VERLIB: Concurrent Versioned Pointers

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

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

Abstract

Recent work has shown how to augment any CAS-based concurrent data structure to support taking a snapshot of the current memory state. Taking the snapshot, as well as loads and CAS (Compare and Swap) operations, take constant time. Importantly, such snapshotting can be used to easily implement linearizable queries, such as range queries, over any part of a data structure.

In this paper, we make two significant improvements over this approach. The first improvement removes a subtle and hard to reason about restriction that was needed to avoid a level of indirection on pointers. We introduce an approach, which we refer to as *indirection-on-need*, that removes the restriction, but yet almost always avoids indirection. The second improvement is to efficiently support snapshotting with lock-free locks. This requires supporting an idempotent CAS. We show a particularly simple solution to the problem that leverages the data structures used for snapshotting.

Based on these ideas we implemented an easy-to-use C++ library, VERLIB, centered around a *versioned pointer* type. The library works with lock (standard or lock-free) and CAS based algorithms, or any combination. Converting existing concurrent data-structures to use the library takes minimal effort. We present results for experiments that use VERLIB to convert state-of-the-art data structures for ordered maps (a B-tree), radix-ordered maps (an ART-tree), and unordered maps (an optimized hash table) to be snapshottable. The snapshottable versions perform almost as well as the original versions and far outperform any previous implementations that support atomic range queries.

CCS Concepts: • **Computing methodologies** → **Concurrent algorithms**.

Keywords: multiversioning, concurrent data structures, snapshots, lock-based, lock-free

PPoPP '24, March 2–6, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0435-2/24/03
https://doi.org/10.1145/3627535.3638501

1 Introduction

The ability to query a concurrent data structure atomically across multiple locations has many applications, such as searching for all keys within a range. Supporting such *multi-point queries* has therefore garnered significant interest over the past decade [1, 2, 4, 6, 14, 16, 18, 27, 28, 37, 45, 47, 51, 62, 64]. Multi-point queries can be supported with atomic snapshots of the memory state. Recent work [62] (henceforth the WBB+ approach) has shown how to efficiently support such snapshots for concurrent data structures that use loads and compare-and-swaps (CASs) on shared memory. The approach uses version lists [53] and maintains all the (asymptotic) time bounds on data structures it is applied to. It is reported to be efficient in practice [62] outperforming prior methods.

In this paper, we make two significant improvements over the WBB+ approach, one that avoids a subtle restriction on how pointers are used that is needed to avoid a level of indirection, and the second is to support multiversioning with lock-free locks [8]. The first is difficult because of sharing of meta-data on objects being pointed to, and the second because previous technique for lock-free locks did not support a CAS operation while CAS is at the heart of the WBB+ approach. Furthermore, we abstract the ideas into a simple library interface for supporting snapshotting, almost always without indirection, and for easily combining the approach with lock-free locks.

Avoiding indirection with multiversioning is critical for performance since it can avoid an extra cache miss on every access.[1] The difficulty of avoiding indirection in a general and lock-free manner is inherent to almost all concurrent approaches that are based on version lists. In particular a version list maintains a historic list of values stored at a location, and each link in the list (version) contains the value and two pieces of meta-data, a timestamp of when that value was written, and a pointer to the previous version (see Figure 1a). Accessing even the most recent value therefore requires first reading the head of the list and then indirectly the value itself. Storing the most recent value directly is difficult in a lock-free setting because of the need to update the version (value, timestamp and previous pointer) atomically.

Leveraging earlier work on a specific concurrent binary tree data structure [28], WBB+ suggest an alternative to get around this problem when values are always pointers, which

[1] Our experiments show up to a factor of two improvement by avoiding indirection.

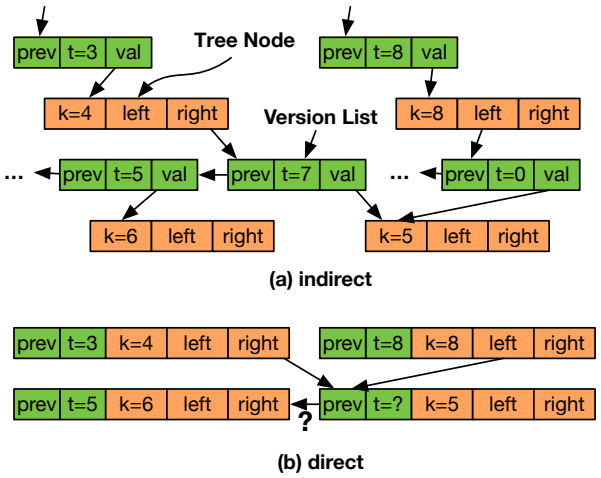


Figure 1. Avoiding indirection in a binary tree by putting version links on the node being pointed to. Structures in orange are tree nodes, with key k, and in green are version links where *prev* means previous and t is the timestamp. The problem with removing indirection is sharing. For example, the nodes with $k = 4$ and $k = 8$ both point to the node with $k = 5$, and seem to need different timestamps and *prev* pointers. In this case, it is OK and the timestamp 7 with *prev* pointing to $k = 6$ is correct since $(t = 0) < (t = 7) < (t = 8)$.

is to store the meta-data on the object being pointed to. However this means, in general, an object cannot over time have two different pointers point to it since then the meta-data could be different for each (see Figure 1b). They show, however, that in some cases this sharing is not problematic and define a property, called *recorded-once*, which limits the use of pointers to avoid improper sharing of meta-data. WBB+ point out that any data structure can be converted into a recorded-once form. Unfortunately, the recorded-once requirement is subtle and many algorithms require non-trivial changes to make them recorded-once, often requiring extra copying. All the experiments they reported were for recorded-once variants of data structures.[2]

Our first contribution is greatly simplifying avoiding indirection by supplying an interface for snapshotted (multiversioned) pointers that does not require the recorded-once condition. The approach avoids indirection in most cases, and when indirection is added, gets rid of it quickly via a shortcut. This is all done under the hood and is invisible to the user. The key ideas here are (1) a light-weight check for when it is safe to avoid indirection, which is possible in the majority of cases, and (2) detection of when indirection is no longer needed and safe removal of indirection at that time. We refer to this approach as *indirection-on-need*.

The second improvement we make is with respect to using locks, and in particular lock-free locks (i.e., implementations of locks that guarantee progress even if processes stall or fail). The idea of a lock-free lock is to have processes help each other run the critical sections when they need a lock that is taken. The idea dates back 30+ years [5, 60] but has only been made practical recently [8]. The recent work has shown that lock-free locks can far outperform standard locks when a machine is *oversubscribed*—i.e., more software threads than hardware threads.

The difficulty with lock-free locks is that when helping, multiple processes can be running the same code, but the code needs to appear as if it ran exactly once: referred to as *idempotence* [15, 21]. Recent research has shown how to implement idempotence cheaply [8], but the approach does not support an idempotent CAS. CAS is difficult because it is hard to tell if a CAS succeeded or not since it could have failed either due to another process running the same instance of a critical section, in which case, if any one succeeds, they should all succeed, or due to a different instance. It is known, theoretically, how to implement an idempotent CAS [3, 10], but with significant practical cost and requiring a double-word-width CAS.

In this paper we show, perhaps surprisingly, that in conjunction with multiversioning an idempotent CAS can be implemented at almost no additional cost and in a simple manner. In particular the method for timestamping in the WBB+ approach can be overloaded to keep track of who succeeded on the CAS allowing all helpers to see the same result. The approach only requires a single-word CAS.

Based on these ideas we have developed and implemented an easy-to-use C++ library called VERLIB. The library revolves around a *versioned pointer* type, which can be used for both lock-based and CAS-based concurrent data structures, as well any combination. As with atomic locations in many programming languages, the versioned pointer supports atomic loads, stores, and CASes. The user can convert their existing concurrent data structure to use VERLIB with only a couple changes: (1) replacing atomic locations holding pointers that need to be part of the snapshotted state with versioned pointers, and (2) inheriting a "versioned" class into any objects pointed to by such pointers. Then the user can wrap a collection of loads in a `with_snapshot` and all the loads will see an atomic view—i.e., the state of the versioned pointers at some fixed point in time.

Once a concurrent data structure is modified to use VERLIB, compiler flags can be set to run it either with or without multiversioning, and either with lock-free versions of the locks or standard versions. If used without multiversioning, then loads within a `with_snapshot` are not atomic. The library also supports different timestamping techniques, including both hardware timestamping and software approaches.

