Constant-Time Snapshots with Applications to Concurrent Data Structures

Yuanhao Wei
yuanhao1@cs.cmu.edu
Carnegie Mellon University, USA

Naama Ben-David
bendavidn@vmware.com
VMware Research, USA

Guy E. Blelloch
guyb@cs.cmu.edu
Carnegie Mellon University, USA

Panagiota Fatourou
faturu@csd.uoc.gr
FORTH ICS and University of Crete, Greece

Eric Ruppert
ruppert@cse.yorku.ca
York University, Canada

Yihan Sun
yihans@cs.ucr.edu
University of California, Riverside, USA

Abstract

Given a concurrent data structure, we present an approach for efficiently taking snapshots of its constituent CAS objects. More specifically, we support a constant-time operation that returns a snapshot handle. This snapshot handle can later be used to read the value of any base object at the time the snapshot was taken. Reading an earlier version of a base object is wait-free and takes time proportional to the number of successful writes to the object since the snapshot was taken. Importantly, our approach preserves all the time bounds and parallelism of the original data structure.

Our fast, flexible snapshots yield simple, efficient implementations of atomic multi-point queries on a large class of concurrent data structures. For example, in a search tree where child pointers are updated using CAS, once a snapshot is taken, one can atomically search for ranges of keys, find the first key that matches some criteria, or check if a collection of keys are all present, simply by running a standard sequential algorithm on a snapshot of the tree.

To evaluate the performance of our approach, we apply it to three search trees, one balanced and two not. Experiments show that the overhead of supporting snapshots is low across a variety of workloads. Moreover, in almost all cases, range queries on the trees built from our snapshots perform as well as or better than state-of-the-art concurrent data structures that support atomic range queries.

CCS Concepts: • **Theory of computation → Concurrent algorithms**.

PPoPP '21, February 27-March 3, 2021, Virtual Event, Republic of Korea
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8294-6/21/02...$15.00
https://doi.org/10.1145/3437801.3441602

Keywords: snapshot, concurrent data structure, versioned CAS, range query

1 Introduction

The widespread use of multiprocessor machines for large-scale computations has underscored the importance of efficient concurrent data structures. Unsurprisingly, there has been significant work in recent years on designing practical lock-free and wait-free data structures to meet this demand and guarantee system-wide progress. Many applications that use concurrent data structures require querying large portions or multiple parts of the data structure. For example, one may want to filter all elements by a certain property, perform range queries, or simultaneously query multiple locations. However, such "multi-point" queries have been notoriously hard to implement efficiently. Although it is easy to support multi-point queries by locking large or multiple parts of the data structure, this approach lacks parallelism. Some concurrent data structures resort to multi-point queries that provide no guarantee of atomicity [41, 43]. Other efforts have implemented specific queries (e.g., range queries, iterators) [2, 4, 14, 18, 27, 28, 47].

A general way to support efficient multi-point queries is to provide the ability to take a *snapshot* of the data structure. Conceptually, a snapshot saves a read-only version of the state of the data structure at a single point in time [1, 3, 30]. Multi-point queries can be performed by taking a snapshot and reading the necessary parts of that version to answer the query, while updates run concurrently. Snapshots are also used in database systems for multiversioning and recovery [9, 20, 40, 44, 48, 51, 58], and in persistent sequential data structures [23, 24, 53]. However, known approaches for taking snapshots either limit the programming model (e.g. purely functional [8, 21]), use locks with no progress guarantees [9, 38, 40], or are lock- or wait-free but have large running times [1, 11, 26, 29, 36].

Given any concurrent data structure, we present an approach for efficiently taking snapshots of its constituent

Compare&Swap (CAS) objects [1]. Importantly, our approach preserves all the time bounds and parallelism of the original algorithm/data structure. Our interface is based on creating a *camera* object that has a collection of associated *versioned CAS objects*, which support read and CAS operations like normal CAS objects, as well as a versioned read operation. The camera object supports a single operation `takeSnapshot` that takes a snapshot of the values stored in all the associated versioned CAS objects. The `takeSnapshot` operation does not make a copy of these objects. Instead it returns, in constant time, a handle that can be used to query (via the versioned read operation) the state of any subset of the versioned CAS objects at the time when the handle was acquired. New versioned CAS objects can be associated with an existing camera object, so our construction is applicable to dynamically-sized data structures.

Our interface is more flexible than the one traditionally used for a *snapshot object* [1], which stores an array and provides *update* operations that write to individual components and *scan* operations that return the state of the entire array. Instead of creating a copy of the state of the entire shared memory in the local memory of a process, our `takeSnapshot` simply makes it possible for a process to later read *only* the memory locations it needs from shared memory, knowing that the collection of all such reads will be atomic. Although partial snapshot objects [5, 34] allow scans of part of the array, they require the set of locations to be specified in advance, whereas our approach allows the locations to be chosen dynamically as a query is executed.

Our algorithm has the following important properties.

1. Taking a snapshot of the current state and returning a handle to it takes a constant number of instructions.
2. A CAS or read of the current state of a versioned CAS object takes constant time. Therefore, adding snapshots to a CAS-based data structure preserves the data structure's asymptotic time bounds.
3. Reading the value of a versioned CAS object from a snapshot takes time proportional to the number of successful CAS operations on the object after the snapshot and before the start of the read. Thus, all reads are *wait-free* (i.e., every read is completed within a finite number of instructions.)
4. The algorithm is implemented using single-word read and CAS, which are supported by modern architectures. It does, however, require an unbounded counter.

We know of no previous general mechanism for snapshotting the state of memory satisfying even the first two properties.

Similarly to previous work [9, 29, 38, 40, 51, 53, 58], we use a version list for each CAS object. The list has one node per update (successful CAS) on the object. Each node contains the value stored by the update and a *timestamp* indicating when the update occurred. The list is ordered by timestamps, most recent first. A difficulty in implementing version lists without locks, which we address, is the need to add a node to the version list, read a global timestamp, and save that timestamp in the node, all atomically.

Snapshots and Multi-point Queries. Our interface provides a simple way of converting a concurrent data structure built out of CAS objects into one that supports snapshots: simply replace all CAS objects with our versioned CAS objects, all associated with a single camera object. If all shared mutable state is stored in the CAS objects, then taking a snapshot will effectively provide access to an atomic copy of the entire state of the data structure at the snapshot's linearization point[2]. After taking a snapshot, a read-only query is free to visit any part of the data structure state at its leisure, even as updates proceed concurrently. Often, the query can be performed by simply taking a snapshot and then running a standard sequential algorithm on the data structure by replacing each read with our versioned read.

In Section 4, we define more precisely when multi-point queries can be computed from snapshots. In particular, we discuss how our approach can be used for arbitrary queries on Michael-Scott queues [39], Harris's linked-lists [32], and two different binary search trees [15, 25]. On the binary search trees, for example, one can support atomic queries for finding the smallest key that matches a condition, reporting all keys in a range, determining the height of the tree, or multisearching for a set of keys. The time complexity of each query is the sequential cost of the query plus the number of `vCAS` operations it is concurrent with.

Avoiding Indirection and Other Optimizations. Our algorithm introduces only constant overhead for existing operations, and allows the implementation of wait-free queries. However, our construction does introduce a level of indirection: to access the value of a 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 an optimization to avoid this in Section 5. This optimization applies to many concurrent data structures that satisfy the *recorded-once* property we introduce. Roughly speaking, recorded-once means that each data structure node is the new value of a successful `CAS` at most once. This allows us to store information for maintaining the version lists (in particular the timestamp and the pointer to the next older version) directly in the nodes themselves, thus removing the level of indirection (see Figure 2 for an example). In Section 6, we describe other optimizations to reduce contention and shorten version lists.

[1] A CAS object V stores a value and supports two atomic *operations*: V.`read`() returns the value of V; V.CAS(*old*, *new*) compares the value of V to *old* and if they are equal, it changes the value of V to *new* and returns `true`; otherwise, it returns `false` without changing V's value.

[2] We use the standard definition of *linearizability* [33], which roughly states that every operation must appear to have taken effect atomically at its linearization point, between its invocation and response.

Memory Reclamation. Maintaining all old versions of a versioned CAS object may be infeasible. In Section 7, we describe how to garbage collect old versions using an approach based on Epoch Based Memory Reclamation (EBR) [31]. Experiments indicate that our approach works well in practice and has low memory overhead.

Implementation and Experiments. To study the time and space overhead of our approach, we applied it to three existing concurrent binary search trees, one balanced and two not [4, 15, 25]. Adding support for snapshots was very easy and required minimal changes to the original code. The experiments demonstrate that the overhead is small. For example, the time overhead of supporting snapshots is about 9% for a mix of updates and queries on the current version of the tree. We also compare to state-of-the-art data structures that support atomic range queries, including `KiWi` [7], `LFCA` [57], `PNB-BST` [28], and `SnapTree` [12]. In almost all cases, our data structure performs as well as or better than all of these special-purpose structures even though our approach is general purpose. Finally, we implement a variety of other atomic multi-point queries and show that the overhead compared to non-atomic implementations, which are correct only when there are no concurrent updates, is small.

Contributions. In summary, our contributions are:

- A simple, constant-time approach to take a snapshot of a collection of CAS objects.
- A technique to use snapshots to implement linearizable multi-point queries on many lock-free data structures.
- Optimizations that make the technique more practical, for example, by avoiding indirection.
- Experiments showing our technique has low overhead, often outperforming other state-of-the-art approaches, despite being more general.

