Practically and Theoretically Efficient Garbage Collection for Multiversioning

Yuanhao Wei
yuanhao1@cs.cmu.edu
Carnegie Mellon University, 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@eecs.yorku.ca
York University, Canada

Abstract

Multiversioning is widely used in databases, transactional memory, and concurrent data structures. It can be used to support read-only transactions that appear atomic in the presence of concurrent update operations. Any system that maintains multiple versions of each object needs a way of efficiently reclaiming them. We experimentally compare various existing reclamation techniques by applying them to a multiversion tree and a multiversion hash table.

Using insights from these experiments, we develop two new multiversion garbage collection (MVGC) techniques. These techniques use two novel concurrent version list data structures. Our experimental evaluation shows that our fastest technique is competitive with the fastest existing MVGC techniques, while using significantly less space on some workloads. Our new techniques provide strong theoretical bounds, especially on space usage. These bounds ensure that the schemes have consistent performance, avoiding the very high worst-case space usage of other techniques.

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

Keywords: lock-free, memory reclamation, MVCC

1 Introduction

Multiversion concurrency control is used widely in database systems [6, 32, 39, 41, 45, 55], transactional memory [18, 27, 30, 42, 43], and shared data structures [4, 16, 38, 53], mainly for supporting *read-only transactions* (abbreviated as *rtxs*), including complex multi-point queries that appear atomic, while update operations are executed concurrently. This is usually achieved by maintaining a list of previous versions for each object, sorted by timestamps indicating when the object was updated. An rtx gets a timestamp t and traverses objects' *version lists*, to read the latest version at or before t.

Multiversion garbage collection (MVGC) is the problem of removing and collecting obsolete versions. Suppose two consecutive versions in the version list have timestamps t_1 and t_2. The version with timestamp t_1 is *needed* if there is an ongoing rtx whose timestamp t satisfies $t_1 \leqslant t < t_2$, and *obsolete* otherwise. The latest version is always needed. Efficient MVGC is an important problem in multiversioning systems [5, 7, 18, 28, 32, 35, 38, 39, 53], since keeping obsolete versions can lead to excessive space usage.

The simplest approach to MVGC is to use epoch-based reclamation (EBR) [18, 28, 38, 39, 53]. EBR manages version lists quickly and simply by removing only the earliest versions from the end of a version list when no active rtxs need them. However, it cannot remove obsolete versions from the middle of a version list and it may therefore maintain an unbounded number of obsolete versions. In practice, this can lead to a blowup in space when there are long-lived rtxs.

To avoid this problem, Lu and Scott designed an MVGC system GVM [35] that can remove intermediate versions by occasionally traversing all version lists and removing obsolete versions. The HANA system uses a similar approach [32]. The problem with these approaches is that they scan all lists even if they are rarely or never updated. Böttcher et al. improve on these techniques in the Steam system [7] by tying the scanning of lists to updates on the list. Experiments with Steam [7] showed that removing intermediate versions can significantly improve space usage and throughput compared to some previous MVGC methods in workloads with lengthy rtxs and frequent updates. However, none of these systems provide strong worst-case time bounds and for P processes and M version lists achieve bounds of at best $O(PM)$ space.

Ben-David et al. [5] recently gave a MVGC technique (henceforth BBF+) that has the best known time and space bounds. It removes intermediate versions without traversing the full list. It is wait-free, maintains only a constant factor more versions than needed, and uses $O(1)$ steps on average per allocated version. To achieve this, BBF+ defined a *range tracking* object to precisely identify obsolete versions. However, BBF+ has not previously been implemented and it is unclear if it can be made time-efficient in practice. It uses a sophisticated doubly-linked list implementation, which we call TreeDL because it has an implicit tree laid over it.

We study the efficiency of several MVGC schemes in practice by implementing and comparing them in a single system. Based on our experiments, we develop two novel MVGC schemes that combine and extend techniques from both

Steam and BBF+ to provide protocols that are as space-efficient as BBF+ while achieving time efficiency comparable to Steam and EBR (sometimes slightly better, sometimes slightly worse). We use the MVGC schemes to unlink obsolete versions from version lists, allowing them to be reclaimed by a garbage collector later. The new schemes relax some of the worst-case time guarantees of BBF+ to improve time performance in the common case. However, they maintain the strong worst-case space bounds proved in [5]. Our experimental analysis reveals the practical merit of state-of-the-art techniques for MVGC. It shows that their performance depends significantly on the workload and that no one technique always dominates for both time and space. However, our new schemes show the most consistent performance without any particularly adverse workloads.

Starting from the state-of-the-art MVGC schemes in [5, 7], we had to combine, simplify and extend techniques to obtain the two new MVGC schemes (as discussed in Section 3). They focus on speeding up the removal of obsolete versions once they are identified. This is an important part of MVGC. Our removal algorithms greatly sped up the theoretically efficient BBF+ scheme. One of them was also used to come up with a lock-free version of Steam. This version is much more efficient than the original, which locked entire version lists, for the lock-free multiversion data structures we use in our experiments. Our algorithms use the range tracking (RT) object from BBF+ to identify obsolete versions. A main challenge is that removing these versions can be concurrent with other operations on the version list.

For the first new scheme, called DL-RT, we develop a simpler doubly-linked list implementation supporting removal from the middle of version lists (see Section 4). The implementation, called PDL (Practical Doubly-linked List), sacrifices TreeDL's guarantee of constant amortized time per operation and instead guarantees $O(c)$ amortized time, where c is the number of concurrent removals of consecutive list nodes. This choice was driven by the experimental insight that removing chains of adjacent nodes is very rare in practice: in our experiments the average c was no more than 1.01 across a wide variety of workloads. Experiments show the new, simpler list is much faster than the constant-time list TreeDL when c is small.

Our second algorithm, SL-RT, further improves practical performance for many common workloads, in which version lists tend to be very short, so that linear searches are reasonably efficient. In this case, we use a singly-linked list, which requires less space (no back pointers) and less time for updates due to fewer pointer changes. GVM [35], Steam [7] and HANA [32] also rely on this observation [7]. We develop a singly-linked list implementation, called SSL (Simple Singly-linked List), in Section 5 and use it to implement version lists in SL-RT. Unlike those in Steam and HANA it is lock-free, and unlike GVM, which is lock-free, it does not require any mark bits on pointers. Such mark bits require an extra level of indirection in languages that do not allow directly marking pointers.

Our new list structures support sorted lists with appends on one end, and deletes anywhere, and are likely of independent interest. We prove both correct. Combining our new lists with the range tracking of [5] significantly improves space in many workloads over previous MVGC schemes. The range tracking object identifies list elements to remove (in DL-RT) or lists to traverse and collect (in SL-RT) more accurately than traversing a list on every update as in Steam or traversing all lists periodically as in HANA and GVM.

In our experiments (Section 6), we implemented and studied several schemes: an epoch-based collector [53], BBF+ [5], our new MVGC schemes, and an optimized variant of Steam [7] we developed using SSL. MVGC schemes encompass several mechanisms that may have a crucial effect on their time or space overhead. We focussed on the following mechanisms that our experiments showed to have high impact on performance: (1) data structure for storing versions and (2) how to choose when to remove obsolete versions.

Our experiments study these MVGC schemes in the context of multiversion concurrent data structures. These are data structures that leverage version lists to support atomic read-only transactions (e.g., range queries) alongside single-key insert, delete and lookup operations. They can be used as database indexes [33, 52] and have gained a lot of recent attention [29, 38, 48, 48, 53].

We experimentally compared the MVGC schemes on two quite different concurrent data structures, a multiversion balanced binary search tree and a multiversion hash table. As in previous work [7], we saw that reclaiming intermediate versions is vital for reducing memory overhead (space), especially when there are long rtxs or oversubscription (i.e., when there are more threads than available logical cores). In particular, EBR, which does not collect intermediate versions, requires up to 10 times more space than the others. Perhaps surprisingly, and unexpected to us, Steam sometimes has particularly bad space usage for trees even though it does collect intermediate versions. We found this is due to versioned pointers pointing to objects containing other versioned pointers, as described in Section 6. This leads to cases that require as much as 8 times more space. We found that SL-RT almost always performed best in terms of space.

Steam and EBR typically performed best in terms of update throughput, but for rtx throughput the performance was mixed. For combined throughput there was little difference among the schemes; sometimes Steam and EBR are better and sometimes SL-RT and DL-RT are better. BBF+ is almost always the slowest.

In conclusion, the experiments indicate MVGC schemes using range tracking avoid the high space anomalies of EBR and Steam, with throughput that is similar on mixed workloads and only slightly worse on update-heavy workloads.

Many existing practical MVGC schemes are designed for database systems that use multiversioning and are often heavily intertwined with other parts of those multiversion systems. We believe many of the MVGC techniques are common across both database systems and multiversion data structures. A previous study [7] compares MVGC schemes in various database systems (including [7, 31, 32]) that use different transaction management protocols. For our experiments, we controlled for these confounding factors by applying different MVGC schemes to the same pair of multiversion data structures. We know of no previous comprehensive, apples-to-apples comparison of MVGC techniques.

We summarize the paper's contributions as follows.

- Two novel MVGC schemes that borrow and extend ideas from state-of-the-art space- and time-efficient schemes. The new schemes maintain the strong worst-case space bounds provided by BBF+ [5], while achieving throughput comparable to Steam [7] in most cases.
- An experimental analysis that *fairly* compares state-of-the-art MVGC schemes by applying them to the same multiversion data structures. This sheds light on the practical merit of the schemes, and the reasons that no single MVGC approach is a clear winner in terms of both space and time.
- Several insights that can drive the design of new MVGC schemes, as they have for our new schemes.
- A new lock-free doubly-linked list PDL that is efficient when used in MVGC, and which may be useful elsewhere.
- A new simple singly-linked list implementation SSL that is suitable for MVGC and has better throughput than PDL.

2 Background and Related Work

Multiversioning. In general, using multiversioning involves maintaining a global timestamp representing the current time. Depending on the implementation, this global timestamp can be incremented by rtxs [16, 53], update operations [38], or system clocks [29, 34]. Each update marks any new object (or version of an object) with its timestamp, and adds the new object to the head of the appropriate version list. Each rtx reads the global timestamp and uses it to navigate the version lists. More specifically, an rtx with timestamp t traverses a version list until it finds a version whose timestamp is at most t, and then reads the desired value from that version. The details of how update operations and rtxs are implemented differ between implementations, but this high-level picture is sufficient for our study of MVGC.

A *versioned CAS object* [53] is a specific implementation of multiversioning that we use in our experiments when comparing different MVGC schemes. It is a CAS object that supports looking up older values previously stored in it given a timestamp. Our experiments use these objects to add support for rtxs to a CAS-based lock-free balanced binary search tree [8] and a simple lock-free hash table.

Epoch-based reclamation (EBR). EBR [9, 20] is a memory reclamation technique which divides the execution into epochs. Each operation begins by reading and announcing the current epoch via an announcement array. When all active processes have announced the current epoch, the global epoch counter is incremented and a new epoch begins. This epoch counter is separate from the timestamps used for multiversioning. EBR ensures that all active operations started during either the current or the previous epoch, and therefore any nodes removed during earlier epochs are safe to reclaim. This idea can be extended to work for multiversioning by observing that a rtx will never access any version that was overwritten before the start of the rtx. Therefore, all versions that were overwritten before the previous epoch are safe to reclaim, as they are no longer needed by any active rtxs and are also not on the path to any needed versions.

Because EBR is simple and fast, variants of it are widely used for MVGC [18, 39, 53, 55]. However, EBR-based MVGC schemes reclaim only the oldest versions from the end of the version list, and not obsolete versions in the middle of the list. As a result, EBR does not guarantee space bounds, and can result in high space overhead, particularly if processes execute long rtxs, thus preventing the epoch from being advanced, or if processors are oversubscribed.

Compaction-based reclamation. To address EBR's weakness, compaction-based MVGC schemes [7, 32, 35] identify obsolete versions and remove them from version lists. A version's *time interval* is the interval of timestamps when it was the latest version. In compaction-based schemes, rtxs announce the timestamp they intend to use for traversing the version lists. A version list is compacted by first reading the announced timestamps, sorting them, and then traversing the version list to identify and remove versions whose time intervals do not include any announced timestamp. Since the versions are in order, the traversal takes constant time per version. Steam [7] traverses and compacts a version list whenever a new version is added to it. HANA [32] and GVM [35] do not tie the compaction to adding versions, but instead either go through all lists using background threads, or have the main threads compact once in a while. These techniques have the disadvantage of traversing a version list even if it contains few or no obsolete versions to be collected. HANA and GVM even visit lists when no changes have been made to them. GVM [35] and Steam [7] guarantee that each version list contains $O(P)$ versions, where P is the number of processes, so $O(PM)$ versions are maintained for M lists.

Range-tracking. BBF+ [5] uses a range tracking object to directly identify obsolete versions, avoiding the traversal of the entire list. A range tracking object tracks a set of non-current versions, each of which has been assigned an integer range indicating its time interval. A version in the range tracking object can be reclaimed if its interval does not intersect any of the timestamps announced by rtxs. In

RangeTracker (the linearizable range tracking implementation of [5]), each thread p appends non-current versions in a local list. If the size of p's list becomes $P \log P$, p performs a *flush*. The flush appends p's local list to a shared FIFO queue Q (of lists) [15]. It then dequeues two lists from Q, merges their contents and compares the merged list against a sorted sequence of the current announcements to determine which versions are still needed. The list of needed versions is enqueued to Q, and the set of obsolete versions is returned so that they can be removed from their lists. A flush phase requires $O(P \log P)$ steps and is performed once every $\Theta(P \log P)$ times an element is inserted in the local list. RangeTracker thus ensures amortized constant time for each operation it supports.

Removing from Lists. After identifying an obsolete version, it must be spliced out of its list. Steam [7] and HANA [32] use locks to safely do so. GVM [35] uses a variant of Harris's lock-free singly-linked list [23]. This works since lists are always traversed from their head. However, BBF+ requires safely removing obsolete nodes from the middle of a list, given only a pointer to the node. To do this safely and efficiently, BBF+ uses TreeDL, a custom wait-free doubly-linked list. TreeDL uses an implicit binary tree whose in-order traversal gives the list nodes in order. List nodes that are leaves of the tree are not adjacent in the list and can therefore be removed safely and concurrently. If only leaves are removed, then leaves that store versions that are still needed by rtxs may prevent obsolete versions at internal nodes from being removed, resulting in high space bounds. Thus, TreeDL provides intricate helping mechanisms that also permit the removal of internal list nodes in the tree. This is done with care to ensure list consistency and low space usage.