We have converted several state-of-the-art concurrent data structures to use this approach, including a doubly linked list, a hash table, an adaptive radix tree (ART) [40], and

[2]The changes were small, but subtle, requiring an expert to make them.

a B-tree. All but the hash table are taken from the FLOCK library[3] [8], and the hash table uses array bucket copying [20]. We believe our baseline implementations are the fastest or competitive with the fastest current implementations for sorted lists, sorted sets, radix-sorted sets and unsorted-sets. In the paper, we present several experimental results comparing the different data structures, with the different settings of the flags mentioned above, and under a variety of workloads. The workloads include various mixes of updates (inserts and deletes), finds, range queries, and multi-finds. We also vary the data structure sizes and the skewness of the key distribution using a Zipfian distribution. We then compare performance to some existing data structures that directly support range queries.

The experiments demonstrate several points. They show that the cost of versioning is typically small. They show that indirection-on-need is much more efficient than using indirection, while not requiring changes to algorithms to make them recorded-once. They show that combining multiversioning with lock-free locks is efficient, performing much better than standard locks when oversubscribed. And they show that software approaches to timestamping are almost as good as hardware timestamps.

The contributions of the paper include:

- A new *indirection-on-need* approach for version lists that mostly avoids indirection, while not requiring that objects are only recorded-once.
- Efficient and full support of versioned pointers inside of both blocking and lock-free locks. This includes a new mechanism to support an idempotent CAS.
- A easy-to-use portable library, VERLIB, for adding versioning to existing or new concurrent data structures.
- The first B-tree we know of that is lock-free and versioned. It is also significantly faster than previous data structures that support linearizable range queries.
- First versioned radix tree, whether lock-free or not.
- A collection of experiments demonstrating the various tradeoffs of our approaches including the first comparison we know that compares a variety of timestamping approaches.

2 Related Work

Multiversioning using version lists dates back to the 70s [53] and is often used for efficiently supporting read-only transactions in databases [11, 17, 22, 25, 30, 38, 41, 46, 48–50, 52–54, 66]. None of this database work considers multiversioning concurrent data structures, and only one [30] is lock-free and it sequentializes commits.

More recent work has considered efficient multi-point read-only operations in the context of concurrent data structures. These techniques most often support atomic single point updates and atomic multi-point queries and are mostly data structure specific. *Range queries* on ordered sets (maps) have been studied extensively. Brown and Avni [16] gave an obstruction-free range query for k-ary search trees. Avni, Shavit and Suissa [4] described how to support range queries on skip lists. Basin *et al.* [6] described a concurrent implementation of a key-value map that supports range queries. Fatourou, Papavasileiou and Ruppert [28] gave a persistent implementation of a binary search tree with wait-free range queries. The last two both use version list. Winblad, Sagonas and Jonsson [64] also gave a concurrent binary search tree that supports range queries.

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. Petrank and Timnat [51] 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.* [1] 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 [2] described how to implement range queries for concurrent set data structures that use epoch-based memory reclamation.

As described in Section 4, WBB+ describe a general approach to support snapshots for any concurrent algorithm that uses CASes and loads to access shared memory. It introduces the idea of set-stamp helping. Nelson-Slivon, Hassan and Palmiery [45] describe a technique for supporting range queries on a variety of ordered data structures (e.g. linked list, skip list and binary search tree). Kobus and Kokociński, and Wojciechowski describe a linked-list data structure that supports arbitrary snapshots well as atomic batch updates [37]. Sheffi, Ramalhete and Petrank present lock-free data structures supporting linearizable range queries that also bound memory usage by aborting long-lived queries that force the system to hold onto too many old versions [57]. All these use version lists and the last one also uses set-stamp helping.

Several works have studied removing a level of indirection in transactional memory systems, where the main purpose is to avoid extra cache misses. Harris et. al. [32] reduce the two levels of indirection required by DSTM [35] to one in the common case when there is no ongoing transaction involving a location. Marathe et. al. [43] improve this to one level in all cases. Both approaches are obstruction-free. The Cicada system [41] completely removes indirection in certain cases, but requires locks. Furthermore it can only avoid indirection for the first value written to a location (we do not have this restriction). None of these system store the meta-data on the target of a pointer, and hence none have to deal with the issue of sharing meta-data when there are multiple pointers to an object, which we need to handle. Shortcutting of indirection is also supported by Cicada. Again this is only

[3] A library for lock-free locks

```
1  // a versioned pointer to object of type T
2  struct versioned_ptr<T> {
3    versioned_ptr(T v);  // constructor with value v
4    T load();            // read the value
5    void store(T v);     // store a new value
6    bool cas(T old_v, T new_v); }; // compare and swap
7  // inherited in objects pointed to by versioned pointers
8  struct versioned;
9  // function f applied on an atomic snapshot
10 // where R is the return type of f
11 R with_snapshot(F f);
12 // The following is only needed for lock-free locks
13 T* flck::New<T>(args); // idempotent memory allocation
14 void flck::Retire<T>(T*);
15 T flck::with_epoch(F f);
16 struct flck::atomic<T>;
17 struct flck:lock {
18    T try_lock<F> (F f); // f is critical section returning type T
19    T with_lock<F> (F f); }
```

Algorithm 2. The VERLIB interface. C++ template declarations for F and T left out.

under a lock. We note that the Cicada approach does have the advantage that it works for arbitrary values while ours is just for pointers.

The idea of lock-free locks was introduced by Turek, Shasha and Prakash [60] and independently by Barnes [5]. Both approaches use helping and allow arbitrary nesting of locks, and, as long as there are no lock cycles, ensure that the code runs in a lock-free manner—i.e. that the system will make progress for any schedule. The approaches were widely considered to be impractical due to their approach to idempotence, requiring effectively a context switch on every read and write. Therefore, most lock-free data-structures have instead used custom approaches for helping [9, 23, 26, 29, 31, 34, 56, 61, 65]. Ben-David, Blelloch and Wei [8] developed a much more efficient approach to idempotence, outlined in Section 4. We know of no work prior to this paper that combines lock-free locks and multiversioning.

Researchers have studied reducing the memory required by multiversioning [9, 13, 42, 57, 63]. In this paper, we use a simple epoch based collector, but we expect these approaches can also be applied.

3 VERLIB

Here we present the rather minimal VERLIB interface. Although presented and implemented in C++, it should not be hard to embed the ideas in libraries for other programming languages. Our implementation is available in the following repository: https://github.com/cmuparlay/verlib.

The interface is listed in Figure 2. It consists of two classes:

- A `versioned_ptr<T>` class which is used to store versioned pointers to objects of type `T`.
- A `versioned` class that must be inherited by every type `T` that is used in a `versioned_ptr<T>`. It has no user accessible fields.

The library also supports the function: `with_snapshot`(f), which takes a thunk (i.e. a function without arguments) f and runs it such that all calls to `load()` on a versioned pointer return values at a fixed point in the linearized order of updates which falls between the invocation and response of the `with_snapshot`. In other words, the loads in f on versioned pointers appear as if they ran on a snapshot of memory. The `with_snapshot`(f) function returns the value returned by f.

If the structure is to be used with lock-free locks (not required) then it must use FLOCK locks, meaning that all `std::atomic<T>` types (i.e. mutable shared locations holding values of type `T`) must be replaced with `flck::atomic`, and any code that allocates or frees within a lock must use the FLOCK idempotent memory management routines. Note that if not using lock-free locks, any safe memory reclamation scheme can be used. Currently VERLIB implements a `store` with a `load`-and-`cas`. This means that concurrent stores and CASes to the same location will not necessarily linearize.

Cost Bounds. In the following discussion, the term "steps" refers to the number of machine instructions executed. The `store` and `cas` operations each take a constant number of steps.[4] The `load` operation outside a `with_snapshot` takes a constant number of steps, and inside, the number of steps is at most proportional to the number of `store` and `cas` operations on the same versioned pointer that are concurrent with the containing `with_snapshot`. The overhead of the `with_snapshot` is a constant additive number of steps, and if using OPTTS (one of our timestamping schemes) then the thunk f in a `with_snapshot`(f) might be run twice.

