Space and Time Bounded Multiversion Garbage Collection

Naama Ben-David
VMware Research, USA

Guy E. Blelloch
Carnegie Mellon University, USA

Panagiota Fatourou
FORTH ICS and University of Crete, Greece

Eric Ruppert
York University, Canada

Yihan Sun
University of California, Riverside, USA

Yuanhao Wei
Carnegie Mellon University, USA

Abstract

We present a general technique for garbage collecting old versions for multiversion concurrency control that simultaneously achieves good time and space complexity. Our technique takes only $O(1)$ time on average to reclaim each version and maintains only a constant factor more versions than needed (plus an additive term). It is designed for multiversion schemes using version lists, which are the most common.

Our approach uses two components that are of independent interest. First, we define a novel *range-tracking data structure* which stores a set of old versions and efficiently finds those that are no longer needed. We provide a wait-free implementation in which all operations take amortized constant time. Second, we represent version lists using a new lock-free doubly-linked list algorithm that supports efficient (amortized constant time) removals given a pointer to any node in the list. These two components naturally fit together to solve the multiversion garbage collection problem–the range-tracker identifies which versions to remove and our list algorithm can then be used to remove them from their version lists. We apply our garbage collection technique to generate end-to-end time and space bounds for the multiversioning system of Wei et al. (PPoPP 2021).

2012 ACM Subject Classification Theory of computation → Concurrent algorithms; Theory of computation → Data structures design and analysis

Keywords and phrases Lock-free, data structures, memory management, snapshot, version lists

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.39

Related Version *Full Version*: `https://arxiv.org/abs/2108.02775`

Funding *Guy E. Blelloch and Yuanhao Wei: NSF CCF-1901381, CCF-1910030, and CCF-1919223. Eric Ruppert: NSERC Discovery Grant. Yihan Sun: NSF grant CCF-2103483. Panagiota Fatourou: EU Horizon 2020, Marie Sklodowska-Curie GA No 101031688.*

Acknowledgements *We thank the anonymous referees for their helpful comments and suggestions.*

1 Introduction

Supporting multiple "historical" versions of data, often called multiversioning or multiversion concurrency control, is a powerful technique widely used in database systems [42, 10, 38, 32, 36, 51], transactional memory [40, 22, 39, 31, 29], and shared data structures [7, 21, 35, 49].

This approach allows complex queries (read-only transactions) to proceed concurrently with updates while still appearing atomic because they get data views that are consistent with a single point in time. If implemented carefully, queries do not interfere with one another or with updates. The most common approach for multiversioning uses version lists [42] (also called version chains): the system maintains a global timestamp that increases over time, and each object maintains a history of its updates as a list of value-timestamp pairs, each corresponding to a value written and an update time. Each node in the list has an associated interval of time from that node's timestamp until the next (later) node's timestamp. A query can first read a timestamp value t and then, for each object it wishes to read, traverse the object's version list to find the version whose interval contains t.

Memory usage is a key concern for multiversioning, since multiple versions can consume huge amounts of memory. Thus, most previous work on multiversioning discusses how to reclaim the memory of old versions. We refer to this as the multiversion garbage collection (MVGC) problem. A widely-used approach is to keep track of the earliest active query and reclaim the memory of any versions overwritten before the start of this query [22, 36, 30, 35, 49]. However, a query that runs for a long time, either because it is complicated or because it has been delayed, will force the system to retain many unneeded intermediate versions between the oldest required version and the current one. This has been observed to be a major bottleneck for database systems with Hybrid Transaction and Analytical Processing (HTAP) workloads [14] (i.e., many small updates concurrent with some large analytical queries). To address this problem in the context of software transactional memory, Lu and Scott [33] proposed a non-blocking algorithm that can reclaim intermediate versions. Blocking techniques were later proposed by the database community [14, 32]. However, these techniques add significant time overhead in worst-case executions.

We present a wait-free MVGC scheme that achieves good time and space bounds, using $O(1)$ time[1] on average per allocated version and maintaining only a constant factor more versions than needed (plus an additive term). The scheme is very flexible and it can be used in a variety of multiversioning implementations. It uses a three-step approach that involves 1) identifying versions that can be reclaimed, including intermediate versions, 2) unlinking them from the version lists, and 3) reclaiming their memory. To implement these three steps efficiently, we develop two general components—a *range-tracking* data structure and a *version-list* data structure—that could be of independent interest beyond MVGC.

The range-tracking data structure is used to identify version list nodes that are no longer needed. It supports an `announce` operation that is used by a query to acquire the current timestamp t as well as protect any versions that were current at t from being reclaimed. A corresponding `unannounce` is used to indicate when the query is finished. The data structure also supports a `deprecate` operation that is given a version and its time interval, and indicates that the version is no longer the most recent—i.e., is safe to reclaim once its interval no longer includes any announced timestamp. When a value is updated with a new version, the previous version is deprecated. A call to `deprecate` also returns a list of versions that had previously been `deprecated` and are no longer cover any announced timestamp—i.e., are now safe to reclaim. We provide a novel implementation of the range-tracking data structure for which the amortized number of steps per operation is $O(1)$. We also bound the number of versions on which `deprecate` has been called, but have not yet been returned. If H is the maximum, over all configurations, of the number of needed deprecated versions, then the number of deprecated versions that have not yet been returned is at most $2H + O(P^2 \log P)$,

[1] For time/space complexity, we count both local and shared memory operations/objects.

where P is the number of processes. To achieve these time and space bounds, we borrow some ideas from real-time garbage collection [6, 11], and add several new ideas such as batching and using a shared queue.

The second main component of our scheme is a wait-free version-list data structure that supports efficient (amortized constant time) removals of nodes from anywhere in the list. When the `deprecate` operation identifies an unneeded version, we must splice it out of its version list, without knowing its current predecessor in the list, so we need a doubly-linked version list. Our doubly-linked list implementation has certain restrictions that are naturally satisfied when maintaining version lists, for example nodes may be appended only at one end. The challenge is in achieving constant amortized time per remove, and bounded space. Previously known concurrent doubly-linked lists [47, 43] do not meet these requirements, requiring at least $\Omega(P)$ amortized time per remove. We first describe the implementation of our version list assuming a garbage collector and then we show how to manually reclaim removed nodes while maintaining our desired overall time and space bounds.

To delete elements from the list efficiently, we leverage some recent ideas from randomized parallel list contraction [12], which asynchronously removes elements from a list. To avoid concurrently splicing out adjacent elements in the list, which can cause problems, the approach defines an implicit binary tree so that the list is an in-order traversal of the tree. Only nodes corresponding to leaves of the tree, which cannot be adjacent in the list, may be spliced out. Directly applying this technique, however, is not efficient in our setting. To reduce space overhead, we had to develop intricate helping mechanisms for splicing out internal nodes rather than just leaves. To achieve wait-freedom, we had to skew the implicit tree so that it is right-heavy. The final algorithm ensures that at most $2(L - R) + O(P \log L_{max})$ nodes remain reachable in an execution with L appends and R removes across an arbitrary number of version lists, and at most L_{max} appends on a single version list. This means the version lists store at most a constant factor more than the $L - R$ required nodes plus an additive term shared across all the version lists. Combining this with the bounds from the range tracker, our MVGC scheme ensures that at most $O(V + H + P^2 \log P + P \log L_{max})$ versions are reachable from the V version lists. This includes the current version for each list, H needed versions, plus additive terms from the range tracking and list building blocks.

After a node has been spliced out of the doubly-linked list, its memory must be reclaimed. This step may be handled automatically by the garbage collector in languages such as Java, but in non-garbage-collected languages, additional mechanisms are needed to safely reclaim memory. The difficulty in this step is that while a node is being spliced out, other processes traversing the list might be visiting that node. We use a reference counting reclamation scheme and this requires modifying our doubly-linked list algorithm slightly to maintain the desired space bounds. We apply an existing concurrent reference counting implementation [2] that employs a local hash table per process which causes the time bounds of our reclamation to become amortized $O(1)$ *in expectation*. It also requires an additional fetch-and-add instruction, whereas the rest of our algorithms require only read and CAS.