Other reclamation schemes. A recent improvement on EBR is version-based reclamation [47]. It bounds memory by restarting long running operations, and has very recently been applied to a multiversion data structure [48]. Other memory reclamation schemes have been proposed, but they do not solve the MVGC problem [25, 36, 44, 54], or require special support, in hardware [1, 13] or through the operating system [9, 49]. BBF+ uses Hazard pointers [36] and a recent implementation [2] of reference counting [11, 12, 25] to deallocate nodes that have been spliced out of the lists. Our new MVGC schemes rely on the automatic Java garbage collector to deallocate unreachable nodes.

Other lock-free list structures. We designed SSL and PDL as special-purpose lock-free linked lists for representing version lists. They avoid the extra level of indirection of Valois's singly-linked list [51], which places auxiliary nodes between real nodes. Harris's singly-linked list [24] and others that build on it [19, 37] also add a level of indirection when implemented in Java, since they require a mark bit on list pointers. Besides TreeDL, discussed above, other existing lock-free doubly-linked list implementations are quite complex because they support more operations than we need for version lists [46, 50] or rely on multi-word CAS instructions [3, 21], which are not widely available in hardware (although they can be simulated in software [22, 24]).

3 Proposed MVGC Schemes

We propose two new MVGC schemes, DL-RT and SL-RT. As with BBF+, we use the RangeTracker to identify which versions in the middle of a list can be removed. Our new schemes can be applied in the same way as BBF+; namely, rtxs announce and unannounce their timestamps using the operations provided by the RangeTracker, and whenever a version is overwritten by a newer version, it is passed to the RangeTracker along with its time interval. The two new schemes preserve the correctness conditions (e.g., linearizability, sequential consistency) of the multiversion data structure they are applied to.

Our first scheme, DL-RT is based on a novel doubly-linked list implementation, called PDL, whereas SL-RT employs a new, simple implementation of a singly-linked list, called SSL. The lists are sorted by a key field (e.g., the version timestamp). PDL supports the following operations, introduced in [5, 53]: 1) `tryAppend(x,y)` attempts to append a new element `y` at the end of the list, given that `x` is currently the last list element; 2) `remove(x)` removes the element `x` from the list (and is used for garbage collection); 3) `peekHead` returns the last element of the list (i.e., the current version); 4) `search(key)` returns the latest element of the list whose key is less than or equal to `key` (i.e., the version that was current when `key` was the current timestamp). Consistent with how MVGC uses version lists, PDL assumes a `remove` is called only once on each node, and not on the head node.

The list interface above supports *versioned CAS objects*, as described in [53]. In this case, a CAS operation op that wants to change the value from `v` to `v′` calls `peekHead`, and checks if the value stored in the latest version is `v`. If so, op calls `tryAppend` to add a version containing the value `v′` after `v`. If `tryAppend` fails, a concurrent CAS must have successfully changed the value of the CAS object to something different from `v`. Then, op returns false.

To avoid some of the intricacies of TreeDL we designed our novel implementation PDL based on a simple idea: to remove a node we mark it and then traverse outwards to the first unmarked node in either direction and make them point to each other. PDL ensures that at most $L - R + P$ nodes in total are in the version lists at the end of an execution containing a total of L appends and R removes. This bound is better than the $2(L - R) + O(P \log L_{max})$ bound provided by TreeDL [5], where L_{max} is the maximum number of appends on a single version list. In terms of time, PDL ensures that `tryAppend` and `peekHead` maintain the $O(1)$ bound of TreeDL, but the number of steps for `remove` is $O(c)$, where c is the number of concurrent removals of consecutive nodes in the list. The average value of c was at most 1.01 in each of our experiments. Thus, from a practical perspective, relaxing the time bound for remove to $O(c)$ is a good compromise.

SSL is a singly-linked list implementation, and thus lacks back pointers. In a doubly-linked list, a version can be removed by accessing only the node's immediate neighbourhood. To remove a node from a singly-linked list, we must find its predecessor in the list by traversing the list from the head. Thus, instead of focusing on removing individual nodes, SSL supports a compact routine that traverses the entire list, removing obsolete nodes. Experiments showed that, in many cases, the version lists tend to be short, so traversing a list is often inexpensive. Indeed, in most experiments, SL-RT, which employs SSL, exhibits better throughput than DL-RT. SSL ensures the same $L-R+P$ bound on the number of nodes contained in the version lists as PDL.

PDL and SSL are presented in Sections 4 and 5. Due to lack of space, their correctness proofs and complexity bounds are in the appendices. We used PDL to develop DL-RT, and SSL to develop SL-RT, as well as STEAM+LF, a lock-free version of Steam that we describe below. Specifically, instead of using TreeDL for implementing the version lists as in BBF+, DL-RT employs PDL and SL-RT employs SSL. RangeTracker [5] is used by both DL-RT (to decide when to splice out an individual node), and by SL-RT (to decide when to traverse and compact a version list).

RangeTracker is proved in [5] to use $O(H + P^2 \log P)$ space, where H is the maximum number of needed versions at any point during the execution. We call the left and right pointers of nodes in PDL and the left pointers of nodes in SSL, *access pointers*. A node is *reachable* in PDL or in SSL at some point in time, if it can be reached starting from the list head and following access pointers. Recall that both PDL and SSL ensure that at most $L-R+P$ nodes in total remain reachable in the version lists. This bound and the bound for RangeTracker imply the following.

Theorem 1. *Consider an implementation of a concurrent multi-versioning data structure that uses* DL-RT *or* SL-RT *for garbage collection. At any time t of an execution, the total number of versions that are reachable in the version lists is $O(H + P^2 \log P)$, where H is the maximum, over all times before t, of the number of needed versions.*

If the number of needed versions is large at some point in an execution, this will only influence the size of the version lists for a limited number of steps. More specifically, if the number of needed versions remains below some quantity h in a suffix of an execution, then the number of reachable versions in Theorem 1 will eventually be $O(h + P^2 \log P)$.

4 Doubly-Linked List

Algorithm 1 provides the pseudocode for PDL. Each node stores a left and right pointer, as well as a mark bit to facilitate deletions. It also stores a key key and a value val, which are set when the node is created and never change.

```
1  class Node {
2    int key; Value val; bool mark; // initially false
3    Node* left, right; }
4  class DoublyLinkedList {
5    Node* head;
6    Value peekHead() { return head->val; }
7    Value search(int k) {
8      Node* x = head;
9      while(x->key > k) {
10       x = x->left; }
11     return x->val; } }
12   bool tryAppend(Node* x, Node* y) {
13     Node* w = x->left;
14     // first, help tryAppend(w, x) if necessary
15     if(w != null) CAS(&(w->right), null, x);
16     y->left = x;
17     if(CAS(&head, x, y)) {
18       CAS(&(x->right), null, y);
19       return true;
20     } else return false; }
21   void remove(Node* x) {
22     x->mark = true;
23     Node* left = x->left;
24     Node* right = x->right;
25     Node* leftRight, rightLeft;
26     while(true) {
27       while(left->marked) left = left->left;
28       while(right->marked) right = right->right;
29       rightLeft = right->left;
30       leftRight = left->right;
31       if(left->marked || right->marked) continue;
32       if(!CAS(&right->left, rightLeft, left)) continue;
33       if(!CAS(&left->right, leftRight, right)) continue;
34       break; } }
```

Algorithm 1. Doubly linked list implementation (PDL).

List elements are appended to the right end of the list and a head pointer points to the rightmost node. The list initially contains a sentinel node with key $-\infty$, which remains at the left end of the list at all times.

The peekHead operation simply reads head and returns the node's val field. A search(k) starts at head and follows left pointers until reaching a node whose key is at most k.

A tryAppend(x,y) attempts to append a new node y after a node x that has previously been read from the head pointer. It updates the left pointer of y to x (line 16) and swings the head pointer from x to y using a CAS (line 17). If swinging the head fails, the operation returns false. Otherwise, tryAppend attempts to update the right pointer of x to y using the CAS at line 18. If swinging the head succeeds, but tryAppend pauses before updating x's right pointer, the list is left in an inconsistent state. So, before any subsequent tryAppend adds a node beyond y, it first helps complete the append of

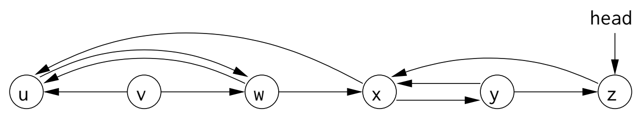


Figure 2. An example state of the list. A `remove(v)` has completed. A pending `remove(w)` and `remove(y)` have performed the CAS at line 32 successfully but not the CAS at line 33. A `search` may be visiting any of the nodes shown.

node `y` by updating `x`'s `right` pointer (line 15). The use of CAS (lines 15 and 18) to initialize `x`'s `right` pointer ensures that the pointer is initialized only once.

A `remove(x)` first marks the node `x`. (Line 22 does this with a simple write, since a marked node remains marked forever.) It then seeks the nearest unmarked nodes on either side and updates them to point to each other using CAS instructions. If either CAS fails (or if the validation at line 31 that the two nodes are still unmarked fails) the `remove` operation tries again to find the nearest unmarked nodes, continuing outwards from where the previous attempt left off. Once both pointers are updated, the `remove` terminates.

`PDL` does not maintain perfect consistency between `left` and `right` pointers: a `remove` may update one but pause or die before updating the other. Moreover, to keep the implementation lightweight, a `remove` does not help other nearby `remove` operations to make the list consistent before applying its own changes; it simply makes its changes (which may, incidentally, help other `remove` operations take effect). See Figure 2 for an example. In addition, a `search` concurrent with `remove` operations may traverse removed nodes. Thus, a careful proof is needed to show some fairly weak invariants suffice for linearizability [26]. We sketch the proof in Section 4.1; for details see Appendix A.

4.1 Correctness

We show that any execution α of the doubly-linked list is linearizable. We assume the following preconditions. When `tryAppend(x,y)` is called, `y` is a new node that has never been used as the second argument of `tryAppend` before and it is not the sentinel node. Moreover, `x` has been read from `head` and `y`'s key is greater than or equal to `x`'s key. There is at most one call to `remove(x)` for each `x`, and it can be called only after a call to `tryAppend(x,*)` has returned true.

If `x` and `y` are pointers to nodes, we use the notation $\mathtt{x} \leftarrow \mathtt{y}$ to indicate that `y->left = x` and $\mathtt{x} \rightarrow \mathtt{y}$ to indicate that `x->right = y`. We use $\mathtt{x} \leftsquigarrow \mathtt{y}$ to indicate there is a path of `left` pointers from `y` to `x`. We say that `x` is added to the list when the `head` pointer is set to `x`. Since there is at most one call to tryAppend with `x` as the second argument, there is at most one step in α that sets `head` to `x`. We define a total order on nodes that are added to the list during α: $\mathtt{x} \prec \mathtt{y}$ if the `head` pointer was set to `x` before it was set to `y` in α.

The following invariant captures some relationships between a node and its neighbours. Essentially, it says that `left` pointers always point to older nodes and `right` pointers point to newer nodes. Moreover, if one of those pointers skips over a node, the skipped node must be marked for deletion. To prove these invariants, we also need similar claims for the local variables `left` and `right` of pending `remove`s.

Invariant 2. *Let C be a configuration and* `y` *be a node.*

1. *If* `y` *has been added to the list (and is not the sentinel), then* `y->left` *is also a node that was added to the list and* `y->left` $\prec$ `y`*. For all* `w`*, if* `y->left` $\prec$ `w` $\prec$ `y` *then* `w` *is marked.*
2. *If* `y->right` *was non-null in a configuration at or before C, then in C* `y->right` *is a node that was added to the list and* `y->right` $\succ$ `y`*. For all* `w`*, if* `y` $\prec$ `w` $\prec$ `y->right` *then* `w` *is marked.*
3. *For any call to* `remove(y)` *in progress, the local variables* `left` *and* `right` *are nodes that have been added to the list and* `left` $\prec$ `y` $\prec$ `right`*. Moreover, for all* `w`*, if* `left` $\prec$ `w` $\prec$ `right` *then* `w` *is marked.*
4. *The* `left` *pointer of the sentinel node is null.*

The `head` pointer is initialized to the sentinel node, and can only be updated on line 17 to a non-null pointer. This fact, Invariant 2 and the preconditions of the operations imply that a null pointer is never dereferenced.

Because `remove` operations update `left` pointers concurrently, and may splice out multiple nodes at once, there is a danger that one `remove` may undo the effects of another. For example, suppose we have four nodes $\mathtt{w} \leftarrow \mathtt{x} \leftarrow \mathtt{y} \leftarrow \mathtt{z}$. If a `remove(x)` updates `y->left` to `w` and a concurrent `remove(y)` updates `z->left` to `x`, then `x` would remain reachable. The following lemma ensures that this cannot happen. (In fact, it is more general in that it applies even when the four nodes are not consecutive in the list.)

Lemma 3. *Suppose* $\mathtt{w} \prec \mathtt{x} \prec \mathtt{y} \prec \mathtt{z}$ *and* $\mathtt{w} \leftarrow \mathtt{y}$ *at some time during the execution. Then there is never a time when* $\mathtt{x} \leftarrow \mathtt{z}$*.*

Invariant 2 can be used to show that each time a `left` pointer changes, it points to an older node.

Lemma 4. *If line 32 does a* `CAS(z->left, y, w)`*, then* $\mathtt{w} \preceq \mathtt{y}$*.*

The *abstract list AL* in a configuration is the sequence of nodes reachable from the head by following `left` pointers. The next three lemmas ensure that a node `x` is removed from *AL* once `remove(x)` has terminated, and only if `remove(x)` has been invoked. Lemma 5 implies that updates to a `left` pointer only remove elements from *AL*: the list after the update is a subsequence of the list prior to the update.

Lemma 5. *If a CAS at line 32 changes* `z->left` *from* `y` *to* `w`*, then* $\mathtt{w} \leftsquigarrow \mathtt{y}$ *in the preceding configuration C.*

Lemma 6. *When a node is removed from AL, it is marked.*

Lemma 7. *When* `remove(x)` *terminates,* `x` *is not in AL.*

These lemmas can be used to verify that the following linearization is correct. We linearize a successful `tryAppend` when it updates the `head` pointer. We linearize a `remove(x)` when a CAS at line 32 causes `x` to be removed from *AL*. If several nodes are removed by a single CAS, we order them in decreasing order by $\prec$. Thus, at all times, the abstract list *AL* agrees with the update operations linearized so far. The following lemma, which requires a technical proof, allows us to linearize `search` operations.

Lemma 8. *Let C be a configuration and* `x` *be the local variable of a pending* `search(k)` *in C. There was a time between the invocation of the* `search` *and C when* `x` *was in AL and either* `x` *was the first node in AL or its predecessor in AL had a key greater than* `k`.

Thus, we can linearize `search(k)` when the returned node `x` is the first in *AL* with key at most `k`.