3.1 VERLIB Example: Doubly Linked List

As an example of how to use the interface, we present the code for a doubly-linked sorted lists [8] that supports snapshots. In addition to insertions, deletions and finds, the snapshots allow for atomic range queries and any other queries involving a snapshot of the state of the list. We present the code for insertions and range queries in Algorithm 3. Usages of the VERLIB library are marked in red. The code supports lock-free locks using FLOCK, but could also use standard locks, in which case all the code in blue can be replaced with generic versions (e.g., `std::mutex::try_lock`, `std::atomic` and any safe memory reclamation scheme).

Each `node` of the list holds a key and value, previous and next pointers, and a flag indicating whether the node has been removed. The `versioned_ptr` on (line 3) indicates that the `next` pointer should be versioned (since it is used in an atomic snapshot). The `versioned` class needs to be inherited for any class `X` that is used as `versioned_ptr<X>` (line 1).

The `range` implements an atomic range query from key `k1` (inclusive) to key `k2` (exclusive). It finds the first key greater or equal to `k1` using `find_node`, and then continues traversing the list while pushing keys onto `result` until

[4] If used with FLOCK's lock-free locks and because of the way it avoids ABA with tagging, in the infrequent event that the tags run out, then the store and CAS can take longer.

```
struct node : verlib::versioned, flck:lock {
  Key k; Value v;
  verlib::versioned_ptr<node> next;
  flck::atomic<node*> prev;   // not versioned
  flck::atomic<bool> removed; }; // not versioned
node* find_node(node* head, Key k) {
  node* cur = (head->next).load();
  while (k > cur->k) cur = (cur->next).load();
  return cur; }
std::vector<Key> range(node* head, Key k1, Key k2) {
  return verlib::with_snapshot([=] {
    std::vector<Key> result;
    node* cur = find_node(head, k1);
    while (cur->k < k2) {
      result.push_back(cur->k);
      cur = (cur->next).load(); }
    return result; }); }
bool insert(node* head, Key k, Value v) {
  return flck::with_epoch([=] {
    while (true) {
      node* next = find_node(head, k);
      if (next->k == k) return false; // already there
      node* prev = (next->prev).load();
      if (prev->k < k && prev->try_lock([=] {
            if (prev->removed.load() || // validate
                (prev->next).load() != next)
              return false; // try again
            node* cur = flck::New<node>(k,v,next,prev,false);
            prev->next.store(cur); // splice in
            next->prev.store(cur);
            return true;}))
        return true;}});} // success
```

Algorithm 3. Using VERLIB for a sorted doubly-linked list with atomic range queries. Uses of VERLIB are marked in red. Uses of FLOCK lock-free locks are marked in blue and, if using standard locks, can be replaced with standard C++ locks, `std::atomic` and any safe memory reclamation scheme. Code for `find` and `remove` can be found in the full version of the paper.

finding a key greater or equal to k2. We assume the list has a sentinel infinite key at the end. The `with_snapshot` takes as its only argument a thunk `f` (lambda with no argument)[5] and runs it such that all its loads see an atomic view of the memory state (i.e. of all versioned pointers). The range query will therefore be atomic (i.e., linearizable with updates). Note that only the `next` pointer needs to be versioned since only it is followed in the query.

The `insert` searches for the first node `next` with a key greater or equal to `k` and tries to acquire a lock on its previous node (`prev`). If the lock is successfully acquired, `prev` has not been removed, and `prev->next` still points to `next`, the algorithm allocates a new node and splices it in. Otherwise it makes another attempt by repeating in the `while` loop. The `lck->try_lock(f)` from the FLOCK library attempts to take the lock and if successful runs the thunk `f`. It returns true if successful and the thunk returned true.

4 Background on WBB+

Here we review the WBB+ approach for snapshotting [62] since we build on it. The approach is designed to support

[5]In C++ "`[=] { body }`" creates a lambda with no arguments where the free variables of the body are captured by value.

```
struct versioned {};
struct ver_link { // A link in a version list
  std::atomic<long> time_stamp;
  ver_link* prev_version;
  void* value; };
// Timestamps
std::atomic<long> global_stamp;
const long tbd = -1; // to be determined stamp
thread_local long local_stamp = -1;
void take_snapshot() { // increment and return old timestamp
  long stamp = global_stamp.load();
  global_stamp.primcas(stamp,stamp+1);
  return stamp;}
template <typename F> // Run f on a snapshot
auto with_snapshot(F f) {
  local_stamp = take_snapshot(); // set local_stamp
  auto r = f();  local_stamp = -1; return r;}
template <typename V> // Versioned pointer type
struct versioned_ptr {
  std::atomic<ver_link*> v;
  ver_link* set_stamp(ver_link* ptr) {
    if(ptr->time_stamp.load() == tbd)
      ptr->time_stamp.primcas(tbd, global_stamp.load());
    return ptr;}
  V* read_snapshot(long timestamp) {
    ver_link* head = set_stamp(v.load());
    while (head->time_stamp.load() > timestamp)
      head = head->prev_version;
    return (V*) head->value; }
  bool primcas(V* old_v, V* new_v) {
    return v.primcas(old_v, new_v); }
public:
  versioned_ptr(V* ptr) : v(New<ver_link>(zero, nullptr, ptr}) {}
  ~versioned_ptr() { Retire<ver_link>(v.load()); }
  V* load() {
    if (local_stamp != -1) return read_snapshot(local_stamp);
    else return (V*) set_stamp(v.load())->value;}
  bool cas(V* old_v, V* new_v) {
    ver_link* old_l = set_stamp(v.load());
    if (old_v != old_l->value) return false;
    if (old_v == new_v) return true;
    ver_link* new_l = New<ver_link>(tbd, old_l, new_v);
    if (primcas(old_l, new_l)) {
      set_stamp(new_l);
      Retire<ver_link>(old_l);
      return true; }
    set_stamp(v.load());
    Retire<ver_link>(new_l);
    return false; }
  void store(V* new_v) {cas(load(), new_v);}
} // end versioned_ptr
```

Algorithm 4. The VERLIB interface supported using the WBB+ approach with indirection.

taking atomic snapshots of the state of the memory of concurrent algorithms. The approach can be applied to any concurrent algorithm that accesses shared memory through locations supporting loads and compare-and-swaps (CASs). Stores can be implemented with a load and then CAS. To support snapshots, the user replaces all locations they want to included in a snapshot with "versioned" locations[6]. Versioned locations support a `read_snapshot` operation that, given a snapshot handle, returns the value of that location for that snapshot. The interface then supplies a `take_snapshot`

[6]The authors refer to them as CAS objects and versioned CAS (vCAS) objects.

operation that returns a handle to a snapshot of the state of all versioned locations.

The approach is implemented using version lists. It keeps a global timestamp and each versioned location keeps a reverse time-ordered list of all the successful CAS operations applied to it. A versioned location points to the head of its list. Each link in the list contains a value, a timestamp and a previous pointer (Figure 1a). In the following discussion we use PRIMCAS to indicate a machine-level CAS and CAS to indicate a user level CAS on a versioned location. The complete C++ code for the approach, modified to support the VERLIB interface, is given in Algorithm 4.

The CAS operation appends a new version onto the front of the version list by allocating a new version link (Line 42), pointing its previous pointer to the current version, and using a PRIMCAS to try to install it in the head pointer (Line 43). If there are concurrent CAS operations only one will succeed. The tricky part is installing the timestamp on the link. Installing the stamp before or after the PRIMCAS can lead to incorrect results. To fix this, WBB+ introduce a technique, which we will refer to as *set-stamp helping*, that has all operations help set the timestamp at the head of a version list. This technique is a simplification over previous solutions that use double-compare-single-swap [2]. The set-stamp helping approach is implemented by initially setting the new version's timestamp to a special TBD (to be determined) value (Line 42). It then links in this new version with a PRIMCAS. If successful, it sets the new version's timestamp by reading the current global timestamp and using a PRIMCAS to update the new version's timestamp from TBD (Line 47). To ensure the timestamp for the previous version is set, before appending the new version, the CAS operation also helps set the previous version's timestamp (Line 39). Any reads or readSnapshots also help set the timestamp if they encounter a TBD. A readSnapshot is implemented by following the version list of the object to the first link with a timestamp at or earlier than the one requested (Line 27). The `take_snapshot` operation simply increments the global timestamp returning the old one (Line 10).

