342, 2014.

[14] B. Chatterjee. Lock-free linearizable 1-dimensional range queries. In *Proc. 18th Intl Conf. on Dist. Computing and Networking*, pages 9:1–9:10, 2017.

[15] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server’s memory-optimized oltp engine. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1243–1254, 2013.

[16] T. Dickerson. Adapting persistent data structures for concurrency and speculation, 2020.

[17] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Computer and System Sciences*, 38(1):86–124, 1989.

[18] J. R. Driscoll, D. D. K. Sleator, and R. E. Tarjan. Fully persistent lists with catenation. In *ACM-SIAM Symp. on Disc. Algorithms*, pages 89–99, 1991.

[19] F. Ellen, P. Fatourou, J. Helga, and E. Ruppert. The amortized complexity of non-blocking binary search trees. In *ACM Symposium on Principles of Distributed Computing*, pages 332–340, 2014.

[20] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *ACM Symp. on Principles of Distributed Computing*, 2010. See also Technical Report CSE-2010-04, EECS Department, York University, 2010.

[21] P. Fatourou, Y. Nikolakopoulos, and M. Papatriantafilou. Linearizable wait-free iteration operations in shared double-ended queues. *Parallel Processing Letters*, 27(2):1–17, 2017.

[22] P. Fatourou, E. Papavasileiou, and E. Ruppert. Persistent non-blocking binary search trees supporting wait-free range queries. In *Proc. 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 275–286, 2019.

[23] F. E. Fich. How hard is it to take a snapshot? In *Proc. 31st Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, volume 3381 of LNCS, pages 28–37, 2005.

[24] K. Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

[25] W. Golab, L. Higham, and P. Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *ACM Symposium on Theory of Computing (STOC)*, 2011.
[26] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing*, pages 300–314, 2001.

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

[28] P. Jayanti. F-arrays: Implementation and applications. In *Proc. 21st Symposium on Principles of Distributed Computing*, pages 270–279, 2002.

[29] N. D. Kallimanis and E. Kanellou. Wait-free concurrent graph objects with dynamic traversals. In *Proc. 19th International Conference on Principles of Distributed Systems*, Leibniz International Proceedings in Informatics, 2015.

[30] P. Kumar, S. Peri, and K. Vidyasankar. A timestamp based multi-version STM algorithm. In *Int'l Conf. on Dist. Computing and Networking*, 2014.

[31] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *ACM Symposium on Principles of Distributed Computing*, 1996.

[32] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 317–328, 2014.

[33] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 677–689. ACM, 2015.

[34] Y. Nikolakopoulos, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. A consistency framework for iteration operations in concurrent data structures. In *Proc. IEEE International Parallel and Distributed Processing Symposium*, pages 239–248, 2015.

[35] Y. Nikolakopoulos, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. Of concurrent data structures and iterations. In *Algorithms, Probability, Networks and Games: Scientific Papers and Essays Dedicated to Paul G. Spirakis on the Occassion of his 60th Birthday*, pages 358–369. Springer, 2015.

[36] C. H. Papadimitriou and P. C. Kanellakis. On concurrency control by multiple versions. *ACM Transactions on Database Systems*, 9(1):89–99, 1984.

[37] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *ACM Symp. on Principles of Dist. Computing*, pages 16–25, 2010.

[38] E. Petrank and S. Timnat. Lock-free data-structure iterators. In *Proc. 27th Intl Symposium on Distributed Computing*, pages 224–238, 2013.

[39] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in PostgreSQL. *Proc. of the VLDB Endowment*, 5(12):1850–1861, Aug. 2012.

[40] A. Prokopec. Snapqueue: lock-free queue with constant time snapshots. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*, pages 1–12, 2015.

[41] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proc. 17th ACM Symposium on Principles and Practice of Parallel Programming*, pages 151–160, 2012.

[42] D. Reed. Naming and synchronization in a decentralized computer system. Technical Report LCS/TR-205, EECS Dept., MIT, Sept. 78.
[43] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986.

[44] K. Winblad, K. Sagonas, and B. Jonsson. Lock-free contention adapting search trees. In *Proc. 30th Symposium on Parallelism in Algorithms and Architectures*, pages 121–132, 2018.

[45] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment (PVLDB)*, 10(7), Mar. 2017.
A Detailed Proof of Correctness of versioned CAS and camera Objects

In this section, we prove that Fig. 1 is a linearizable implementation of versioned CAS and camera objects. First we argue that it suffices to prove linearizability for histories consisting of a single versioned CAS object and a single camera object. Suppose two versioned CAS objects are associated with different camera objects. Then we can prove linearizability for the two sets of objects independently because they do not access any common variables and do not affect each other in terms of sequential specifications. Suppose two versioned CAS objects $O_1$ and $O_2$ are associated with the same camera object $S$. Let $H'$ be a history of operations on these three objects. Furthermore, let $H_1'$ be the history $H'$ restricted to only operations from $S$ and $O_1$, and similarly, let $H_2'$ be the history $H'$ restricted to only operations from $S$ and $O_2$. We will define the linearization points of $S$ so that they are not affected by operations on $O_1$ or $O_2$. Therefore, showing that both $H_1'$ and $H_2'$ are linearizable is sufficient for showing that $H'$ is linearizable because $S$ will be linearized the same way in both $H_1'$ and $H_2'$.

Let $H$ be a history of a versioned CAS object $O$ and a camera object $S$. We assume that $S$ and $O$ are initialized by their constructors (Line 3 and 19, respectively) before the beginning of $H$. We assume this history satisfies the precondition (described in Definition 1) that whenever readSnapshot($ts$) is invoked, there must be a completed takeSnapshot operation that returned $ts$. When referring to the variables $O.VHead$ and $S.timestamp$, we will often abbreviate them to $VHead$ and timestamp.

We first introduce some useful terminology. We say that a $VNode$ has a valid timestamp at some configuration $C$ if the value of its $ts$ field is not TBD at $C$. Otherwise, the timestamp of the node is called invalid. We use the term version list to refer to the list that results from starting at the $VNode$ pointed to by $VHead$ and following the nextv pointers. The head of the version list is the $VNode$ pointed to by $VHead$.

A modifying $vCAS$ operation is one that performs a successful CAS on line 48. Due to the if statement on line 46, if $vCAS(oldV, newV)$ is a modifying $vCAS$ operation, then $oldV \neq newV$. Note that modifying $vCAS$ operations can return only on line 50 and any operation that returns on line 50 is a modifying $vCAS$. A $vCAS$ is successful if it is a modifying $vCAS$ or if it returns true at line 46. Otherwise, it is unsuccessful.