2 Related Work

Implementing a snapshot object is a classic problem in shared-memory computing with a long history. Fich surveyed some of this work [30]. A *partial snapshot* object allows operations that take a snapshot of selected entries of the array instead of the whole array [5, 34]. An f*-array* [35] 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. As mentioned above, snapshot objects have a less flexible interface than our approach to snapshotting.

We describe in Section 4 how to use our snapshots to support multi-point queries on a wide variety of data structures. Previous work has focused on supporting such queries on *specific* data structures. Bronson *et al.* [12] gave a blocking implementation of AVL trees that supports a `scan` operation that returns the state of the entire data structure. Prokopec *et al.* [50] 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 [41, 42, 49] and deques [27]. Kallimanis and Kanellou [37] gave a dynamic graph data structure that allows atomic dynamic traversals of a path.

Range queries, which return all keys within a given range, have been studied for various implementations of ordered sets. Brown and Avni [14] gave an obstruction-free range query for k-ary search trees. Avni, Shavit and Suissa [6] described how to support range queries on skip lists. Basin *et al.* [7] 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 [28] gave a persistent implementation of a binary search tree with wait-free range queries, also based on version lists. Our work borrows some of these ideas, but avoids the cumbersome handshaking and helping mechanism they use to synchronize between scan and update operations. This more streamlined approach makes our approach easier to generalize to other data structures. Winblad, Sagonas and Jonsson [57] also gave a concurrent binary search tree that supports range queries.

Some researchers have also taken steps towards the design of general techniques for supporting multi-point queries that can be applied to classes of data structures, although none are as general as our approach.

Petrank and Timnat [47] described how to add a non-blocking `scan` operation to non-blocking data structures such as linked lists and skip lists that implement a set abstract data type; `scan` returns the state of the entire data structure. Updates and scan operations must coordinate carefully using auxiliary *snap collector* objects. Agarwal *et al.* [2] discussed what properties a data structure must have in order for this technique to be applied. Chatterjee [18] adapted Petrank and Timnat's algorithm to support range queries.

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.

Within the database and software transactional memory (STM) literature there has been a long history of having transactions capture a snapshot of the state using multi-versioning [8, 9, 17, 20, 22, 29, 38, 40, 44–46, 48, 51, 52, 54, 58]. This avoids conflicts between read-only transactions and write transactions. Indeed, the idea of version lists for this purpose dates back to Reed's thesis on transactions [51] and is implemented in many modern-day database systems. Much of the work, especially the earlier work, is lock-based. Fernandes and Cachopo [29] introduced a lock-free approach to transactional multiversioning. Their approach, however, fully sequentializes transactions that require updates by adding each successful transaction to the end of a transactional log. Other work has, for example, studied how to make updates in the past [22] by splicing elements into the version lists.

As compared to this work on effectively snapshotting the state after whole transactions, our focus is on standard concurrent algorithms and snapshots at the level of individual memory operations. A key objective in our work is maintaining the time bounds of the original algorithms when operating on the current copy, and thus maintaining an algorithm's progress properties, e.g., wait-freedom, constant-time. For this purpose, concurrent updates are crucial—even wait-freedom for simulating a CAS would by itself be insufficient. Since we focus on the more restricted problem of making concurrent data structures snapshottable, we can use techniques and optimizations that would be difficult if at all possible in the more general STM setting.

3 Versioned CAS Objects

Our approach uses "time-stamped" versioned lists to maintain the state of each object, as in previous work (e.g., [9, 29, 38, 40, 51, 58]). Unlike most of this work, updates do not increment the timestamps—only taking a snapshot might increment the timestamp.[3] An important aspect of our algorithm is how it attaches a timestamp to a new version when updating an object (with a CAS). This involves temporarily setting the new version's timestamp to an undetermined value (TBD) and then updating this to the "current" timestamp only after it is inserted into a version list. The new version's timestamp might be updated by the CAS that created it or via helping by a concurrent operation accessing the object. This ability to help is crucial.

We begin with a sequential specification of our objects.

Definition 1 (Camera and Versioned CAS Objects)**.** A *versioned CAS* object stores a *value* and supports three operations, vRead, vCAS, and readVersion. The first two operate on the *current value* and the third is to access a snapshotted value. A *camera* object supports a single operation, takeSnapshot. Each versioned CAS object O is associated with a single camera object when it is created. Consider a sequential history of operations on a camera object S and the set Λ_S of vCAS objects associated with it. The behavior of operations on S and O for all $O \in \Lambda_S$, is specified as follows:

- An 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*.
- An O.vRead() returns the current value of O.
- The behavior of readVersion and takeSnapshot are specified simultaneously. A precondition of O.readVersion(ts) is that there must have been an earlier S.takeSnapshot() that returned the handle ts. For any S.takeSnapshot() operation T that returns ts and any O.readVersion(ts) operation R, R must return the value O had when T occurred.

[3]When there are concurrent snapshots, only one needs to increment the timestamp, avoiding sequentializing snapshots.

Multiple takeSnapshot operations on a camera object S may return the same handle, but Definition 1 implies that two takeSnapshot operations can return the same handle ts only if each associated versioned CAS object has the same value when these two takeSnapshot operations occurred.

3.1 A Linearizable Implementation

Algorithm 1 is a linearizable implementation of versioned CAS and camera objects. The vCAS, vRead and takeSnapshot operations all take constant time.

The Camera Object. The camera object behaves like a global clock for all versioned CAS objects associated with it. It is implemented as a counter called timestamp that stores an integer value. A takeSnapshot simply returns the current value ts of variable timestamp as the handle and attempts to increment timestamp using a CAS. If this CAS fails, it means that another concurrent takeSnapshot has incremented the counter, so there is no need to try again. The handle will be used by future readVersion operations to find the latest version of any versioned CAS object that existed when the counter was incremented from ts to $ts + 1$.

The Versioned CAS Object. Each versioned CAS object is implemented as a singly-linked list (a *version list*) that preserves all earlier values committed by vCAS operations, where each version is labeled by a timestamp read from the camera's counter during the vCAS. The list is ordered with more recent versions closer to the head of the list. A vRead operation just returns the version at the head of the list. A successful vCAS adds a node to the head of the list. *After* the node has been added to the list, the value of the camera object's counter is recorded as the node's timestamp. A readVersion(ts) operation traverses the version list and returns the value in the first node with timestamp at most ts.

The versioned CAS object stores a pointer VHead to the last node added to the object's version list. Each node in this list is of type VNode and stores

- a value val, which is immutable once initialized,
- a timestamp ts, and
- a pointer nextv to the next VNode of the list, which contains the next (older) version of the object.

The version list essentially stores the history of the object.

Timestamps. We use a special timestamp TBD (to-be-decided) as the default timestamp for any newly-created VNode. TBD is not a valid timestamp and must be substituted by a concrete value later, once the VNode has been added to the version list. When a VNode x is added to the version list, we call the initTS subroutine (Lines 23–25) to assign it a valid timestamp read from the camera object's timestamp field. Once x's timestamp changes from TBD to a valid value, it will never change again, because the CAS on Line 25 succeeds only if the current value is TBD. This initTS function can be performed either by the process that added x to the list, or by another process that is trying to help.

```
1  class Camera {
2    int timestamp;
3    Camera() { timestamp = 0; }
4    int takeSnapshot() {
5      int ts = timestamp;
6      CAS(&timestamp, ts, ts+1);
7      return ts; }
8  }
9  class VNode {
10   Value val; VNode* nextv; int ts;
11   VNode(Value v, VNode* n){
12     val = v; ts = TBD; nextv = n;}
13 };
14 class VersionedCAS {
15   VNode* VHead;
16   Camera* S;
17   VersionedCAS(Value v, Camera* s) {
18     S = s;
19     VHead = new VNode(v, NULL);
20     initTS(VHead);
21   }
22   void initTS(VNode* n) {
23     if(n->ts == TBD) {
24       int curTS = S->timestamp;
25       CAS(&(n->ts), TBD, curTS); }
26   }
27   Value readVersion(int ts) {
28     VNode* node = VHead;
29     initTS(node);
30     while (node->ts > ts)
31       node = node->nextv;
32     return node->val; }
33   Value vRead() {
34     VNode* head = VHead;
35     initTS(head);
36     return head->val; }
37   bool vCAS(Value oldV, Value newV) {
38     VNode* head = VHead;
39     initTS(head);
40     if(head->val != oldV) return false;
41     if(newV == oldV) return true;
42     VNode* newN = new VNode(newV, head);
43     if(CAS(&VHead, head, newN)) {
44       initTS(newN);
45       return true;
46     } else {
47       delete newN;
48       initTS(VHead);
49       return false; } } };
```

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

***Implementing* `readVersion(ts)` *and* `vRead()`.** The goal of a `readVersion(ts)` is to return the latest version whose timestamp is at most `ts`. It first reads `VHead` and if necessary helps set the timestamp of the `VNode` that `VHead` points to by calling `initTS`. The `readVersion` 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 looks only 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 first reads `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`, the `vCAS` operation fails and returns `false` (Line 40). 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 because without it, a successful `vCAS`(a, a) could cause a concurrent `vCAS`(a, b) to fail. 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` with value `newV` to the version list. It first allocates a new `VNode` `newN` (Line 42) to store `newV` and lets it point to `head` as its next 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 43). If successful, it then calls `initTS` on the new `VNode` to ensure its timestamp is valid, and returns `true` to indicate success. 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 43 fails, then `VHead` must have changed during the `vCAS`. In this case, the new `VNode` is not appended to the version list. The algorithm deallocates the new `VNode` (Line 47) and returns `false`. An unsuccessful `vCAS` also helps the first `VNode` in the version list acquire a valid timestamp.