The time for read, CAS and takeSnapshot is constant. Therefore the asymptotic time of a concurrent algorithm is not affected by using a versioned location. The time for a readSnapshot is at most proportional to the number of CAS operations on the location between when the takeSnapshot was applied and the readSnapshot is called on that timestamp (i.e. the depth of the desired version in the version list).

The `with_snapshot` is simply a wrapper function that calls `take_snapshot` and stores the resulting snapshot handle in a thread local variable called `local_stamp`. Note that in this code, the structure `versioned` is empty. Fields will be added to support indirection-on-need.

The WBB+ approach as just described, and in the code, requires a level of indirection through a version link to read each location. The WBB+ paper describes an optimization that avoids the indirection. The idea is to store the timestamp and the pointer to the previous version directly in each data structure object—i.e. the objects a location points to, such as a node of a tree or linked list. The versioned location then points directly to the object instead of indirectly through a version link (Figure 1b). The problem is that two pointers to the same location will share the timestamp and previous version data. WBB+ observed that such sharing is safe if the data structure is *recorded-once*—meaning that each pointer can only be used as the new field of a CAS at most once [62]. Any data structure can be converted so that all objects are recorded-once, but this conversion is often non-trivial and can cause extra cost. Our goal is to remove the restriction.

5 Indirection-on-need

In this section, we introduce a new mechanism, which we call *indirection-on-need*, for avoiding the level of indirection added by the baseline WBB+ approach. As in the WBB+ approach, we augment each data structure object with additional fields to store version list meta-data which consists of a timestamp field and a pointer to the previous version (see Figure 1b). This can be done through our library by inheriting from the `vp:versioned` class (e.g. Line 1 of Algorithm 3). Our mechanism differs from the one in WBB+ in two main ways. First, when a versioned pointer is written to, our library automatically determines if indirection can be avoided and does so whenever possible. Second, we show that any indirection that is added is only needed temporary and our library automatically removes it when it is no longer needed.

For the first contribution, we identify two cases where indirection can be avoided. Suppose O is a newly allocated object and we wish to change a versioned pointer vp to point to it for the first time. In this case, our library sees that the two meta-data fields of O are unused and initializes them to store version list meta-data for vp (Lines 45 and 48 of Algorithm 5). Then it changes vp to point directly to O (Line 49). For this to work, we make one reasonable restriction, which is that when an object O is allocated by a process, a versioned pointer to it must be written using a store or CAS before any other process can see it—i.e., no side channels can be used to communicate the pointer to O. This is to avoid races among processes each trying to be the first to write a pointer to a newly allocated object.

In the second case, even if O is not a newly allocated object and its meta-data fields were already set by another versioned pointer, we can still initialize a new versioned pointer vp to point directly to O. This is because O is the oldest version in vp's version list, so it just needs to contain enough information to indicate this. In particular, we just need to make sure any `read_snapshot` operation (Line 25) on vp never tries to follow O's `prev_version` pointer. This is the case because O's timestamp was set before vp was initialized, so any call to `read_snapshot` on vp will have

```
1  struct versioned {
2    std::atomic<long> timestamp;
3    ver_link* prev_version;
4    versioned(ver_link* prev) : // constructor
5      timestamp(tbd), prev_version(prev) {} };
6  struct ver_link : versioned {
7    void* value;
8    ver_link(ver_link* prev, void* value) : // constructor
9      versioned(prev), value(value) {} };
10 template <typename V>
11 struct versioned_ptr {
12   std::atomic<ver_link*> v;
13   // adds/strips/tests bit of pointer to mark as indirect
14   ver_link* add_indirect(ver_link* ptr);
15   ver_link* strip_indirect(ver_link* ptr);
16   bool is_indirect(ver_link* ptr);
17   static V* get_value(versioned* ptr) {
18     if (!is_indirect(ptr)) return (V*) ptr;
19     return (V*) ((ver_link*) strip_indirect(ptr))->value; }
20   void shortcut(ver_link* ptr) {
21     ver_link* ptr_ = strip_indirect(ptr)
22     if (ptr_->timestamp.load() <= done_stamp)
23       if (v.compare_exchange_strong(ptr, (ver_link*) ptr_->value))
24         Retire<ver_link>(ptr_); }
25   V* read_snapshot(ver_link* head, long timestamp) {
26     while (strip_indirect(head)->timestamp.load() > timestamp)
27       head = strip_indirect(head)->prev_version;
28     return get_value(head); }
29 public:
30   versioned_ptr(V* ptr) : v((ver_link*) ptr) {
31     if (ptr != nullptr && ptr->timestamp.load() == tbd)
32       ptr->timestamp = zero_stamp; }
33   V* load() {
34     ver_link* head = v.load();
35     shortcut(set_stamp(strip_indirect(head)));
36     if (local_stamp != -1)
37       return read_snapshot(head, local_stamp);
38     else return get_value(head); }
39   bool cas(V* exp, V* ptr) {
40     ver_link* old_v = v.load();
41     if (exp == ptr) return true;
42     if (get_value(old_v) != exp) return false;
43     set_stamp(old_v);
44     ver_link* new_v = (ver_link*) ptr;
45     bool indirect = (ptr==null || ptr->timestamp.load() != tbd);
46     if (indirect)
47       new_v = add_indirect(New<ver_link>(old_v, new_v));
48     else ptr->prev_version = old_v;
49     bool succeeded = primcas(old_v, new_v);
50     if (!succeeded && is_indirect(old_v)) {
51       old_v = ((ver_link*) strip_indirect(old_v))->value;
52       succeeded = primcas(old_v, new_v); }
53     if (succeeded) {
54       set_stamp(new_v);
55       if (is_indirect(old_v))
56         Retire<ver_link>(strip_indirect(old_v));
57       if (indirect) shortcut(new_v);
58       return true; }
59     if (indirect) Retire<ver_link>(strip_indirect(new_v));
60     set_stamp(v.load());
61     return false; }
62 } // end versioned_ptr
```

Algorithm 5. The VERLIB interface implemented using indirection-on-need. Variables and functions unchanged from Figure 4 are omitted.

timestamp greater than or equal to O's timestamp. Therefore, the `read_snapshot` will never traverse past O and it will treat O as if it was the end of the version list. Therefore, it is safe to not add any indirection when initializing versioned pointers, effectively allowing multiple versioned pointers to share the same meta-data.

If neither of the previous two cases hold when writing to a versioned pointer, then we create a `ver_link` object as before and have the versioned pointer point indirectly to O via this `ver_link`. Our library steals a bit from the pointer to distinguish between direct and indirect versioned pointers. See the `cas` operation in Algorithm 5 for the full implementation of this approach.

This approach is very effective in practice because in many commonly used concurrent data structures [26, 33, 44], indirection is only added when deleting a node since inserts always write newly allocated nodes. Furthermore, each update operation usually only performs a small number of pointer swings and most of the writes are initialization writes, which do not add any indirection with this approach. However, the indirect version links added by deletes eventually build up, and we need an efficient strategy for shortcutting them out.

Shortcutting. To identify indirect `ver_links` that can be safely shortcutted out, the idea is to make use of the memory reclamation scheme. Memory reclamation is essentially the problem of determining when an object or a version link is safe to deallocate. If a versioned pointer is stored indirectly, and all of the versions in its version list are safe to deallocate except the current one, then it is safe to shortcut out the version list by storing the versioned pointer directly. This is done by the `shortcut` procedure in Algorithm 5.

In the discussion that follows, we will assume a shared `done_stamp` is maintained that is guaranteed at all times to be no greater than the minimum of the `local_stamps` of any ongoing `with_snapshots` as well as the global stamp. This ensures that no current or future `read_snapshot` will ask for a version older than `done_stamp`. In the full version of the paper, we describe how to maintain the `done_stamp` with epoch-based memory reclamation (EBR).

Since all ongoing `with_snapshots` have timestamps no less than `done_stamp` (by assumption) we can determine if a version list is no longer needed by checking if the timestamp of the current version is no more than `done_stamp`. This check is performed by the `shortcut` function (Line 22). If the check passes, then no ongoing or future `with_snapshot` will access any of the old versions from this list, so it safely shortcuts out the version link vl (line 23). This causes the versioned pointer vp to point directly to some object O. Effectively, this sets O as the tail of vp's version list. One complication is that vl might have a different timestamp than O. We can show that O's timestamp must be strictly less than vl's timestamp because an indirect `ver_link` is only created if O's timestamp is already set. Since all active and future `with_snapshots` have timestamp greater than or equal to vl's timestamp, none of them can distinguish between O's timestamp and vl's timestamp, so it is safe to use O's timestamp instead. Shortcutting is another situation that results in multiple versioned pointers safely sharing the same version list meta-data.