We first show that the only change to a version list is inserting a $VNode$ at the beginning of it.

Lemma 15. Once a $VNode$ is in the version list, it remains in the version list forever.

Proof. The only way to change a version list is a successful CAS at line 48, which changes $VHead$ from head to newN. When this happens, newN->nextv = head, so all $VNodes$ that were in the version list before the CAS are still in the version list after the CAS. 

It is easy to check that every time we access some field of an object via a pointer to that object, the pointer is not NULL. $VHead$ always points to a $VNode$ after it is initialized on Line 20 of $O$’s constructor. It follows that every call to initTS is on a non-null pointer. The precondition of readSnapshot($ts$) ensures that $ts$ is a timestamp obtained from $S$ after $O$ was initialized and is therefore greater than or equal to the timestamp that $O$’s constructor stored in the initial $VNode$ of the version list. Thus, the readSnapshot will stop traversing the version list when it reaches that initial $VNode$, ensuring that node is never set to NULL on line 34.

Linearization Points. Before we can define the linearization points, we need a few simple lemmas that describe when $VNodes$ have valid timestamps. We start with an easy lemma about initTS.

Lemma 16. The following hold:

1. Before initTS is called on a $VNode$, $VHead$ has contained a pointer to that $VNode$.

2. After a complete execution of initTS on some $VNode$, that $VNode$’s timestamp is valid.
Proof. All calls to \texttt{initTS} are done on a pointer that has either been read from \texttt{VHead} or successfully CASed into \texttt{VHead}. Once a timestamp is valid, it can never be modified again, since only a CAS on line 25 modifies the \texttt{ts} variable of any \texttt{VNode}. The CAS on Line 25 can fail only if the \texttt{ts} variable is already a valid timestamp. \qed

Lemma 17. In every configuration \(C\), the only \texttt{VNode} in the version list that can have an invalid timestamp is the head of the version list.

Proof. No \texttt{VNode}'s \texttt{nextv} pointer changes after the \texttt{VNode} is created, so the only way the version list can change is when \texttt{VHead} is updated. Moreover, no \texttt{VNode}'s timestamp ever changes from valid to invalid. So, we must only show that updates to \texttt{VHead} preserve the claim.

The value of \texttt{VHead} changes only when a successful CAS is executed on Line 48 of an instance of \texttt{vCAS}. Consider any such successful CAS by some process \(p\) and assume the claim holds in the configuration before the CAS to show that it holds immediately after the CAS. This CAS changes \texttt{VHead} from \texttt{head} to \texttt{newN}. By the initialization of \texttt{newN} on Line 47, that \texttt{VNode}'s \texttt{nextv} pointer is \texttt{head}. So, we must show that \texttt{head} and all \texttt{VNodes} reachable from \texttt{head} by following \texttt{nextv} pointers have valid timestamps when the CAS occurs. Before executing this CAS, \(p\) executes \texttt{initTS(head)}, so, by Lemma 16(2), that \texttt{VNode}'s timestamp is valid at the time that the CAS is executed. Since the CAS is successful, \texttt{VHead} was equal to \texttt{head} immediately before the CAS, so all nodes reachable from that \texttt{VNode} had valid timestamps, by our assumption. \qed

The next lemma is used to define the linearization point of a modifying \texttt{vCAS}.

Lemma 18. Suppose an invocation of \texttt{initTS} makes the timestamp of some \texttt{VNode} \(n\) valid. Then, \(n\) is the head of the version list when that \texttt{initTS} executes Line 24 and 25.

Proof. By Lemma 16(1), every call to \texttt{initTS} is on a pointer that has previously been in \texttt{VHead}, so \(n\) has been in the version list before \texttt{initTS} is called. By Lemma 15, \(n\) is still in the version list when Line 24 and 25 are executed. By Lemma 17, \(n\) remains at the head of the version list until its timestamp becomes valid when \texttt{initTS} performs Line 25. \qed

We are now ready to define linearization points. As we define them, we argue that the linearization point of each operation is well-defined and within the interval of the operation.

• A \texttt{vCAS} operation is linearized depending on how it executes.
  
  – If the \texttt{vCAS} performs a successful CAS on Line 48 that adds a node \(n\) to the version list, and \(n\)'s timestamp eventually becomes valid, then the \texttt{vCAS} is linearized on Line 24 of the \texttt{initTS} method that makes \(n\)'s timestamp valid. Lemma 18 implies that the linearization point occurs after the \texttt{vCAS} adds \(n\) to the version list at Line 48. If the \texttt{vCAS} terminates, it first calls \texttt{initTS} on \(n\) at line 49 so Lemma 16(2) ensures the \texttt{vCAS} is linearized and that the linearization point comes before the end of that \texttt{initTS}.
  
  – Let \(h\) be the value of \texttt{VHead} at Line 43 of a \texttt{vCAS} operation. If the \texttt{vCAS} operation returns on Line 45 or 46 then it is linearized either at Line 43 if \(h\)'s timestamp is valid at that time, or the first step afterwards that makes \(h\)'s timestamp valid. Lemma 16(2) ensures this step exists and is within the interval of the \texttt{vCAS}, since \texttt{initTS} is called on \(h\) at line 44.
  
  – Finally, consider a \texttt{vCAS}(\texttt{oldV}, \texttt{newV}) operation \(V\) that returns \texttt{false} on Line 54. This is the most subtle case. The return on Line 54 is only reached when \(V\) fails its CAS on Line 48 because some other \texttt{vCAS} operation changed \texttt{VHead} after \(V\) read it at Line 43. We linearize the \texttt{vCAS} immediately after the \texttt{vCAS} operation \(V'\) that made the first such change. (If several \texttt{vCAS} operations that return on Line 54 are linearized immediately after \(V'\), they can be ordered arbitrarily.)
To argue that this linearization point is well-defined, we must show that the VNode \( n \) that \( V' \) added to the version list gets a valid timestamp, so that \( V' \) is assigned a linearization point as described in the first paragraph above. By Lemma 15, \( n \) is still in the version list when \( V \) reads VHead at Line 53. If \( n \) is no longer at the head of the version list, then \( n \)'s timestamp must be valid, by Lemma 17. Otherwise, if \( n \) is still the head of the version list, then \( n \)'s timestamp is guaranteed to be valid after \( V \) calls initTS on \( n \) (Line 53), by Lemma 16. So, in either case, \( V' \) is assigned a linearization point, which is before the timestamp of \( n \) becomes valid. Thus, \( V' \) (and therefore \( V \)) is linearized before the end of \( V \). Lemma 18 implies that the linearization point of \( V' \) (and therefore of \( V \)) is after \( V' \) adds \( n \) to the version list, which is after \( V \) reads VHead. This proves that \( V' \)'s linearization point is inside the interval of \( V \).