Helping. As mentioned, a `vRead`, `readVersion` and an unsuccessful `vCAS` all help (by calling `initTS`) to ensure that the timestamp of the `VNode` at the head of the version list is valid before they return. This is necessary to overcome the main difficulty in implementing version lists without locks, i.e., making the following three steps appear atomic: adding a node to the version list, reading a global timestamp, and recording a valid timestamp in the node. (See the discussion of correctness below, and the full correctness proof in [56].)

Initialization. We assume that the constructor (Line 3) for the camera object completes before invoking the constructor (Line 17) for any associated versioned CAS object. We require, as a precondition of any `readVersion`(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 using versioned CAS objects to implement a pointer-based data structure (like a tree or linked list), this constraint will be satisfied naturally.

Correctness. Theorem 2 states the algorithm's properties.

Theorem 2 (Linearizability and Time Bounds)**.** *Algorithm 1 is a linearizable implementation of versioned CAS and camera objects such that*

1. *the number of instructions performed by* `read`, `vCAS`, *and* `takeSnapshot` *is constant, and*
2. *the number of instructions performed by the operation* O.`readVersion`*(ts) is proportional to the number of successful* O.`vCAS` *operations linearized between the linearization point of the* `takeSnapshot` *operation that returned* `ts` *and the start of the* `readVersion`.

A complete proof of Theorem 2 appears in [56]. Below, we just describe the linearization points used in that proof. We say that a timestamp of a `VNode` is *valid* at some point if the `ts` field is not `TBD` at that point; it is *invalid* otherwise.

- For a `vCAS` operation V:
  - If V performs a successful `CAS` on Line 43 adding a node x to the version list, and x's timestamp eventually becomes valid, then V is linearized on Line 24 of the `initTS` method that makes x's timestamp valid.
  - Let x be the node `VHead` points to on Line 38 of V. If V returns on Line 40 or 41, it is linearized either at Line 38 if x's timestamp is valid at that time, or the first step afterwards that makes x's timestamp valid.
  - If V returns `false` on Line 49, then V failed its `CAS` on Line 43. Thus, some other `vCAS` operation changed `VHead` after V read it at Line 38. 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 49 are linearized immediately after V', they can be ordered arbitrarily.
- For a `vRead` operation that terminates, let x be the `VNode` read from `VHead` at Line 34. The `vRead` is linearized at Line 34 if x's timestamp is valid at that time, or at the first step afterwards that makes x's timestamp valid.
- A `readVersion` operation that terminates is linearized at its last step.
- For a `takeSnapshot` operation T that terminates, let ts be the value read from `timestamp` on Line 5, T is linearized when `timestamp` changes from ts to $ts + 1$.

Intuitively, the correctness of an O.`readVersion` operation depends on ensuring that the timestamp associated with a value is *current* (i.e., in the `timestamp` field of the camera object S associated with O) at the linearization point of the `vCAS` that stored the value in O. Hence, we linearize a successful `vCAS` at the time when the successfully installed timestamp was read from S. Thus, a `VNode` x can appear at the head of the version list before the `vCAS` that created x 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 to prove that the linearization points described above are well-defined and within the intervals of their respective operations.

4 Supporting Linearizable Wait-free Queries

We 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. Our approach is general enough to allow transforming many multi-point read-only operations on a sequential data structure into linearizable queries on the corresponding concurrent data structure. To achieve this, we define the concept of a *solo* query, i.e., a query that only reads the shared state, and once invoked, is correct if run to completion without any other process taking steps during its execution. Typically, solo queries can be implemented by adapting standard sequential queries.

The approach works as follows. Each `CAS` or `read` on a CAS object is replaced by a `vCAS` or `vRead` (respectively) on the corresponding versioned CAS object, all of which are associated with the same camera object. To perform a solo query operation q, a process p first executes `takeSnapshot` on the camera object, to obtain a handle ts. Then, for any CAS object that q would have accessed in the data structure, p performs `readVersion`(ts) on the corresponding versioned CAS object. Intuitively, `takeSnapshot` takes a snapshot of shared state, and solo queries then run on this snapshot while other threads may be updating concurrently.

Not all concurrent data structures can support solo queries. Herlihy and Wing [33] describe an array-based queue implementation in which the linearization order of the enqueue operations depends on future dequeue operations. For that algorithm, no solo query is possible. However, for most data structures it is straightforward to implement solo queries. Here we give examples of several concurrent data structures that support solo queries. A thorough treatment of the conditions under which solo queries are sufficient, all the formalism for our approach, necessary proofs, and more examples, are provided in [56].

FIFO Queue. We first consider Michael and Scott's concurrent queue (`MSQ`) [39], which supports atomic enqueues and dequeues, as well as finding the oldest and newest elements. Our scheme additionally provides an easy atomic implementation of more powerful operations such as returning the i-th element, or all elements, etc. The mutable locations in a `MSQ` consist of a `head` pointer, a `tail` pointer, and the `next` pointer of each node in a linked list of elements, pointing from oldest to newest. The `head` points indirectly to the oldest remaining element, and the `tail` points to the newest element, or temporarily to the element behind the newest. The newest element always has a null `next` pointer. After applying our approach, all these pointers become `vCAS` objects, and a `takeSnapshot` operation, T, will atomically capture the state of all of them. Any query can then easily reconstruct the part of the queue state it requires. For example, the i-th query can start at the `head` and follow the list (calling `readVersion` on each node, using the handle returned by T) until it reaches the i-th element in the queue. Each `next` pointer in the linked list is only successfully updated once, so each `readVersion` of a `next` pointer takes constant time. Therefore, for example, finding the i-th element (from the head) in a queue takes time $O(i + c)$ where c denotes the number of successful dequeues between the read of the timestamp by T and the read of the `head`.

Original data structure	Operation	Our Time Bounds	Parameters
Michael Scott Queue [39]	`i-th`(i):	$O(i + c)$	c: number of dequeues concurrent with the query
	`enqueue/dequeue`:	same as original	
Harris Linked List [32]	`range`(s, e):	$O(m + P + c)$	m: number of keys in the linked list
	`multisearch`(L):	$O(m + P + c)$	c: number of inserts and deletes concurrent with the query
	`ith`(i):	$O(i + P + c)$	
	`insert/delete/lookup`:	same as original	P: number of processes
NBBST [25] and CT [15]	`successor`(k)	$O(h + c)$	m: number of keys in the BST
	`multisearch`(L):	$O(\|L\| \times h + c)$	h: height of tree. In the case of CT, $h \in O(\log(m) + P)$
	`range`(s, e):	$O(h + K(s, e) + c)$	$K(s, e)$: number of keys in BST between $[s, e]$
	`height`():	$O(m + c)$	c: number of inserts, deletes, rotations concurrent with the query
	`insert/delete/lookup`:	same as original	

Table 1. Time bounds for various operations on concurrent queues, lists, and BSTs using our snapshot approach. Parameters such as the number of keys in the data structure are measured at the linearization point of the operation.

Sorted Linked List. Harris's data structure [32] maintains an ordered set as a sorted linked list (`HLL`), and supports insertions, deletions, and searches. Our approach adds atomic versions of multi-point query operations, such as range queries, finding the first element that satisfies a predicate, or multisearches (i.e., finding if all or any of a set of keys is in the list). To implement concurrent insertions and deletions properly, `HLL` marks a node before splicing it out of the list. The mark is kept as one bit on the pointer to the next list node. Deletes are linearized when the mark is set. The mutable state comprises the `next` pointers of each link, which contains the mark bit. If these are versioned, a `takeSnapshot` captures the full state. A query can then follow the snapshotted linked list from the head, using `readVersion` on each node; all marked nodes should be skipped.

Time bounds for range query, multisearch and finding the i-th element are given in Table 1. Each insert or delete performs up to two successful `vCAS` operations and each successful `vCAS` may cause a query to traverse an extra version node. So, in the worst case, queries incur an additive cost of c (defined in Table 1). Each query also incurs an additive cost of P since it may encounter up to P marked nodes.

Binary Search Trees. We now consider concurrent binary search trees (BST). Many such data structures have been designed [4, 7, 10, 12, 14, 15, 25, 55, 57]. All the BST structures we looked into work with solo queries allowing for multi-point queries of the same type as in `HLL` (e.g., range queries and multisearches), but potentially much faster since they can often visit a small part of the tree. Queries on the structure of the tree (e.g., finding its height) can also be supported. Here we consider two such trees (which are also used in our experiments in Section 9): the non-blocking binary search trees (`NBBST`) of Ellen *et al.* [25], and the balanced non-blocking chromatic tree (`CT`) of Brown *et al.* [15].