The shortcut function is called each time an indirect versioned pointer is loaded and also at the end of each store and cas. If there are no concurrent with_snapshots, then store/cas will immediately shortcut out any indirect nodes that it creates, in which case indirect nodes are only reachable for a brief moment of time. Shortcutting adds an additional write to each store/cas, but we see in our experiments that the benefits almost always outweigh the cost.

This shortcutting technique requires us to make some additional changes to the versioned pointer's CAS operation. The primcas on Line 49 can fail not only because another CAS succeeded, but also because an indirect ver_link got shortcutted out. In the latter case, the value of the versioned pointer did not change, so we need to retry the primcas on Line 52. Another subtlety is that the cas needs to know if it overwrote an indirect pointer and is thus responsible for retiring it. This check and the subsequent retire is done on Line 59.

6 Snapshots with Locks-free-Locks

The WBB+ approach to snapshotting works for lock-based code, as does the indirection-on-need approach just described. At the end of the section we describe a slightly more efficient version of store that is specialized for the case where there are no write-write races. This is typically true when using locks. The WBB+ approach, however, does not work with the lock-free lock technique of Ben-David, Blelloch and Wei [8] (FLOCK). This is because FLOCK does not allow CAS to be used inside a lock-free lock's critical section and CAS is crucial for the WBB+ approach. In this section, we cover how to combine snapshots and lock-free locks, including a technique for implementing an idempotent CAS that is safe to use with lock-free locks, optimizations to maintaining timestamps, and a specialized implementation of store.

In the FLOCK approach, a lock request takes two arguments: the requested lock and a thunk containing the critical code to run when the lock is acquired. A thunk is a closure containing both the code pointer to the critical section and the values of captured free variables. When a process acquires a lock, it leaves a pointer to the thunk on the lock so others can help run it. When attempting to acquire a lock that is already locked, the process will help run the thunk stored on the lock and help reset the lock back to the unlocked state.

The difficult part is helping since multiple threads might be running the same thunk concurrently, which semantically should only run once. FLOCK ensures code is idempotent, guaranteeing it appears as if it ran exactly once. The library-based approach replaces operations on shared memory (loads, stores, CAMs[7], allocation, and deallocation) with idempotent versions. It uses a log for each thunk, and ensures that (1) for all loads the original thunk and all helpers read the same value, (2) for all stores and CAMs only the first among the original and helpers will have an effect, and (3) memory allocations and deallocations happen once.

[7]A limited form of CAS that does not return whether it succeeded.

```
1  struct versioned {
2    flck::atomic<long> timestamp; ...} // idempotent link stamp
3  ...
4  std::atomic<long> global_stamp; // non-idempotent global stamp
5  ...
6  struct versioned_ptr {
7    flck::atomic<ver_link*> v; // idempotent pointer
8    ...
9    ver_link* set_stamp(ver_link* ptr) {
10     long new_s = global_stamp.load(); //not idempotent
11     ptr->time_stamp.cas_nonidempotent(tbd, new_s);
12     return ptr;}
13   ...
14   void shortcut(ver_link* ptr) {
15     ver_link* ptr_ = strip_indrct(ptr)
16     if (ptr_->timestamp.load() <= done_stamp)
17       if (v.cas_nonidempotent(ptr, (ver_link*) ptr_->value))
18         Retire_nonidempotent<ver_link>(ptr_); }
19   ...
20   bool primcas(V* old_v, V* new_v) {
21    v.cam(old_v, new_v)
22    return (v.load()==new_v ||
23            new_v->time_stamp.load() != tbd);}
24   ...
25   void store_norace(V* ptr) {
26     ver_link* old_v = v.load();
27     ver_link* new_v = (ver_link*) ptr;
28     bool indirect = (ptr==null || ptr->timestamp.load() != tbd);
29     if (indirect) new_v = add_indrct(New<ver_link>(old_v, new_v));
30     else ptr->prev_version = old_v;
31     if (is_indrct(old_v) && primcas(old_v, new_v))
32       Retire<ver_link>(strip_indrct(old_v));
33     else v.store(new_v);
34     if (indirect) shortcut(new_v);}
35 } // end versioned_ptr
```

Algorithm 6. Combining snapshotting with lock-free-locks. Only changes from Figure 5 are shown.

Importantly, the approach does not support a general CAS since it is difficult to determine, in a general and idempotent way, if a CAS succeeded. Using a CAM followed by a load to check for success does not work since another CAM could succeed between the two operations, making it appear that the first failed. It is possible to implement an idempotent CAS using a double-word wide regular CAS [10], but this is impractical since it would require that all versioned pointers be maintained as double words. Furthermore, the approach is also quite complicated.

Here we describe a simple technique to implement an idempotent CAS that works within the FLOCK framework when used with snapshotting. The code is shown in Figure 6 with the redefinition of primcas. The code is deceptively simple, and the correctness a somewhat subtle. The implementation relies on the fact that when updating a pointer with a versioned CAS, the timestamp of the new version is initially set to TBD, and before it is updated again, it must be set to a real timestamp. This means that a versioned CAS succeeded if and only if either the value in the location is the same as the new value of the CAS, or the timestamp was set (i.e., the conditions on Lines 22 and 23). This is shown more formally by the following theorem.

Theorem 6.1. *The* `primcas` *on Line 20 of Algorithm 6 and as used on Lines 49 and 52 of Algorithm 5 implements a linearizable* CAS.

Proof. In the following, we will use Line $x.y$ to indicate Line y of Algorithm x. We first note that two concurrent `primcass` must have different new values since they both just allocated new objects on Line 5.47. Therefore, if the CAM on Line 6.21 failed, then the check on Line 6.22 will always fail since no concurrent `primcas` can be writing the same value. Now consider two concurrent CAS operations and for now just the first `primcas` on Line 5.49. Without loss of generality assume the CAM (Line 6.21) of the first CAS linearizes first and succeeds. If the CAM of the second CAS linearizes after the first CAS runs the check on Line 6.22, then the first CAS passes the check and correctly reports success. However, if the CAM of the second CAS succeeds and linearizes before the first runs the check, then the first will fail the check. This is exactly the problem with trying to implement a CAS with a CAM then load to check. But, in this case, for the second CAS to succeed on its CAM, it must have loaded the result of the first CAM into `old_v` on Line 5.40. Hence, the first PRIMCAS properly reports success or failure. A similar argument can be made for the second `primcas` on Line 5.52 but here the `old_v` is from Line 5.51. However, in this case the timestamp on `old_v` must be set since it is an indirect value. □

There is a second problem we found with using lock-free locks with snapshotting. The problem is that using idempotent operations when accessing timestamps causes a significant bottleneck. This is because the timestamp is heavily contended. We show that although using a non-idempotent load and CAM to update the timestamp can lead to non-idempotent execution (different helpers on the same thunk can see different timestamps), this does not effect correctness. We can therefore use a non-idempotent atomic for the global timestamp (Line 4) and also non-idempotent CAS in `set_stamp` (Line 11) . Note that the timestamp within each version link (Line 2) needs to be idempotent since the load on Line 28 needs to be idempotent. The use of non-idempotent global timestamps is justified with the following theorem along with the fact that with lock-free locks, helpers run in the same epoch as the original [62]. The proof is in the full version of the paper.

Theorem 6.2. *Any call to* `set_stamp` *on Line 9 of Algorithm 6 can be repeated any number of times by helper operations without affecting the correctness of versioning.*

Another similar optimization can be made to use a non-idempotent CAS and a non-idempotent Retire for shortcutting (Lines 17–18). This is safe since shortcutting is a "helping" step anyway (many processes might be attempting a shortcut at the same time and it has to appear to happen once), so it is already idempotent.

Finally we show how to directly implement `store` on Lines 25–34, which avoids several steps that would have been required if a `load` and CAS were instead used to implement the `store`. However, this version assumes there are no write-write races—i.e., that locks prevent two processes storing to the same location concurrently. We therefore refer to it as `store_norace`. The code for this direct store is less than half as many lines as for the CAS.