We apply our MVGC scheme to a specific multiversioning scheme [49] to generate end-to-end bounds for a full multiversioning system. This multiversioning scheme takes a given CAS-based concurrent data structure and transforms it to support complex queries (e.g., range queries) by replacing each CAS object with one that maintains a version list. Overall, we ensure that the memory usage of the multiversion data structure is within a constant factor of the needed space, plus $O(P^2 \log P + P^2 \log L_{max})$. In terms of time complexity, our garbage collection scheme takes only $O(1)$ time on average for each allocated version.

Detailed proofs of correctness and of our complexity bounds appear in the full version [8].

2 Related Work

Garbage Collection. One of the simplest, oldest techniques for garbage collection is reference counting (RC) [16, 17, 28]. In its basic form, RC attaches to each object a counter of the number of references to it. An object is reclaimed when its counter reaches zero. Some variants of RC are wait-free [2, 46]. In Section 6, we apply the RC scheme of [2] to manage version list nodes as it adds only constant time overhead (in expectation) and it is the only concurrent RC scheme that maintains our desired time bounds.

Epoch-based reclamation (EBR) [23, 15] employs a counter that is incremented periodically and is used to divide the execution into epochs. Processes read and announce the counter value at the beginning of an operation. An object can be reclaimed only if it was retired in an epoch preceding the oldest announced. EBR is often the preferred choice in practice, as it is simple and exhibits good performance. However, a slow or crashed process with timestamp t can prevent the reclamation of *all* retired objects with timestamps larger than t. EBR, or variants, are used in a variety of MVGC schemes [22, 36, 49] to identify versions that are older than any query. An advantage of these schemes is that identified versions can be immediately reclaimed without first being unlinked from the version lists because the section of the version list they belong to is old enough to never be traversed. However, they inherit the same problem as EBR and are not able to reclaim intermediate versions between the oldest needed version and the current version when a long-running query holds on to an old epoch. This can be serious for multiversioned systems since EBR works best when operations are short, but a key motivation for multiversioning is to support lengthy queries.

Hazard pointers (HP) [28, 34] can be used to track which objects are currently being accessed by each process and are therefore more precise. Combinations of HP and EBR have been proposed (e.g. [41, 50]) with the goal of preserving the practical efficiency of EBR while lowering its memory usage. However, unlike EBR, none of these techniques directly solve the MVGC problem. Other memory reclamation schemes have been studied that require hardware support [1, 18] or rely on the signaling mechanism of the operating system [15, 45]. Hyaline [37] implements a similar interface to EBR and can be used for MVGC, but like EBR, it cannot reclaim intermediate versions.

We are aware of three multiversioning systems based on version lists that reclaim intermediate versions: GMV [33], HANA [32] and Steam [14]. To determine which versions are safe to reclaim, all three systems merge the current version list for an object with the list of active timestamps to check for overlap. The three schemes differ based on when they decide to perform this merging step and how they remove and reclaim version list nodes. In GMV, when an update operation sees that memory usage has passed a certain threshold, it iterates through all the version lists to reclaim versions. Before reclaiming a version, it has to help other processes traverse the version list to ensure traversals remain wait-free. HANA uses a background thread to identify and reclaim obsolete versions while Steam scans the entire version list whenever a new version is added to it. In HANA and Steam, nodes are removed by locking the entire version list, whereas in GMV, nodes are removed in a lock-free manner by first logically marking a node for deletion, as in Harris's linked list [26]. If a `remove` operation in GMV experiences contention (i.e., fails a CAS), it restarts from the head of the version list. None of these three techniques ensure constant-time removal from a version list. Both Steam and GMV ensure $O(PM)$ space where M is the amount of space required in an equivalent sequential execution. In comparison, we use a constant factor more than the required space plus an additive term of $O(P^2 \log P + P^2 \log L_{max})$, where L_{max} is the maximum number of versions added to a single version list. This can be significantly less than $O(PM)$ in many workloads.

Lock-Free Data Structures and Query Support. We use doubly-linked lists to store old versions. Singly-linked lists had lock-free implementations as early as 1995 [48]. Several implementations of doubly-linked lists were developed later from multi-word CAS instructions [5, 24], which are not widely available in hardware but can be simulated in software [27, 25]. Sundell and Tsigas [47] gave the first implementation from single-word CAS, although it lacks a full proof of correctness. Shafiei [43] gave an implementation with a proof of correctness and amortized analysis. Existing doubly-linked lists are not efficient enough for our application, so we give a new implementation with better time bounds.

Fatourou, Papavasileiou and Ruppert [21] used multiversioning to add range queries to a search tree [19]. Wei et al. [49] generalized this approach (and made it more efficient) to support wait-free queries on a large class of lock-free data structures. Nelson, Hassan and Palmieri [35] sketched a similar scheme, but it is not non-blocking. In Appendix A, we apply our garbage collection scheme to the multiversion system of [49].

3 Preliminaries

We use a standard model with asynchronous, crash-prone processes that access shared memory using CAS, read and write instructions. For our implementations of data structures, we bound the number of steps needed to perform operations, and the number of shared objects that are allocated but not yet reclaimed.

We also use *destination objects* [13], which are single-writer objects that store a value and support `swcopy` operations in addition to standard reads and writes. A `swcopy(ptr)` atomically reads the value pointed to by `ptr`, and copies the value into the destination object. Only the owner of a destination object can perform `swcopy` and `write`; any process may `read` it. Destination objects can be implemented from CAS so that all three operations take $O(1)$ steps [13]. They are used to implement our range-tracking objects in Section 4.

Pseudocode Conventions. We use syntax similar to C++. The type `T*` is a pointer to an object of type `T`. `List<T>` is a `List` of objects of type `T`. If `x` stores a pointer to an object, then `x->f` is that object's member `f`. If `y` stores an object, `y.f` is that object's member `f`.

4 Identifying Which Nodes to Disconnect from the Version List

We present the *range-tracking* object, which we use to identify version nodes that are safe to disconnect from version lists because they are no longer needed. To answer a query, a slow process may have to traverse an entire version list when searching for a very old version. However, we need only maintain list nodes that are the potential target nodes of such queries. The rest may be spliced out of the list to improve space usage and traversal times.

We assign to each version node `X` an interval that represents the period of time when `X` was the current version. When the next version `Y` is appended to the version list, `X` ceases to be the current version and becomes a potential candidate for removal from the version list (if no query needs it). Thus, the left endpoint of `X`'s interval is the timestamp assigned to `X` by the multiversioning system, and the right endpoint is the timestamp assigned to `Y`.

We assume that a query starts by announcing a timestamp t, and then proceeds to access, for each relevant object o, its corresponding version at time t, by finding the first node in the version list with timestamp at most t (starting from the most recent version). Therefore, an announcement of t means it is unsafe to disconnect any nodes whose intervals contain t.

As many previous multiversioning systems [22, 32, 35, 36, 49] align with the general scheme discussed above, we define the range-tracking object to abstract the problem of identifying versions that are not needed. We believe this abstraction is of general interest.

▶ **Definition 1** (Range-Tracking Object). *A range-tracking object maintains a multiset A of integers, and a set O of triples of the form* `(o,low,high)` *where* `o` *is an object of some type T and* `low` $\leq$ `high` *are integers. Elements of A are called* active announcements*. If* `(o,low,high)` $\in O$ *then* `o` *is a* deprecated object *with associated half-open interval* [`low`, `high`)*. The range-tracking object supports the following operations.*

- `announce(int* ptr)` *atomically reads the integer pointed to by* `ptr`*, adds the value read to A, and returns the value read.*
- `unannounce(int i)` *removes one copy of* `i` *from A, rendering the announcement inactive.*
- `deprecate(T* o, int low, int high)`*, where* `low` $\leq$ `high`*, adds the triple* `(o,low,high)` *to O and returns a set S, which contains the deprecated objects of a set $O' \subseteq O$ such that for any $o \in O'$, the interval of o does not intersect A, and removes O' from O.*

The specification of Definition 1 should be paired with a progress property that rules out the trivial implementation in which `deprecate` always returns an empty set. We do this by bounding the number of deprecated objects that have not been returned by `deprecate`.