A `tryAppend` takes $O(1)$ steps. A `search` is wait-free, and `remove` is lock-free. A `remove(x)` operation takes $O(c)$ steps, where c is the length of the chain of adjacent nodes containing `x` that are marked before the `remove` ends. Here, `y` and `z` are *adjacent* if, during the `remove(x)`, `y` $\leftarrow$ `z` or `y` $\rightarrow$ `z`.

5 Singly-Linked List Compaction

We now describe our simple singly-linked list SSL designed to store version lists (Algorithm 3). Each list node stores a version and has an associated timestamp `ts` and a pointer `left` to a preceding version. A `head` pointer stores the most recently appended node. Like PDL, SSL provides a `tryAppend` and `search` operation. Nodes are appended to the list with timestamps in non-decreasing order, so that the list is always sorted by timestamp. Instead of a `remove` operation that splices out individual nodes, SSL provides a `compact` operation that traverses the whole list splicing out obsolete nodes. We assume the list initially contains a sentinel node with timestamp $-\infty$, which always remains at the list's tail.

The `compact` operation is given a sorted list `A` of timestamps announced by rtxs, a threshold timestamp `t` that was read from the global timestamp counter and a node `h` read from the `head` pointer of a version list, which specifies where to begin traversing the list to compact. We describe below exactly how these arguments are obtained so that any rtx that is concurrent with the `compact` or begins after the `compact` will have a timestamp that is either in `A` (because the rtx's announcement is in `A`) or greater than or equal to `t` (because the rtx grabs its timestamp after `t` has been read from the global counter). The following definition describes the nodes that `compact` should retain. A node `x` is *needed w.r.t. to* `A` *and* `t` (abbreviated *needed*(`A`, `t`)) if (1) `x.ts` $>$ `t`, or (2) `x` is the last appended node with timestamp at most `t`, or (3) for some list element `A`$[i]$, `x` is the last appended node whose timestamp is less than or equal to `A`$[i]$.

Since both the version list and `A` are sorted by timestamp, `compact` makes a pass across both the version list and `A` using an algorithm similar to merging two sorted arrays to determine which nodes of the list can be removed. When a removable node is found, it is spliced out of the list by a CAS instruction on its predecessor's `next` field. If multiple consecutive nodes are all found to be removable, a single CAS attempts to splice them all out of the list at once.

```
1  class Node { Node* left; int ts; Value val;}
2  class AnnScan { List<int> A; int t; }
3  AnnScan* scanAnnounce() {
4    AnnScan* newScan = new AnnScan;
5    repeat twice {
6      AnnScan* old = GlobalAnnScan;
7      newScan.t = GlobalTimeStamp;
8      read Announce[1..P] one by one, and sort into newScan.A;
9      if CAS(&GlobalAnnScan, oldScan, newScan) return newScan; }
10   return GlobalAnnScan }
11 void compact(List<int> A, int t, Node* h) { // A is sorted
12   A.appendFront(-1); // padding
13   int i = index of last element of A;
14   Node* cur = h;
15   while(cur != sentinel) {
16     Node* next = cur->left;
17     // skip nodes whose timestamps exceed threshold t
18     if(cur->ts > t) {
19       cur = next;
20     } else {
21       while(A[i] >= cur->ts) i--;
22       if(A[i] >= next->ts) { // next is needed
23         cur = next;
24       } else { // next is not needed
25         Node* newNext = next->left;
26         while(A[i] < newNext->ts) {
27           newNext = newNext->left; }
28         while(!CAS(&(cur->left), next, newNext)) {
29           next = cur->left;
30           if(next->ts <= newNext->ts) break; }
31         cur = cur->left; } } } }
32 bool tryAppend(Node* x, Node* y) {
33   y->left = x;
34   return CAS(&head, x, y); }
35 Value search(int k) {
36   Node* x = head;
37   while(x->ts > k) {
38     x = x->left; }
39   return x->val; }
```

Algorithm 3. Singly linked list (SSL) compaction routine.

In more detail, the variables `i` and `cur` of `compact` are used as pointers into `A` and the version list, respectively, starting at the highest reserved timestamp and the most recent version. Lines 18–19 skip past nodes in the version list whose timestamps are greater than `t`. Line 21 finds the appropriate element of `A` to compare to the timestamp of `cur`. Lines 22–23 skip past a node if it is needed for the timestamp

A[i]. Lines 25–30 handle the removal of next if it is found to be unnecessary. The compact routine first finds the first needed node after next and stores a pointer to it in newNext (lines 25–27). It then tries to splice out a block of consecutive nodes, including next, by changing cur->left from next to newNext at line 28. If the CAS fails, it repeatedly tries to update cur->left to newNext until it succeeds or finds that cur->left points to a node whose timestamp is less than or equal to newNext's. Once the block of nodes is removed, the compact routine proceeds down the list (line 31).

We now discuss how to obtain the arguments for a compact. Each process performing a rtx writes its timestamp into a global array Announce. If a process simply copies the global timestamp into t and copies Announce element by element into a local copy A before calling compact(A,t,h), updates to the list may behave strangely. Because the copies are not made atomically, two concurrent compact operations may disagree about the set of needed nodes: for example, if there are 4 nodes x_1, x_2, x_3, x_4 in the list with timestamps 1, 2, 3, 4, one compact may think x_1, x_3, x_4 are needed while the other thinks x_1, x_2, x_4 are needed. The two operations may get poised to splice out x_2 and x_3, respectively. If they then perform their CAS steps to splice out the nodes, node 2 will be removed by the CAS of the first operation and then become reachable again when the CAS of the second operation occurs. We wish to avoid this situation to maintain good worst-case space bounds. A similar problem may occur if the copy h of the head pointer used as the starting point of a compact is not obtained at the same time as A and t.

This problem could be avoided by taking an atomic snapshot of the global timestamp, Announce and head, but that is too expensive. It suffices that the intervals of time that different processes use to copy the global timestamp and Announce are non-overlapping. This can be guaranteed using a more lightweight synchronization mechanism. (See the scanAnnounce routine in Algorithm 3.) Whenever a process p reads the global timestamp and Announce into local copies t and A, it attempts to install an AnnScan object, which stores the pair (A, t), into a shared variable GlobalAnnScan using a CAS. If p's CAS fails, p tries again to make a local copy and install it in GlobalAnnScan. If p's second CAS on GlobalAnnScan also fails, some other process must have stored an AnnScan object there that it obtained after p's first failed CAS, so p can simply use the AnnScan object it finds in GlobalAnnScan, since it is guaranteed to be recent. Before calling compact(A,t,h), a process gets a snapshot of just two variables, GlobalAnnScan (for the values of A and t) and head (for the value of h).

The lock-free compact routine allows multiple processes to splice nodes out of the list concurrently, while other processes executing search traverse the list. A careful proof of linearizability is in Appendix B. We linearize tryAppend operations when they update head and search operations when they read head. We also show that compact and search are wait-free and that after a compact(A,t,h) routine terminates, all nodes reachable from the head of the list that were appended to the list before h are *needed*(A, t).

6 Experiments

We experimentally compared throughput and memory usage of state-of-the-art MVGC schemes and our new schemes. We tested DL-RT, SL-RT, BBF+, STEAM+LF, and an epoch-based scheme EBR. We implemented all for two different multiversion data structures and used the benchmarking suite from [53]. Only the MVGC code varies. Our code is publicly available on GitHub [1].

6.1 Setup

Machine/Compiler. 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 108MB L3 cache), and 256GB memory. Each core is 2-way hyperthreaded, giving 128 hyperthreads. The machine runs Ubuntu 22.04.1 LTS. Our experiments were written in Java and we used OpenJDK 19.0.1 with heap size set to 64GB. We found that overall, the runtime overhead of Java's garbage collector is small, usually accounting for less than 5% of the overall time in most workloads. We also tried running on smaller heap sizes down to 10GB and did not see much difference in performance. For each data point in our graphs, we ran the processes for 25 seconds to warm up the JVM and then measured 5 runs, each of 5 seconds. We report the average of those 5 runs. Error bars indicate variance. To measure memory usage, at the end of each run, we measure the amount of reachable memory used by the multiversion data structure. This includes metadata maintained by the data structure's MVGC scheme.

Data Structures. We apply MVGC schemes to CAS-based implementations of two concurrent data structures by replacing each CAS object with a versioned CAS object, as in [53]. This adds support for linearizable rtxs on top of the usual insert, delete, and lookup operations. Our experiments test a hash table and a chromatic tree [8], which have quite different characteristics, as the experiments will show.

The hash table is based on separate chaining. Each chain is a sorted linked list of elements. The lists are immutable: to insert or delete an element, we use path copying to create a new copy of the list, and change the hash table entry to point to the new copy using CAS. For short chains, path copying yields a simple, efficient list implementation that avoids the need for the mark bits on pointers that are used in many lock-free lists. In our experiments, we pick the size of the hash table so that the load factor is about 0.5, so chains are very short on average. An important property of this structure is that vCAS objects never point indirectly to other vCAS objects—i.e., the head pointer of the list is a vCAS object but none of the links are. The experiments run by Böttcher et

[1] https://github.com/cmuparlay/ppopp23-mvgc

al. [7] also had this property. The hashtable workload has similarities with many database workloads where the table entries are versioned but not the index [39, 55].

The chromatic tree is a binary search tree with a lazy version of the red-black balancing scheme. Unlike the hash table, in a chromatic tree, or any concurrent tree, the vCAS objects *do* point indirectly to others. In particular, a node points to its child, which can contain other vCAS objects. This has a significant impact on some of the experimental results, since not collecting a node soon enough means that its children will not be collectible. This represents an interesting aspect that does not show up in previous database MVGC work.

Both data structures store 32-bit integer keys and values and support inserts, deletes, and rtxs. On the trees we support range queries—i.e., returning the keys within a range. For hash tables we support returning the value of multiple keys. All operations are linearizable.

Optimizations. We use a backoff scheme in all implementations to reduce contention on the global timestamp counter. A process reads the counter and if, after some variable delay, the counter has increased, it simply uses the incremented value. Otherwise, it updates the counter using CAS. When using version lists as in [53], reading an object requires first reading the location of the head of its version list, and then reading the version the head points to. As suggested in [53], we avoid this level of indirection by satisfying the *recorded-once* property they define and placing the timestamp and pointer to the next older version in the object itself.

We tuned parameters of the range tracking object [5], including the number of lists to dequeue from the shared queue Q in a flush and the number of elements per list. When adding a list to Q, we omit already obsolete versions from it.

The authors of Steam [7] suggest a heuristic of periodically scanning timestamps announced by rtxs, instead of scanning every time a list is compacted. This optimization does not preserve the $O(P)$ bound on the size of a version list but it is crucial for performance. In STEAM+LF we apply this optimization and we scan announcements every 1ms.

Workload. In our experiments, we vary the following parameters: (a) size of the multiversion data structure (denoted by n), (b) operation mix, (c) size of rtxs, (d) number of threads, and (e) the distribution from which keys are drawn. In most experiments, we prefill each data structure with either $n = 100K$ or $n = 10M$ keys. These sizes are chosen to illustrate performance when the data set fits and does not fit into the L3 cache. We perform a mix of operations, consisting of inserts and deletes (done in equal numbers), as well as read transactions. Keys for operations and the initial values of the data structure are drawn randomly from the range $[1, 2n]$, ensuring that the size of the data structure remains approximately n throughout the experiment.

Our rtxs search for all keys in the range $(a, a+s)$ where a is drawn uniformly at random from the range $[1, 2n-s]$ (s is the rtx size). The trees search by using the ordering while hash tables search by checking each individual key in the range. For insert and delete operations, we draw keys from both the uniform distribution and Zipfian distribution with parameter 0.99, which is the default in the YCSB benchmark [10].

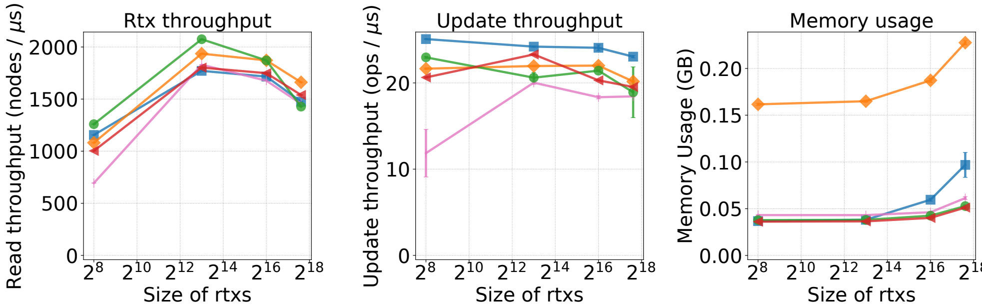
By tuning the number of threads, we can vary the amount of contention, and also study the effects of oversubscription. We see that these have a big impact on how MVGC algorithms perform since they lead to longer version lists.

6.2 Evaluation

Figures 4–8 give a cross section of our experimental results. There was no qualitative difference between experiments on the uniform and Zipfian distributions, so we just include the Zipfian here. More graphs are given in Appendix C. All experiments cover all five GC techniques. Figures 4-6 show rtx and update throughput separately by placing rtxs and updates on separate threads. This highlights the performance differences on updates, since rtxs tend to be affected less. The workload in these figures consists of 40 update threads, 40 threads performing rtxs of size 16, and 40 threads performing variable-sized rtx. The throughputs of the variable-sized rtxs are shown in the leftmost graph of each figure. Figures 7 and 8 aggregate throughput by having each thread perform a mix of both types of operations, which is more representative of a real workload. Based on the experiments, no one algorithm consistently outperforms the others; relative performance depends on various factors described below.

Space. We first consider space usage. Perhaps the most interesting aspect of space is how poorly STEAM+LF performs on trees relative to hash tables. This is explained by our previous discussion of vCAS objects indirectly pointing to other vCAS objects in the chromatic tree but not the hash table. STEAM+LF generally keeps around 1.5 to 2 versions per version list (see tables of Figures 4 and 5) because if there is an ongoing rtx when a new version is added, the existing version is still needed by the rtx and cannot be collected. Soon after, when all current rtxs finish, it becomes obsolete, but STEAM+LF will not compact the list until that location is updated again. This is sometimes called the *dusty corners problem* [31] because corners (i.e., version lists) that are visited infrequently will be cleaned infrequently. This delay only causes a space overhead factor of at most 2 when there is no indirection, but can be much worse with indirection. In particular, the old version that is not collected can point to another node containing versioned objects, and that node and its versions will not be collected. This effect can be seen for the chromatic trees in Figures 4 and 7a. Even for small rtxs, STEAM+LF performs much worse than the others. The experiments in the original Steam paper [7] used a typical database implementation where only table entries are multi-versioned, so versions do not indirectly point to one another and they did not see this issue as severely as we do.