7 Optimistic Timestamps

One of the bottlenecks of multiversioning is incrementing the shared global timestamp. For this reason, there has been significant work on reducing the cost of maintaining timestamps [11, 24, 36, 41, 55, 59, 62, 67]. On x86-based machines, the RDTSC instruction implements a hardware timestamp that is synchronous across cores [36, 55]. This, however, is not portable, and manufacturers are not guaranteeing the clock will be synchronous across processing nodes in future platforms. We present a software optimistic timestamping technique called OPTTS which simplifies the low contention version-clock proposed in TL2 [24]. Our variant never increments the timestamp during updates and only sometimes increments for queries.

The OPTTS technique first runs each query optimistically and only increments the global timestamp if the optimistic execution aborts. Specifically the execution only needs to abort if the query comes across a timestamp equal to its own. The approach runs the query at most twice since the second run is guaranteed to see a consistent snapshot.

The code for the approach is given in Algorithm 7. It uses the `global_stamp` defined in Algorithm 4. The approach modifies `read_snapshot` so that after locating the version with the largest timestamp less than or equal to the current local stamp, it checks if that stamp is equal to the current stamp (Line 5). If so, and if running optimistically, it sets the abort flag. The approach then modifies the `with_snapshot(f)` so that it first runs the query `f` without incrementing the stamp (Line 11). It then checks if the query aborted and, if so, increments the stamp and reruns (Line 15). The second run is guaranteed to produce a linearizable return value because it is essentially the same as the old `with_snapshot` implementation in Algorithm 4. Note that this technique requires `f` to be safe to run twice. This is a natural requirement since `f` is a read-only query on the data structure. As an optimization, queries passed to `with_snapshot` can periodically check the abort flag and finish early if they see it set.

8 Experimental Evaluation

We apply VERLIB to several concurrent set data structures to add support for linearizable range queries and groups of k find operations that act atomically (multi-finds). Our goal is to (1) measure the overhead VERLIB adds to the original data

```
1  thread_local bool aborted, optimistic;
2  void increment_timestamp(long stamp) {
3    global_stamp.primcas(stamp,stamp+1);}
4  // add after Line 27 of Algorithm 5, in read_snapshot
5  if (strip_indirect(head)->time_stamp.load() == timestamp)
6    aborted = optimistic;
7  template <typename F>
8  auto optimistic_with_snapshot(F f) {
9    local_stamp = global_stamp;
10   aborted = false; optimistic = true;
11   auto r = f();   // run optimistically
12   if (aborted) {  // rerun with incremented stamp if aborted
13     aborted = false; optimistic = false;
14     increment_global_stamp(local_stamp);
15     r = f(); }
16   local_stamp = -1; return r; }
```

Algorithm 7. Optimistic Timestamping.

structure, (2) measure the improvements from indirection-on-need with and without shortcutting, and (3) compare with state-of-the-art concurrent set data structures. We also compare different timestamping approaches described below.

Setup. Our experiments ran on a 64-core Amazon Web Service c6i-metal instance with 2x Intel(R) Xeon(R) Platinum 8375C (32 cores, 2.9GHz and 54MB L3 cache each), and 256GB memory. Each core is 2-way hyperthreaded, giving 128 hyperthreads. We used `numactl -i all`, evenly spreading the memory pages across the sockets in a round-robin fashion. The machine runs Ubuntu 22.04.1 LTS. The C++ code was compiled with g++ 11 with `-O3`. For memory allocation in all of our C++ experiments, we used the ParlayLib [12] allocator, which is built on top of Jemalloc. Memory is reclaimed using epoch-based reclamation. For Java, we used OpenJDK 19.0.1 with flags `-server`, `-Xms50G` and `-Xmx50G`. We report the average of 3 runs, each of 5 seconds. For Java, we also pre-ran 3 runs to warm up the JVM.

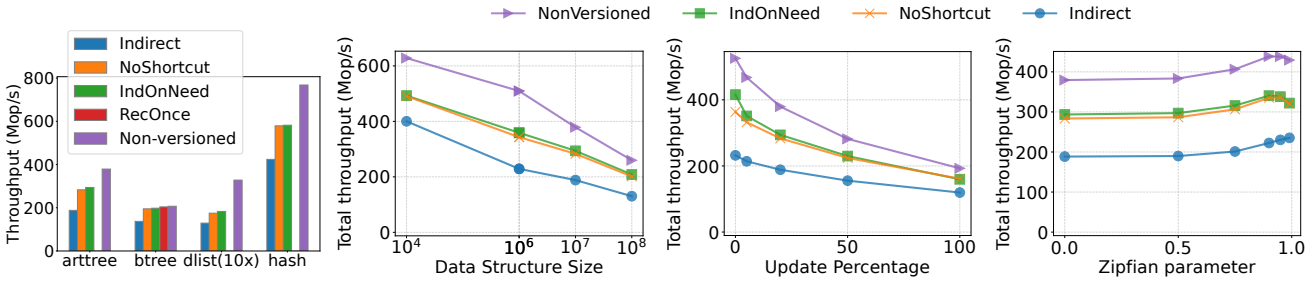
Data Structures. We report on four data structures implemented in C++ with VERLIB: a B-tree (btree), an adaptive radix tree (arttree), a doubly-linked list (`dlist`), and a `hashtable`. For the first three, we used existing data structures from the FLOCK library [7] and applied the modifications described in Section 3. These three are lock based and can be run in either blocking or lock-free mode using FLOCK or VERLIB. The doubly-linked list code is given in Section 3.1. The hash table is a CAS based implementation that maintains an array per bucket and copies the array on update [20]. We set the number of buckets equal to the initial size of the data structure rounded up to the nearest power of two for fast hashing. The btree has internal nodes that hold between 4 and 22 child pointers. As in the original arttree [40], the arttree is byte-based and has three types of internal nodes. For all four data structures, we implemented multi-finds, wrapping them in a `with_snapshot`. For the three ordered set structures we also implemented range queries.

All of the original data structures required recording more than once. However, the only place the btree recorded a node more than once was at the root, so we created a strictly recorded-once version to compare to, but not for the others.

We compare these VERLIB data structures with several state-of-the-art concurrent set data structures: LFCA [64], Jiffy [37], EpochBST [2], BundledSkiplist [45], BundledCitrus [45], LSKN-arttree [39, 40], SB-abtree [58]. The first three are lock-free and all except the last two support linearizable range queries. LSKN-arttree is a concurrent radix tree and the others are comparison based ordered set data structures. We used implementations by the original authors for all these data structures. We made a small change to the LSKN-arttree because it originally only supported key-value pairs that fit in 63-bits. Our VERLIB arttree supports arbitrary key-value types by storing them at a level of indirection and we modified LSKN-arttree to do this as well for a fair comparison. We use the C++ version of most data structures to be consistent with our implementations. For LFCA and Jiffy, we were unable to find a reliable C++ implementation and had to use the Java implementation instead.

Workloads. In our experiments, we vary the following parameters: (a) data structure size (denoted by n), (b) operation mix, (c) size of range queries (denoted by s), (d) number of threads, and (e) the distribution from which keys are drawn. In most experiments, we initialize each data structure with $n = 10M$ keys by running a mix of inserts and deletes on an initially empty data structure. This size was chosen to make sure the datasturcture does not fit in L3 cache. For linked lists, we instead use $n = 1000$ as the default size. In the timed portion of the code, each thread performs a mix of operations, consisting of inserts and deletes (done in equal numbers), as well as either finds, range queries or multi-finds. We use a universe U of $2n$ distinct, uniform random 64-bit keys. Keys for all operations (including initialization) are drawn randomly from U, which ensures that the size of the data structure remains approximately n throughout the experiment. Keys are drawn from Zipfian distribution with parameter ranging from 0 (uniform distribution) to .99 (highly skewed). The Zipfian distribution is used in the YCSB benchmark [19] to model real world access patterns. Our range queries search for all keys in the range $[a, b]$ where a is drawn from U as before and b is chosen so that the expected number of keys in the range is s.

Timestamps. We experimented with several different techniques for maintaining the global timestamp. Firstly, we use hardware timestamping with the RDTSC instruction on x86 [36, 55] (HWTS), which is supported by the machine we use. This is generally the fastest in our experiments, but not always. We also use a traditional timestamp increment on updates [53] (UPDATETS), as well as an increment on snapshotted queries [62] (QUERYTS). Our implementation of these include a tuned delay to reduce contention. We implemented the optimized timestamping scheme from TL2 [24] (TL2-TS),



(a) default workload **(b)** arttree, varying size **(c)** arttree, varying update rate **(d)** arttree, zipfian keys

Figure 8. Comparing different **versioned pointer implementations**. Unless otherwise specified, the workload consists of 128 threads performing 20% updates and 80% multi-finds of size 16, with keys drawn from the uniform distribution. The default size for lists is 1000 and for all other data structure is 10M. `dlist(10x)` indicates that its throughput was scaled up by 10x.