▶ **Assumption 2.** To implement the range-tracking object, we assume the following.

1. A process's calls to `deprecate` have non-decreasing values of parameter `high`.
2. If, in some configuration G, there is a pending `announce` whose argument is a pointer to an integer variable `x`, then the value of `x` at G is greater than or equal to the `high` argument of every `deprecate` that has been invoked before G.
3. For every process p, the sequence of invocations to `announce` and `unannounce` performed by p should have the following properties: a) it should start with `announce`; b) it should alternate between invocations of `announce` and invocations of `unannounce`; c) each `unannounce` should have as its argument the integer returned by the preceding `announce`.
4. Objects passed as the first parameter to `deprecate` operations are distinct.

In the context we are working on, we have a non-decreasing integer variable that works as a global timestamp, and is passed as the argument to every `announce` operation. Moreover, the `high` value passed to each `deprecate` operation is a value that has been read from this variable. This ensures that parts 1 and 2 of Assumption 2 are satisfied. The other parts of the assumption are also satisfied quite naturally for our use of the range-tracking object, and we believe that the assumption is reasonably general. Under this assumption, we present and analyze a linearizable implementation of the range-tracking object in Section 4.1.

4.1 A Linearizable Implementation of the Range-Tracking Object

Our implementation, RangeTracker, is shown in Figure 1. Assumption 2.3 means that each process can have at most one active announcement at a time. So, RangeTracker maintains a shared array `Ann` of length P to store active announcements. `Ann[p]` is a destination object (defined in Section 3) that is owned by process `p`. Initially, `Ann[p]` stores a special value $\bot$. To announce a value, a process `p` calls `swcopy` (line 28) to copy the current timestamp into `Ann[p]` and returns the announced value (line 29). To deactivate an active announcement, p writes $\bot$ into `Ann[p]` (line 31). Under Assumption 2.3, the argument to

```
1 class Range { T* t, int low, int high; };
2 class RangeTracker {
3   // global variables
4   Destination Ann[P];
5   Queue<List<Range>> Q; //initially empty
6   // thread local variables
7   List<Range> LDPool; // initially empty
8   Array<int> sortAnnouncements() {
9     List<int> result;
10    for(int i = 0; i < P; i++) {
11      int num = Ann[i].read();
12      if(num != ⊥) result.append(num); }
13    return sort(toArray(result)); }

15  List<T*>, List<Range> intersect(
16      List<Range> MQ, Array<int> ar) {
17    Range r; int i = 0;
18    List<T*> Redundant;
19    List<Range> Needed;
20    for(r in MQ) {
21      while(i < ar.size() &&
22            ar[i] < r.high) i++;
23      if(i == 0 || ar[i-1] < r.low)
24        Redundant.append(r.t);
25      else Needed.append(r); }
26    return <Redundant, Needed>; }
27 int Announce(int* ptr) {
28   Ann[p].swcopy(ptr);
29   return Ann[p].read(); }

31 void unannounce() { Ann[p].write(⊥); }

33 List<T*> deprecate(T* o, int low, int high) {
34   List<T*> Redundant;
35   List<Range> Needed, Needed1, Needed2;
36   // local lists are initially empty
37   LDPool.append(Range(o, low, high));
38   if(LDPool.size() == B) {
39     List<Range> MQ = merge(Q.deq(),Q.deq());
40     Array<int> ar = sortAnnouncements();
41     Redundant, Needed = intersect(MQ, ar);
42     if(Needed.size() > 2*B) {
43       Needed1, Needed2 = split(Needed);
44       Q.enq(Needed1);
45       Q.enq(Needed2); }
46     else if(Needed.size() > B) {
47       Q.enq(Needed); }
48     else {
49       LDPool = merge(LDPool,Needed); }
50     Q.enq(LDPool);
51     LDPool = empty list; }
52   return Redundant; } };
```

Figure 1 Code for process p for our linearizable implementation of a range-tracking object.

unannounce must match the argument of the process's previous announce, so we suppress unannounce's argument in our code. An announce or unannounce performs O(1) steps.

A Range object (line 1) stores the triple (o,low,high) for a deprecated object o. It is created (at line 37) during a deprecate of o. RangeTracker maintains the deprecated objects as *pools* of Range objects. Each pool is sorted by its elements' high values. Each process maintains a local pool of deprecated objects, called LDPool. To deprecate an object, a process simply appends its Range to the process's local LDPool (line 37). Assumption 2.1 implies that objects are appended to LDPool in non-decreasing order of their high values.

We wish to ensure that most deprecated objects are eventually returned by a deprecate operation so that they can be freed. If a process p with a large LDPool ceases to take steps, it can cause all of those objects to remain unreturned. Thus, when the size of p's LDPool hits a threshold B, they are flushed to a shared queue, Q, so that other processes can also return them. The elements of Q are pools that each contain B to $2B$ deprecated objects. For the sake of our analysis, we choose $B = P \log P$. When a flush is triggered, p dequeues two pools from Q and processes them as a batch to identify the deprecated objects whose intervals do not intersect with the values in Ann, and return them. The rest of the dequeued objects, together with those in LDPool, are stored back into Q. We call these actions (lines 38–51), the *flush phase* of deprecate. A deprecate without a flush phase returns an empty set.

During a flush phase, a process p dequeues two pools from Q and merges them (line 39) into a new pool, MQ. Next, p makes a local copy of Ann and sorts it (line 40). It then uses the intersect function (line 41) to partition MQ into two sorted lists: Redundant contains objects whose intervals do not intersect the local copy of Ann, and Needed contains the rest. Intuitively, a deprecated object in MQ is put in Redundant if the low value of its interval is larger than the announcement value immediately before its high value. Finally, p enqueues the Needed pool with its LDPool into Q (lines 44–47 and line 50). To ensure that the size of each pool in Q is between B and $2B$, the Needed pool is split into two halves if it is too large (line 43), or is merged with `LDPool` if it is too small (line 49). A flush phase is performed once every $P \log P$ calls to `deprecate`, and the phase executes $O(P \log P)$ steps. Therefore, the amortized number of steps for `deprecate` is $O(1)$.

The implementation of the concurrent queue `Q` should ensure that an element can be enqueued or dequeued in $O(P \log P)$ steps. The concurrent queue presented in [20] has step complexity $O(P)$ and thus ensures these bounds. To maintain our space bounds, the queue nodes must be reclaimed. This can be achieved if we apply hazard-pointers on top of the implementation in [20]. If `Q` is empty, then `Q.deq()` returns an empty list.

We sketch the proofs of the following three theorems. For detailed proofs, see [8].

▶ **Theorem 3.** *If Assumption 2 holds, then* RangeTracker *is a linearizable implementation of a range-tracking object.*

The linearization points used in the proof of Theorem 3 are defined as follows. An `announce` is linearized at its `swcopy` on line 28. An `unannounce` is linearized at its `write` on line 31. A `deprecate` is linearized at line 50 if it executes that line, or at line 37 otherwise.

The most interesting part of the proof concerns a `deprecate` operation I with a flush phase. I dequeues two pools from `Q` as `MQ` and decides which objects in `MQ` to return based on the local copy of `Ann` array. To show linearizability, we must also show that intervals of the objects returned by I do not intersect the `Ann` array at the linearization point of I. Because of Assumption 2.2, values written into `Ann` after the pools are dequeued cannot be contained in the intervals in `MQ`. Thus, if an object's interval does not contain the value I read from `Ann[i]`, it will not contain the value in `Ann[i]` at I's linearization point.