The `NBBST` data structure is an unbalanced BST with the data stored at the leaves and the internal nodes storing keys for guiding searches. Every insertion involves inserting an internal node and a leaf, and similarly a delete removes an internal node and a leaf. The data structure uses a lock-free implementation of locks, "locking" one or two nodes for each insertion or deletion. The locks are implemented by pointing to a descriptor of the ongoing operation, so other threads can help complete the operation if they encounter a lock. This makes the data structure lock-free. The linearization point is at the pointer swing that splices an internal node (along with a child) in or out. Therefore at any time the child pointers of the internal nodes fully define the contents of the data structure. If these child pointers are kept as versioned CAS objects, then a snapshot will capture the required state. The queries can ignore the locks, and therefore the lock pointers, although mutable, do not need to be versioned.

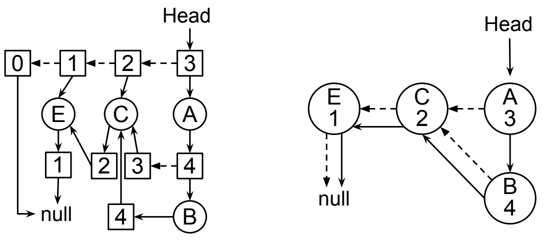
The chromatic tree (`CT`) is a balanced BST that also stores its data at the leaves. It is based on a relaxed version of red-black trees, with colors at each node facilitating rebalancing. Concurrent updates are managed similarly to the `NBBST`. In particular, updates are linearized at a single CAS that adds or removes a key. So, obtaining a snapshot of the tree's child pointers is sufficient to run multi-point queries.

Any query q on `NBBST` or `CT` takes time proportional to the number of nodes it visits plus the write contention of q (i.e., the number of `vCAS` operations concurrent with q on memory locations accessed by q). This assumes q performs `readVersion` on each versioned CAS object at most once. This can be ensured by caching values read from the tree. For the bounds in Table 1, it suffices to show that the number of `vCAS` operations concurrent with q is at most the number of inserts, deletes and rotations concurrent with q. This is because each `vCAS` is either due to a rotation (only applies to `CT`) or is the linearization point of an insert or delete.

Importantly, our snapshot approach maintains the time bounds of all the operations supported by the original data structure. (For example, in the case of `NBBST` and `CT`, the original operations would be insert, delete, and lookup).

5 Avoiding Indirection

In this section and the next, we present ways to optimize our snapshotting approach (and therefore multi-point queries using such snapshots). We present these optimizations in



(a) Indirect vCAS **(b)** Direct vCAS

Figure 2. A simple concurrent linked list using both direct and indirect versioned CAS objects. The state of each results from inserting keys E, C, A, and B (in that order) into an empty list. Circles represent linked list nodes and squares represent VNodes. Numbers represent timestamps and dotted arrows represent version pointers (nextv pointers).

terms of a concurrent data structure D to which we add snapshots and use them to run queries from a set Q. We denote by D' the version of D that also supports the queries in Q. Some CAS objects in D need not be versioned in D' because they are not accessed by any of the queries.

Algorithm 1 has a level of indirection even when accessing the most recent version of a vCAS object since it requires first accessing the head of the version list, and then the object it points to. Figure 2(a) illustrates an example of a linked list updated as described in Algorithm 1 along with its version lists. Here we discuss how this indirection can be avoided. Figure 2(b) illustrates the linked list after applying the optimization (more details later). This optimization has some restrictions, which we define first.

A *versioned node* is a node that versioned CAS objects can point to directly. A *history* H is a sequence of atomic steps executed by an algorithm starting from an initial state. We say that a versioned node is *recorded* in H if a pointer to it is the newV parameter of a *successful* vCAS (on any versioned CAS object) in H.

Definition 3 (Recorded-once)**.** D' is *recorded-once* if for every history of D', the following hold: (1) every versioned node is recorded at most once, (2) the newV parameter of each vCAS is a pointer to a versioned node, (3) vCAS operations with the same newV parameter must have the same oldV parameter, and (4) versioned CAS objects are initialized with null or a pointer to a previously recorded node.

This property allows us to overload a versioned node as both a node and a link in a version list, thus avoiding the indirection. Conditions 2 to 4 are relatively natural to satisfy, so it is Condition 1 that is most important.

Our approach works as follows. For each versioned CAS object O that stores a pointer to a node in D', instead of creating a new VNode to store the version pointer and the timestamp, we store this information directly in the node pointed to by O. To do this, we extend each node with two extra fields, ts and nextv, and modify Algorithm 1 accordingly. The resulting algorithm is described in Algorithm 3 and we call it the *direct* implementation of versioned CAS objects. Naturally, we call Algorithm 1 the *indirect* implementation. Figure 2 gives an example using both versions.

The correctness of Algorithm 3 depends heavily on the recorded-once property (Definition 3). Condition 1 ensures that every versioned node appears as a non-tail element of a version list at most once. Note, however, that a versioned node x can appear as the tail of the version lists of an arbitrary number of vCAS objects since each can set their initial value to x. In the example of Figure 2 the node C is both the tail of the version list from B and a non-tail for the version list from the head. A timestamp is set on a versioned node when it is recorded (Line 31, or by someone helping), and by Condition 1 this means it is set at most once. Furthermore by Condition 4 a node is not used as an initial value until its timestamp is set, meaning that all timestamps stored in D' are already set or in the process of being set (possibly by helping). Condition 2 is required to ensure we have somewhere to record the timestamp and next node in the version list (i.e., newV needs nextv and ts fields). Condition 3 ensures that all vCAS operations with the same newV attempt to write the same version pointer into newV on Line 29.

The direct implementation can be applied to concurrent data structures for which, at any point in time, every object has at most one pointer to it. Examples include tree data structures where pointers go from parents to children, or singly-linked lists. However, this can involve slight modifications to the original concurrent algorithm. For example, a node is being pointed to by one object and is being moved to be pointed to by another object then it would be recorded more than once. This can happen during a delete operation in HLL [32] and NBBST [25]. To avoid this, the object can be copied and a pointer to the new copy can be written into the new location. This modification should be done with care to preserve correctness. We apply this transformation in our NBBST implementation (Section 8).

We note that the recorded-once property is actually fully general, albeit possibly requiring adding back a level of indirection in the data structure itself. Consider the following construction on an arbitrary concurrent data structure D based on CAS objects. We introduce a version-link type which is a versioned node holding a single pointer to a direct value of another object from D. All CAS objects in D are replaced with versioned CAS objects in the construction of D', which will now point to a version link, and then indirectly to the value from D. Now whenever applying a CAS in D, in D' we create a new version link, put the new value into the link, and apply a vCAS from the old version link to the new one (some care needs to be taken to check for equality, as done in lines 40 and 41 of Algorithm 1). Whenever D initializes a CAS object, in D', we initialize the corresponding versioned

```
1   class Node {
2     /* other fields of the Node class */
3     int ts;   // initially TBD
4     Node* nextv; };
5   class DirectVersionedCAS {
6     Node* Head; Camera* S;
7     DirectVersionedCAS(Node* n, Camera* s) {
8       Head = n; S = s; initTS(n); }
9     void initTS(Node* n) {
10      if(n != NULL && n->ts == TBD) {
11        int curTS = S->timestamp;
12        CAS(&(n->ts), TBD, curTS); } }
```

```
13    Node* vRead() {
14      Node* head = Head;
15      initTS(head);
16      return head; }
17    Node* readVersion(int ts) {
18      Node* node = Head;
19      initTS(node);
20      while(node != NULL &&
21                 node->ts > ts)
22        node = node->nextv;
23      return node; }
```

```
24    bool vCAS(Node* oldV, Node* newV) {
25      Node* head = Head;
26      initTS(head);
27      if (head != oldV) return false;
28      if (newV == oldV) return true;
29      newV->nextv = oldV;
30      if(CAS(&Head, head, newV)) {
31        initTS(newV);
32        return true; }
33      else {
34        initTS(Head);
35        return false; } } };
```

Algorithm 3. Linearizable implementation of a versioned CAS object without indirection.

CAS object to null, create a new version link with the desired initial value, and write a pointer to the new version link into the versioned CAS object with a vCAS. This construction satisfies all the recorded once properties—the version link is the newV of exactly one vCAS, it is a versioned node, and all versioned CAS objects are initialized with null.

Although this construction reintroduces a level of indirection, it indicates that it should be relatively simple to have hybrid data structures that use indirection where needed, and not when not needed. It also means an implementation of the indirect variant can be built on top of the direct variant.

6 Other Optimizations

In this section, we present additional optimizations that work for both the direct versioned CAS algorithm from Section 5 and the indirect algorithm from Section 3.

Removing redundant versions. If snapshot operations are infrequent, many consecutive nodes in a version list may have the same timestamp. Since only the most recent such node is needed by readVersion, we can save space by storing only nodes that have distinct timestamps. We can guarantee that all nodes (other than the first two) in a version list have distinct timestamps by splicing out the second node if it has the same timestamp as the first. This splicing step is done after a successful vCAS operation sets the timestamp of the newly added node (i.e., after Line 44 of Algorithm 1). Note that a successful vCAS operation might stall between setting the timestamp and splicing out the next version list node, so each new vCAS operation has to help older vCAS operations splice out the second node in a version list if it has the same timestamp as the first. This helping step is done between Lines 39 and 40 in Algorithm 1. When using this optimization in Algorithm 3, Line 29 should be modified to update nextv with a CAS so that this optimization does not get undone.