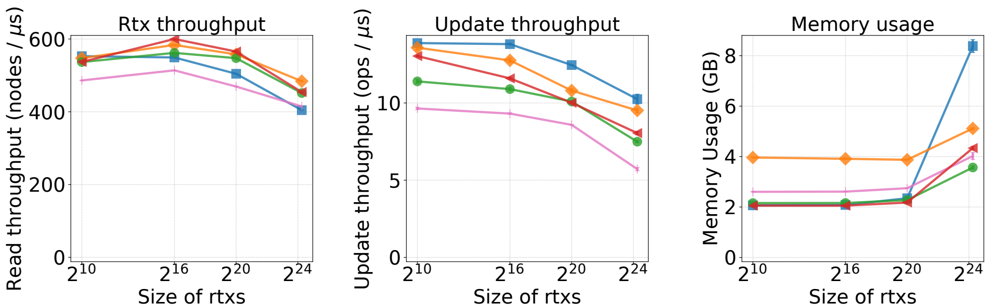
EBR does not suffer from dusty corners since a version can be removed as soon as all active rtxs at the time of an update



Average Version List Lengths

Size of rtxs	2^8	2^{13}	2^{16}	2^{18}
SL-RT	1.03	1.05	1.15	1.39
DL-RT	1.08	1.08	1.17	1.47
BBF+	1.08	1.08	1.2	1.52
Steam+LF	1.52	1.54	1.67	1.95
EBR	1.05	1.05	1.28	2.32

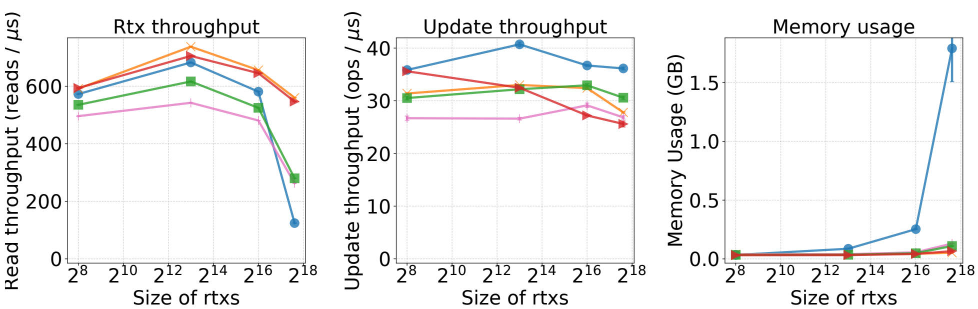
Figure 4. Tree with 100K keys, 40 update threads, 40 fixed-size rtx threads, 40 variable-size rtx threads.



Average Version List Lengths

Size of rtxs	2^{10}	2^{16}	2^{20}	20M
SL-RT	1.0	1.0	1.04	1.42
DL-RT	1.0	1.0	1.04	1.5
BBF+	1.0	1.01	1.05	1.51
Steam+LF	1.29	1.29	1.3	1.49
EBR	1.0	1.01	1.11	2.32

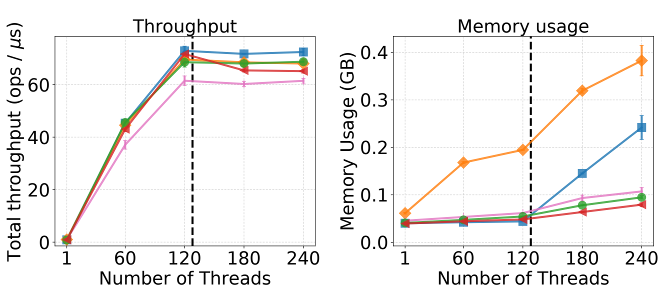
Figure 5. Tree with 10M keys, 40 update threads, 40 fixed-size rtx threads, 40 variable-size rtx threads.



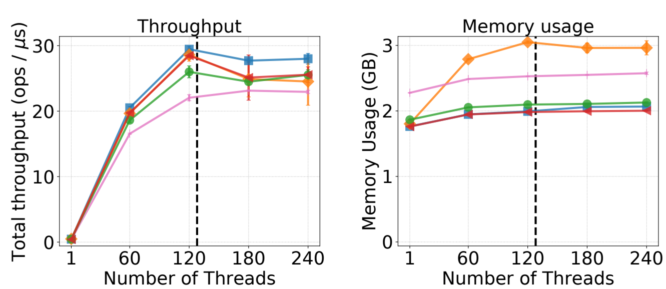
Average Version List Lengths

Size of rtxs	2^8	2^{13}	2^{16}	2^{18}
SL-RT	1.02	1.06	1.34	2.14
DL-RT	1.05	1.08	1.65	4.6
BBF+	1.05	1.08	1.61	4.76
Steam+LF	1.64	1.68	1.93	2.51
EBR	1.01	1.09	2.26	73.63

Figure 6. Hash table with 100K keys, 40 update threads, 40 fixed-size rtx threads, 40 variable-size rtx threads.



(a) Tree with 100K keys



(b) Tree with 10M keys

Figure 7. Workload with each thread performing 50% updates, 49% lookups, and 1% read transactions of size 1024.

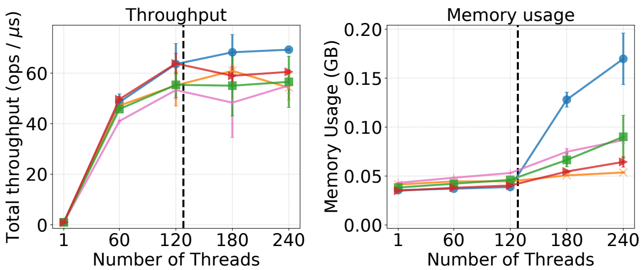


Figure 8. Hash table with 100K keys, each thread performs 50% updates, 49% lookups, and 1% rtxs of size 1024.

have completed, although it has other problems discussed below. The other three algorithms lack dusty corners since a version is removed as soon as the range tracker identifies it, which can be much sooner than the next update.

The space usage of EBR suffers badly for update-heavy workloads with large rtxs and with oversubscription. See, e.g., Figures 5, 6, and 7a. With large rtxs the epochs become long, and update-heavy workloads can create many versions during an epoch that cannot be collected until a following epoch. Oversubscription can further prolong an epoch if a thread in the epoch is delayed. Indeed, this is exactly the problem that removing intermediate versions is trying to solve. Thus, although EBR does fine in many "nice" situations it does very badly in adverse conditions. Figure 6 indicates EBR can use more than an order of magnitude more memory than all the other MVGC schemes in such conditions.

A third observation is that DL-RT, SL-RT, and BBF+ behave as expected in terms of space. In particular, the theory indicates the memory is bounded by three terms: the space needed for the current versions, a constant times the number of needed old versions, and a function of the number of threads. In Figure 7a for a small data structure of size 100K the third term dominates as the number of threads increases for all three algorithms. In Figure 7b for a larger data structure of size 10M the first two terms dominate making it hard to see the increase of space with more threads.

In almost all cases DL-RT, SL-RT, and BBF+ require less space than EBR or STEAM+LF. Importantly, although EBR or STEAM+LF use slightly less memory than the other three in some cases, the memory usage for the other three is much more predictable and never has any particularly bad cases.

Finally, we observe that BBF+ almost always uses more memory than DL-RT, since in BBF+ some nodes are obsolete but not collectable. DL-RT almost always uses more memory than SL-RT because of the additional back pointers in the version lists, and also because SL-RT is able to preemptively splice out versions when traversing a list even before they are returned by the range-tracker. As discussed, the advantages of SL-RT come at the cost of losing time guarantees.

Throughput. We now consider throughput. Varying the GC algorithms affects rtx and update throughput differently. We first consider rtx throughput. The rtx code is the same for all GC schemes, but there are at least two indirect effects.

Firstly, a scheme that compacts version lists less effectively tends to have longer version lists, so rtxs have to follow more version links. This explains the very poor performance of very large rtxs on the hash table with EBR (Figure 6).

Secondly, rtxs are running concurrently with updates and, although they do not communicate with one another directly, there can be indirect effects, such as cache-line interference or faster updates increasing the lengths of version lists. It is hard to draw general conclusions about these effects, but they are likely contributing to the minor variances.

The differences in update throughput among the algorithms is higher than for rtxs. This is to be expected since the code for updates differ significantly—e.g., some traverse whole version lists, and some have to apply operations to the range-tracker data structure. Firstly, we observe that BBF+ almost always does worse than all the others, often significantly. This is the main motivation for designing the DL-RT and SL-RT algorithms. There are a few data points where BBF+ is faster than SL-RT but in these cases the rtx throughput is significantly faster for SL-RT. Secondly, STEAM+LF and EBR mostly dominate the others, although the difference is usually small. This is expected, given the extra cost of maintaining the range-tracker data structure. Thirdly, comparing our two new algorithms, DL-RT and SL-RT, in all graphs except Figure 6, the version lists were short enough that traversing from the beginning was faster than maintaining back pointers and hence SL-RT performs better. Even though the average version list length in Figure 6 is small for SL-RT, the lists that get frequently updated tend to be long and SL-RT traverses an average of 17.9 version list nodes before reaching the one it wishes to splice out

Summary. Our evaluation shows that there is no definitively best MVGC scheme and we provide several insights into the types of workload and data structures each scheme is most suitable for. While the performance of different schemes is often similar, particularly in terms of throughput, there are a few important exceptions. In particular, EBR uses up to 10 times more memory than the other schemes in workloads with long rtxs and Steam uses 8 times more memory in hierarchical data structures like trees. Our new schemes (SL-RT and DL-RT) are consistently faster than BBF+ and do not incur exceptionally high space usage in any workload.

Acknowledgments

We thank the anonymous referees for their helpful comments. This research was supported by NSERC; NSF grants CCF-1901381, CCF-1910030, CCF-1919223, and CCF-2119069; and the Hellenic Foundation for Research and Innovation (HFRI) under the "Second Call for HFRI Research Projects to support Faculty members and researchers" (project number: 3684).

References

[1] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. 2014. StackTrack: An Automated Transactional Approach to Concurrent Memory Reclamation. In *Proc. 9th European Conference on Computer Systems*. 25:1–25:14. https://doi.org/10.1145/2592798.2592808

[2] Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. 2021. Concurrent deferred reference counting with constant-time overhead. In *Proc. 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 526–541.

[3] Hagit Attiya and Eshcar Hillel. 2013. Built-in coloring for highly-concurrent doubly-linked lists. *Theory of Computing Systems* 52, 4 (2013), 729–762.

[4] 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

[5] Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, Yihan Sun, and Yuanhao Wei. 2021. Space and Time Bounded Multiversion Garbage Collection. In *Proc. 35th International Symposium on Distributed Computing*. 12:1–12:20. A full version is available from https://arxiv.org/abs/2108.02775.

[6] 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

[7] 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.

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

[9] Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proc. ACM Symposium on Principles of Distributed Computing*. 261–270. https://doi.org/10.1145/2767386.2767436

[10] 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

[11] Andreia Correia, Pedro Ramalhete, and Pascal Felber. 2021. OrcGC: automatic lock-free memory reclamation. In *Proc. 26th ACM Symposium on Principles and Practice of Parallel Programming*. 205–218.

[12] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele. 2001. Lock-Free Reference Counting. In *Proc. 20th ACM Symposium on Principles of Distributed Computing*. 190–199. https://doi.org/10.1145/383962.384016

[13] Aleksandar Dragojević, Maurice Herlihy, Yossi Lev, and Mark Moir. 2011. On the Power of Hardware Transactional Memory to Simplify Memory Management. In *Proc. 30th ACM Symposium on Principles of Distributed Computing*. 99–108. https://doi.org/10.1145/1993806.1993821

[14] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-Blocking Binary Search Trees. In *Proc. 29th ACM Symposium on Principles of Distributed Computing* (Zurich, Switzerland). 131–140. https://doi.org/10.1145/1835698.1835736

[15] Panagiota Fatourou and Nikolaos D Kallimanis. 2014. Highly-efficient wait-free synchronization. *Theory of Computing Systems* 55, 3 (2014), 475–520.

[16] 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

[17] Yotam M. Y. Feldman, Artem Khyzha, Constantin Enea, Adam Morrison, Aleksandar Nanevski, Noam Rinetzky, and Sharon Shoham. 2020. Proving Highly-Concurrent Traversals Correct. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 128 (Nov. 2020), 29 pages. https://doi.org/10.1145/3428196

[18] Sérgio Miguel Fernandes and João Cachopo. 2011. Lock-Free and Scalable Multi-Version Software Transactional Memory. In *Proc. 16th ACM Symposium on Principles and Practice of Parallel Programming*. 179–188. https://doi.org/10.1145/1941553.1941579

[19] Mikhail Fomitchev and Eric Ruppert. 2004. Lock-Free Linked Lists and Skip Lists. In *Proc. 23rd ACM Symposium on Principles of Distributed Computing*. 50–59. https://doi.org/10.1145/1011767.1011776

[20] Keir Fraser. 2004. *Practical Lock-freedom*. Technical Report UCAM-CL-TR-579. University of Cambridge, Computer Laboratory.

[21] Michael Greenwald. 2002. Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In *Proc. 21st ACM Symposium on Principles of Distributed Computing*. 260–269.

[22] Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi. 2020. Efficient Multi-Word Compare and Swap. In *Proc. 34th International Symposium on Distributed Computing (LIPIcs, Vol. 179)*.

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

[24] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A practical multi-word compare-and-swap operation. In *Proc. International Symposium on Distributed Computing (LNCS, Vol. 2508)*. 265–279.

[25] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. 2005. Nonblocking Memory Management Support for Dynamic-Sized Data Structures. *ACM Trans. Comput. Syst.* 23, 2 (May 2005), 146–196. https://doi.org/10.1145/1062247.1062249

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

[27] Idit Keidar and Dmitri Perelman. 2015. Multi-versioning in Transactional Memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*. LNCS, Vol. 8913. Springer, 150–165. https://doi.org/10.1007/978-3-319-14720-8_7

[28] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. 2019. MV-RLU: Scaling Read-Log-Update with Multi-Versioning. In *Proc. 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 779–792. https://doi.org/10.1145/3297858.3304040

[29] Tadeusz Kobus, Maciej Kokociński, and Paweł T. Wojciechowski. 2022. Jiffy: A Lock-Free Skip List with Batch Updates and Snapshots. In *Proc. 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 400–415. https://doi.org/10.1145/3503221.3508437

[30] Priyanka Kumar, Sathya Peri, and K. Vidyasankar. 2014. A TimeStamp Based Multi-version STM Algorithm. In *Proc. International Conference on Distributed Computing and Networking*. 212–226.