▶ **Theorem 4.** *In the worst case,* `announce` *and* `unannounce` *take* $O(1)$ *steps, while* `deprecate` *takes* $O(P \log P)$ *steps. The amortized number of steps performed by each operation is* $O(1)$*.*

Let H be the maximum, over all configurations in the execution, of the number of *needed* deprecated objects, i.e., those whose intervals contain an active announcement.

▶ **Theorem 5.** *At any configuration, the number of deprecated objects that have not yet been returned by any instance of* `deprecate` *is at most* $2H + 25P^2 \log P$*.*

At any time, each process holds at most $P \log P$ deprecated objects in `LDPool` and at most $4P \log P$ that have been dequeued from `Q` as part of a flush phase. We prove by induction that the number of deprecated objects in `Q` at a configuration G is at most $2H + O(P^2 \log P)$. Let G' be the latest configuration before G such that all pools in `Q` at G' are dequeued between G' and G. Among the dequeued pools, only the objects that were needed at G' are re-enqueued into `Q`, and there are at most H such objects. Since we dequeue two pools (containing at least B elements each) each time we enqueue B *new* objects between G' and G, this implies that the number of such new objects is at most half the number of objects in `Q` at G' (plus $O(P^2 \log P)$ objects from flushes already in progress at G'). Assuming the bound on the size of `Q` holds at G', this allows us to prove the bound at G.

The constant multiplier of H in Theorem 5 can be made arbitrarily close to 1 by dequeuing and processing k pools of `Q` in each flush phase instead of two. The resulting space bound would be $\frac{k}{k-1} \cdot H + \frac{(2k+1)(3k-1)}{k-1} \cdot P^2 \log P$. This would, of course, increase the constant factor in the amortized number of steps performed by `deprecate` (Theorem 4).

5 Maintaining Version Lists

We use a restricted version of a doubly-linked list to maintain each version list so that we can more easily remove nodes from the list when they are no longer needed. We assume each node has a timestamp field. The list is initially empty and provides the following operations.

Figure 2 An example of incorrect removals.

- `tryAppend(Node* old, Node* new)`: Adds `new` to the head of the list and returns true if the current head is `old`. Otherwise returns false. Assumes `new` is not null.
- `getHead()`: Returns a pointer to the `Node` at the head of the list (or `null` if list is empty).
- `find(Node* start, int ts)`: Returns a pointer to the first `Node`, starting from `start` and moving away from the head of the list, whose timestamp is at most `ts` (or `null` if no such node exists).
- `remove(Node* n)`: Given a previously appended `Node`, removes it from the list.

To obtain an efficient implementation, we assume several preconditions, summarized in Assumption 6 (and stated more formally in the full version [8]). A version should be removed from the object's version list only if it is not current: either it has been superseded by another version (6.1) or, if it is the very last version, the entire list is no longer needed (6.2). Likewise, a version should not be removed if a `find` is looking for it (6.3), which can be guaranteed using our range-tracking object. We allow flexibility in the way timestamps are assigned to versions. For example, a timestamp can be assigned to a version after appending it to the list. However, some assumptions on the behaviour of timestamps are needed to ensure that responses to `find` operations are properly defined (6.4, 6.5).

▶ **Assumption 6.**

1. Each node (except the very last node) is removed only after the next node is appended.
2. No `tryAppend`, `getHead` or `find` is called after a `remove` on the very last node.
3. After `remove(X)` is invoked, no pending or future `find` operation should be seeking a timestamp in the interval between `X`'s timestamp and its successor's.
4. Before trying to append a node after a node `B` or using `B` as the starting point for a `find`, `B` has been the head of the list and its timestamp has been set. A node's timestamp does not change after it is set. Timestamps assigned to nodes are non-decreasing.
5. If a `find(X,t)` is invoked, any node appended after `X` has a higher timestamp than `t`.
6. Processes never attempt to append the same node to a list twice, or to remove it twice.

5.1 Version List Implementation

Pseudocode for our list implementation is in Figure 4. A `remove(X)` operation first marks the node `X` to be deleted by setting a `status` field of `X` to `marked`. We refer to the subsequent physical removal of `X` as *splicing* `X` out of the list.

Splicing a node `B` from a doubly-linked list requires finding its left and right neighbours, `A` and `C`, and then updating the pointers in `A` and `C` to point to each other. Figure 2 illustrates the problem that could arise if adjacent nodes `B` and `C` are spliced out concurrently. The structure of the doubly-linked list becomes corrupted: `C` is still reachable when traversing the list towards the left, and `B` is still reachable when traversing towards the right. The challenge of designing our list implementation is to coordinate splices to avoid this situation.

We begin with an idea that has been used for parallel list contraction [44]. We assign each node a priority value and splice a node out only if its priority is greater than both of its neighbours' priorities. This ensures that two adjacent nodes cannot be spliced concurrently.

Conceptually, we can define a *priority tree* corresponding to a list of nodes with priorities as follows. Choose the node with minimum priority as the root. Then, recursively define

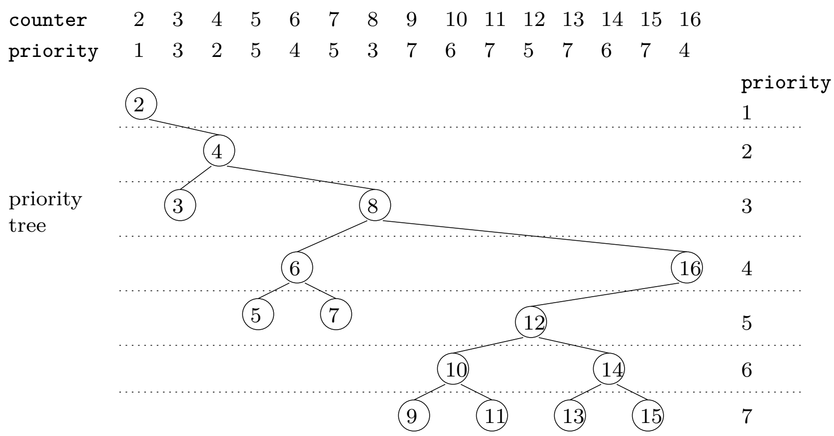

Figure 3 A list and its priority tree.

the left and right subtrees of the root by applying the same procedure to the sublists to the left and right of the root node. The original list is an in-order traversal of the priority tree. See Figure 3 for an example. We describe below how we choose priorities to ensure that (1) there is always a unique minimum in a sublist corresponding to a subtree (to be chosen as the subtree's root), and (2) if L nodes are appended to the list, the height of the priority tree is $O(\log L)$. We emphasize that the priority tree is not actually represented in memory; it is simply an aid to understanding the design of our implementation.

The requirement that a node is spliced out of the list only if its priority is greater than its neighbours corresponds to requiring that we splice only nodes whose descendants in the priority tree have all already been spliced out of the list. To remove a node that still has unspliced descendants, we simply mark it as logically deleted and leave it in the list. If `X`'s descendants have all been spliced out, then `X`'s parent `Y` in the priority tree is the neighbour of `X` in the list with the larger priority. An operation that splices `X` from the list then attempts to *help* splice `X`'s parent `Y` (if `Y` is marked for deletion and `Y` is larger than its two neighbours), and this process continues up the tree. Conceptually, this means that if a node `Z` is marked but not spliced, the last descendant of `Z` to be spliced is also responsible for splicing `Z`.

In this scheme, an unmarked node can block its ancestors in the priority tree from being spliced out of the list. For example, in Figure 3, if the nodes with counter values 10 to 16 are all marked for deletion, nodes 11, 13 and 15 could be spliced out immediately. After 13 and 15 are spliced, node 14 could be too. The unmarked node 9 prevents the remaining nodes 10, 12 and 16 from being spliced, since each has a neighbour with higher priority. Thus, an unmarked node could prevent up to $\Theta(\log L)$ marked nodes from being spliced out of the list.