Avoiding contention. Although takeSnapshot only uses a single CAS, this can still be a bottleneck. To reduce contention on this CAS, we observe that C.takeSnapshot must only ensure that C's timestamp is incremented at some point during its execution interval. Thus, C.takeSnapshot can use exponential backoff to wait for another process to increment C's timestamp. After waiting, if no process has done so, then C.takeSnapshot tries to do the increment itself.

7 Memory Reclamation

To add memory reclamation to our snapshotting approach, we use Epoch Based Memory Reclamation (EBR) [31]. EBR splits an execution into epochs by utilizing a global epoch counter EC (with initial value 1). Interestingly, with our direct implementation of versioning we are able to collect exactly the same nodes as can be collected in non-versioned EBR—i.e., all nodes that were freed prior to the last two epochs. There can still be some additional memory overhead for versioning, however, due to the extra nextv and ts field in each node, and, as mentioned in Section 5, the need in some algorithms to allocate extra nodes when deleting.

EBR supports three operations, BeginOp, EndOp and Retire. A process p executing a BeginOp operation simply reads EC and announces the value read as its current epoch. An EndOp by p clears any previously announced epoch by p. EBR maintains a per-process *limbo list* of objects for each epoch. An object is added to the limbo list of the most recent epoch whenever it is passed to Retire. When all processes have announced an epoch number that is at least b (where b is any integer greater than 2), the limbo list associated with epoch $b-2$ is collected, and the global epoch counter, EC, is incremented. In this way, EBR maintains only the limbo lists of the last three epochs. For our experiments, we use an efficient variation of EBR called DEBRA [16].

Using the notation of Section 5, let D be a concurrent data structure and let D' be the snapshottable version of D that supports a set of query operations Q in addition to the operations supported by D. In languages *without* automatic garbage collection, we can support memory reclamation for D' as follows. BeginOp is invoked at the beginning of each operation, and EndOp is invoked at the end. Retire is called on a node whenever it is removed from the current version of the data structure. If D' uses the indirect implementation from Algorithm 1, before returning from a successful vCAS on Line 45, we also have to Retire the VNode pointed to by local variable head. Furthermore, when a data structure node y is retired, we have to Retire all the VNodes at the head of y's versioned CAS objects. The key observation is that all operations from D' (including query operations) only access nodes that were in the current version of the data structure at some point during the operation's execution interval. This means that whenever EBR determines that a node is retired before the start of the earliest live operation, we can free the node without first unlinking it from any version list because it can no longer be accessed by any live operation.

In languages *with* automatic garbage collection, we first use EBR to unlink nodes from version lists and then rely on the garbage collector to clean up any unreachable nodes. We modify the EBR algorithm so that when a limbo list is collected, for each node x in the limbo list, instead of freeing x, we set its version list pointer, x->nextv, to null. We can think of this as retiring a version list pointer rather than a node. A query operation q working on a snapshot of the data structure protects any version list pointers it may access by calling BeginOp before taking a snapshot and EndOp when it is done using the snapshot. Since q can access version list pointers only of those nodes that were added during q's execution interval, it is safe to retire a version list pointer as soon as the pointer is added to the data structure. EBR ensures that this pointer is not set to null until all operations that were live when it was retired terminate. This means that in Algorithm 1 (Algorithm 3), we retire the pointer newN->nextv (newV->nextv, respectively) before returning from a successful vCAS on Line 45 (Line 32, respectively).

8 Implementation

We implemented our snapshotting approach in both Java and C++. Using the implementations, we then implemented snapshottable versions of three existing lock-free external BST data structures (see details below). We use all the optimizations discussed in Sections 5 and 6. To apply our approach on top of these tree data structures, we make each node in the data structure *versionable* by adding a timestamp and a version pointer field to it, use direct versioned CAS objects for child pointers, and modify the data structure to be recorded-once if necessary. All versioned CAS objects are associated with the same camera object so we avoid storing a pointer to a camera object in each versioned CAS object.

The key and value fields of each node are immutable in all three tree data structures. So, a snapshot of the child pointers completely defines the contents of the tree and can be used to answer arbitrary multi-point queries. For our implementation in Java, we implemented the four queries in Table 2, and for our implementation in C++, we implemented range queries. All are linearizable. We reclaim memory using the EBR based technique described in Section 7. Our code is publicly available on GitHub[4].

Base Data Structures. We applied our snapshotting approach to the two BSTs described in Section 4, NBBST and CT, and to a lock-free unbalanced BST [4] that does not support linearizable range queries. For the first two, we used Brown's Java implementations [13]. For the third, we used the C++ implementation by Arbel-Raviv and Brown [4].

Batching. Previous work has shown that the performance of concurrent BSTs is improved by batching keys. We therefore applied the same batching technique from PNB-BST [28] and LFCA [57] to our Java implementations, storing up to 64 key-value pairs in each leaf (see [28] for more details). We did not apply batching in our C++ code since it was also not used by the C++ implementation [4] we compared with.

Recorded-Once. The recorded-once requirement is naturally satisfied by CT and the BST from [4], but not by NBBST because the delete operation uses CAS to swing a pointer to a node that is already in the data structure. To avoid this, our implementation copies the node and swings the pointer to this new copy instead. This requires some extra marking and helping steps to preserve correctness and lock-freedom.

Names. BST-64 and CT-64 are the non-snapshotted Java BSTs (with batching). VcasBST-64 and VcasCT-64 are our snapshotted versions. BST is the non-snapshotted C++ BST, while VcasBST is our snapshotted version.

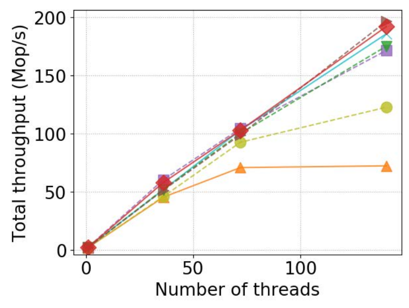
9 Experimental Evaluation

In this section, we provide our experimental analysis, which has two main goals: first, to understand the overhead that our approach introduces to concurrent data structures which originally did not support multi-point queries, and second, to compare the performance of our approach to that of state-of-the-art alternatives which support atomic range queries.

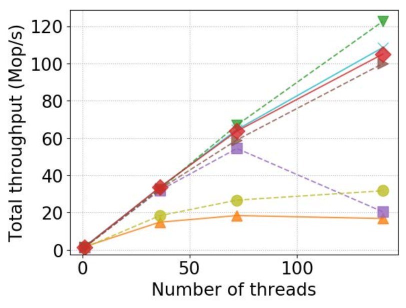
Other Structures that Support Range Queries. We compare with several state-of-the-art dictionary data structures: SnapTree [12], KiWi [7], LFCA [57], PNB-BST [28], KST [14], and EpochBST [4] using code provided by their respective authors. Arbel-Raviv and Brown [4] presented several ways to add range queries to concurrent data structures, implemented in C++. We use EpochBST to refer to their most efficient range queryable lock-free BST. Note that EpochBST and VcasBST add range queries to the same initial BST. All the other data structures are written in Java. They are all lock-free except SnapTree, which uses fine-grained locking. We classify KiWi, SnapTree, and VcasCT-64 as *balanced* data structures because they have logarithmic search time in the absence of contention, and the others as *unbalanced*. For the k-ary tree (KST), we use $k = 64$ which was shown to perform well across a variety of workloads [14]. We used batch size 64 for VcasBST-64 and VcasCT-64, as well as for LFCA and

[4]https://github.com/yuanhaow/vcaslib

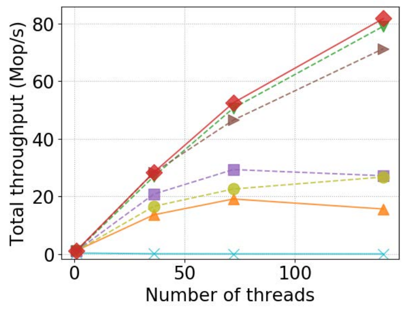
Balanced data structures:
KIWI
SnapTree
VcasCT-64
Unbalanced data structures:
KST
PNB-BST
LFCA
VcasBST-64



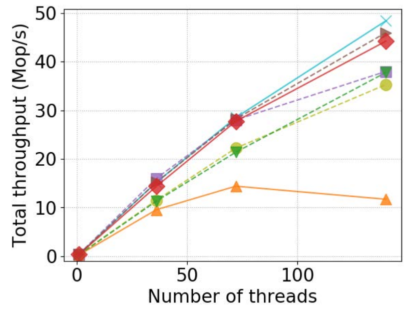
(a) Lookup heavy - 100K Keys: 3%ins-2%del-95%find-0%rq



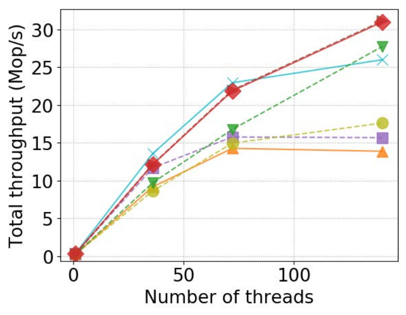
(b) Update heavy - 100K Keys: 30%ins-20%del-50%find-0%rq