[31] Per-Åke Larson, Mike Zwilling, and Kevin Farlee. 2013. The Hekaton Memory-Optimized OLTP Engine. *IEEE Data Eng. Bull.* 36, 2 (2013), 34–40.

[32] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. 2016. Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA. In *Proc. International Conference on Management of Data*. 1307–1318. https://doi.org/10.1145/2882903.2903734

[33] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proc. 29th IEEE International Conference on Data Engineering*. 38–49.

[34] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. 2017. Cicada: Dependably fast multi-core in-memory transactions. In *Proc. ACM International Conference on Management of Data*. 21–35.

[35] Li Lu and Michael L Scott. 2013. Generic multiversion STM. In *Proc. International Symposium on Distributed Computing (LNCS, Vol. 8205)*. 134–148.

[36] M.M. Michael. 2004. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (2004), 491–504. https://doi.org/10.1109/TPDS.2004.8

[37] Maged M. Michael. 2002. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proc. 14th ACM Symposium on Parallel Algorithms and Architectures*. 73–82. https://doi.org/10.1145/564870.564881

[38] Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. 2022. Bundling linked data structures for linearizable range queries. In *Proc. 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 368–384. https://doi.org/10.1145/3503221.3508412

[39] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast serializable multi-version concurrency control for main-memory database systems. In *Proc. ACM SIGMOD International Conference on Management of Data*. 677–689.

[40] Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. Verifying Linearizability with Hindsight. In *Proc. 29th ACM Symposium on Principles of Distributed Computing*. 85–94.

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

[42] Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. 2011. SMV: Selective Multi-Versioning STM. In *Proc. International Symposium on Distributed Computing (LNCS, Vol. 6950)*. 125–140.

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

[44] Pedro Ramalhete and Andreia Correia. 2017. Brief Announcement: Hazard Eras–Non-Blocking Memory Reclamation. In *Proc. 29th ACM Symposium on Parallelism in Algorithms and Architectures*. 367–369. https://doi.org/10.1145/3087556.3087588

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

[46] Niloufar Shafiei. 2015. Non-Blocking Doubly-Linked Lists with Good Amortized Complexity. In *Proc. 19th Int. Conference on Principles of Distributed Systems (LIPIcs, Vol. 46)*. 35:1–35:17.

[47] Gali Sheffi, Maurice Herlihy, and Erez Petrank. 2021. VBR: Version Based Reclamation. In *Proc. 35th International Symposium on Distributed Computing (LIPIcs, Vol. 209)*. 35:1–35:18. https://doi.org/10.4230/LIPIcs.DISC.2021.35

[48] Gali Sheffi, Pedro Ramalhete, and Erez Petrank. 2022. EEMARQ: Efficient Lock-Free Range Queries with Memory Reclamation. https://doi.org/10.48550/ARXIV.2210.17086

[49] Ajay Singh, Trevor Brown, and Ali Mashtizadeh. 2021. NBR: Neutralization Based Reclamation. In *Proc. 26th ACM Symp. on Principles and Practice of Parallel Programming*. 175–190. https://doi.org/10.1145/3437801.3441625

[50] Håkan Sundell and Philippas Tsigas. 2008. Lock-free deques and doubly linked lists. *J. Parallel and Distributed Computing* 68, 7 (2008), 1008–1020.

[51] John D. Valois. 1995. Lock-free linked lists using compare-and-swap. In *Proc. 14th ACM Symposium on Principles of Distributed Computing*. 214–222.

[52] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. 2018. Building a Bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*. 473–488.

[53] 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 *Proc. ACM Symposium on Principles and Practice of Parallel Programming*. 31–46. A full version is available from https://arxiv.org/abs/2007.02372.

[54] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. 2018. Interval-Based Memory Reclamation. In *Proc. 23rd ACM Symp. on Principles and Practice of Parallel Programming*. 1–13. https://doi.org/10.1145/3178487.3178488

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

A Proofs for the Doubly-Linked List

Invariant 2. *Let C be a configuration and* `y` *be a node.*

1. *If* `y` *has been added to the list (and is not the sentinel), then* `y->left` *is also a node that was added to the list and* `y->left` $\prec$ `y`. *For all* `w`, *if* `y->left` $\prec$ `w` $\prec$ `y` *then* `w` *is marked.*
2. *If* `y->right` *was non-null in a configuration at or before C, then in C* `y->right` *is a node that was added to the list and* `y->right` $\succ$ `y`. *For all* `w`, *if* `y` $\prec$ `w` $\prec$ `y->right` *then* `w` *is marked.*
3. *For any call to* `remove(y)` *in progress, the local variables* `left` *and* `right` *are nodes that have been added to the list and* `left` $\prec$ `y` $\prec$ `right`. *Moreover, for all* `w`, *if* `left` $\prec$ `w` $\prec$ `right` *then* `w` *is marked.*
4. *The* `left` *pointer of the sentinel node is null.*

Proof. Initially, the only node is the sentinel node and its `right` pointer is null, and there is no pending call to `remove`, so the claim holds vacuously. We show that every step preserves each of the three claims of the invariant.

1. To prove that claim 1 is preserved, we must only consider steps that can add a node to the list or change the `left` pointer of a node.
   When a node `y` is added to the list at line 17 of a call to `tryAppend(x,y)`, `y->left` has been set to `x` at line 16. By the preconditions, this is the unique call of the form `tryAppend(*,y)`, so a pointer to `y` has not previously been written into shared memory. So, no process could have updated `y->left` and it is therefore still `x`. By the precondition of `tryAppend`, `x` is a node that has previously been read from `head`, so it has been added to the list and `x` $\prec$ `y`. Since the CAS on `head` changes it from `x` directly to `y`, there are no nodes `w` satisfying `x` $\prec$ `w` $\prec$ `y`.
   The only instructions that update a `left` pointer are lines 16 and 32. As mentioned above, a node `y` has not been added to the list yet when line 16 of `tryAppend(x,y)` updates `y->left`, so there is nothing more to prove.
   Suppose an execution of line 32 of a `remove(x)` sets `y->left` to `v`. Then, `v` and `y` are the values of the `remove`'s local variables `left` and `right`. By induction hypothesis 3, `v` has been added to the list and `v` $\prec$ `x` $\prec$ `y` and every node `w` satisfying `v` $\prec$ `w` $\prec$ `y` is marked. Thus, after the CAS, every node `w` satisfying `y->left` $\prec$ `w` $\prec$ `y` is marked.
2. To prove that claim 2 is preserved, we must only consider steps that can change the `right` pointer of a node. The only instructions that set a `right` pointer to a non-null value are lines 15, 18 and 33.
   First, consider line 18 of a `tryAppend(x, y)`, which sets `x->right` to `y`. Since `head` was changed directly from `x` to `y` on line 17, `y` has been added to the list and `x` $\prec$ `y`, and there are no nodes `w` satisfying `x` $\prec$ `w` $\prec$ `y`.
   Next, consider an execution of line 33 of a `remove(y)` sets `x->right` to `z`. Then, `x` and `z` are the values of the local variables `left` and `right`. By the induction hypothesis, `x` $\prec$ `y` $\prec$ `z` and every node `w` satisfying `x` $\prec$ `w` $\prec$ `z` is marked. Thus, after the CAS, every node `w` satisfying `x` $\prec$ `w` $\prec$ `x->right` is marked.
   Finally, consider a CAS step s at line 15 of `tryAppend(y,*)` that changes `x->right` from null to `y`. By the precondition of `tryAppend`, `y` has been added to the list. The `tryAppend` read `x` from `y->left` at line 13 before s. Since `x` is not null at line 15, `y` cannot be the sentinel node, by induction hypothesis 4. So `x` is a node that has been added to the list and `x` $\prec$ `y` by induction hypothesis 1. We prove the remainder of the claim by showing that there is no `w` such that `x` $\prec$ `w` $\prec$ `y`.
   Since `x->right` is null in the configuration before s, it follows from induction hypothesis 2 that
   $$\texttt{x->right} = \text{null at all times before } s. \qquad (1)$$
   Node `x` cannot be marked when s occurs: before a `remove(x)` could have marked `x`, a `tryAppend(x,*)` must have returned true (by the precondition for `remove`), and that `tryAppend` would have set `x->right` to a non-null value at line 18, contrary to (1).
   To derive a contradiction, suppose there is some `w` such that `x` $\prec$ `w` $\prec$ `y`. Consider the minimal such `w` with respect to the total order $\prec$. (I.e., `w` is the successor of `x` in the order $\prec$.) Since `x` $\leftarrow$ `y` at line 13 before s, w is marked, by induction hypothesis 1. This means that a `remove(w)` has been invoked before s. By the precondition of `remove`, a `tryAppend(w,w′)` returned true before s, for some `w′`. Consider line 13 of that `tryAppend`. When it occurs, `x` is unmarked (since it is still unmarked at s). By induction hypothesis 1, `w->left` $\succeq$ `x` (since `x` is unmarked) and `w->left` $\preceq$ `x` (since `x` is the predecessor of `w` in the order $\prec$). Thus, line 13 reads `x` from `w->left`. Line 15 of that `tryAppend(w,w′)` (which also occurs before s) will perform a `CAS(x->right, null, w)`, violating (1). This contradiction proves that there is no `w` such that `x` $\prec$ `w` $\prec$ `y`.
3. We prove that claim 3 is preserved by all updates to the local variables `left` and `right` in a `remove` operation. By the precondition of `remove(y)`, `y` is not the sentinel node. Moreover, a call to `tryAppend(y,*)` has returned true, so by the preconditions of `tryAppend`, we know that `y` has been added to the list before the `remove` begins. Moreover, the `tryAppend` executed line 18, after which `y->right` is non-null. So, by induction hypothesis 2, `y->right` points to a node that has been added to the list.
   So, when the local variable `left` in the `remove(y)` operation is initialized on line 23 to the value read from `y->left`, the claim follows from induction hypothesis 1. Similarly, when the local variable `right` is initialized

on line 24, the claim follows from induction hypothesis 2.

When the local variable `left` is advanced from x to x' at line 27 in a `remove(y)`, we have $x' \leftarrow x$. Since x is marked, so it is not the sentinel node, since the precondition to `remove` ensures that the sentinel node is never marked. By induction hypothesis 1, $x' \prec x$ and each node w satisfying $x' \prec w \prec x$ is marked. By induction hypothesis 3, $x \prec y$ and each node w satisfying $x \prec w \prec y$ is marked. Moreover, by the test on line 27, x is marked. The claim follows.

A symmetric argument shows that advancing the local variable `right` at line 28 also preserves the invariant.

4. Finally we show that claim 4 is preserved because no step can change the `left` pointer of the sentinel node. Line 16 of `tryAppend` changes the `left` field of a non-sentinel node, by the precondition to `tryAppend`. Consider a CAS step at line 32 of `remove(y)` that changes `z->left`. By induction hypothesis 3, $y \prec z$, so z cannot be the sentinel node (which is the minimum element in the ordering $\prec$).

□

Lemma 3. *Suppose* $w \prec x \prec y \prec z$ *and* $w \leftarrow y$ *at some time during the execution. Then there is never a time when* $x \leftarrow z$.

Proof. To derive a contradiction, suppose the claim is false.

Since there is a node x between w and y in the total order $\prec$, the `head` pointer takes the value x between the times it takes the values w and y. Thus, y is not added to the list by a `tryAppend(w,y)` operation, so the initial value of `y->left` is not w. So, there is a CAS on line 32 of a `remove` operation that changes `y->left` to w. This CAS was preceded by a test at line 31 that found y was not marked. At that time, the local variables `left` and `right` of the `remove` are equal to w and y. By Invariant 2.3, x is marked since $w \prec x \prec y$. Thus, x is marked before y.

An identical argument shows that a CAS at line 32 of a `remove` operation changes `z->left` to x and at the execution of line 31 before that CAS, x is not marked and y is marked, so y is marked before x. This is a contradiction. □

Lemma 4. *If line 32 does a* `CAS(z->left, y, w)`*, then* $w \preceq y$.

Proof. To derive a contradiction, suppose $w \succ y$. Consider the configuration C when y is read from `z->left` on line 30 prior to the CAS. At C, w and z are the values of the local variables `left` and `right` of the `remove`. By Invariant 2.3, $w \prec z$. In C, we have `z->left` $= y \prec w \prec z$. By Invariant 2.1, w is marked in C. This contradicts the fact that w is unmarked when line 31 is executed after C. □

Recall that $x \leftsquigarrow z$ if x can be reached from z by following `left` pointers. More formally, $x \leftsquigarrow z$ if (a) $x = z$ or (b) for some y, $y \leftarrow z$ and $x \leftsquigarrow y$.

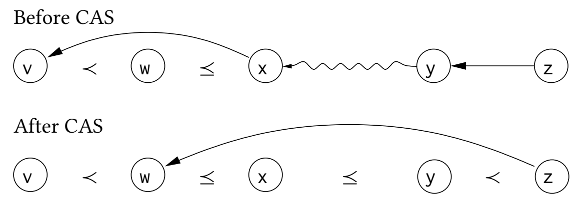


Figure 9. Proof of Lemma 5 and Lemma 16.

Lemma 5. *If a CAS at line 32 changes* `z->left` *from* y *to* w*, then* $w \leftsquigarrow y$ *in the preceding configuration* C.

Proof. Refer to Figure 9. Let x be the minimum node (with respect to $\prec$) such that $x \succeq w$ and $x \leftsquigarrow y$ in C. (This minimum is well-defined, since there exists a node x that satisfies both of these properties: $y \succeq w$ by Lemma 4 and $y \leftsquigarrow y$ holds trivially.) Since $x \leftsquigarrow y \leftarrow z$, we have $x \prec z$ by Lemma 2.1.

Let v be the value of `x->left` in C. By Invariant 2.1, $v \prec x$. It must be the case that $v \prec w$; otherwise we would have $v \succeq w$ and $v \leftsquigarrow y$, which would contradict the fact that x is the *minimum* node that satisfies these two properties.

To derive a contradiction, suppose $w \prec x$. Then we have $v \prec w \prec x \prec z$. We also have $w \leftarrow z$ after the CAS and $v \leftarrow x$ in C, which contradicts Lemma 3. Thus, $w \succeq x$. By definition of x, $x \succeq w$ and $x \leftsquigarrow y$ in C. Thus, $x = w$ and $w \leftsquigarrow y$. □

Lemma 6. *When a node is removed from AL, it is marked.*

Proof. A node can only be removed from the abstract list by a CAS step on a `left` pointer. Consider a CAS at line 32 that changes `z->left` from y to w. By Invariant 2.1 and Lemma 5, only nodes x satisfying $w \prec x \prec y$ can be removed from the abstract list by this CAS. Since w and y are the values of the local variables `left` and `right` in the `remove` operation that performs the CAS, Invariant 2.3 ensures that all such nodes x are marked when the CAS occurs. □

To ensure that we can linearize `remove` operations, we show that a `remove(x)` successfully removes x from the abstract list. The `remove` can only terminate when it performs a successful CAS on the `left` pointer of some node appended to the list after x. The following lemma shows that when that happens, x is no longer in the abstract list.