Improving this space overhead factor to $O(1)$ requires an additional, novel mechanism. If an attempt to remove node `B` observes that `B`'s left neighbour `A` is unmarked and `B`'s priority is greater than `B`'s right neighbour `C`'s priority, we allow `B` to be spliced out of the list using a special-purpose routine called `spliceUnmarkedLeft`, even if `A`'s priority is greater than `B`'s. In the example of the previous paragraph, this would allow node 10 to be spliced out after 11. Then, node 12 can be spliced out after 10 and 14, again using `spliceUnmarkedLeft`, and finally node 16 can be spliced out. A symmetric routine `spliceUnmarkedRight` applies if `C` is unmarked and `B`'s priority is greater than `A`'s. This additional mechanism of splicing out nodes when one neighbour is unmarked allows us to splice out all nodes in a string of consecutive marked nodes, except possibly one of them, which might remain in the list if both its neighbours are unmarked and have higher priority. However, during the spliceUnmarkedLeft routine that is splicing out B, A could become marked. If A's priority is greater than its two neighbours' priorities, there could then be simultaneous splices of A and B. To avoid this, instead of splicing out B directly, the spliceUnmarkedLeft installs a pointer to a *Descriptor* object into node A, which describes the splice of B. If A becomes marked, the information in the Descriptor is used to *help* complete the splice of B before A itself is spliced. Symmetrically, a spliceUnmarkedRight of B installs a Descriptor in C.

Multiple processes may attempt to splice the same node B, either because of the helping coordinated by Descriptor objects or because the process that spliced B's last descendant in the priority tree will also try to splice B itself. To avoid unnecessary work, processes use a CAS to change the status of B from marked to finalized. Only the process that succeeds in this CAS has the responsibility to recursively splice B's ancestors. (In the case of the spliceUnmarkedLeft and spliceUnmarkedRight routines, only the process that successfully installs the Descriptor recurses.) If one process responsible for removing a node (and its ancestors) stalls, it could leave $O(\log L)$ marked nodes in the list; this is the source of an *additive* $P \log L$ term in the bound we prove on the number of unnecessary nodes in the list.

We now look at the code in more detail. Each node X in the doubly-linked list has right and left pointers that point toward the list's head and away from it, respectively. X also has a status field that is initially unmarked and leftDesc and rightDesc fields to hold pointers to Descriptors for splices happening to the left and to the right of X, respectively. X's counter field is filled in when X is appended to the right end of the list with a value that is one greater than the preceding node. To ensure that the height of the priority tree is $O(\log L)$, we use the counter value c to define the priority of X as $p(c)$, where $p(c)$ is either k if c is of the form 2^k, or $2k + 1 -$ (number of consecutive 0's at the right end of the binary representation of c), if $2^k < c < 2^{k+1}$. The resulting priority tree has a sequence of nodes with priorities $1, 2, 3, \ldots$ along the rightmost path in the tree, where the left subtree of the ith node along this rightmost path is a complete binary tree of height $i - 1$, as illustrated in Figure 3. (Trees of this shape have been used to describe search trees [9] and in concurrent data structures [3, 4].) This assignment of priorities ensures that between any two nodes with the same priority, there is another node with lower priority. Moreover, the depth of a node with counter value c is $O(\log L)$. This construction also ensures that remove operations are wait-free, since the priority of a node is a bound on the number of recursive calls that a remove performs.

A Descriptor of a splice of node B out from between A and C is an object that stores pointers to the three nodes A, B and C. After B is marked, we set its Descriptor pointers to a special Descriptor frozen to indicate that no further updates should occur on them.

To append a new node C after the head node B, the tryAppend(B,C) operation simply fills in the fields of C, and then attempts to swing the Head pointer to C at line 36. B's right pointer is then updated at line 37. If the tryAppend stalls before executing line 37, any attempt to append another node after C will first help complete the append of C (line 32). The boolean value returned by tryAppend indicates whether the append was successful.

A remove(B) first sets B's status to marked at line 44. It then stores the frozen Descriptor in both B->leftDesc and B->rightDesc. The first attempt to store frozen in one of these fields may fail, but we prove that the second will succeed because of some handshaking, described below. B is *frozen* once frozen is stored in both of its Descriptor fields. Finally, remove(B) calls removeRec(B) to attempt the real work of splicing B.

The removeRec(B) routine manages the recursive splicing of nodes. It first calls splice, spliceUnmarkedLeft or spliceUnmarkedRight, as appropriate, to splice B. If the splice of B was successful, it then recurses (if needed) on the neighbour of B with the larger priority.

The actual updates to pointers are done inside the splice(A,B,C) routine, which is called

```
class Node {
  Node *left, *right; // initially null
  enum status {unmarked,marked,finalized};
      // initially unmarked
  int counter; // used to define priority
  int priority; // defines implicit tree
  int ts; // timestamp
  Descriptor *leftDesc, *rightDesc;
      // initially null
};

class Descriptor { Node *A, *B, *C; };
Descriptor* frozen = new Descriptor();

class VersionList {
  Node* Head;
  // public member functions:
  Node* getHead() {return Head;}

  Node* find(Node* start, int ts) {
    VNode* cur = start;
    while(cur != null && cur->ts > ts)
      cur = cur->left;
    return cur; }

  bool tryAppend(Node* B, Node* C) {
    // B can be null iff C is the initial node
    if(B != null) {
      C->counter = B->counter+1;
      Node* A = B->left;
      // Help tryAppend(A, B)
      if(A != null) CAS(&(A->right), null, B);
    } else C->counter = 2;
    C->priority = p(C->counter);
    C->left = B;
    if(CAS(&Head, B, C)) {
      if(B != null) CAS(&(B->right), null, C);
      return true;
    } else return false; }

  // public static functions:
  void remove(Node* B) {
    // B cannot be null
    B->status = marked;
    for F in [leftDesc, rightDesc] {
      repeat twice {
        Descriptor* desc = B->F;
        help(desc);
        CAS(&(B->F), desc, frozen); } }
    removeRec(B); }

  // private helper functions:
  bool validAndFrozen(Node* D) {
    // rightDesc is frozen second
    return D != null && D->rightDesc == frozen; }

  void help(Descriptor* desc) {
    if(desc != null && desc != frozen)
      splice(desc->A, desc->B, desc->C); }

  int p(int c) {
    k = floor(log2(c));
    if(c == 2^k) return k;
    else return 2k + 1 - lowestSetBit(c); }
// private helper functions continued:
void removeRec(Node* B) {
  // B cannot be null
  Node* A = B->left;
  Node* C = B->right;
  if(B->status == finalized) return;
  int a, b, c;
  if(A != null) a = A->priority;
  else a = 0;
  if(C != null) c = C->priority;
  else c = 0;
  b = B->priority;
  if(a < b > c) {
    if(splice(A, B, C)) {
      if(validAndFrozen(A)) {
        if(validAndFrozen(C) && c > a) removeRec(C);
        else removeRec(A); }
      else if(validAndFrozen(C)) {
        if(validAndFrozen(A) && a > c) removeRec(A);
        else removeRec(C); } } }
  else if(a > b > c) {
    if(spliceUnmarkedLeft(A, B, C) &&
    validAndFrozen(C)) {
      removeRec(C); } }
  else if(a < b < c) {
    if(spliceUnmarkedRight(A, B, C) &&
    validAndFrozen(A)) {
      removeRec(A); } } } }

bool splice(Node* A, Node* B, Node* C) {
  // B cannot be null
  if(A != null && A->right != B) return false;
  bool result = CAS(&(B->status), marked, finalized);
  if(C != null) CAS(&(C->left), B, A);
  if(A != null) CAS(&(A->right), B, C);
  return result; }

bool spliceUnmarkedLeft(Node* A, Node* B, Node* C) {
  // A, B cannot be null
  Descriptor* oldDesc = A->rightDesc;
  if(A->status != unmarked) return false;
  help(oldDesc);
  if(A->right != B) return false;
  Descriptor* newDesc = new Descriptor(A, B, C);
  if(CAS(&(A->rightDesc), oldDesc, newDesc)) {
    // oldDesc != frozen
    help(newDesc);
    return true;
  } else return false; }

bool spliceUnmarkedRight(Node* A, Node* B, Node* C) {
  // B, C cannot be null
  Descriptor* oldDesc = C->leftDesc;
  if(C->status != unmarked) return false;
  help(oldDesc);
  if(C->left != B || (A != null && A->right != B))
    return false;
  Descriptor* newDesc = new Descriptor(A, B, C);
  if(CAS(&(C->leftDesc), oldDesc, newDesc)) {
    // oldDesc != frozen
    help(newDesc);
    return true;
  } else return false; } };
```