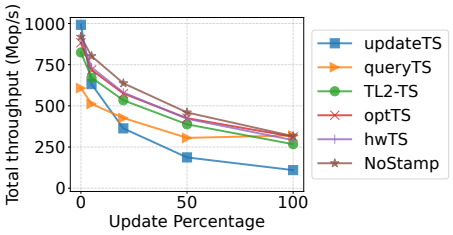


Figure 9. Comparing different **timestamp implementations** when applied to our versioned hashtable, initialized with 10M keys. Keys are drawn from an uniform distribution and each run uses 128 threads.

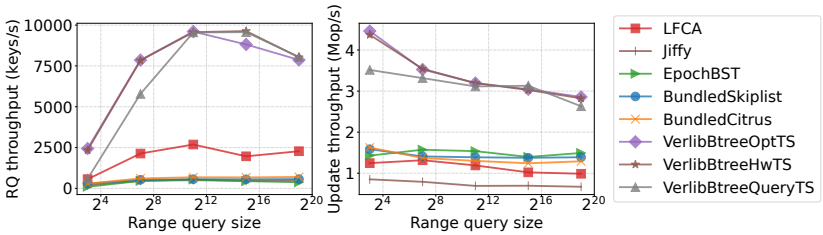


Figure 10. Range queries. Comparing various data structures supporting linearizable range queries. Run with 100 threads: 5 update threads, 95 range query threads, keys drawn from uniform distribution, 10M data structure size

which reduces the number of increments. Finally we implemented our own simpler variant of the TL2 approach (OPTTS) described in Section 7.

8.1 Results

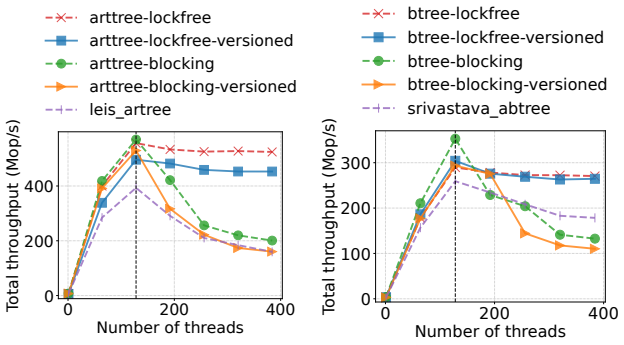
Indirection-on-need. Figure 8 compares the performance of the versioned pointer algorithms presented in this paper. `Indirect` represents the algorithm from Section 6 without the indirection-on-need optimization, `NoShortcut` uses indirection-on-need but without shortcutting, and `IndOnNeed` uses shortcutting and is the default implementation in VERLIB. We also implemented a variant of versioned pointers, called `RecOnce`, which never uses indirect nodes and only works for recorded-once data structures (as with the WBB+ experiments). We applied this to our recorded-once variant of btree. All these variants use lock-free locks and hardware timestamping (HWTS). To measure the overhead of versioned pointers, we also show the original non-versioned data structure (`Non-versioned`) in the graphs. Multi-finds on this data structure are not linearizable (each find can linearize at its own point).

Overall, across the wide variety of workloads and data structures shown in Figure 8, the overhead of applying versioned pointers (with all optimizations) to a `Non-versioned` data structure is generally low. It is higher on lists since everything fits in cache and traversing a list is very cheap—hence the cost of the extra checks is more significant. For arttree, indirection-on-need improves the performance of `Indirect` versioned pointers by almost 2x on the left side of Figure 8c. The shortcutting optimization also consistently helps on these data structures, especially for read mostly workloads, although not as much as initially removing indirection. For btree, `IndOnNeed` versioned pointers achieves essentially the same performance as `RecOnce`, while not requiring the data structure to be recorded-once. Overall, indirection avoidance is more important for larger data structure that do not fit in cache and for read-heavy workloads.

In Figure 8d, we vary the amount of contention by drawing keys from the zipfian distribution and varying its parameter. The relative performance of the versioned pointer implementations generally stayed the same across all contention levels, although at high contention shortcutting no longer helps.

The remaining experiments use `IndOnNeed` as the default implementation of versioned pointers.

Timestamps. Figure 9 compares the five timestamp techniques: QUERYTS, UPDATETS, HWTS, OPTTS and TL2-TS. As a baseline, it also compares with NO-STAMP, which never increments the global timestamp, resulting in non-linearizable snapshots. We applied all six to our lock-free versioned hashtable. In this experiment, the update rate varies from 0-100% with all other operations being multi-finds of size 16. To the best of our knowledge this is the first apples-to-apples comparison among a wide range of timestamp techniques.



(a) arttree **(b)** btree

Figure 11. Scalability. Comparing various arttree and btree implementations. Solid lines used for data structures that support linearizable range queries, dotted lines used otherwise. Run with 10M keys, 5% updates, 95% lookups, and keys drawn from Zipfian distribution (parameter .99). The dotted vertical line indicates the number of cores on our machine.

Across these experiments, HWTS tends to perform the fastest because our machine supports a very light-weight `rdtsc` instruction for reading the hardware clock. Optimistic timestamp (OPTTS) achieves almost the same performance as HWTS, indicating that optimistic executions of multi-find often succeed without having to increment the global timestamp. OPTTS is slightly faster than TL2-TS due to being simpler and more optimized for this setting with just read-only transactions. QUERYTS and UPDATETS perform poorly in multi-point query heavy and update heavy workloads, respectively, due to high contention when incrementing the timestamp. OPTTS outperforms HWTS at high update rates because it does almost no work.

Direct Stores. Section 6 described how to replace a load-then-CAS with a store, avoiding some checks and updates. We ran experiments with and without this optimization. On workloads with 50% updates we saw up to a 8% improvement in performance (e.g., on B-trees with 100K keys and uniform distribution). On workloads with 5% updates the improvement was negligible, as might be expected since the optimization only affects the performance of updates.

Range query. Figure 10 compares our versioned btrees with state-of-the-art data structures supporting linearizable range queries. Updates and especially range queries on our versioned B-trees are significantly faster because of the increased cache locality due to the large fanout at internal nodes and the batching of keys in each leaf. Out of the other range queriable data structures, only LFCA and VcasChromaticTree store a batch of keys in each leaf, but internal nodes still only have fanout 2. Developing a general and easy-to-apply library allowed us to apply versioning to faster baseline data structures than those used in previous work.

Scalability. Figure 11 measures the scalability of our versioned arttree and btree up to oversubscription. The previous

	arttree	btree	hash	dlist	SB	LSKN
Non-versioned	57.2	44.0	40.7	41.0	67	53
Versioned	69.3	45.1	54.6	57.0		

Figure 12. Space. Bytes per entry upon initializing with 10 million entries. Includes the size of the key-value pair (16 bytes), the node meta-data (e.g., size, type, pointers), the versioning meta-data (next pointer and timestamp) and allocator overhead. SB = SB-abtree and LSKN = LSKN-arttree.

experiments were run with VERLIB in lock-free mode, and these graphs also show its performance in blocking mode. Consistent with previous experiments on lock-free locks [8], blocking mode tends to be slightly faster before oversubscription, but drops severely in performance after oversubscription. This motivates the importance of supporting both versioning and lock-free locks.

We also plot the performance of LSKN-arttree and SB-abtree, which are state-of-the-art concurrent radix trees and B-trees, respectively. They both use blocking locks, so they also slow down after oversubscription. Our VERLIB arttrees and btrees perform competitively with these data structures while also being lock-free and supporting linearizable range queries. When oversubscribed, the performance of SB-abtree does not degrade as much as our blocking btree. We believe this is because the SB-abtree takes finer grained locks at the leaves, instead of at the parent.

Space. Figure 12 gives some numbers for space in terms of bytes per entry. The space overhead for versioning is particularly small for btrees since each node is large (up to 512 bytes), and the versioning metadata (next pointer and timestamp) is only needed once per node. The other structures have smaller nodes and hence the space overhead for the metadata is larger.