Lemma 7. *When* `remove(x)` *terminates,* x *is not in AL.*

Proof. Since the `remove(x)` terminates, it performs a successful CAS on a `left` pointer at line 32. Let w and z be the values of the `remove`'s local variables `left` and `right` when the CAS occurs. The CAS updates `z->left` to w. By Invariant 2.3, $w \prec x \prec z$.

To derive a contradiction, suppose that x is still in the abstract list in the configuration C after this CAS.

Since x is in the abstract list and `z->left` $\prec x \prec z$, there must be nodes v' and v in the abstract list such that $x \leftsquigarrow$

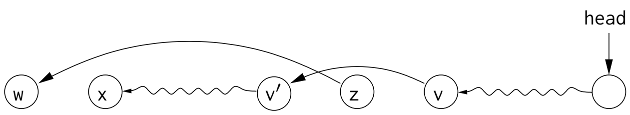


Figure 10. Proof of Lemma 7.

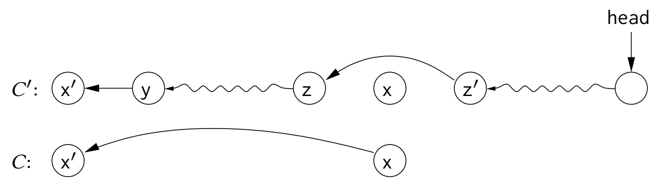


Figure 11. Proof of Lemma 8.

$\mathsf{v}' \leftarrow \mathsf{v} \leftsquigarrow$ head with $\mathsf{v}' \prec \mathsf{z} \prec \mathsf{v}$. Refer to Figure 10. More formally, let v be the minimum node (with respect to the order $\prec$) such that $\mathsf{v} \succeq \mathsf{z}$ and $\mathsf{v} \leftsquigarrow$ head in C. Let $\mathsf{v}' =$ `v->left`. It follows from the definition of v that $\mathsf{v}' \prec \mathsf{z}$. If v were equal to z, then since x is in the abstract list, x would have to be reachable by following `left` pointers from w, which would violate 2.1, since $\mathsf{w} \prec \mathsf{x}$. Thus, $\mathsf{v} \succ \mathsf{z}$.

So, we have $\mathsf{w} \prec \mathsf{x} \preceq \mathsf{v}' \prec \mathsf{z} \prec \mathsf{v}$ and $\mathsf{w} \leftarrow \mathsf{z}$ and $\mathsf{v}' \leftarrow \mathsf{v}$, which violates Lemma 3. □

We now specify the linearization points of update operations so that we maintain the invariant that, for all configurations C, the abstract list L_C in that configuration is exactly the list that would be obtained by performing the operations linearized before C in their linearization ordering. We linearize a successful `tryAppend` operation (i.e., one that returns `true`) when it updates the `head` pointer at line 17. We linearize a `remove(x)` when a CAS at line 32 removes x from the abstract list. This step may be performed by the `remove` itself or by another `remove` operation. By Lemma 5, n will remain out of the abstract list forever once it is removed. By Lemma 6, only marked nodes are removed from the abstract list when a `left` pointer is updated. By Lemma 7, there is a well-defined linearization point for each `remove` operation that terminates, and it is before the `remove` terminates. By Lemma 6, it is also after the `remove` operation performs line 22, so the linearization point is during the execution interval of the `remove`. Thus, the linearization is consistent with the abstract list.

A CAS of a `left` pointer may remove several nodes from the abstract list, and therefore be used as the linearization point of several `remove` operations. When linearizing such a batch of `remove` operations, we order them in decreasing order by $\prec$. In other words, we remove the batch of nodes from right to left. Thus, when a CAS on a `left` pointer splices several nodes out of the abstract list, we can think of the abstract list as undergoing a sequence of changes by performing those `remove` operations one by one. This will be important, because it is sometimes necessary to linearize searches in the *middle* of such a batch of `remove` operations.

We now describe how to linearize `search` operations. It is a common technique to argue that each node visited by a `search` *was* in the data structure at some time during the search (e.g., [14, 40]). Such proofs often rely on the fact that once a node is removed from the data structure, its pointers can no longer be changed. This does not hold for our data structure, but we can still establish the required property using Lemmas 3 and 5. (An alternative approach would be to use the forepassed condition defined in [17].)

Lemma 8. *Let C be a configuration and* x *be the local variable of a pending* `search(k)` *in C. There was a time between the invocation of the* `search` *and C when* x *was in AL and either* x *was the first node in AL or its predecessor in AL had a key greater than* k.

Proof. Consider any `search(k)` operation in the execution. We prove the claim holds for every configuration C during the `search` by induction on the number of steps the `search` has performed.

For the base case, consider the configuration C after the `search` is invoked and executes line 8, x is initialized to `head`, so it is the first node in the abstract list in C.

For the induction step, we must just verify that steps that modify the local variable x preserve the claim. Consider an execution of line 10 that advances the `search` from node x to x′, and let C be the configuration after that step. Then, in C, $\mathsf{x}' \leftarrow \mathsf{x}$. By the test at line 9, `x->key` $>$ k.

First, consider the case where x is in L_C. Since $\mathsf{x}' \leftarrow \mathsf{x}$ in C, x′ is also in L_C and its predecessor x in L_C has a key greater than k. Thus, the claim is satisfied at C.

So, for the remainder of the proof, assume x is not in L_C. By the induction hypothesis, there was an earlier time during the `search` when x was in the abstract list. So there was a CAS step s during the `search` that removed x from the abstract list.

Lemma 5 implies that if $\mathsf{x}' \leftsquigarrow \mathsf{x}$ after some CAS on a `left` pointer, then $\mathsf{x}' \leftsquigarrow \mathsf{x}$ before that CAS too. Since $\mathsf{x}' \leftsquigarrow \mathsf{x}$ in C, it follows that $\mathsf{x}' \leftsquigarrow \mathsf{x}$ holds in all configurations between the addition of x to the abstract list and C. In particular, this means that $\mathsf{x}' \leftsquigarrow \mathsf{x}$ in the configuration before s. So x′ is in the abstract list in that configuration. We consider the two possible cases.

- Suppose s removes both x and x′ from the abstract list. Then s removes some sequence of nodes from the abstract list by updating the `left` pointer of some node z. Since we linearize a batch of `remove` operations from right to left, z is the predecessor of x′ in the abstract list just before the linearization of `remove(x′)`. It follows from Lemma 5 that $\mathsf{x} \leftsquigarrow \mathsf{z}$ in the configuration before s. By the precondition that the list is always sorted, we have `z->key` $\geqslant$ `x->key` $>$ k. Thus, the claim is satisfied just before the linearization point of the remove(x′).

- Suppose x′ is still in the abstract list in the configuration C' following s. If x′ is the first element of $L_{C'}$, then the claim holds.
  Otherwise, there is some y such that in C', x′ ← y ⇜ head. To derive a contradiction, suppose y ≺ x. Then there must be nodes z and z′ such that y ⪯ z ≺ x ≺ z′ and y ⇜ z ← z′ ⇜ head in C'. (See Figure 11: at some point in the path from head to y, there must be consecutive nodes z ← z′ such that z ≺ x ≺ z′ since x is not in the abstract list.) So we have x′ ≺ x ≺ x ≺ z′ and z ← z′ in C' and x′ ← x in C. This violates Lemma 3. Hence, y ⪰ x. We cannot have y = x, since y in in $L_{C'}$ and x is not. So, y ≻ x. By the precondition that nodes are appended with non-decreasing keys, this means that y->key ⩾ x->key > k. So, x′ is in $L_{C'}$ and its predecessor y has a key greater than k, as required.

□

Suppose a search(k) returns the value of a node x. By the exit condition on line 9, vx->key ⩽ k. Lemma 8 says there is a time during the search when x is in the abstract list and its predecessor in the list has a key greater than k. Thus, at that time, x is the first node in the abstract list whose key is less than or equal to k. We choose that time as the linearization point of the search.

B Correctness of the Singly-Linked List

We assume that the following preconditions are satisfied.

1. If tryAppend(x,y) is invoked, then x has been read from head, and no other tryAppend(*,y) has been invoked.
2. If a node x is appended to the list before y, x.ts ⩽ y.ts.
3. While a search(t) is in progress, t appears in Announce.
4. At all times after a compact(A,t,*) is invoked and for all i, either Announce[i] ∈ A or Announce[i] ⩾ t.

These conditions are satisfied naturally in MVGC applications. Since compact takes its parameters from an AnnScan object (A,t), scanAnnounce ensures that t was copied from shared memory before A. Since the global timestamp is non-decreasing, any value stored in Announce[i] after that entry was copied into Awill be greater than t. (We assume rtxs announcing a timestamp copy the global timestamp into Announce atomically, as described in [53]. An optimization that avoids this is described in Appendix B.2.)

As in Section 4.1, we say a node is appended to the list when a pointer to the node is stored in head. We consider the sentinel node to be appended to the list when the list is created. We also use the notation x ← y and x ≺ y as in Section 4.1. We first prove some simple facts about the algorithm.

Invariant 9.

1. *If* y *has been appended to the list and is not the sentinel node and* w ← y *then* w *is a node that has been appended to the list and* w ≺ y.
2. *If a CAS at line 28 changes* y->left *from* x *to* w*, then* w ≺ x.
3. *A* compact *never sets* cur*,* next *or* newNext *to null.*

Proof. We show that each step preserves the invariant. Assume the invariant holds prior to the step. We need only consider steps that append a node, successfully CAS a left pointer and update the local variables cur, next and newNext.

First, consider a step that appends a node y to the list (line 34). If w ← y when y is appended to the list by a CAS at line 34, then head is changed from w directly to y. Thus, w is a node that was appended to the list before y.

The only step that changes a left pointer after it is initialized is a successful CAS at line 28. Suppose this step changes y->left from x to w. Then, x and w are the values of local variables next and newNext when the CAS occurs. Prior to the CAS, w has been reached from y by following left pointers, first at line 16 and then at lines 25–27. Moreover, before advancing from one node to the next by reading a left pointer (at line 16, 25 or 27), the algorithm first checks (at line 15, 22 or 26, respectively) that the former node is not the sentinel, either explicitly, or by checking that the node's ts field is greater than A[i]. Since the invariant holds before the CAS, w has been appended and w ≺ x ≺ y.

As mentioned in the previous paragraph, when we set next at line 16 or set newNext at lines 25 or 27 by reading the left field of a node, we first check that that node is not the sentinel. Since the invariant holds before reading this left field, the value we store in next or newNext is non-null. Similarly, if next is updated at line 29, the test at line 15 guarantees that cur is not the sentinel, so its left pointer is non-null.

We check that all steps that set the cur variable give it a non-null value. The cur variable is set to the value read from head at line 14, and head is never null. Lines 19 and 23 copy the value next into cur, and this value is non-null, by the induction hypothesis. Line 31 sets cur to the value read from the left field of a node that is known not to be the sentinel (by the test at line 15), so its left field is non-null by the induction hypothesis. □

It follows from Invariant 9 that all pointers dereferenced in compact are non-null.

The following technical lemma shows that compact updates i appropriately to carry out the test of whether node next is needed.

Lemma 10. *If the test on line 22 fails, then either* i *is the index of the last entry of* A *or* cur->ts ⩽ A[i+1].

Proof. Let C be the configuration after the test on line 22 fails. Consider the configuration C_i after the last time the local variable i was updated prior to C. If this was when i was initialized at line 13, then i is the index of the last element of A, which makes the claim true. Otherwise, i was updated at line 21. Then, A[i+1] $\geqslant$ cur->ts held at C_i. Between C_i and C, we argue that changes to cur can only cause cur->ts to decrease. If cur is updated at line 19 or 23 it is changed to the value read from the left pointer of the previous cur node at line 16. Similarly, line 31 updates cur to the value read from the left pointer of the previous cur node. By Lemma 9 and precondition 2, updating cur between C_i and C by traversing left pointers cannot cause cur->ts to increase. Thus, at C, we still have cur->ts $\leqslant$ A[i+1]. ◻

We say that (A, t) *is written before* $(\mathsf{A}', \mathsf{t}')$ if an AnnScan object (A, t) is stored in GlobalAnnScan before $(\mathsf{A}', \mathsf{t}')$. The following lemma is proved using the fact that creating the earlier copy completes before creating the later copy begins, because of the way we use CAS to update GlobalAnnScan. This, in turn, guarantees that once a node becomes unneeded, it is never needed by a later AnnScan pair.

Lemma 11. *Suppose* $(\mathsf{A}_1, \mathsf{t}_1)$ *is written before* $(\mathsf{A}_2, \mathsf{t}_2)$*. For any node* x*, if* x *is needed*$(\mathsf{A}_2, \mathsf{t}_2)$ *then* x *is needed*$(\mathsf{A}_1, \mathsf{t}_1)$*.*

Proof. Since every CAS on GlobalAnnScan tries to install a newly created object, there is no ABA problem on GlobalAnnScan. Thus, GlobalAnnScan is not updated between the time it is read on line 6 and the time a successful CAS is applied to it on line 9. Thus, $(\mathsf{A}_1, \mathsf{t}_1)$ is stored in GlobalAnnScan before the process that stores $(\mathsf{A}_2, \mathsf{t}_2)$ begins reading the values $(\mathsf{A}_2, \mathsf{t}_2)$ from the Announce array and the global timestamp. Thus, $t_1 \leqslant t_2$. Moreover, if some value $\mathsf{t} < \mathsf{t}_1$ appears in A_2, then it must also appear in A_1 since we assume values written into the Announce array are atomically copied from the global timestamp.

Assume x is *needed*$(\mathsf{A}_2, \mathsf{t}_2)$. We consider several cases.

If x.ts $> \mathsf{t}_1$, then x is also *needed*$(\mathsf{A}_1, \mathsf{t}_1)$, by definition.

If x.ts $\leqslant \mathsf{t}_1$ and x is the last appended node with timestamp at most t_2, then it is also the last appended node with timestamp at most t_1 since $\mathsf{t}_1 \leqslant \mathsf{t}_2$. Thus, x is *needed*$(\mathsf{A}_1, \mathsf{t}_1)$.