Figure 4 Linearizable implementation of our doubly-linked list.

after reading `A` in `B->left` and `C` in `B->right`. The routine first tests that `A->right` = `B` at line 96. This could fail for two reasons: `B` has already been spliced out, so there is no need to proceed, or there is a `splice(A,D,B)` that has been partially completed; `B->left` has been updated to `A`, but `A->right` has not yet been updated to `B`. In the latter case, the `remove` that is splicing out `D` will also splice `B` after `D`, so again there is no need to proceed with the splice of `B`. If `A->right` = `B`, `B`'s `status` is updated to `finalized` at line 97, and the pointers in `C` and `A` are updated to splice `B` out of the list at line 98 and 99.

The `spliceUnmarkedLeft(A,B,C)` handles the splicing of a node `B` when `B`'s left neighbour `A` has higher priority but is unmarked, and `B`'s right neighbour `C` has lower priority. The operation attempts to CAS a Descriptor of the splice into `A->rightDesc` at line 109. If there was already an old Descriptor there, it is first helped to complete at line 106. If the new Descriptor is successfully installed, the `help` routine is called at line 111, which in turn calls `splice` to complete the splicing out of `B`. The `spliceUnmarkedLeft` operation can fail in several ways. First, it can observe that `A` has become marked, in which case `A` should be spliced out before `B` since `A` has higher priority. (This test is also a kind of handshaking: once a node is marked, at most one more Descriptor can be installed in it, and this ensures that one of the two attempts to install `frozen` in a node's Descriptor field during the `remove` routine succeeds.) Second, it can observe at line 107 that `A->right` ≠ `B`. As described above for the `splice` routine, it is safe to abort the splice in this case. Finally, the CAS at line 109 can fail, either because `A->rightDesc` has been changed to `frozen` (indicating that `A` should be spliced before `B`) or another process has already stored a new Descriptor in `A->rightDesc` (indicating either that `B` has already been spliced or will be by another process).

The `spliceUnmarkedRight` routine is symmetric to `spliceUnmarkedLeft`, aside from a slight difference in line 120 because `splice` changes the `left` pointer before the `right` pointer. The return values of `splice`, `spliceUnmarkedLeft` and `spliceUnmarkedRight` say whether the calling process should continue recursing up the priority tree to splice out more nodes.

5.2 Properties of the Implementation

Detailed proofs of the following results appear in the full version [8]. We sketch them here.

▶ **Theorem 7.** *Under Assumption 6, the implementation in Figure 4 is linearizable.*

Since the implementation is fairly complex, the correctness proof is necessarily quite intricate. We say that `X` $<_c$ `Y` if node `X` is appended to the list before node `Y`. We prove that `left` and `right` pointers in the list always respect this ordering. Removing a node has several key steps: marking it (line 44), freezing it (second iteration of line 49), finalizing it (successful CAS at line 97) and then making it unreachable (successful CAS at line 99). We prove several lemmas showing that these steps take place in an orderly way. We also show that the steps make progress. Finally, we show that the coordination between `remove` operations guarantees that the structure of the list remains a doubly-linked list in which nodes are ordered by $<_c$, except for a temporary situation while a node is being spliced out, during which its left neighbour may still point to it after its right neighbour's pointer has been updated to skip past it. To facilitate the inductive proof of this invariant, it is wrapped up with several others, including an assertion that overlapping calls to `splice` of the form `splice(W,X,Y)` and `splice(X,Y,Z)` never occur. The invariant also asserts that unmarked nodes remain in the doubly-linked list; no `left` or `right` pointer can jump past a node that has not been finalized. Together with Assumption 6.3, this ensures a `find` cannot miss the node that it is supposed to return, regardless of how `find` and `remove` operations are linearized. We linearize `getHead` and `tryAppend` when they access the `Head` pointer.

▶ **Theorem 8.** *The number of steps a* `remove(X)` *operation performs is* $O($`X->priority`$)$ *and the* `remove` *operation is therefore wait-free.*

Proof. Aside from the call to `removeRec(X)`, `remove(X)` performs $O(1)$ steps. Aside from doing at most one recursive call to `removeRec`, a `removeRec` operation performs $O(1)$ steps. Each time `removeRec` is called recursively, the node on which it is called has a smaller priority. Since priorities are non-negative integers, the claim follows. ◀

▶ **Theorem 9.** *The* `tryAppend` *and* `getHead` *operations take* $O(1)$ *steps. The amortized number of steps for* `remove` *is* $O(1)$*.*

Consider an execution with R `remove` operations. Using the argument for Theorem 8, it suffices to bound the number of calls to `removeRec`. There are at most R calls to `removeRec` directly from `remove`. For each of the R nodes `X` that are removed, we show that at most one call to `removeRec(X)` succeeds either in finalizing `X` or installing a Descriptor to remove `X`, and only this `removeRec(X)` can call `removeRec` recursively.

We say a node is *lr-reachable* if it is reachable from the head of the list by following `left` or `right` pointers. A node is *lr-unreachable* if it is not lr-reachable.

▶ **Theorem 10.** *At the end of any execution by P processes that contains L successful* `tryAppend` *operations and R* `remove` *operations on a set of version lists, and a maximum of L_{max} successful* `tryAppend`*s on a single version list, the total number of lr-reachable nodes across all the version lists in the set is at most* $2(L - R) + O(P \log L_{max})$*.*

Theorem 10 considers a set of version lists to indicate that the $O(P \log L_{max})$ additive space overhead is shared across all the version lists in the system. A node `X` is *removable* if `remove(X)` has been invoked. We must show at most $(L - R) + O(P \log L_{max})$ removable nodes are still lr-reachable. We count the number of nodes that are in each of the various phases (freezing, finalizing, making unreachable) of the removal. There are at most P removable nodes that are not yet frozen, since each has a pending `remove` operation on it. There are at most P finalized nodes that are still lr-reachable, since each has a pending `splice` operation on it. To bound the number of nodes that are frozen but not finalized, we classify an unfinalized node as Type 0, 1, or 2, depending on the number of its subtrees that contain an unfinalized node. We show that each frozen, unfinalized node `X` of type 0 or 1 has a pending `remove` or `removeRec` at one of its descendants. So, there are $O(P \log L_{max})$ such nodes. We show that at most half of the unfinalized nodes are of type 2, so there are at most $L - R + O(P \log L_{max})$ type-2 nodes. Summing up yields the bound.

6 Memory Reclamation for Version Lists

We now describe how to safely reclaim the nodes spliced out of version lists and the Descriptor objects that are no longer needed. We apply an implementation of Reference Counting (RC) [2] with amortized expected $O(1)$ time overhead to a slightly modified version of our list. To apply RC in Figure 4, we add a reference count field to each Node or Descriptor and replace raw pointers to Nodes or Descriptors with reference-counted pointers. Reclaiming an object clears all its reference-counted pointers, which may lead to recursive reclamations if any reference count hits zero. This reclamation scheme is simple, but not sufficient by itself because a single pointer to a spliced out node may prevent a long chain of spliced out nodes from being reclaimed (see Figure 5, discussed later). To avoid this, we modify the `splice` routine so that whenever the `left` or `right` pointer of an node `Y` points to a descendant in

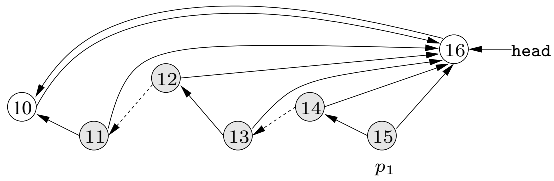

Figure 5 A portion of a version list where shaded nodes 15, 14, ..., 11 have been removed, in that order. Dotted pointers represent left and right pointers set to $\top$ by our modified `splice` routine. Node labels are counter values and vertical positioning represents nodes' priorities (cf. Figure 3).

the implicit tree, we set the pointer to $\top$ after `Y` is spliced out. Thus, only `left` and `right` pointers from spliced out nodes to their ancestors in the implicit tree remain valid. This ensures that there are only $O(\log L)$ spliced out nodes reachable from any spliced out node.