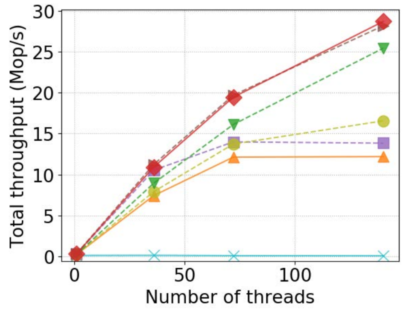
(c) Update heavy with RQ - 100K Keys: 30%ins-20%del-49%find-1%rq-1024size



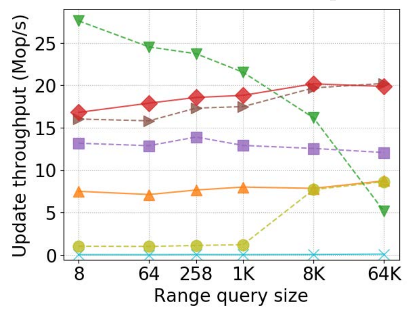
(d) Lookup heavy - 100M Keys: 3%ins-2%del-95%find-0%rq



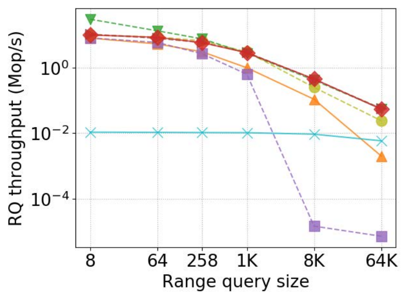
(e) Update heavy - 100M Keys: 30%ins-20%del-50%find-0%rq



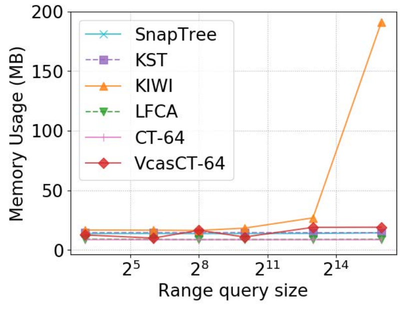
(f) Update heavy with RQ - 100M Keys: 30%ins-20%del-49%find-1%rq-1024size



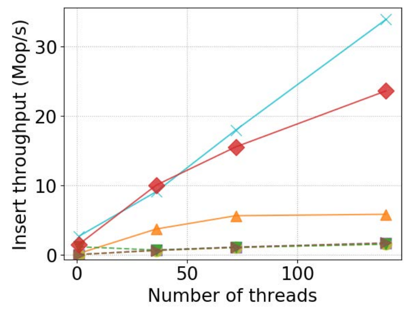
(g) Update Throughput - 100K Keys: 36 Update Threads, 36 RQ Threads



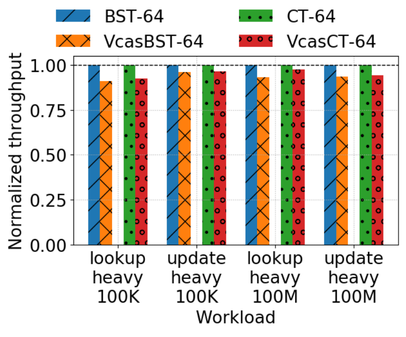
(h) RQ Throughput - 100K Keys: 36 Update Threads, 36 RQ Threads



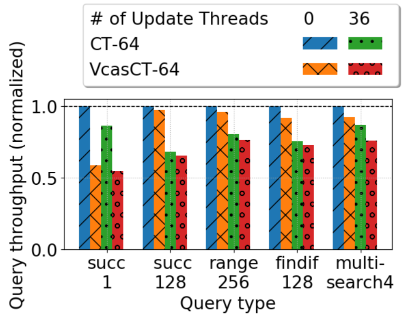
(i) Memory Usage - 100K keys: 36 Update Threads, 36 RQ Threads



(j) Insert Only, Sorted Sequence



(k) Overhead of Vcas, 140 threads, measured across various workloads

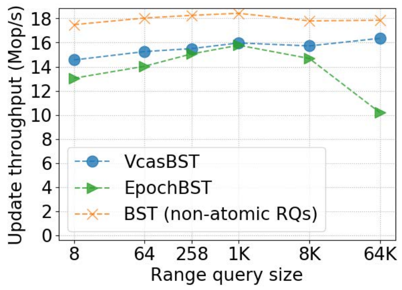


(m) Throughput of atomic queries on VcasCT-64

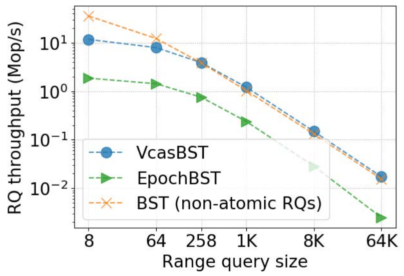
Figure 4. Java experiments.

Query	Definition	Parameters in Figure 4m
`range`(s, e):	All keys in range $[s, e]$	`range256`: $e = s + 256$
`succ`(k, a)	The first a key-values with key greater than k	`succ1`: $a = 1$, or `succ128`: $a = 128$
`findif`(s, e, f) [19]:	The first key-value pair in range $[s, e)$	`findif128`: $f(k) = (k \ mod\ 128 \ is\ 0)$
`multisearch`(L):	For a list of keys in L, return their values (null if not found)	`multisearch4`: $\|L\| = 4$

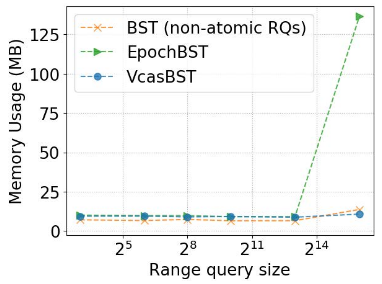
Table 2. The multi-point queries and their parameters we use in experiments.



(a) Update Throughput - 100K Keys: 36 Update Threads, 36 RQ Threads



(b) RQ Throughput - 100K Keys: 36 Update Threads, 36 RQ Threads



(c) Memory Usage - 100K keys: 36 Update Threads, 36 RQ Threads

Figure 5. C++ experiments.

`PNB-BST`. This batch size has been shown to yield good range query performance for `LFCA` and `PNB-BST` in [28, 57].

We also applied the contention avoiding technique from Section 6 to `KiWi`, `PNB-BST`, and `EpochBST` because we found that it improves their performance in some workloads by reducing contention on the global timestamp.

Setup. Our experiments ran on a 72-core Dell R930 with 4x Intel(R) Xeon(R) E7-8867 v4 (18 cores, 2.4GHz and 45MB L3 cache), and 1Tbyte memory. Each core is 2-way hyperthreaded giving 144 hyperthreads. We used `numactl -i all`, evenly spreading the memory pages across the sockets in a round-robin fashion. The machine runs Ubuntu 16.04.6 LTS. The C++ code was compiled with g++ 9.2.1 with `-O3`. Jemalloc was used for scalable memory allocation. For Java, we used OpenJDK 11.0.5 with flags `-server`, `-Xms300G` and `-Xmx300G`. The latter two flags reduce interference from Java's GC. We report the average of 5 runs, each of 5 seconds. For Java we also pre-ran 5 runs to warm up the JVM. The variance is small in almost all tests. Experiments showed that the throughput of our approach in longer experiments (lasting up to two hours) is similar to that reported below.

Workload. We vary four parameters: data structure size n, operation mix, range query size *rqsize*, and number of threads. In most experiments, we prefill a data structure with either $n = 100K$ or $n = 100M$ keys. These sizes show the performance when fitting and not fitting into the L3 cache. Keys for operations, and in the initial tree, are drawn uniformly at random from a range $[1, r]$, where r is chosen to maintain the initial size of the data structure. For example, for $n = 100K$ and a workload with 30% inserts and 20% deletes, we use $r = n \times (30 + 20)/30 \approx 166K$. We perform a mix of operations, represented by four values, *ins*, *del*, *find*, *rq*, which are the probabilities for each thread to execute an `insert`, `delete`, `find`, or `range`, respectively. Unbalanced trees can be balanced in expectation using uniformly random keys, so we also run a workload with keys inserted in sorted order.

Scalability. Figures 4a-4f show scalability (in Java) under a variety of workloads. Note that in Figures 4c and 4f, although range queries are only performed with 1% probability, they occupy a significant fraction of execution time.

Generally, `VcasCT-64` and `VcasBST-64` (our two implementations), and `LFCA` have the best (almost-linear) scalability across all workloads. `LFCA` outperforms our implementation in Figure 4b, but it is consistently slower in the 100M-key experiments (Figures 4d-f). `SnapTree` is competitive with our trees in the absence of range queries, but it has no scalability with range queries due to its lazy copy-on-write mechanism. Overall, `VcasCT-64` is always among the top three algorithms and in most cases has the best performance.

Varying Range Query Size. We show the effect of varying range query size in Figures 4g and 4h (Java), and Figures 5a and 5b (C++), in which 36 dedicated threads ran range queries and 36 ran updates. Each update thread performs 50% inserts and 50% deletes on a data structure initialized to 100K keys. To better understand the cost of updates and range queries, we plot the throughput of each operation separately.

In Figure 4g, `PNB-BST` has low update throughput when *rqsize* $\leq$ 1024. This is because its update operations are forced to abort and restart whenever a new range query begins, and thus decreasing range query size lowers update throughput. `KST` performs decently in most workloads except when each range query covers a significant fraction of the key range. This is because their range query performs a double collect of the desired range and is forced to restart if it sees an update in that range.