Otherwise, x.ts $\leqslant \mathsf{t}_1$ and x is the last appended node with timestamp at most $\mathsf{A}_2[\mathsf{i}]$ for some i. If $\mathsf{A}_2[\mathsf{i}] \geqslant \mathsf{t}_1$, then x is also the last appended node with timestamp at most t_1, and is therefore *needed*$(\mathsf{A}_1, \mathsf{t}_1)$. If $\mathsf{A}_2[\mathsf{i}] < \mathsf{t}_1$ then the value $\mathsf{A}_2[\mathsf{i}]$ must also appear in A_1. So, again x is *needed*$(\mathsf{A}_1, \mathsf{t}_1)$. ◻

The following key lemma describes how a compact(A,t,*) works: it traverses nodes that are *needed*(A, t) and splices out those nodes that are not *needed*(A, t). The proof is necessarily quite technical because, even though the set of needed nodes can only be reduced when GlobalAnnScan is updated, there many compact routines simultaneously traversing a version list, whose arguments are (A, t) pairs read from GlobalAnnScan at different times–some out of date and some more current.

Lemma 12. *If an invocation of* compact(A,t,h) *sets its local variable* cur *to a node* x $\neq$ h*, then* x *is needed*(A, t)*. If an invocation of* compact(A,t,h) *performs a successful CAS that stores* v *in* x->left*, then* x *is needed*(A, t) *(unless* x $=$ h*) and* v *is needed*(A, t) *and for all* w *satisfying* v $\prec$ w $\prec$ x*,* w *is not needed*(A, t)*.*

Proof. We prove the lemma by induction on steps: we assume the lemma is true in a prefix of an execution and show that it remains true when an additional step s is appended to the prefix. We first prove a couple of technical claims that will be used several times in the induction step.

Claim 12.1. *If* x $\leftarrow$ z *at some time before* s *and either* x.ts $\leqslant$ A[j] $<$ z.ts *for some* j *or* z.ts $>$ t*, then* x *is needed*(A, t)*.*

Proof of Claim. We consider several cases.

1. If x.ts $>$ t, then x is *needed*(A, t) by definition.
2. If x is the last appended node whose timestamp is at most t, then x is *needed*(A, t) by definition.
3. Suppose x.ts $\leqslant$ t $<$ z.ts but x is not the last node whose timestamp is at most t. Let y be the last appended node whose timestamp is at most t. By definition, y is *needed*(A, t). Since z.ts $>$ t, we have x $\prec$ y $\prec$ z by precondition 2. By the hypothesis of the claim, some invocation compact(A′,t′) set z->left to x before s. By the induction hypothesis, x is *needed*$(\mathsf{A}', \mathsf{t}')$ but y is not *needed*$(\mathsf{A}', \mathsf{t}')$. Since y is *needed*$(\mathsf{A}, \mathsf{t})$ but not *needed*$(\mathsf{A}', \mathsf{t}')$, (A, t) is written before $(\mathsf{A}', \mathsf{t}')$, by Lemma 11. Since x is *needed*$(\mathsf{A}', \mathsf{t}')$, x is also *needed*$(\mathsf{A}, \mathsf{t})$, again by Lemma 11.
4. If x is the last appended node whose timestamp is at most A[j] (for some j), then x is *needed*(A, t), by definition.
5. Otherwise, x.ts $\leqslant$ A[j] $<$ z.ts for some j, but x is not the last appended node with timestamp at most A[j]. This case is very similar to Case 3. Let y be the last appended node whose timestamp is at most A[j]. By definition, y is *needed*(A, t). Since z.ts $>$ A[j], we have x $\prec$ y $\prec$ z by precondition 2. By the hypothesis of the claim, some invocation compact(A′,t′) set z->left to x before s. By the induction hypothesis, x is *needed*$(\mathsf{A}', \mathsf{t}')$ but y is not *needed*$(\mathsf{A}', \mathsf{t}')$. Since y is *needed*$(\mathsf{A}, \mathsf{t})$ but not *needed*$(\mathsf{A}', \mathsf{t}')$, (A, t) is written before $(\mathsf{A}', \mathsf{t}')$, by Lemma 11. Since x is *needed*$(\mathsf{A}', \mathsf{t}')$, x is also *needed*$(\mathsf{A}, \mathsf{t})$, again by Lemma 11.

◻

Claim 12.2. *If* s *is an execution of line 28 or 31, let* v *and* x *be the nodes stored in* newNext *and* cur*, respectively, in the configuration before* s*. Then* v $\prec$ x *and* v *is needed*(A, t) *and for all* w *satisfying* v $\prec$ w $\prec$ x*,* w *is not needed*(A, t)*.*

Proof of Claim. The node v was reached from x by following left pointers, at line 16 and then at lines 25–27. By Invariant 9, v $\prec$ x. Let x $=$ $\mathsf{w}_1, \mathsf{w}_2, \ldots, \mathsf{w}_k$ $=$ v be the sequence of nodes traversed to get from x to v. By the tests on lines 22 and 26,

$\mathtt{w}_k$.ts $\leqslant$ A[i] $<$ $\mathtt{w}_{k-1}$.ts. Since $\mathtt{w}_k$ was read from $\mathtt{w}_{k-1}$ before *s*, Claim 12.1 implies that v $=$ $\mathtt{w}_k$ is *needed*(A, t).

It remains to show that all nodes w satisfying v $\prec$ w $\prec$ x are not *needed*(A, t). Since the test at line 18 failed, x.ts $\leqslant$ t. So a node w $\prec$ x cannot be the last appended node with timestamp at most t, nor can w have a timestamp greater than t. To complete the proof that w is not *needed*(A, t), we must show that it is not the last appended node with timestamp at most A[j] for any j. We do this in two parts, first considering j $>$ i and then j $\leqslant$ i. (Recall that A[0] $= -1$.)

Since x is the value of cur when the test at line 22 fails, it follows from Lemma 10 that either i is the index of the last element of A or x->ts $\leqslant$ A[i + 1]. Thus, any node w $\prec$ x cannot be the last appended node whose timestamp is at most A[j] for any j $>$ i, since then i would not be the last index and so x.ts $\leqslant$ A[i + 1] $\leqslant$ A[j].

To derive a contradiction, suppose that there is some w such that v $\prec$ w $\prec$ x and w is *needed*(A, t). Since we have already eliminated all other possibilities, this means that for some $j \leqslant i$, w is the last appended node whose timestamp is at most A[j]. Since w.ts $\leqslant$ A[j] $\leqslant$ A[i] $<$ $\mathtt{w}_{k-1}$.ts, we must have v $=$ $\mathtt{w}_k$ $\prec$ w $\prec$ $\mathtt{w}_{k-1}$. (In particular, this means that $\mathtt{w}_{k-1}$ cannot be *needed*(A, t).) Since $\mathtt{w}_k$ was read before *s* from $\mathtt{w}_{k-1}$->left, the induction hypothesis implies that some compact(A′,t′) did a CAS that set $\mathtt{w}_{k-1}$->left to $\mathtt{w}_k$ and $\mathtt{w}_{k-1}$ is *needed*(A′, t′) but w is not *needed*(A′, t′). Since $\mathtt{w}_{k-1}$ is *needed*(A′, t′) but not *needed*(A, t), Lemma 11 implies that the AnnScan object (A′, t′) is written before (A, t). Since w is not *needed*(A′, t′), it follows from Lemma 11 that w is not *needed*(A, t). This contradiction completes the proof of the claim. ◻

Now we are ready to prove that step *s* preserves the truth of the lemma. We need only consider steps *s* that update the cur variable of a compact routine or perform a successful CAS at line 28.

Now, suppose *s* executes line 19 to change the cur pointer from node z to x. Then x was read from x->left at line 16. By the test at line 18, z.ts $>$ t. So, by Claim 12.1, x is *needed*(A, t).

Now, suppose *s* executes line 23 to change the cur pointer from node z to x. Then, x was read from x->left at line 16. Since the while loop at line 21 terminated, we have z.ts $>$ A[i]. By the test at line 22, we have x $\leqslant$ A[i]. So, by Claim 12.1, x is *needed*(A, t).

Now, suppose *s* performs a successful CAS at line 28 that changes x->left to v. Then x and v are the values of the local variables cur and newNext, respectively, when the CAS occurs. Since x is stored in cur, x is *needed*(A, t), by the induction hypothesis. Claim 12.2 says that v is *needed*(A, t) and any node w satisfying v $\prec$ w $\prec$ x is not *needed*(A, t).

Finally, suppose *s* executes line 31 to change the cur pointer from node z to w. Let y be the node stored in newNext. By Claim 12.2, y $\prec$ z and y is *needed*(A, t). We argue that, prior to *s*, a node x $\preceq$ y was stored in z->left by considering two cases. If the loop at lines 28–30 terminated because the CAS at line 28 succeeded, then that CAS stored y in z->left. Otherwise, the test at line 30 was true, so the node x read from z->left on line 29 satisfied x.ts $\leqslant$ y.ts. Since the test at line 18 failed, z.ts $\leqslant$ t. Since y $\prec$ z, y.ts $\leqslant$ z.ts $\leqslant$ t by precondition 2. Since y is *needed*(A, t), it must be the last appended node with its timestamp. Therefore, x.ts $\leqslant$ y.ts implies that x $\preceq$ y. Thus, in either case, a node x $\preceq$ y was stored in z->left before *s*. By Lemma 9, the node w that *s* reads in z->left satisfies w $\preceq$ x. We show that w is *needed*(A, t) by considering two cases. If w $=$ y, we already know that y is *needed*(A, t). If w $\neq$ y, we have w $\preceq$ x $\preceq$ y $\prec$ z. Assume the CAS that stored w in x->left was performed by a compact(A′,t′). By the induction hypothesis, y is not *needed*(A′, t′) since w $\prec$ y $\prec$ z and a CAS in compact(A′,t′) stored w in z->left. Since y is *needed*(A, t), Lemma 11 implies that (A, t) was written before (A′, t′). Since w is *needed*(A′, t′), it follows from Lemma 11 that w is *needed*(A, t). ◻

We linearize successful tryAppend operations when they CAS the head pointer of the list. We linearize a search operation when it reads the head at line 36. The following theorem guarantees that every search operation returns the correct response.

Theorem 13. *If a* search(k) *operation returns the value from the last node appended to the list before the linearization point of the* search *whose timestamp is at most* k.

Proof. The search returns the value from the node stored in the local variable x.

If x is the value read from head at line 36, then x is the last node appended to the list before the linearization point of the search, and its timestamp is at most t since the test at line 37 failed.

Otherwise, there is a configuration C during the search just before x is read from the left pointer of some node z at line 38. By the test at line 37, we know that x->ts $\leqslant$ t $<$ z->ts. To derive a contradiction, suppose x is not the last appended node whose timestamp is at most k. Let y be the last appended node whose timestamp is at most k. Then, x $\prec$ y $\prec$ z.

By Lemma 12, there is an invocation of compact(A,t) before C such that y is not *needed*(A, t). We consider two cases.

If k $\in$ A, then y is not the last appended node whose timestamp is at most k, since y is not *needed*(A, t). This is the desired contradiction.

Otherwise, k $\notin$ A. At C, k is the in the Announce array, by precondition 3. By precondition 4, this means that k $\geqslant$ t. Since y is not *needed*(A, t), y->ts $\leqslant$ t, and there is a node appended after y whose timestamp is also at most t. Since k $\geqslant$ t, there is another node appended after y whose timestamp is at most k. Again, this is a contradiction. ◻

B.1 Analysis

We now consider progress properties. We use Invariant 9 to prove wait-freedom and that `search` does not fall off the end of the list.

Lemma 14. *The* `compact` *and* `search` *routines are wait-free. A* `search` *never sets its local variable* $\mathtt{x}$ *to null.*

Proof. We first show that `compact` is wait-free. Each iteration of the outer loop updates `cur` at line 19, 23 or 31 to a value read from `cur->left` at line 16 or 31. Consider the sequence of values $\mathtt{x}_1, \mathtt{x}_2, \ldots$ assigned to the `cur` variable of the `compact` routine. By Invariant 9, all the values are non-null and we have $\mathtt{x}_1 \succ \mathtt{x}_2 \succ \cdots$. Since there are finitely many nodes that precede $\mathtt{x}_1$ in the order $\prec$, the loop will terminate.

We must also check that the inner loops terminate. Due to the test at line 15, `cur` is not the sentinel node when the loop at line 21 is executed. Thus, $\mathtt{cur\text{->}ts} \geqslant 0$, so when the index $\mathtt{i}$ reaches 0, $\mathtt{A}[0] = -1 < \mathtt{cur\text{->}ts}$ and the loop terminates. (This argument also ensures that we never use an out-of-bounds index for $\mathtt{A}$.)

Each iteration of the loop at lines 26–27 sets `newNext` to an earlier node in the order $\prec$, by Invariant 9. So, it must terminate.

Each time the test at line 28 fails, some other operation has performed a successful CAS on `cur->left`. By Invariant 9, each such CAS changes `cur->left` to an earlier node in the order $\prec$, so it must eventually terminate.

We now show that `search` is wait-free and does not set $\mathtt{x}$ to null. Line 36 initializes $\mathtt{x}$ to the node read from `head`, which has been appended (by definition). If line 38 changes $\mathtt{x}$, then the node previously stored in $\mathtt{x}$ was not the sentinel, since its timestamp was greater than $\mathtt{k} \geqslant 0$. Thus, the value read from its left pointer is non-null, by Lemma 9.

If the sequences of nodes visited by the `search` is $\mathtt{x}_1, \mathtt{x}_2, \ldots$, then by Lemma 9, we have $\mathtt{x}_1 \succ \mathtt{x}_2 \succ \cdots$. Since there are only finitely many nodes that precede $\mathtt{x}_1$ in the order $\prec$, the loop must terminate. ◻

For PDL, we proved Lemmas 3 and 5 to show that CAS steps on `left` pointers can only remove nodes; they cannot cause a remove node to return to the list. We now prove analogous results for SSL.

Lemma 15. *Suppose* $\mathtt{w} \prec \mathtt{x} \prec \mathtt{y} \prec \mathtt{z}$ *and* $\mathtt{w} \leftarrow \mathtt{y}$ *at some time during the execution. Then there is never a time when* $\mathtt{x} \leftarrow \mathtt{z}$.

Proof. To derive a contradiction, suppose the claim is false.