9 Conclusion

In conclusion, we present an efficient implementation of concurrent versioned pointers that is compatible with both blocking and lock-free locks and is optimized to avoid indirection whenever possible. It is significantly easier to apply than previous work on versioned CAS objects [62], which requires the user to often modify their data structure in non-trivial ways to get good performance. We wrap these ideas in a VERLIB library and apply it to several data structures to support linearizable range queries. Experiments show that these data structures are significantly faster than existing concurrent, range queriable data structures.

Acknowledgments

We thank the anonymous referees for their comments. This work was supported by the National Science Foundation grants CCF-2119352 and CCF-1919223.

References

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

[2] 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.

[3] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. 2018. Nesting-Safe Recoverable Linearizability: Modular Constructions for Non-Volatile Memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[4] 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.

[5] Greg Barnes. 1993. A method for implementing lock-free shared-data structures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 261–270.

[6] 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

[7] Naama Ben-David, Guy Blelloch, and Yuanhao Wei. 2022. The flock library. https://github.com/cmuparlay/flock.

[8] Naama Ben-David, Guy Blelloch, and Yuanhao Wei. 2022. Lock-Free Locks Revisited. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[9] Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, Yihan Sun, and Yuanhao Wei. 2021. Space and Time Bounded Multiversion Garbage Collection. In *International Symposium on Distributed Computing (DISC)*. 12:1–12:20.

[10] Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. 2019. Delay-free concurrency on faulty persistent memory. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 253–264.

[11] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control - Theory and Algorithms. 8, 4 (Dec. 1983), 465–483. http://doi.acm.org/10.1145/319996.319998

[12] Guy E Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib-A toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 507–509.

[13] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable garbage collection for in-memory MVCC systems. *Proceedings of the VLDB Endowment* 13, 2 (2019), 128–141.

[14] 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.

[15] Jeremy Brown, J. P. Grossman, and Tom Knight. 2002. A Lightweight Idempotent Messaging Protocol for Faulty Networks. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 248–257.

[16] 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.

[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.

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing*. 143–154. https://doi.org/10.1145/1807128.1807152

[20] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized concurrency: The secret to scaling concurrent search data structures. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 631–644.

[21] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static Analysis and Compiler Design for Idempotent Processing. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

[22] 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.

[23] Dave Dice and Alex Garthwaite. 2002. Mostly lock-free malloc. In *International Symposium on Memory Management (ISMM)*. 163–174.

[24] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *International Symposium on Distributed Computing (DISC)*.

[25] 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.

[26] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking binary search trees. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[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.

[29] Steven Feldman, Pierre LaBorde, and Damian Dechev. 2015. A wait-free multi-word compare-and-swap operation. *International Journal of Parallel Programming* 43, 4 (2015), 572–596.

[30] 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

[31] Rachid Guerraoui, Alex Kogan, Virendra J Marathe, and Igor Zablotchi. 2020. Efficient multi-word compare and swap. *International Symposium on Distributed Computing (DISC)*.

[32] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. 2006. Optimizing memory transactions. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 14–25.

[33] Timothy L Harris. 2001. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing (DISC)*. Springer, 300–314.

[34] Timothy L Harris, Keir Fraser, and Ian A Pratt. 2002. A practical multi-word compare-and-swap operation. In *International Symposium on Distributed Computing (DISC)*. 265–279.

[35] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. 2003. Software Transactional Memory for Dynamic-Sized Data Structures. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[36] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. 2018. A Scalable Ordering Primitive for Multicore Machines. In *ACM European Conference on Computer Systems (EuroSys)*. Article 34, 15 pages.

[37] Tadeusz Kobus, Maciej Kokociński, and Paweł T. Wojciechowski. 2022. Jiffy: A Lock-Free Skip List with Batch Updates and Snapshots. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[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] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *IEEE International Conference on Data Engineering (ICDE)*.

[40] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *Proc. International Workshop on Data Management on New Hardware*.

[41] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 21–35.

[42] Li Lu and Michael L Scott. 2013. Generic multiversion STM. In *Proc. International Symposium on Distributed Computing*. Springer, 134–148.

[43] Virendra Marathe, Michael Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William Scherer, and Michael Scott. 2006. Lowering the overhead of nonblocking software transactional memory. In *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*.

[44] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-Free Binary Search Trees. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[45] Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. 2022. Bundling Linked Data Structures for Linearizable Range Queries. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 368–384.

[46] 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.

[47] 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.

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

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

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

[51] Erez Petrank and Shahar Timnat. 2013. Lock-Free Data-Structure Iterators. In *Proc. 27th Intl Symposium on Distributed Computing*. 224–238.

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

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

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

[55] Wenjia Ruan, Yujie Liu, and Michael Spear. 2013. Boosting Timestamp-Based Transactional Memory by Exploiting Hardware Cycle Counters. *ACM Trans. Archit. Code Optim.* 10, 4 (dec 2013), 21 pages.

[56] Niloufar Shafiei. 2014. Non-Blocking Doubly-Linked Lists with Good Amortized Complexity. arXiv:1408.1935 [cs.DC]

[57] Gali Sheffi, Pedro Ramalhete, and Erez Petrank. 2023. EEMARQ: Efficient Lock-Free Range Queries with Memory Reclamation. In *26th International Conference on Principles of Distributed Systems (OPODIS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[58] Anubhav Srivastava and Trevor Brown. 2022. Elimination (a,b)-Trees with Fast, Durable Updates. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 416–430. https://doi.org/10.1145/3503221.3508441

[59] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *ACM Symposium on Operating Systems Principles (SOSP)*. 18–32.

[60] John Turek, Dennis Shasha, and Sundeep Prakash. 1992. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Principles of Database Systems (PODS)*. 212–222.

[61] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy lock-free indexing in non-volatile memory. In *IEEE International Conference on Data Engineering (ICDE)*. IEEE, 461–472.

[62] Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. 2021. Constant-time snapshots with applications to concurrent data structures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 31–46.

[63] Yuanhao Wei, Guy E. Blelloch, Panagiota Fatourou, and Eric Ruppert. 2023. Practically and Theoretically Efficient Garbage Collection for Multiversioning. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[64] 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.

[65] Kjell Winblad, Konstantinos Sagonas, and Bengt Jonsson. 2021. Lock-free contention adapting search trees. *ACM Transactions on Parallel Computing (TOPC)* 8, 2 (2021), 1–38.

[66] 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.

[67] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1629–1642.

A Artifact Evaluation Appendix

A.1 Abstract

This artifact contains the source code and scripts to reproduce all the graphs in Section 8. For an up-to-date version of the VERLIB library, please visit our repository on GitHub https://github.com/cmuparlay/verlib.

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

- **Algorithm:** The VERLIB data structures described in Section 8.
- **Program:** microbenchmarks
- **Compilation:** g++11, OpenJDK 19.0.1
- **Run-time environment:** Ubuntu 22.04.1 LTS
- **Hardware:** Multi-core machine, preferably with at least 64 logical cores
- **Output:** Graphs from Section 8 as pdf files.
- **Experiments workflow:** One script for compiling the experiments and one script for generating all the graphs.
- **Disk space required (approximately):** 2 GB
- **Time needed to prepare workflow:** approximately 5 minutes
- **Time needed to complete experiments:** approximately 4 hours
- **Publicly available:** yes
- **Code licenses:** MIT License

A.3 Description

A.3.1 How delivered. The artifact is available on Zenodo https://zenodo.org/records/10447617.

A.3.2 Hardware dependencies. To accurately reproduce our experimental results, a multi-core machine with at least 64 logical cores is recommended.

A.3.3 Software dependencies. Our artifact is expected to run correctly under a variety of Linux x86_64 distributions. 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) as well as the ParlayLib memory allocator, which is included with the artifact. 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 `bash compile_all.sh`.

A.5 Experiment workflow

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

A.6 Evaluation and expected results

On a machine with 128 logical cores, the results should be very similar to those reported in this paper. For machines with different numbers of cores, we recommend using the following settings to reproduce the general shape of our graphs (where X is the number of logical cores):

- Scalability experiments should be run with $[1, X/2, X, 1.5X, 2.5X, 3X]$ threads
- All other experiments should be run with X threads.

A.7 Experiment customization

For instructions on how to customize the number of threads, workload, and data structure size in each experiment, please see the README file included in the artifact.

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