This modification requires some changes to `find`. When a `find` reaches a node whose `left` pointer is $\top$, the traversal moves right instead; this results in following a valid pointer because whenever `splice(A, B, C)` is called, it is guaranteed that either `A` or `C` is an ancestor of `B`. For example in Figure 5, a process p_1, paused on node 15, will next traverse nodes 14, 16, and 10. Breaking up chains of removed nodes (e.g., from node 15 to 11 in Figure 5) by setting some pointers to $\top$ is important because otherwise, such chains can become arbitrarily long and a process paused at the head of a chain can prevent all of its nodes from being reclaimed. In the full version of the paper, we prove that traversing backwards does not have any significant impact on the time complexity of `find`. Intuitively, this is because backwards traversals only happen when the `find` is poised to read a node that has already been spliced out and each backwards traversal brings it closer to a non-removed node.

Using the memory reclamation scheme described above, we prove Theorems 11 and 12 that provide bounds similar to Theorems 9 and 10 in [8]. Both theorems include the resources needed by the RC algorithm, such as incrementing reference counts, maintaining retired lists, etc. Since the RC algorithm uses process-local hash tables, the amortized time bounds in Theorem 9 become amortized *in expectation* in Theorem 11. Using this scheme requires that `getHead` and `find` return reference counted pointers rather than raw pointers. Holding on to these reference counted pointers prevents the nodes that they point to from being reclaimed. For the space bounds in Theorem 12, we consider the number of reference counted pointers K, returned by version list operations that are still used by the application code. In most multiversioning systems (including the one in Appendix A), each process holds on to a constant number of such pointers, so $K \in O(P)$.

▶ **Theorem 11.** *The amortized expected time complexity of* `tryAppend`, `getHead`, `remove`, *and creating a new version list is* $O(1)$. *The amortized expected time complexity of* `find(V, ts)` *is* $O(n + \min(d, \log c))$, *where* n *is the number of version nodes with timestamp greater than* `ts` *that are reachable from* `V` *by following* `left` *pointers (measured at the start of the* `find`*),* d *is the depth of the VNode* `V` *in the implicit tree and* c *is the number of successful* `tryAppend` *from the time* `V` *was the list head until the end of the* `find`*. All operations are wait-free.*

▶ **Theorem 12.** *Assuming there are at most* K *reference-counted pointers to VNodes from the application code, at the end of any execution that contains* L *successful* `tryAppend` *operations,* R `remove` *operations and a maximum of* L_{max} *successful* `tryAppend`*s on a single version list, the number of VNodes and Descriptors that have been allocated but not reclaimed is* $O((L - R) + (P^2 + K) \log L_{max})$.

In RC, cycles must be broken before a node can be reclaimed. While there are cycles in our version lists, we show that VNodes that have been spliced out are not part of any cycle.

A Application to Snapshottable Data Structures

We present a summary of the multiversioning scheme of Wei et al. [49], and describe how the techniques in this paper can be applied to achieve good complexity bounds.

The Multiversioning Scheme. Wei et al. [49] apply multiversioning to a concurrent data structure (DS) implemented from CAS objects to make it *snapshottable*. It does so by replacing each CAS object by a VersionedCAS object which stores a version list of all earlier values of the object. VersionedCAS objects support `vRead` and `vCAS` operations, which behave like ordinary read and CAS. They also support a `readVersion` operation which can be used to read earlier values of the object. Wei et al. present an optimization for avoiding the level of indirection introduced by version lists. For simplicity, we apply our MVGC technique to the version without this optimization.

Wei et al. also introduce a *Camera* object which is associated with these *VersionedCAS* objects. The Camera object simply stores a timestamp. A `takeSnapshot` operation applied to the Camera object attempts to increment the timestamp and returns the old value of the timestamp as a snapshot handle. To support read-only query operations on the concurrent DS (such as range-query, successor, filter, etc.), it suffices to obtain a snapshot handle s, and then read the relevant objects in the DS using `readVersion`(s) to get their values at the linearization point of the `takeSnapshot` that returned s. This approach can be used to add arbitrary queries to many standard data structures.

For multiversion garbage collection, Wei et al. [49] uses a variation of EBR [23], inheriting its drawbacks. Applying our range-tracking and version-list data structures significantly reduces space usage, resulting in bounded space without sacrificing time complexity.

Applying Our MVGC Scheme. Operations on snapshottable data structures (obtained by applying the technique in [49]) are divided into *snapshot queries*, which use a snapshot handle to answer queries, and *frontier operations*, which are inherited from the original non-snapshottable DS. We use our doubly-linked list algorithm (with the memory reclamation scheme from Section 6) for each VersionedCAS object's version list, and a range-tracking object `rt` to announce timestamps and keep track of required versions by ongoing snapshot queries. We distinguish between objects inherited from the original DS (DNodes) and version list nodes (VNodes). For example, if the original DS is a search tree, the DNodes would be the nodes of the search tree. See [8] for the enhanced code of [49] with our MVGC scheme.

At the beginning of each snapshot query, the taken snapshot is announced using `rt.announce()`. At the end of the query, `rt.unannounce()` is called to indicate that the snapshot that it reserved is no longer needed. Whenever a `vCAS` operation adds a new VNode `C` to the head of a version list, we `deprecate` the previous head VNode `B` by calling `rt.deprecate(B, B.timestamp, C.timestamp)`. Our announcement scheme prevents VNodes that are part of any ongoing snapshot from being returned by `deprecate`.

Once a VNode is returned by a `deprecate`, it is removed from its version list and the reclamation of this VNode and the Descriptors that it points to is handled automatically by the reference-counting scheme of Section 6. Thus, we turn our attention to DNodes. A DNode can be reclaimed when neither frontier operations nor snapshot queries can access it.

We assume that the original, non-snapshottable DS comes with a memory reclamation scheme, MRS, which we use to determine if a DNode is needed by any frontier operation. We assume that this scheme calls `retire` on a node X when it becomes unreachable from the roots of the DS, and `free` on X when no frontier operations need it any longer. This assumption is naturally satisfied by many well-known reclamation schemes (e.g., [28, 41, 23]).

Even when MRS `frees` a DNode, it may not be safe to reclaim it, as it may still be needed by ongoing snapshot queries. To solve this problem, we tag each DNode with a birth timestamp and a retire timestamp. A DNode's birth timestamp is set after a DNode is allocated but before it is attached to the data structure. Similarly, a DNode's retire timestamp is set when MRS calls `retire` on it. We say that a DNode is *necessary* if it is not yet freed by MRS, or if there exists an announced timestamp in between its birth and retire timestamp. We track this using the same range-tracking data structure `rt` that was used for VNodes. Whenever MRS `frees` a DNode `N`, we instead call `rt.deprecate(N, N.birthTS, N.retireTS)`. When a DNode gets returned by a `deprecate`, it is no longer needed so we reclaim its storage space.

We say that a VNode is *necessary* if it is pointed to by a DNode that has not yet been deprecated (i.e. freed by MRS) or if its interval contains an announced timestamp. Let D and V be the maximum, over all configurations in the execution, of the number of necessary DNodes and VNodes, respectively. Theorem 13 bounds the overall memory usage of our memory-managed snapshottable data structure. Theorem 14 is an amortized version of the time bounds proven in [49].

▶ **Theorem 13.** *Assuming each VNode and DNode takes $O(1)$ space, the overall space usage of our memory-managed snapshottable data structure is $O(D + V + P^2 \log P + P^2 \log L_{max})$, where L_{max} is the maximum number of successful* `vCAS` *operations on a single* `VCAS` *object.*

▶ **Theorem 14.** *A snapshot query takes amortized expected time proportional to its sequential complexity plus the number of* `vCAS` *instructions concurrent with it. The amortized expected time complexity of frontier operations is the same as in the non-snapshottable DS.*

References

1. D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit. StackTrack: An automated transactional approach to concurrent memory reclamation. In *Proc. 9th European Conference on Computer Systems*, pages 25:1–25:14, 2014.
2. D. Anderson, G. E. Blelloch, and Y. Wei. Concurrent deferred reference counting with constant-time overhead. In *Proc. 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 526–541, 2021.
3. J. Aspnes, H. Attiya, and K. Censor. Max registers, counters, and monotone circuits. In *Proc. 28th ACM Symposium on Principles of Distributed Computing*, pages 36–45, 2009.
4. H. Attiya and A. Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, 2001.
5. H. Attiya and E. Hillel. Built-in coloring for highly-concurrent doubly-linked lists. *Theory of Computing Systems*, 52(4):729–762, 2013.
6. H. G. Baker. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, Apr. 1978.
7. D. Basin, E. Bortnikov, A. Braginsky, G. Golan-Gueta, E. Hillel, I. Keidar, and M. Sulamy. KiWi: A key-value map for scalable real-time analytics. *ACM Trans. Parallel Comput.*, 7(3):16:1–16:28, June 2020.
8. N. Ben-David, G. E. Blelloch, P. Fatourou, E. Ruppert, Y. Sun, and Y. Wei. Space and time bounded multiversion garbage collection, 2021. Available from https://arxiv.org/abs/2108.02775.
9. J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.*, 5(3):82–86, 1976.
10. P. A. Bernstein and N. Goodman. Multiversion concurrency control–theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, Dec. 1983.

11 G. E. Blelloch and P. Cheng. On bounding time and space for multiprocessor garbage collection. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 104–117, 1999.

12 G. E. Blelloch, J. T. Fineman, Y. Gu, and Y. Sun. Optimal parallel algorithms in the binary-forking model. In *Proc. ACM Symp. on Parallelism in Algorithms and Architectures*, pages 89–102, 2020.

13 G. E. Blelloch and Y. Wei. LL/SC and atomic copy: Constant time, space efficient implementations using only pointer-width CAS. In *Proc. 34th International Symposium on Distributed Computing*, volume 179 of *LIPICS*, pages 5:1–5:17, 2020.

14 J. Böttcher, V. Leis, T. Neumann, and A. Kemper. Scalable garbage collection for in-memory MVCC systems. *Proceedings of the VLDB Endowment*, 13(2):128–141, 2019.

15 T. A. Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 261–270, 2015.

16 A. Correia, P. Ramalhete, and P. Felber. Orcgc: automatic lock-free memory reclamation. In *Proc. of the 26th ACM Symp. on Principles and Practice of Parallel Programming*, pages 205–218, 2021.

17 D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele. Lock-free reference counting. In *Proc. 20th ACM Symposium on Principles of Distributed Computing*, pages 190–199, 2001.

18 A. Dragojević, M. Herlihy, Y. Lev, and M. Moir. On the power of hardware transactional memory to simplify memory management. In *Proc. 30th ACM Symposium on Principles of Distributed Computing*, pages 99–108, 2011.

19 F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proc. 29th ACM Symposium on Principles of Distributed Computing*, pages 131–140, 2010.

20 P. Fatourou and N. D. Kallimanis. Highly-efficient wait-free synchronization. *Theory of Computing Systems*, 55(3):475–520, 2014.

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

22 S. M. Fernandes and J. Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proc. 16th ACM Symposium on Principles and Practice of Parallel Programming*, pages 179–188, 2011.

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

24 M. Greenwald. 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*, pages 260–269, 2002.

25 R. Guerraoui, A. Kogan, V. J. Marathe, and I. Zablotchi. Efficient multi-word compare and swap. In *34th International Symposium on Distributed Computing*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

26 T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proc. International Symposium on Distributed Computing*, pages 300–314. Springer, 2001.

27 T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *International Symposium on Distributed Computing*, pages 265–279. Springer, 2002.

28 M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, May 2005.

29 I. Keidar and D. Perelman. Multi-versioning in transactional memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, volume 8913 of *LNCS*, pages 150–165. Springer, 2015.

30 J. Kim, A. Mathew, S. Kashyap, M. K. Ramanathan, and C. Min. MV-RLU: Scaling read-log-update with multi-versioning. In *Proc. 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 779–792, 2019.

31 P. Kumar, S. Peri, and K. Vidyasankar. A timestamp based multi-version STM algorithm. In *Proc. Int. Conference on Distributed Computing and Networking*, pages 212–226, 2014.

32 J. Lee, H. Shin, C. G. Park, S. Ko, J. Noh, Y. Chuh, W. Stephan, and W.-S. Han. Hybrid garbage collection for multi-version concurrency control in SAP HANA. In *Proc. International Conference on Management of Data*, page 1307–1318, 2016.

33 L. Lu and M. L. Scott. Generic multiversion STM. In *Proc. International Symposium on Distributed Computing*, pages 134–148. Springer, 2013.

34 M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

35 J. Nelson, A. Hassan, and R. Palmieri. Poster: Bundled references: An abstraction for highly-concurrent linearizable range queries. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, pages 448–450, 2021.

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

37 R. Nikolaev and B. Ravindran. Snapshot-free, transparent, and robust memory reclamation for lock-free data structures. In *Proceedings of the 42nd ACM International Conference on Programming Language Design and Implementation*, pages 987–1002, 2021.

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

39 D. Perelman, A. Byshevsky, O. Litmanovich, and I. Keidar. SMV: Selective multi-versioning STM. In *Proc. International Symposium on Distributed Computing*, pages 125–140, 2011.

40 D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 16–25, 2010.

41 P. Ramalhete and A. Correia. Brief announcement: Hazard eras–non-blocking memory reclamation. In *Proc. 29th ACM Symp. on Parallelism in Algorithms and Architectures*, pages 367–369, 2017.

42 D. Reed. Naming and synchronization in a decentralized computer system. Technical Report LCS/TR-205, EECS Dept., MIT, Sept. 1978.

43 N. Shafiei. Non-blocking doubly-linked lists with good amortized complexity. In *Proc. 19th Int. Conference on Principles of Distributed Systems*, volume 46 of *LIPIcs*, pages 35:1–35:17, 2015.

44 J. Shun, Y. Gu, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *Proc. 26th ACM-SIAM Symposium on Discrete Algorithms*, pages 431–448, 2015.

45 A. Singh, T. Brown, and A. Mashtizadeh. NBR: Neutralization based reclamation. In *Proc. 26th ACM Symp. on Principles and Practice of Parallel Programming*, pages 175–190, 2021.

46 H. Sundell. Wait-free reference counting and memory management. In *Proc. 19th IEEE Symposium Parallel and Distributed Processing*, 2005.

47 H. Sundell and P. Tsigas. Lock-free deques and doubly linked lists. *J. Parallel and Distributed Computing*, 68(7):1008–1020, 2008.

48 J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pages 214–222, 1995.

49 Y. Wei, N. Ben-David, G. E. Blelloch, P. Fatourou, E. Ruppert, and Y. Sun. Constant-time snapshots with applications to concurrent data structures. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, pages 31–46, 2021. A full version is available from `https://arxiv.org/abs/2007.02372`.

50 H. Wen, J. Izraelevitz, W. Cai, H. A. Beadle, and M. L. Scott. Interval-based memory reclamation. In *Proc. 23rd ACM Symp. on Principles and Practice of Parallel Programming*, pages 1–13, 2018.

51 Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. of the VLDB Endowment*, 10(7):781–792, Mar. 2017.