Since there is a node $\mathtt{x}$ between $\mathtt{w}$ and $\mathtt{y}$ in the total order $\prec$, the `head` pointer takes the value $\mathtt{x}$ between the times it takes the values $\mathtt{w}$ and $\mathtt{y}$. Thus, $\mathtt{y}$ is not added to the list by a `tryAppend(w,y)` operation, so the initial value of `y->left` is not $\mathtt{w}$. So, there is a CAS on line 28 of a `compact(A,t,h)` operation that changes `y->left` to $\mathtt{w}$. Similarly, there is a CAS of a `compact(A,t,h)` operation that changes `z->left` to $\mathtt{x}$. By Lemma 12, $\mathtt{x}$ is *needed*$(\mathtt{A}', \mathtt{t}')$ but not *needed*$(\mathtt{A}, \mathtt{t})$. By Lemma 11, $(\mathtt{A}', \mathtt{t}')$ is written before $(\mathtt{A}, \mathtt{t})$. Since the arguments of `compact` are obtained by taking a snapshot of `GlobalAnnScan` and `head`, this means that $\mathtt{h}'$ was stored in `head` before $\mathtt{h}$, so $\mathtt{h}' \preceq \mathtt{h}$. Thus, $\mathtt{w} \prec \mathtt{x} \preceq \mathtt{h}' \preceq \mathtt{h}$, so $\mathtt{w} \neq \mathtt{h}$. By Lemma 12, $\mathtt{w}$ is *needed*$(\mathtt{A}, \mathtt{t})$ but not *needed*$(\mathtt{A}', \mathtt{t}')$. This contradicts Lemma 11. ◻

Lemma 16. *If a CAS at line 28 changes* `z->left` *from* $\mathtt{y}$ *to* $\mathtt{w}$*, we have* $\mathtt{w} \leftsquigarrow \mathtt{y}$ *in the preceding configuration* C.

Proof. Refer to Figure 9. Let $\mathtt{x}$ be the minimum node (with respect to $\prec$) such that $\mathtt{x} \succeq \mathtt{w}$ and $\mathtt{x} \leftsquigarrow \mathtt{y}$ in C. (This minimum is well-defined, since there exists a node $\mathtt{x}$ that satisfies both of these properties: $\mathtt{y} \succeq \mathtt{w}$ by Lemma 4 and $\mathtt{y} \leftsquigarrow \mathtt{y}$ holds trivially.) Since $\mathtt{x} \leftsquigarrow \mathtt{y} \leftarrow \mathtt{z}$, we have $\mathtt{x} \prec \mathtt{z}$ by Lemma 9.

Let $\mathtt{v}$ be the value of `x->left` in C. By Lemma 9, $\mathtt{v} \prec \mathtt{x}$. It must be the case that $\mathtt{v} \prec \mathtt{w}$; otherwise we would have $\mathtt{v} \succeq \mathtt{w}$ and $\mathtt{v} \leftsquigarrow \mathtt{y}$, which would contradict the fact that $\mathtt{x}$ is the *minimum* node that satisfies these two properties.

To derive a contradiction, suppose $\mathtt{w} \prec \mathtt{x}$. Then we have $\mathtt{v} \prec \mathtt{w} \prec \mathtt{x} \prec \mathtt{z}$. We also have $\mathtt{w} \leftarrow \mathtt{z}$ after the CAS and $\mathtt{v} \leftarrow \mathtt{x}$ in C, which contradicts Lemma 15. Thus, $\mathtt{w} \succeq \mathtt{x}$. By definition of $\mathtt{x}$, $\mathtt{x} \succeq \mathtt{w}$ and $\mathtt{x} \leftsquigarrow \mathtt{y}$ in C. Thus, $\mathtt{x} = \mathtt{w}$ and $\mathtt{w} \leftsquigarrow \mathtt{y}$. ◻

Lemma 16 allows us to prove that the `compact` routine permanently removes unneeded nodes.

Proposition 17. *Suppose a* `compact(A,t,h)` *routine on a version list has terminated prior to some configuration* C*. Then all nodes* $\mathtt{x}$ *satisfying sentinel* $\prec \mathtt{x} \prec \mathtt{h}$ *that are reachable from the* `head` *of the version list in* C *are needed*$(\mathtt{A}, \mathtt{t})$.

Proof. To derive a contradiction, assume that some node $\mathtt{x}$ is reachable from `head` in C and $\mathtt{x}$ is not *needed*$(\mathtt{A}, \mathtt{t})$. Consider the sequence of values $\mathtt{h} = \mathtt{y}_1, \mathtt{y}_2, \ldots, \mathtt{y}_\ell =$ sentinel that are assigned to the local variable `cur` during the `compact(A,t,h)` routine. Since each is read from the `left` pointer of the previous one, $\mathtt{y}_\ell \prec \mathtt{y}_{\ell-1} \prec \cdots \prec \mathtt{y}_1$. Since $\mathtt{x}$ is not *needed*$(\mathtt{A}, \mathtt{t})$, Lemma 12 implies that $\mathtt{x}$ does not appear in this sequence. So, for some j, $\mathtt{y}_{j+1} \prec \mathtt{x} \prec \mathtt{y}_j$.

Now, let $\mathtt{head} = \mathtt{z}_1, \mathtt{z}_2, \ldots, \mathtt{z}_k = \mathtt{x}$ be the path from `head` to $\mathtt{x}$ by following `left` pointers. Consider the maximum i such that $\mathtt{z}_i \succeq \mathtt{y}_j$. (Such an i exists, since $\mathtt{z}_1 = \mathtt{head} \succeq \mathtt{h} = \mathtt{y}_1 \succeq \mathtt{y}_j$.) So, we have $\mathtt{y}_{j+1} \prec \mathtt{x} = \mathtt{z}_k \prec \mathtt{z}_{k-1} \prec \cdots \mathtt{z}_{i+1} \prec \mathtt{y}_j \preccurlyeq \mathtt{z}_i$. Now, $\mathtt{y}_{j+1} \leftarrow \mathtt{y}_j$ at some time during the `compact` (before C). If $\mathtt{y}_j$ were equal to $\mathtt{z}_i$, then in C, $\mathtt{z}_i\text{->}\mathtt{left} = \mathtt{y}_j\text{->}\mathtt{left} \preceq \mathtt{y}_{j+1}$ by Invariant 9, and this would contradict the fact that $\mathtt{z}_i\text{->}\mathtt{left} = \mathtt{z}_{i+1} \succ \mathtt{y}_{j+1}$. So, we have $\mathtt{y}_{j+1} \prec \mathtt{z}_{i+1} \prec \mathtt{y}_j \prec \mathtt{z}_i$. However, $\mathtt{y}_{j+1} \leftarrow \mathtt{y}_j$ at some time before C and $\mathtt{z}_{i+1} \leftarrow \mathtt{z}_i$ in C. This contradicts Lemma 15. ◻

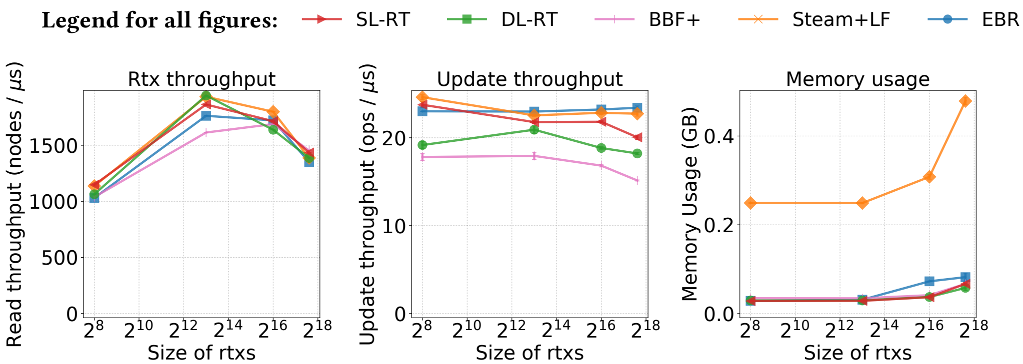
The head of a version list represents the current version, so it is not obsolete. Thus, when a node is added to the range tracker data structure, it is no longer the head of its list. Once a process receives a node x returned by the range tracker, it gets up-to-date copies A, t and h of the announcements, global timestamp and the list's head and invoke compact(A,t,h). Since x was already returned by the range tracker, x will not be *needed*(A, t) and $x \prec h$. Proposition 17 ensures that x is permanently removed from its version list before the compact terminates. Thus, aside from the lists where compact routines are still pending, all nodes that have been returned by the range tracker are no longer reachable from the head of the list.

B.2 Reserving Timestamps

Both BBF+ and our singly-linked list compaction algorithm from Section 5 use atomic copy (a primitive which allows us to atomically read from one memory location and write to another) to announce timestamps in a wait-free manner. Instead, in our experiments, we use a lighter-weight lock-free scheme for announcing timestamps which consist of (A1) reading the current timestamp, (A2) announcing it, and (A3) checking if the current timestamp is still equal to the announced value and, if not, going back to step A1. This means that a process scanning the announcement array might see a very old timestamp get announced, which breaks the monotonicity property required for the correctness of our singly-linked list. We fix this by having all processes work on updating a global scan of the announcement array as described in Section 5. When reading the announcement array, we ignore any timestamp that (1) is smaller than the timestamp at which the current global scan was taken and (2) does not appear in the current global scan. This is safe because any announced timestamp satisfying these two conditions will fail the check in step A3, and not be used. This change to how we scan announcement arrays ensures that the global scan satisfies the monotonicity property required by our singly-linked list. This announcement scanning technique is also used in our implementation of STEAM+LF, BBF+, and SL-RT.

C Additional Experiments

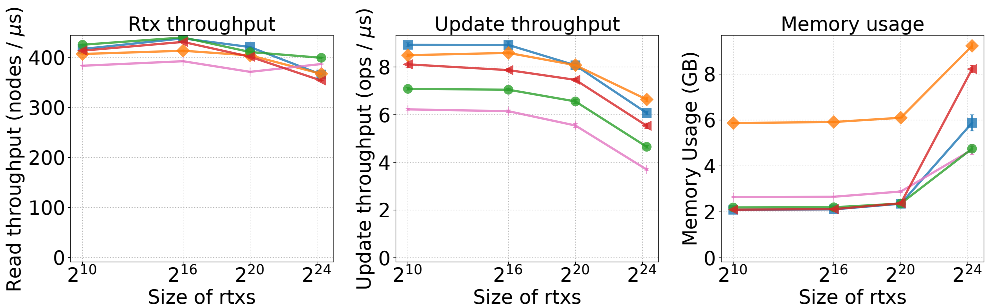
Figures 12–16 showcase the same workloads as Figures 4–8, but with keys drawn from uniform rather than zipfian distribution.



Average Version List Lengths

Size of rtxs	2^8	2^{13}	2^{16}	2^{18}
SL-RT	1.06	1.07	1.24	1.66
DL-RT	1.07	1.08	1.23	1.63
BBF+	1.07	1.08	1.23	1.64
Steam+LF	1.51	1.53	1.68	2.06
EBR	1.02	1.04	1.33	2.0

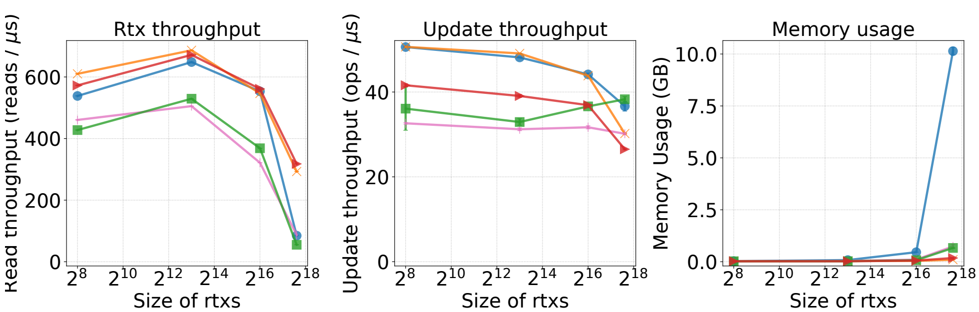
Figure 12. Tree with 100K keys, 40 update threads, 40 fixed-size rtx threads, 40 variable-size rtx threads.



Average Version List Lengths

Size of rtxs	2^{10}	2^{16}	2^{20}	20M
SL-RT	1.0	1.01	1.09	1.63
DL-RT	1.01	1.01	1.07	1.57
BBF+	1.01	1.01	1.06	1.55
Steam+LF	1.43	1.43	1.46	1.68
EBR	1.01	1.01	1.07	1.71

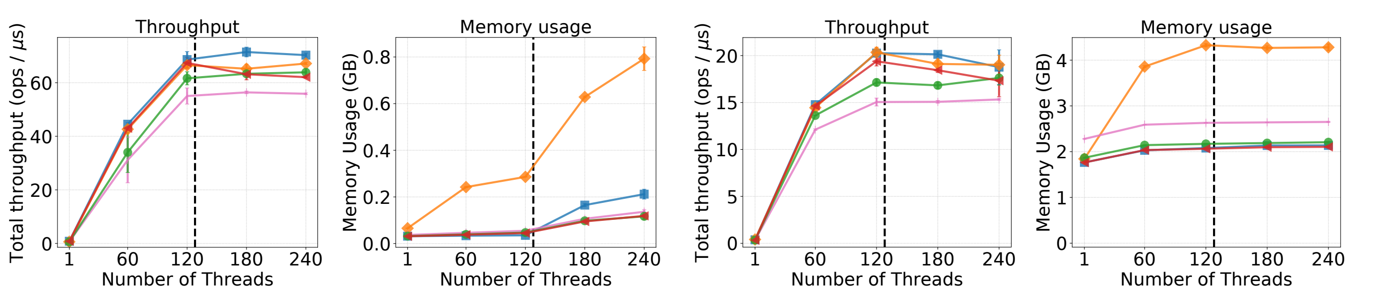
Figure 13. Tree with 10M keys, 40 update threads, 40 fixed-size rtx threads, 40 variable-size rtx threads.



Average Version List Lengths

Size of rtxs	2^8	2^{13}	2^{16}	2^{18}
SL-RT	1.04	1.15	2.35	5.46
DL-RT	1.05	1.14	2.42	25.27
BBF+	1.05	1.12	2.5	19.39
Steam+LF	1.64	1.76	2.74	5.57
EBR	1.01	1.19	3.34	451.99

Figure 14. Hash table with 100K keys, 40 update threads, 40 fixed-size rtx threads, 40 variable-size rtx threads.



(a) Tree with 100K keys **(b)** Tree with 10M keys

Figure 15. Workload with each thread performing 50% updates, 49% lookups, and 1% read transactions of size 1024.

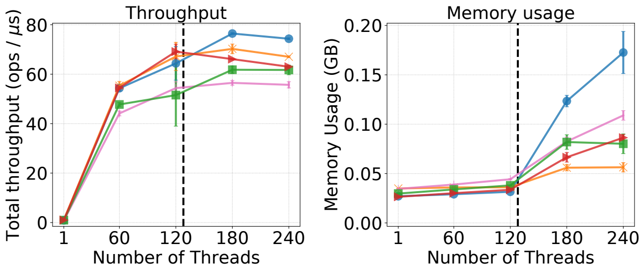


Figure 16. Hash table with 100K keys, each thread performs 50% updates, 49% lookups, and 1% rtxs of size 1024.