- For a vRead operation that terminates, let \( h \) be the VNode read from VHead at Line 38. The vRead is linearized at Line 38 if \( h \)’s timestamp is valid at that time, or at the first step afterwards that makes \( h \)’s timestamp valid. Lemma 16 ensures that this step exists and is during the interval of the vRead, since the vRead calls initTS on \( h \) at Line 39.

- A readSnapshot operation that terminates is linearized at its last step.

- For takeSnapshot operations, let \( t \) be the value read from timestamp on line 5. A takeSnapshot operation that terminates is linearized when the value of timestamp changes from \( t \) to \( t + 1 \). We know that this occurs between the execution of Line 5 and 6. Either the takeSnapshot operation made this change itself if the CAS at line 6 succeeds, or some other takeSnapshot operation did so, causing the CAS on line 6 to fail.

Note that all operations that terminate are assigned linearization points. In addition, some vCAS operations that do not terminate are assigned linearization points.

**Proof that Linearization Points are Consistent with Responses**

Recall that \( H \) is the history that we are trying to linearize. In the rest of this section, we prove that each operation returns the same response in \( H \) as it would if the operations were performed sequentially in the order of their linearization points.

**Lemma 19.** Assume VHead points to a node \( h \) in some configuration \( C \). If \( h.t.\)s is valid in \( C \) then either \( h \) is the VNode created by the constructor of \( O \), or the vCAS that created \( h \) is linearized before the configuration that immediately precedes \( C \).

**Proof.** Suppose \( h.t.\)s is valid in \( C \) but \( h \) is not the VNode created by the constructor of \( O \). Then \( h \) is created by some vCAS operation \( V \) that added \( h \) to the head of the version list. Since \( h.t.\)s is valid in \( C \), some step prior to \( C \) set \( h.t.\)s by executing Line 25. The linearization point of \( V \) is at the preceding execution of Line 24. Thus, the linearization point precedes the configuration before \( C \).

We define the value of the versioned CAS object in configuration \( C \) to be the value that a versioned CAS object would store if all of the vCAS operations linearized before \( C \) are done sequentially in linearization order (starting from the initial value of the versioned CAS object). The following crucial lemma describes how the value of the versioned CAS object is represented in our implementation. It also says that the responses returned by all readSnapshot and vCAS operations are consistent with the linearization points we have chosen.

**Lemma 20.** In every configuration \( C \) of \( H \) after the constructor of the versioned CAS object has completed,

1. if VHead points to the VNode created by the constructor of the versioned CAS object, then VHead->val is the value of the versioned CAS object,

2. if the linearization point of the vCAS that created the first node in the version list is before \( C \), then VHead->val is the value of the versioned CAS object, and
3. otherwise, $\text{VHead->nextv->val}$ is the value of the versioned CAS object.

Moreover, each $\text{vRead}$ and $\text{vCAS}$ operation that is linearized at or before $C$ returns the same result in $H$ as it would return when all operations are performed sequentially in their linearization order.

Proof. We prove this by induction on the length of the prefix of $H$ that leads to $C$. In the configuration immediately after the constructor of the versioned CAS object terminates, $\text{VHead->val}$ stores the initial value of the versioned CAS object.

Since $\text{nextv}$ and $\text{val}$ fields of a $\text{VNode}$ do not change after the $\text{VNode}$ is created, we must only check that the invariant is preserved by steps that modify $\text{VHead}$ or are linearization points of $\text{vCAS}$ operations (which may change the value of the versioned CAS object) or $\text{vRead}$ operations. We consider each such step $s$ in turn and show that, assuming the claim holds for the configuration $C$ before $s$, then it also holds for the configuration $C'$ after $s$.

First, suppose $s$ is a successful CAS on $\text{VHead}$ at line 48 of a $\text{vCAS}$ operation. It changes $\text{VHead}$ from head to newN, where newN->next = head. By Lemma 17, head->ts is valid when this CAS occurs, since head becomes the second node in the version list. By our assumption, the value of the versioned CAS object prior to the CAS is head->val. Since this step is not the linearization point of any $\text{vCAS}$ operation, the value after the CAS is still head->val. By Lemma 16, $\text{initTS}$ is only called on a pointer that has been in $\text{VHead}$ previously, and newN has never been in $\text{VHead}$ before this CAS, we know that newN->ts is TBD. So the invariant holds after the CAS, since $\text{VHead->nextv->val } = \text{head->val}$.

Now, consider a step $s$ that is the linearization point of a modifying $\text{vCAS(oldV, newV)}$, which we denote $V$, possibly followed by the linearization points of some other $\text{vCAS}$ operations that return false on Line 54. Since $V$ is a modifying $\text{vCAS}$, it added a new $\text{VNode}$ $n_1$ to the head of the version list in front of node $n_2$. This happens after $V$ checks that $n_2.val = oldV \neq newV$ on Line 45 and sets $n_1.nextv$ to point to $n_2$ and sets $n_1.val$ to newV on Line 47. By Lemma 18, $n_1$ is still the head of the version list when step $s$ occurs. So in the configuration $C$ before $s$, the value in the versioned CAS object is $n_2.val = oldV$, by our assumption that the claim holds in $C$. Thus, when $V$ occurs in the sequential execution, it returns true and changes the value of the versioned CAS object to newV. Note that $\text{VHead->val } = \text{newV}$ in $C'$. It remains to check that all other $\text{vCAS}$ operations that return false at line 54 and are linearized immediately after $V$ should return false in the sequential execution and therefore do not change the value of the versioned CAS object. Consider any such $\text{vCAS} V'$ of the form $\text{vCAS(oldV', newV')}. By the definition of the linearization point of $V'$, $V$ makes the first change to $\text{VHead}$ after $V'$ reads it on Line 43. So, $V'$ must have read a pointer to $n_2$ on Line 43. Since $V'$ returns false at Line 54, it must have seen $n_2.val = oldV'$ at Line 45. Thus, oldV' = newV, so when each of the $\text{vCAS}$ operations $V'$ is executed sequentially in linearization order, it should return false and leave the state of the versioned CAS object equal to newV. The claim for $C'$ follows.