Data structures that increment a global timestamp with every range query become bottlenecked by this increment when range queries are frequent. This applies to our trees as well as PNB-BST, KiWi, and EpochBST. Consequently, with *rqsize* = 8, LFCA has 3x faster range queries when compared to our trees (Figure 4h). However, LFCA avoids using a global timestamp by having update operations help ongoing range query operations. This helping becomes more frequent and more costly when *rqsize* is large, as shown in Figure 4g. For *rqsize* = 64*K* (about a third of the key range), the update throughput of our trees is 4x faster than LFCA. Other than LFCA, all the other implementations have mostly stable update throughput with varied range size, among which VcasCT-64 has the best overall performance.

Figures 5a and 5b compare the performance of the C++ version of VcasBST with that of EpochBST. Range queries on VcasBST are 5–7x faster than EpochBST. This is because a range query on EpochBST has to visit three nodes in the retired list for each concurrent delete. Thus, EpochBST visits 1.5–5.5x more nodes in range queries than VcasBST. For updates, VcasBST is at least as fast as EpochBST, and up to 60% faster on the largest range query size.

Sorted Workload. In Figure 4j, we test the Java implementations under a sorted workload. We insert an array of sorted keys into an initially empty tree by splitting the array into chunks of size 1024 and placing the chunks on a shared work queue; when a thread runs out of work, it grabs a new chunk from the head of the shared work queue. As expected, the balanced trees, VcasCT-64, KiWi and SnapTree, outperform the unbalanced ones. On 140 threads, SnapTree is 1.4x faster than VcasCT-64, which is in turn 4.1x faster than KiWi.

Overhead of Our Approach. In Figure 4k, we compare the throughput of our Java implementations VcasBST-64 and VcasCT-64 with the original data structures, BST-64 and CT-64, using 140 threads. The numbers in Figure 4k are normalized to the throughput of BST-64 and CT-64 to make the overheads easier to read. The overall overhead of our approach is low, ranging between 2.7% and 9.1% depending on the workload. This overhead includes the time for epoch-based memory management and the cost of using vCAS and vRead. For VcasBST-64, it also includes the actions we take to ensure that deletes satisfy the recorded-once property. The overhead is low because a vRead rarely sees a node with timestamp TBD, and therefore rarely performs a CAS. While a vRead sometimes incurs an extra cache miss to read the timestamp of the next node (Line 10 of Algorithm 3), in most cases, this node will later be accessed by the operation that invoked vRead anyway, so the overall number of cache misses is not increased by much.

We also measure the overhead of our multi-point queries, range, succ, findif, and multisearch, with parameters shown in Figure 4m. We compare throughputs for VcasCT-64 with *non-atomic* multi-point queries on the original CT-64, which simply run their sequential algorithms (and are not linearizable). Non-atomic multisearch, for example, simply calls find for each key. Figure 4m shows the cost that our approach has to pay to provide query atomicity.

All queries other than succ1 exhibit low overhead: they are between 2.9% and 12.8% slower than their non-atomic counterparts. For succ1, our scheme exhibits larger overheads (36.8-41.4%) due to the counter bottleneck when the query size is small.

Memory Usage Figures 4i and 5c show memory usage graphs for the Java and C++ data structures, respectively. In our Java experiments, we measured the amount of heap memory in use after Java's garbage collector cleans up all unreachable objects. We found that VcasCT-64's memory usage is within a factor of 2.2 of both CT-64 and LFCA, which tie for having the smallest memory footprints. We omitted PNB-BST from Figure 4i because it does not allow for garbage collection and uses significantly more memory than the rest.

For the C++ experiments, we measured memory usage by multiplying the number of allocated nodes by the size of each node. As discussed in Section 7, VcasBST has little memory overhead with respect to the non-versioned BST because they both use EBR and keep around approximately the same number of nodes. Most of the overhead comes from storing an extra timestamp and version pointer in each node.

We tried running Figures 4i and 5c in an oversubscribed case by only using 64 out of the 144 hyperthreads on our machine. We found that the memory usage of EBR based data structures (VcasBST, VcasBST-64, VcasCT-64, BST, and EpochBST) became several times higher. This is because whenever a thread gets swapped out in the middle of an operation, nothing can be collected until it is scheduled again.

Summary. Overall, our snapshot approach has low overhead and, despite its generality, performs well compared to existing special-purpose data structures. In particular, VcasCT-64 had the best overall throughput among all the range queryable Java data structures we tested. VcasBST-64 is also competitive on uniform workloads.

10 Conclusion and Discussion

In this paper, we show a simple and efficient approach for snapshotting and supporting multi-point queries on a large class of concurrent data structures. Our paper focuses mostly on CAS based data structures, but these ideas can be extended to work for LL/SC based data structures as well.

Acknowledgments

We thank the anonymous referees for their comments as well as Faith Ellen for the insightful discussion. Funding was provided by the Natural Sciences and Engineering Research Council of Canada as well as National Science Foundation grants CCF-1901381, CCF-1910030, and CCF-1919223.

References

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1993. Atomic Snapshots of Shared Memory. *J. ACM* 40, 4 (Sept. 1993), 873–890. https://doi.org/10.1145/153724.153741

[2] Archita Agarwal, Zhiyu Liu, Eli Rosenthal, and Vikram Saraph. 2017. Linearizable Iterators for Concurrent Data Structures. *CoRR* abs/1705.08885 (2017). arXiv:1705.08885 http://arxiv.org/abs/1705.08885

[3] James H. Anderson. 1994. Multi-Writer Composite Registers. *Distributed Comput.* 7, 4 (1994), 175–195. https://doi.org/10.1007/BF02280833

[4] Maya Arbel-Raviv and Trevor Brown. 2018. Harnessing Epoch-Based Reclamation for Efficient Range Queries. In *Proc. 23rd ACM Symposium on Principles and Practice of Parallel Programming*. 14–27. https://doi.org/10.1145/3178487.3178489

[5] Hagit Attiya, Rachid Guerraoui, and Eric Ruppert. 2008. Partial Snapshot Objects. In *Proc. 20th Symposium on Parallelism in Algorithms and Architectures*. 336–343. https://doi.org/10.1145/1378533.1378591

[6] Hillel Avni, Nir Shavit, and Adi Suissa. 2013. Leaplist: Lessons Learned in Designing TM-Supported Range Queries. In *Proc. 2013 ACM Symposium on Principles of Distributed Computing*. 299–308. https://doi.org/10.1145/2484239.2484254

[7] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. 2020. KiWi: A Key-Value Map for Scalable Real-Time Analytics. *ACM Trans. Parallel Comput.* 7, 3, Article 16 (June 2020), 28 pages. https://doi.org/10.1145/3399718

[8] Naama Ben-David, Guy E. Blelloch, Yihan Sun, and Yuanhao Wei. 2019. Multiversion Concurrency with Bounded Delay and Precise Garbage Collection. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 241–252.

[9] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control - Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (Dec. 1983), 465–483. https://doi.org/10.1145/319996.319998

[10] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 253–264.

[11] Alex Brodsky and Faith Ellen Fich. 2004. Efficient Synchronous Snapshots. In *ACM Symposium on Principles of Distributed Computing*. 70–79. https://doi.org/10.1145/1011767.1011778

[12] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A Practical Concurrent Binary Search Tree. In *Proc. 15th ACM Symposium on Principles and Practice of Parallel Programming*. 257–268.

[13] Trevor Brown. [n.d.]. Java Lock-Free Data Structure Library. Available from https://bitbucket.org/trbot86/implementations/src/master/java.

[14] Trevor Brown and Hillel Avni. 2012. Range Queries in Non-blocking k-ary Search Trees. In *Proc. 16th International Conference on Principles of Distributed Systems (LNCS, Vol. 7702)*. 31–45.

[15] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A General Technique for Non-Blocking Trees. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 329–342. https://doi.org/10.1145/2555243.2555267

[16] Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way. In *ACM Symposium on Principles of Distributed Computing*. 261–270.

[17] João Cachopo and António Rito-Silva. 2006. Versioned Boxes as the Basis for Memory Transactions. *Science of Computer Programming* 63, 2 (2006), 172–185.

[18] Bapi Chatterjee. 2017. Lock-free Linearizable 1-Dimensional Range Queries. In *Proc. 18th Intl Conf. on Dist. Computing and Networking*. 9:1–9:10. https://doi.org/10.1145/3007748.3007771

[19] Cplusplus.com. [n.d.]. Documentation for std::find_if. http://www.cplusplus.com/reference/algorithm/find$_i$f/ accessed 31 December, 2020.

[20] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1243–1254. https://doi.org/10.1145/2463676.2463710

[21] Thomas Dickerson. 2020. Adapting Persistent Data Structures for Concurrency and Speculation. arXiv:2003.07395 [cs.DC]

[22] Nuno Diegues and Paolo Romano. 2015. Time-Warp: Efficient Abort Reduction in Transactional Memory. *ACM Trans. Parallel Comput.* 2, 2, Article 12 (June 2015), 44 pages. https://doi.org/10.1145/2775435

[23] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. 1989. Making Data Structures Persistent. *J. Computer and System Sciences* 38, 1 (1989), 86–124.