Finally, consider a step $s$ that is the linearization point of one or more $\text{vRead}$ operations or $\text{vCAS}$ operations that return at Line 45 or 46. Consider any such operation $op$. Let $h$ be the node at the head of the version list when $op$ reads $\text{VHead}$ at Line 53. Then $s$ is either this read or a subsequent execution of Line 25 that makes $h$’s timestamp valid. Either way, $\text{VHead}$ points to $h$ in $C'$, by Lemma 18. By Lemma 19, either case (1) or (2) of the claim applies to configuration $C$. Either way, the value of the versioned CAS object in $C$ is $h.val$. If $op$ is a $\text{vRead}$, then it returns $h.val$ as it should. If $op$ is a $\text{vCAS}$ that returns false at Line 45, it would do the same in the sequential execution in linearization order because $op$ reads the state of the versioned CAS object in $C'$ from $h.val$ on Line 45 and sees that it does not match its oldV argument. If $op$ returns true at Line 46, it would also return true when performed in linearization order because the state of the versioned CAS object in $C'$ matches both $op$’s oldV and newV values. In all cases the value of the versioned CAS object does not change as a result of $op$, so it is still h.val in C', and the invariant is preserved. □

The following observation follows directly from the way modifying $\text{vCAS}$ operations are linearized.
Observation 21. Consider a VNode $n$ that was added to the version list by a modifying vCAS $V$. If the timestamp of $n$ is valid, then $n$.ts stores the value of $S$.timestamp at the linearization point of $V$.

The following key lemma asserts that version lists are properly sorted.

Lemma 22. The modifying vCAS operations are linearized in the order they insert VNodes into the version list.

Proof. Consider any two consecutive VNodes $n_1$ and $n_2$ in the version list, where $n_1$ is inserted into the list before $n_2$, and let $V_1$ and $V_2$ be the vCAS operations that inserted $n_1$ and $n_2$ to the list, respectively. Recall that the linearization point of a modifying vCAS is at the read of the timestamp (Line 24) of the initTS call that validates the timestamp on the VNode that this vCAS appended to the version list. In particular, a modifying vCAS is linearized after it inserts its VNode into the list (since initTS cannot be called on a VNode before it is inserted, by Lemma 16), but before its VNode is assigned a valid timestamp on Line 25 of initTS. By Lemma 17, a VNode is assigned a valid timestamp before it is replaced as the head of the version list. That is, $V_1$ must be linearized before $n_1$’s timestamp was valid, and $n_1$’s timestamp became valid before $n_2$ was added to the list. Furthermore, $V_2$ was linearized after $n_2$ was added to the list. Therefore, $V_1$ is linearized before $V_2$. □

Now, we prove our main theorem which says that our versioned CAS and camera algorithms are linearizable and have the desired time bounds.

Proof (Theorem 2). We show that the return values of each operation is correct with respect to their linearization points. For vCAS and vRead operations, this follows from Lemma 20.

We prove this for takeSnapshot and readSnapshot simultaneously. Suppose a S.takeSnapshot operation $T$ returns a timestamp $t$, which is passed into a O.readSnapshot operation $R$. We show that $R$ returns the value of $O$ at the linearization point of $T$. Let $h$ be the value of VHead on line 32 of $R$. The timestamp of $h$ is valid after line 33 of $R$, and by Lemma 17 the timestamps of all the nodes in the version list starting from $h$ are valid. This means that on line 34 node->ts is never TBD. Let $n$ be the value of node at the last line of $R$ and let $V$ be the modifying vCAS operation that appended $n$. We know that $n$ is the first node in the version list starting from $h$ with timestamp less than or equal to $t$. Since $T$ is linearized when $S$.timestamp gets incremented from $t$ to $t + 1$, by Observation 21 $V$ is linearized before the linearization point of $T$. Since $R$ returns the value written by $V$, it suffices to show that no modifying vCAS operation gets linearized between the linearization points of $V$ and $T$. By Lemma 22 modifying vCAS operations are linearized in the order they appended VNodes to the version list. Therefore, for all nodes that are older than $n$ in the version list, their modifying vCAS operations are linearized before the linearization point of $V$. Next, we show that all nodes in the version list that are newer than $n$ are linearized after $T$. From the while loop on line 34 we can see that all nodes that lie between $h$ and $n$ (including $h$, excluding $n$) have timestamps that are larger than $t$. All nodes in the version list that are newer than $h$ also have timestamp larger than $t$ because they are appended after line 32 of $R$ and $S$.timestamp is already greater than $t$ at this step. Therefore, by Observation 21 all nodes in the version list newer than $n$ are linearized after the linearization point of $T$. This means $V$ is the last modifying vCAS operation to be linearized before the linearization point of $T$, as required.

The bounds on the step complexity of the operations can be derived trivially by inspection of the pseudocode. □

B Proof of Theorem 7

Proof. We construct $D_t$ from $D$ as described by Construction 5. We want to show that $D_t$ is a linearizable implementation of the data structure $D$ with all of its operations in $Q_D$ and $L_D$. We do so by mapping each history $H_t$ of $D_t$ to a history $H$ of $D$ in which all solo linearizable queries are run in isolation, and in which all
read and CAS operations return the same values as the readSnapshot, vRead and vCAS operations in \( H_\ell \). Furthermore, \( H_\ell \) and \( H \) will have the exact same high-level history.

Given a history \( H_\ell \) of \( D_\ell \), we map it to a history \( H \) of \( D \) as follows. (1) For every query operation \( q \in Q_D \) we move all readSnapshot\((h)\) operations executed by \( q \) to appear immediately after the takeSnapshot that returned \( h \), in the same order. We then remove the takeSnapshot, and replace all readSnapshot\((h)\) operations with reads of the corresponding CAS objects of \( D \). (2) For every vRead or vCAS operation that appears in \( H_\ell \), we simply map it to a read or CAS (respectively) on the corresponding CAS object in \( D \), without moving it in the history.

Note that by the definition of the versioned CAS object, vRead and vCAS behave the same as read and CAS in CAS objects. Furthermore, note that the only operations that are moved in \( H_\ell \) to form \( H \) are readSnapshot operations, which do not affect the state of the versioned CAS object they operate on. Thus, all read and CAS operations in \( H \) return the same values as they return in \( H_\ell \). (Note that all \( D \) and \( H \) are isomorphic in this part.)

Since \( D \) is a solo linearizable data structure and \( H \) is a legal history of \( D \) in which all solo operations run in isolation, \( H \) is a linearizable history. Since all operations in \( H \) return the same values as they return in \( H_\ell \), \( H_\ell \) is also linearizable. Furthermore, we can linearize any operation \( \ell \in L_D \) in \( H_\ell \) at the same step as it linearizes in \( H \) (where we map steps of \( H_\ell \) to steps of \( H \) in the same way as above), and any operation \( q \in Q_D \) at its takeSnapshot operation.

To show the required running time bounds, note that operations in \( D \) and operations in \( D_\ell \) access the same number of base objects. The difference in running time for operations in \( L_D \) is strictly due to the time it takes to access the versioned CAS object for vCAS and vRead operations. By Theorem 2, this amounts to constant overhead. Operations \( q \in Q_D \) execute one takeSnapshot operation, which takes constant time, and then replace every read they would do in the implementation of \( D \) with a readSnapshot\((h)\) where \( h \) is the handle returned by the takeSnapshot. By Theorem 2, the running time of each readSnapshot\((h)\) is proportional to the number of successful vCAS operations on that object since \( h \) was produced. Note that all such vCAS operations on all versioned CAS objects that the query accesses are concurrent with the query itself. Thus, we get our desired time bounds.

\[\square\]

C Relationship of Strong Linearizability to Direct Linearizability

Strong linearizability was introduced by Golab, Higham and Woelfel \[25\] to provide a stronger guarantee that permits reasoning about concurrent executions that involve randomness. At first glance, it seems that this condition might be what is required for our transformation to be applicable. We show here that strong linearizability is not comparable to direct linearizability (defined in Definition 4), which is the property required for our transformation.

Intuitively, an implementation is strongly linearizable if linearization points for each operation can be chosen as the execution proceeds, without needing to know what happens later in the execution. (See \[25\] for the formal definition; the informal definition will suffice for this discussion.)

We first show that strong linearizability does not imply direct linearizability. Consider a non-deterministic ADT \( A \) stores a single bit and provides the following two operations. \texttt{Write-random-bit}, which sets the bit to either 0 or 1, non-deterministically, and returns \texttt{ack}. \texttt{Read} simply returns the current value of the bit. Let \( D \) be an implementation that delays the choice of the random bit written by a \texttt{write-random-bit} until the first subsequent \texttt{read} operation. More precisely, \( D \) uses a writable CAS object \( X \) with three possible states, \( \perp, 0, 1 \). A \texttt{write-random-bit} operation simply writes \( \perp \) into \( X \). A \texttt{read} does a \texttt{CAS}(\( X, \perp, \text{random}(0, 1) \)) and returns the new value if the CAS is successful, or the old value if the CAS is unsuccessful. Since each operation

25
performs only one shared-memory access in \( D \), that access must serve as the linearization point of the operation. It is easy to see that this linearization is correct. Thus, linearization points can be determined without having to know what happens later in the execution. In other words, \( D \) is strongly linearizable. However, \( D \) is not directly linearizable: when \( X \) is in state \( \perp \), there is no abstract state that can be used as the value of the abstraction function \( F \). If \( F(\perp) = 0 \), then an execution in which the subsequent read returns 1 would violate Definition 8. A similar problem arises if \( F(\perp) = 1 \).

Indeed, our transformation would fail if we tried to apply it to \( D \): if a \texttt{takeSnapshot} is performed between a \texttt{write-random-bit} and the first subsequent \texttt{read}, reading the snapshot would yield \( \perp \), and it would be impossible to conclude what state of \( A \) this corresponds to. Thus, strong linearizability is not a sufficient condition for our transformation to be applicable.

Next, we show that direct linearizability does not imply strong linearizability. The snapshot ADT \([2]\) stores a vector of values and allows processes to update components of the vector or perform a \texttt{scan} that reads the whole vector atomically. The classic implementation of \([2]\) implements a snapshot using an array of values (with associated timestamps to avoid ABA problems). Updates are performed by writing the new value to the appropriate location in the array and changing its timestamp. A \texttt{scan} reads the array repeatedly until getting identical results twice. This implementation is not strongly linearizable \([25]\), but it is directly linearizable: the abstraction function simply strips the timestamps from the elements stored in the array to get the state of the ADT.

## D Examples

Our first example focuses on the concurrent queue implementation by Michael and Scott presented in \([31]\). We will call this implementation \texttt{MS-QUEUE}. Thus, we have \( D = \texttt{MS-QUEUE} \), and \( A \) is the abstract data type of a FIFO queue that stores integers and supports the operations \texttt{enqueue} and \texttt{dequeue}.

We start by describing how \texttt{MS-QUEUE} works. \texttt{MS-QUEUE} \([31]\) implements the queue using a simply-linked list of Node objects, each storing a key and a next pointer pointing to the next Node. Two pointers, called \texttt{Head} and \texttt{Tail}, point to the first and the last element of the list that implements the queue, respectively. The first Node of the list is always a dummy Node. Thus, the elements of the queue are the keys of the Nodes starting from the second Node of the list up until its last Node. Initially, the list contains just the dummy Node, whose key can be arbitrary and its next pointer is equal to NULL. At each point in time, the list contains those elements that have been inserted in the queue and have not yet been deleted, in the order of insertion. It also contains the last element that has been dequeued as the first element of the list (i.e., as the dummy Node).

To insert a key \( k \) in the queue, a process \( q \) has to call \texttt{Enqueue}(\( k \)). \texttt{Enqueue} first allocates a new Node \( nd \) with key \( k \) and its next field equal to NULL. It then reads \texttt{Tail} and checks whether the next field of the Node it points to is equal to NULL. If this is so, \texttt{Tail} points to the last element of the queue, and \texttt{Enqueue} attempts to insert \( nd \) after this Node using a \texttt{CAS}. If this \texttt{CAS} is successful, then \( q \) performs one more \texttt{CAS} trying to update \texttt{Tail} to point to \( nd \). Otherwise, some other process managed to insert its own Node as the next to the last one, so \( q \) has to retry. If the Node pointed to by \texttt{Tail} does not have its next field equal to NULL, then some process has managed to insert its own Node as the next Node to the one pointed to by \texttt{Tail} but it has not yet updated \texttt{Tail} to point to this Node (i.e., \texttt{Tail} is falling behind). To ensure lock-freedom, whenever \( q \) discovers that \texttt{Tail} is falling behind, it helps by updating \texttt{Tail} to point to the last Node of the list, before it restarts its own operation.

A process \( q \) executing \texttt{Dequeue}, reads both \texttt{Head} and \texttt{Tail}. If they both point to the same Node and the next field of this Node is NULL, then the queue is empty (it contains just the dummy Node) so \texttt{false} is returned. If they point to the same Node, but the next field of this Node is not NULL, then \texttt{Tail} is falling behind, so \( q \) has to help by performing a \texttt{CAS} to update \texttt{Tail} to point to the last Node of the queue before it retries its own operation. If \texttt{Head} and \texttt{Tail} do not point to the same Node, \texttt{Dequeue} reads the key of the second Node of the list and performs a \texttt{CAS} in an effort to update \texttt{Head} to point to this Node. If the \texttt{CAS} is
successful, Dequeue completes by returning the key that it read (and the Node from where it read this key becomes the dummy Node). Otherwise, \( q \) restarts the execution of Dequeue.

In MS-QUEUE, the Head pointer always points to the first element of the list, whereas the Tail pointer always points either to the last or to the second last pointer of the list. This implies that whenever the next pointer of the last element of the list changes to point to a newly inserted Node, Tail points to the last Node of the list. Moreover, whenever Head is updated, Tail does not point to the first element of the list. These properties and the way helping is performed make it possible to assign linearization points to the queue operations in two different ways. A Dequeue is linearized when Head is updated to point to the list Node whose element the Dequeue returns. An Enqueue can be linearized either at the point the next field of the last Node changes to point to the newly inserted Node, or it can be linearized when the Tail pointer changes to point to the newly inserted Node (notice that the latter change might not be performed by the same process that initiated the Enqueue). Note that whenever Dequeue interferes with Enqueue, i.e., whenever there is just one element in the queue, Dequeue first updates Tail to point to the last Node (if needed) and then performs the deletion. In this way, Head is never ahead of Tail and therefore the linearization point of an Enqueue always precedes the linearization point of the Dequeue that deletes the element that the Enqueue inserted in the list. (Note that this is true for both ways of assigning linearization points.) It is also not hard to prove that the list is always connected, and the Nodes are appended at the end of the list, and that they are extracted from the beginning of it, in the order defined by the sequence of the linearization points assigned to Enqueue operations.

Note that the way we choose to assign linearization points allows us to determine the annotations for the executions of MS-QUEUE (in a straightforward way). Note that both linearization schemes, assign the linearization point of an operation at the point in time that a concrete CAS is executed, i.e. each linearization point is assigned at the point that an internal actions of the MS-QUEUE I/O automaton occurs. This allows us to come up with an abstraction function in each case.

Figures 2 and 3 show how to implement two kinds of read-only queries on top of MS-QUEUE using versioned CAS objects. The first, called peekEndPoints, returns the values of the first and the last element in the queue. The second implements scan, i.e., it returns a set containing the keys of all queue Nodes.

To implement these queries, we have to perform the simple changes to MS-QUEUE described in Section 5. We call the resulting algorithm \( \text{VER-QUEUE} \). In \( \text{VER-QUEUE} \), Head and Tail are versioned CAS objects storing references to VNode objects whose val field points to the first and the last Node of the list, respectively. Similarly, the next field of each Node is a versioned CAS object storing a reference to a VNode object that contains the pointer to the next Node in the queue. The code for Enqueue and Dequeue remains unchanged but every read to Head, Tail or to the next field of a Node has to be replaced with an invocation of \( \text{vRead} \) (on the same object). Similarly, every CAS on each of these objects, has to be replaced with a \( \text{vCAS} \) (on the same object with the same old and new values). We remark that all versioned CAS objects are associated with a single camera object \( S \). A takeSnapshot is invoked on this camera object at the beginning of every query.

peekEndPoints starts by performing a takeSnapshot and storing the resulting handle into a local variable ts. Finally, it executes readSnapshot(ts) to read both Head and Tail and returns the values it read.

A scan first executes takeSnapshot to get a handle ts, and also reads Head and Tail using readSnapshot(ts). Then, it executes a while loop to traverse the list starting from the node pointed to by the value read in Head until the node pointed to by the value read in Tail. It uses a set to collect the pointers of the nodes it traverses (other than the first one) and returns this set at the end. On each node, it calls readSnapshot(ts) to move to the next node. This ensures that updates that occurred in the list after the point that the global timestamp was increased will not be included in the set.
// View both ends of the queue
<Value, Value> peekEndPoints () {
    Node* HNode, TNode;
    int ts = TS.takeSnapshot();
    HNode = Head.readSnapshot(ts);
    TNode = Tail.readSnapshot(ts);
    if (HNode != TNode) {
        HNode = HNode->next.readSnapshot(ts);
        return <HNode->val, TNode->val>; }
    return <⊥, ⊥> ;
}

List<Node*> SCAN () {
    List<Node*> Result; \ initially empty
    int ts = TS.takeSnapshot();
    Node* q = Head.readSnapshot(ts);
    Node* last = Tail.readSnapshot(ts);
    while (q != last) {
        q = q->next.readSnapshot(ts);
        Result.append(q); }
    return Result;
}

Figure 2: Two Example Query Operations for VER-QUEUE.

class Node {
    Value val;
    VersionedCAS <Node*> next;
    Node(Value v, Node* n) :
    {val = v; next = n;}
};

class Queue {
    VersionedCAS <Node*> Head, Tail;
    Camera TS;
    /*
    Same code except replacing each read of Head, Tail or the next
    field of a Node with readSnapshot
    and replacing each CAS on these
    variables with a vCAS
    */
    ...
};

Figure 3: Node representation and Code updates for Enqueue and Dequeue in MS-QUEUE.

D.1 A Versioned Concurrent BST Implementation based on the EFRB BST

We start with a brief, informal description of the concurrent binary search tree (BST) implementation provided in [20], which we will call EFRB. EFRB implements a leaf-oriented tree, i.e., a tree that represents a set whose elements are the keys stored only in the leaf nodes of the tree. The tree is full, i.e., every internal node has exactly two children. Moreover, the tree satisfies the following sorting property: for every internal node \(v\) with key \(K\), the key of every node in the left subtree of \(v\) is smaller than \(K\), whereas every node in the right subtree of \(v\) has key larger than or equal to \(K\).

EFRB supports three operations, Insert\((k)\), Delete\((k)\), and Find\((k)\), where \(k\) is a key. All three operations start by calling Search\((k)\), a routine that searches for \(k\) by following the standard BST searching algorithm. Search returns a pointer \(l\) to the leaf that it arrives, a pointer \(p\) to its parent node, and a pointer \(gp\) to the grandparent of this leaf. Find simply checks whether the leaf node returned by Search contains key \(k\). If it does, it returns true, otherwise false is returned.

In its sequential version, Insert replaces the leaf that the Search arrives at with a BST of three nodes, two leaves containing the key of the node pointed to by \(l\) and the newly-inserted key, and an internal node containing the larger key among the keys of the two leaves. The replacement is performed by switching the appropriate pointer of \(p\) from \(l\) to the root of this BST. Delete essentially performs the inverse action: it switches the appropriate child pointer of \(gp\) from \(p\) to the sibling of the node pointed to by \(l\), thus replacing a part of the tree that is comprised of three nodes (one of which is the leaf to be deleted) with one node, namely the sibling of the node to be deleted.

To avoid synchronization problems, EFRB uses CAS to apply a change to a child pointer of a node. Such a CAS is called a child CAS. Moreover, it flags a node when its child pointer is to be changed, and unflags it after the change of the child pointer has been performed. These two types of CAS are called flag CAS, and unflag CAS, respectively. Thus, EFRB uses flagging to “lock” a node (in a non-blocking manner) whose child pointer
is to be changed. EFRB marks an internal node when the node is to be deleted. It does so by executing a mark CAS. A marked node remains marked forever. To implement flagging and marking, each node has a two-bit status field, which can have one of the following four values: CLEAN, FLAG for insertion, FLAG for deletion, or MARK. A flag or mark CAS can succeed only if it is applied on a node whose status is CLEAN. If a flag CAS fails, the process that performed the flag CAS retries the execution of its operation by starting it from scratch. If the operation is a Delete and the flag CAS succeeds but the mark CAS fails, then the process first unflags gp, using a backoff CAS, and then retries the execution of its operation.

To ensure lock-freedom, each process executing an operation records in an Info object all the information needed by other processes to complete the operation. A pointer to such an object is stored together with the status field of a node (and they are manipulated atomically). A process q that fails to flag or mark a node helps the operation that has already flagged or marked the node to complete (by reading the necessary information in the Info object pointed to by the status field of the node). Then, q restarts its own operation. This ensures that a single operation cannot repeatedly block another operation from making progress. Thus, lock-freedom is ensured.

For EFRB, it is proved that each node a Search(k) visits was in the tree, on the search path for k, at some time during the Search. Search(k) is linearized at the point when the leaf it returns was on the search path for k. The insert and delete operations that return false are linearized at the same point as the Search they perform. Every insert or delete operation that returns true has a unique successful child CAS and the operation is linearized at that child CAS.

Figures 4 and 5 show how we can modify EFRB to support queries using versioned CAS objects. We call the resulting algorithm VER-BST. They also provide pseudocode for RangeSum(a, b), a query that returns the set of those keys in the implemented set that are larger than or equal to a and smaller than or equal to b.

In VER-BST, the child[LEFT] or child[RIGHT] field of each internal Node v is a versioned CAS object storing a reference to a VNode object that contains the pointer to the left or right child, respectively, of v in the tree. Moreover, every read of the child[LEFT] or child[RIGHT] field of a Node is replaced with an invocation of vRead (on the same object). Similarly, every CAS on each of these fields, is replaced with a vCAS (on the same object with the same old and new values). We remark that the status field of a Node does not have to be a versioned CAS object, as queries simply ignore the flag and mark signs on the Nodes. Notice that all versioned CAS objects are paired with the same camera on which a takeSnapshot is invoked at the beginning of the execution of every query.

RangeSum(a, b) first performs a takeSnapshot. Then, it calls the recursive function RSTraverse which traverses part of the tree to perform the required calculation. Pseudocode for RangeSum is provided in Figure 5.

In VER-BST, linearization points can be assigned to insert and delete operations the same way as in EFRB. A query is linearized at the linearization point of the takeSnapshot it invokes on Line 2.

E  Solo queries for Harris’s Linked List

Each node in the Harris linked list contains a key field and a next field. The next field stores a pointer that is potentially marked, indicating that the node containing this next field has been logically deleted. A state of the Harris linked list consists of a set of nodes and a pointer to the first node in the linked list.

The technique we use from constructing solo is similar to the technique we applied to the EFRB in Section 5.2.1. Just like the EFRB, Harris’s linked list implements an ordered set ADT. First, we define $F_A$ to be the standard mapping from sequential linked lists to ordered sets which basically maps a linked list to the set of keys that appear in its nodes. If $q_I$ is a read-only, sequential linked list operation implementing the abstract query $q_A$, then by the correctness of $q_I$, we know that $q_I(S) = q_A(F_A(S))$ for all sequential states $S$. Just like in Section 5.2.1, we use $q_I(S)$ to denote the return value of $q_I$ when run on state $S$. Next, we define a mapping $F_I$ from concurrent states to sequential linked list states such that $F = F_A \circ F_I$ is an abstraction function.
Figure 4: Node representation and Code updates for insert and delete in EFRB.

For each sequential, read-only linked list operation $q_I$ implementing the abstract query $q_A$, we show how to modify $q_I$ into a read-only operation $q$ for the Harris linked list such that for all reachable concurrent states $C$, $q(C) = q_I(F_I(C))$. Once we have these facts, it is fairly straightforward to complete the proof. From the equalities we’ve proven $q(C) = q_I(F_I(C)) = q_A(F_A(F_I(C))) = q_A(F(C))$ for all reachable concurrent states $C$, so by Observation 12, $q$ can be added to Harris’s linked list as a solo query.

In Harris’s original paper [26], he linearizes successful insert operations when the node being inserted gets connected to the data structure and successful delete operations when the node being deleted gets marked for deletion (i.e. when the node gets logically deleted, not physically deleted). We need to define $F_I$ so that the abstraction function $F_A \circ F_I$ is consistent when these linearization points. To compute $F_I(C)$, we physically delete (i.e. unlink) all the logically deleted nodes from $C$ and return the resulting linked list as the sequential state. Using an argument similar to the proof of Proposition 10, we can show that $F = F_A \circ F_I$ is an abstraction function for Harris’s linked list.

Now we show how to transform $q_I$ into $q$. Let $q_I$ be a read-only, sequential linked list operation implementing the abstract query $q_A$. To construct $q$, whenever $q_I$ reads the next pointer of a node, change it to call the getNext function implemented in Figure 6. This function basically skips over any marked nodes and returns the next unmarked node. This effectively ignores logically deleted nodes. Therefore, running $q$ on a reachable concurrent state $C$ has the same effect as running $q_I$ on $F_I(C)$. Plugging all this into the proof framework we specified earlier shows that $q$ can be added to Harris’s linked list as a solo query.

F Memory Reclamation

For memory reclamation, we use Epoch Based Memory Reclamation (EBMR) [24]. EBMR splits an execution into epochs by utilizing a global epoch counter $EC$ (with initial value 1). It supports the operations BeginOp and Retire. A process $p$ that invokes an operation $op$ calls BeginOp to read $EC$ and announces the value read. EBMR maintains a limbo list of objects for each epoch. Objects that are retired (e.g., Nodes that are deleted from the data structure) are stored in the limbo list of the current epoch, until it is ensured that no process holds a pointer to these objects. When all processes have announced they have seen an epoch number that is at least $r$, the limbo list associated with epoch $r - 2$ is collected. In this way, EBMR maintains only the
Integer RangeSum(Key a, Key b) {
    int ts = TS.takeSnapshot();
    return RSTraverse(Root, ts, a, b); }

Integer RSTraverse(Node *nd, int ts, Key a, Key b) {
    if (nd points to a leaf node and nd->key is in [a,b]) return nd->key;
    else {
        if (a <= nd->key) {
            return RSTraverse(nd->child[RIGHT].readSnapshot(ts), ts, a, b); }
        else if (b < nd->key) {
            return RSTraverse(nd->child[LEFT].readSnapshot(ts), ts, a, b); }
        else {
            return RSTraverse(nd->child[LEFT].readSnapshot(ts), ts, a, b) +
                  RSTraverse(nd->child[RIGHT].readSnapshot(ts), ts, a, b); }
    }
}

Figure 5: An Example Query Operation for VER-BST.

struct Node { Key key; Node* next; }

Node* getNext(Node* node) {
    Node* n = node->next;
    while(n != NULL && is_marked(n->next))
        n = unmark(n->next);
    return n; }

Figure 6: Implementation of the getNext operation for Harris Linked List.

limbo lists of the last three epochs.

This scheme can be applied out of the box in both the indirect (unoptimized) and direct (optimized) version of our transformation. We assume that the original concurrent data structure $D$ uses EBMR, and a process invokes Retire on a Node $n$ right after the code instructions that renters $n$ unreachable. In an execution $\alpha$ of $D$, a Node $n$ is unreachable if it cannot be reached by following pointers in other Nodes starting from any entry point of the data structure (otherwise, it is reachable).

When using VER-CAS, we also invoke Retire on the old VNode after every successful vCAS (so that VNodes are also retired). Furthermore, we call Retire on the head of the version list when the Node containing the versioned CAS object is retired. In this way, we retain the same space bounds that are guaranteed by EBMR.

Note that when Retire is called on a Node $n$ in an execution $\alpha$ of $D_\ell$, $n$ may still be reachable through the version list of some versioned CAS object. However, we prove that it is safe to deallocate (to free) $n$ by proving that no operation will ever traverse as far as $n$ into the version list of the versioned CAS object. We say that it is safe to deallocate an object in an execution if no process ever refers to this object at some later point in the execution.

Theorem 23. Consider a concurrent implementation $D$ of a data structure that uses EBMR for memory reclamation, and let $D_\ell$ be the concurrent implementation we get from $D$ using Construction 5 (and the technique for memory reclamation described above). In every execution $\alpha$ of $D_\ell$, whenever a Node is deallocated, it is safe to do so.

Proof. Consider any Node $n$ which is deallocated in $\alpha$, and let $r$ be the value of the epoch counter $EC$ when the
deallocation occurs. By the way EBMR works, it must be that $r > 2$, and $n$ resides in the limbo list of epoch $r - 2$. Therefore, $n$ must have been deleted from the data structure during epoch $r - 2$, i.e., the head of the version list of $n$’s versioned CAS object has been updated to point from $n$ to another VNode $n'$ during epoch $r - 2$. Recall that the timestamp of $n'$ is greater than or equal to that of $n$.

Assume that $n$ remains reachable (after its deletion) through the nextv pointers of the version list of some versioned CAS object $O$. Recall that it is only queries that could access the version list of an object, and they may do so only by calling readSnapshot. Let $op$ be such a query and let readSnapshot($ts$) be any invocation of $op$ that accesses the version list of $O$. Recall that timestamp $ts$ was acquired when readSnapshot($ts$) invoked takeSnapshot at the beginning of its execution. Recall also that readSnapshot($ts$) traverses the version list of the versioned CAS object starting from the most recent version (with the largest timestamp among all versions in $O$’s version list) until it reaches the first version whose timestamp is smaller than or equal to $ts$.

To reach epoch $r$, all processes must have announced epoch $r - 1$. In particular, this means that no process is still executing a query operation that started before epoch $r - 1$. Thus, all timestamps being used by queries at the point that $n$ is deleted, must have been produced after the beginning of epoch $r - 1$. Therefore, the timestamps they use are greater than or equal to that of $n'$ in the version list of $O$, and therefore no readSnapshot operation will attempt to access $n$ after its deallocation (they will stop when reaching $n'$). Thus, deallocating $n$ is safe.

G Pseudocode for Direct versioned CAS Algorithm
using Value = Node*;

class Camera {
  int timestamp;
  Camera() { timestamp = 0; }
  int takeSnapshot() {
    // same as in VerCAS
  }
}

Camera S; // global camera object

class Node {
  /* other fields of Node struct of D */
  Value nextv;
  int ts;
};

class OptVersionedCAS {
  Value Head;
  OptVersionedCAS(Value v) {
    Head = v;
    initTS(Head);
  }
  void initTS(Value n) {
    if(n->ts == TBD) {
      int curTS = S.timestamp;
      CAS(&n->ts, TBD, curTS);
    }
  }
  Value OptreadSnapshot(int ts) {
    Value node = Head;
    initTS(node);
    while(node->ts > ts)
      node = node->nextv;
    return node;
  }
  Value OptvRead() {
    Value head = Head;
    initTS(head);
    return head;
  }
  bool OptvCAS(Value oldV, Value newV) {
    Value head = Head;
    initTS(head);
    if(head != oldV) return false;
    if(newV != oldV) return true;
    newV->nextv = oldV; newV->ts = TBD;
    if(CAS(&Head, head, newV)) {
      initTS(newV);
      return true;
    }
    else {
      initTS(Head);
      return false;
    }
  }
};

Figure 7: Linearizable implementation of a versioned CAS object without indirection where vRead(∞) and vCAS take constant time. Major differences between this and Figure [1] are highlighted.