[24] James R. Driscoll, Daniel D. K. Sleator, and Robert E. Tarjan. 1994. Fully Persistent Lists with Catenation. *J. ACM* 41, 5 (Sept. 1994), 943–959. https://doi.org/10.1145/185675.185791

[25] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-Blocking Binary Search Trees. In *ACM Symp. on Principles of Distributed Computing*. 131–140. See also Technical Report CSE-2010-04, EECS Department, York University, 2010.

[26] Panagiota Fatourou and Nikolaos D. Kallimanis. 2007. Time-Optimal, Space-Efficient Single-Scanner Snapshots & Multi-Scanner Snapshots Using CAS. In *ACM Symposium on Principles of Distributed Computing*. 33–42. https://doi.org/10.1145/1281100.1281108

[27] Panagiota Fatourou, Yiannis Nikolakopoulos, and Marina Papatriantafilou. 2017. Linearizable Wait-Free Iteration Operations in Shared Double-Ended Queues. *Parallel Processing Letters* 27, 2 (2017), 1–17.

[28] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. 2019. Persistent Non-Blocking Binary Search Trees Supporting Wait-Free Range Queries. In *Proc. 31st ACM Symposium on Parallelism in Algorithms and Architectures*. 275–286. https://doi.org/10.1145/3323165.3323197

[29] Sérgio Miguel Fernandes and João Cachopo. 2011. Lock-Free and Scalable Multi-Version Software Transactional Memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 179–188. https://doi.org/10.1145/1941553.1941579

[30] Faith Ellen Fich. 2005. How Hard Is it to Take a Snapshot?. In *Proc. 31st Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM) (LNCS, Vol. 3381)*. 28–37.

[31] Keir Fraser. 2004. *Practical Lock-freedom.* Technical Report. University of Cambridge, Computer Laboratory.

[32] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-Blocking Linked-Lists. In *International Symposium on Distributed Computing*. 300–314.

[33] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[34] Damien Imbs and Michel Raynal. 2012. Help When Needed, but No More: Efficient Read/Write Partial Snapshot. *J. Parallel and Distrib. Comput.* 72, 1 (2012), 1–12.

[35] Prasad Jayanti. 2002. f-Arrays: Implementation and Applications. In *Proc. 21st Symposium on Principles of Distributed Computing*. 270–279. https://doi.org/10.1145/571825.571875

[36] Prasad Jayanti. 2005. An Optimal Multi-Writer Snapshot Algorithm. In *ACM Symposium on Theory of Computing (STOC)*. 723–732. https://doi.org/10.1145/1060590.1060697

[37] Nikolaos D. Kallimanis and Eleni Kanellou. 2015. Wait-free Concurrent Graph Objects with Dynamic Traversals. In *Proc. 19th International Conference on Principles of Distributed Systems (Leibniz International Proceedings in Informatics).*

[38] Priyanka Kumar, Sathya Peri, and K. Vidyasankar. 2014. A TimeStamp Based Multi-version STM Algorithm. In *Intl Conf. on Dist. Computing and Networking*. 212–226.

[39] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms.

In *ACM Symposium on Principles of Distributed Computing*. 267–275. https://doi.org/10.1145/248052.248106

[40] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-version Concurrency Control for Main-Memory Database Systems. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 677–689.

[41] Yiannis Nikolakopoulos, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. 2015. A Consistency Framework for Iteration Operations in Concurrent Data Structures. In *Proc. IEEE International Parallel and Distributed Processing Symposium*. 239–248.

[42] Yiannis Nikolakopoulos, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. 2015. 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*. Springer, 358–369.

[43] Oracle. [n.d.]. Java Weakly Consistent Iterators. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html #Weakly accessed 31 December 2020.

[44] Christos H Papadimitriou and Paris C Kanellakis. 1984. On Concurrency Control by Multiple Versions. *ACM Transactions on Database Systems* 9, 1 (1984), 89–99.

[45] Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. 2011. SMV: Selective Multi-Versioning STM. In *Proc. International Symposium on Distributed Computing*. 125–140.

[46] Dmitri Perelman, Rui Fan, and Idit Keidar. 2010. On Maintaining Multiple Versions in STM. In *ACM Symp. on Principles of Dist. Computing*. 16–25.

[47] Erez Petrank and Shahar Timnat. 2013. Lock-Free Data-Structure Iterators. In *Proc. 27th Intl Symposium on Distributed Computing*. 224–238. https://doi.org/10.1007/978-3-642-41527-2$_1$6

[48] Dan R. K. Ports and Kevin Grittner. 2012. Serializable Snapshot Isolation in PostgreSQL. *Proc. of the VLDB Endowment* 5, 12 (Aug. 2012), 1850–1861. https://doi.org/10.14778/2367502.2367523

[49] Aleksandar Prokopec. 2015. Snapqueue: Lock-free Queue with Constant Time Snapshots. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*. 1–12.

[50] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent Tries with Efficient Non-blocking Snapshots. In *Proc. 17th ACM Symposium on Principles and Practice of Parallel Programming*. 151–160. https://doi.org/10.1145/2145816.2145836

[51] D. Reed. 1978. *Naming and synchronization in a decentralized computer system*. Technical Report LCS/TR-205. EECS Dept., MIT.

[52] Torvald Riegel, Pascal Felber, and Christof Fetzer. 2006. A Lazy Snapshot Algorithm with Eager Validation. In *International Symposium on Distributed Computing*. Springer, 284–298.

[53] Neil Sarnak and Robert E Tarjan. 1986. Planar Point Location Using Persistent Search Trees. *Commun. ACM* 29, 7 (1986), 669–679.

[54] Yihan Sun, Guy E. Blelloch, Wan Shen Lim, and Andrew Pavlo. 2019. On Supporting Efficient Snapshot Isolation for Hybrid Workloads with Multi-versioned Indexes. *Proc. of the VLDB Endowment* 13, 2 (2019), 211–225.

[55] Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. 2018. PAM: Parallel Augmented Maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 290–304.

[56] Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. 2020. Constant-Time Snapshots with Applications to Concurrent Data Structures. arXiv:2007.02372 [cs.DC]

[57] Kjell Winblad, Konstantinos Sagonas, and Bengt Jonsson. 2018. Lock-Free Contention Adapting Search Trees. In *Proc. 30th Symposium on Parallelism in Algorithms and Architectures*. 121–132. https://doi.org/10.1145/3210377.3210413

[58] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-memory Multi-version Concurrency Control. *Proceedings of the VLDB Endowment (PVLDB)* 10, 7 (March 2017), 781–792.

A Artifact Evaluation Appendix

A.1 Abstract

This artifact contains the source code and scripts to reproduce all the graphs in Section 9, as well as an easy-to-use library implementation of our snapshotting approach.

A.2 Artifact check-list (meta-information)

- **Algorithm:** all the algorithms from Figure 4
- **Program:** microbenchmarks
- **Compilation:** g++ 9.3.0 and OpenJDK 11.0.9.1
- **Binary:** binary not included
- **Run-time environment:** Ubuntu 16.04.6 LTS
- **Hardware:** any multi-core machine with at least 60 GB main memory.
- **Output:** graphs from Section 9 as PNG files.
- **Experiments workflow:** one script for compiling the experiments and one script for generating all the graphs.
- **Disk space required (approximately):** 46 MB
- **Time needed to prepare workflow:** approximately 10 minutes
- **Time needed to complete experiments:** approximately 20 hours
- **Publicly available:** yes
- **Code licenses:** MIT License

A.3 Description

A.3.1 How delivered Available as open source under the MIT software license: https://github.com/yuanhaow/vcaslib.

A.3.2 Hardware dependencies To accurately reproduce our experimental results, a multi-core machine with at least 300 GB of main memory is recommended. When using less than 300 GB, you may see some small overhead due to Java's garbage collector running more frequently. At least 60 GB of main memory is required to run the PNB-BST in Figure 4 because it never garbage collects nodes.

A.3.3 Software dependencies Our artifact is expected to run correctly under a variety of Linux x86_64 distributions. Our Java experiments require OpenJDK 11 and our C++ experiments were compiled using g++ 9. For scalable memory allocation in C++, we used jemalloc 5.2.1 (https://github.com/jemalloc/jemalloc/releases/download/5.2.1/jemalloc-5.2.1.tar.bz2). Our scripts for running experiments and drawing graphs require a Python 3 installation with mathplotlib. We used the `numactl` command to evenly interleave memory across the NUMA nodes.

A.3.4 Data sets None.

A.4 Installation

Source code can be complied by running `./compile_all.sh`.

A.5 Experiment workflow

After compiling, run `./generate_graphs_from_paper.sh` to generate all the graphs and store them in the `graphs/` directory.

A.6 Evaluation and expected results

On a machine with 140 or more logical cores, the results should be very similar to those reported in this paper. Figures 4i and 5c require at least 72 logical cores to reproduce. On a machine with fewer cores, the memory usage of `BST`, `EpochBST`, `VcasBST`, and `VcasCT-64` will be several times higher due to Epoch Based Memory Reclamation and oversubscription. This effect is explained in more detail in Section 9.

A.7 Experiment customization

For instructions on how to customize the number of threads, workload, range query size, and data structure size in each experiment, please see the README file in the GitHub repository (https://github.com/yuanhaow/vcaslib).

A.8 Notes

None.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://ctuning.org/ae/submission-20190109.html
- https://ctuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging