# VBR: Version Based Reclamation

**Gali Sheffi**
Department of Computer Science, Technion, Haifa, Israel

**Maurice Herlihy**
Department of Computer Science, Brown University, Providence, USA

**Erez Petrank**
Department of Computer Science, Technion, Haifa, Israel

## Abstract

Safe lock-free memory reclamation is a difficult problem. Existing solutions follow three basic methods (or their combinations): epoch based reclamation, hazard pointers, and optimistic reclamation. Epoch-based methods are fast, but do not guarantee lock-freedom. Hazard pointer solutions are lock-free but typically do not provide high performance. Optimistic methods are lock-free and fast, but previous optimistic methods did not go all the way. While reads were executed optimistically, writes were protected by hazard pointers. In this work we present a new reclamation scheme called *version based reclamation* (VBR), which provides a full optimistic solution to lock-free memory reclamation, obtaining lock-freedom and high efficiency. Speculative execution is known as a fundamental tool for improving performance in various areas of computer science, and indeed evaluation with a lock-free linked-list, hash-table and skip-list shows that VBR outperforms state-of-the-art existing solutions.

**2012 ACM Subject Classification** Software and its engineering → Concurrent programming languages; Software and its engineering → Concurrent programming structures; Theory of computation → Parallel computing models; Theory of computation → Concurrent algorithms; Software and its engineering → Garbage collection; Software and its engineering → Multithreading

**Keywords and phrases** Safe memory reclamation, concurrency, linearizability, lock-freedom

## 1 Introduction

Lock-freedom guarantees eventual system-wide progress, regardless of the behavior of the executing threads. Achieving this desirable progress guarantee in practice requires a lock-free memory reclamation mechanism. Otherwise, available memory space may be exhausted and the executing threads may be indefinitely blocked while attempting to allocate, foiling any progress guarantee. Automatic garbage collection could solve this problem for high-level managed languages, but while some efforts have been put into designing a garbage collector that supports lock-free executions [28, 30, 42–44], a lock-free garbage collector for the entire heap is not available in the literature. Consequently, lock-free implementations must use manual memory reclamation schemes.

Manual reclamation methods rely on *retire* invocations by the program, announcing that a certain object has been unlinked from a data-structure. After an object is retired, the task of the memory reclamation mechanism is to decide when it is safe to reclaim it, making its memory space available for reuse in future allocations. The memory reclamation mechanism ensures that an object is not freed by one thread, while another thread is still using it. Accessing a memory address which is no longer valid may result in unexpected and undesirable program behavior. Conservative reclamation methods make sure that no thread accesses reclaimed space, while optimistic methods allow threads to speculatively access reclaimed memory, taking care to preserve correctness nevertheless.

Conservative manual reclamation schemes can be roughly classified as either *epoch–based* or *pointer–based*. In epoch–based reclamation (EBR) schemes [10, 22, 23, 33], all threads share a global epoch counter, which is incremented periodically. Additionally, the threads share an announcements array, in which they record the last seen epoch. During reclamation, only

objects that had been retired before the earliest recorded epoch are reclaimed. These schemes are often fast, but they are not robust. I.e., a stalled thread may prevent the reclamation of an unbounded number of retired objects. At a worst-case scenario, these objects will consume too much memory space resulting in blocking all new allocations and consequently, foiling system-wide progress.

Pointer–based reclamation methods [26, 35, 37, 49] allow threads to protect specific objects. Namely, before accessing an object, a thread can announce its access in order to prevent this object from being reclaimed. While pointer-based methods can guarantee robustness (and consequentially, lock-freedom), they incur significant time overheads because they need to protect each dereference to shared memory, and issue an expensive memory synchronization fence to make sure the protection is visible to all threads before accessing the derefeneced object. Furthermore, pointer-based schemes are not applicable to many concurrent data-structures (e.g., to Harris's linked-list [22]).

Hybrid schemes that enjoy stronger progress guarantees and smaller time overheads have been proposed. Some epoch-based algorithms try to minimize the chance a non-cooperative thread will block the entire system, by allowing reclamation of objects whose life cycle do not overlap with the activity of the non-responsive thread, e.g., HE (Hazard Eras [45]) and IBR (Interval-Based Reclamation [55]). However, while these algorithms are effectively non-blocking in most practical scenarios, a halted thread can still prevent the reclamation of a large space that relates to the size of the heap. There exists a wait-free variant of the Hazard Eras algorithm [41] which provides a better guarantee but is naturally slower. Another hybrid approach, called *PEBR* [31], obtains lock-freedom at a lower cost, but relies on the elimination of the costly memory fences using the mechanism of Dice et al. [17], which in turn relies on hardware modifications or on undocumented operating systems assumptions that might not hold in the future. Another hybrid of pointer- and epoch-based reclamation is the DEBRA+ and the subsequent NBR [10, 48] reclamation schemes. In order to deal with stuck threads in the epoch-based reclamation, these schemes signal non-cooperative threads and prevent them from postponing the reclamation procedure. While DEBRA+ and NBR are fast, their lock-free property relies on the system's lock-free implementation of signaling. This assumption is not currently available in most existing operating systems, but it may become available in the future.

The first optimistic approach to lock-free memory reclamation was the *Optimistic Access* scheme [14] (also denoted as OA), speculatively allowing reads from retired objects, but protecting writes from modifying retired objects through the use of conservative pointer-based reclamation. After each read, designated per-thread flags signify whether reclamation took place. This allows the reading thread to avoid using stale data loaded from reclaimed space. Subsequent work [12, 13] increased automation and applicability. While the optimistic access reclamation scheme initiated speculative memory reclamation, its mechanism only allowed speculative read instructions. Write instructions were still applied conservatively, using hazard pointers (also denoted as HP) protection to avoid writes on reclaimed space, limiting the benefits of speculative execution.

In this paper we present *VBR*, a novel optimistic memory reclamation scheme that allows full speculative execution, achieving safe and highly efficient lock-free memory reclamation. Both read and write instructions are allowed to access reclaimed space. VBR uses a global epoch counter to assign versions to objects and to (mutable) fields. The versioning of objects ensures that read and write accesses to reclaimed space are guarded from affecting program semantics. The key invariant is that the global epoch counter is guaranteed to increment (at least once) between the time an object is retired and the time its space is re-allocated

as a new object. Each logical object is associated with its birth epoch and (eventually) its retire epoch, and each of its mutable fields is associated with a version (representing an epoch smaller or equal to its last update). A speculative read access prudently backs out and retries if the global epoch counter advances while the data structure operation is active. A speculative write operation always fails to modify a re-allocated object due to the modified versions of the object's mutable fields. To support this, each mutable field within a node is represented as a ⟨value, epoch#⟩ pair, and modified only with a double-width compare and swap.

VBR is fully optimistic. Unlike OA [12–14], writes are also speculative and do not require costly fences. Memory fences are used infrequently, upon updating the global epoch. VBR provides full lock-freedom and it does not allow a non-cooperative thread to stall the reclamation process. In fact, VBR never prevents the reclamation of any retired memory object. VBR does not rely on hardware or operating system assumptions (or modifications), except for the existence of a double-word compare and swap instruction, which is available on most existing architectures (e.g., x86).

The proposed VBR can reuse any retired object immediately after it is retired without jeopardizing correctness, which makes it highly space efficient. However, VBR does encounter an issue that pops up in several other schemes [10, 13, 14, 37, 48]. The memory manager sometimes causes a read or a write instruction to "fail" due to a memory reclamation validation test. This failure is not part of the original program control flow and thus, an adequate handling of such failure should be added. In [37] Michael proposes informally to "skip over the hazards and follow the path of the target algorithm when conflict is detected, i.e., try again, backoff, exit loop, etc.". Indeed in many known lock-free data structures [19, 20, 22, 32, 34, 38, 40, 46], handling validation failures is easy, which makes the use of VBR, and other reclamation schemes easy in practice. However, the question arises whether there is a methodology to handle failed validations for all data structures, even when we do not master their specific algorithm. A first rigorous treatment of these failures was provided in [14] for data structures that are written in the normalized form of lock-free data structures [52]. Subsequently, a weaker version of normalized concurrent data structures was presented in [48], where the failure problem is somewhat more severe, as signals may occur at an arbitrary point in the program flow. In this paper, we follow this line of work, and provide a rigorous treatment of a failed read or write validation.

We have implemented VBR on a linked-list, a skip-list, and a hash table and evaluated it against epoch-based reclamation, hazard pointers, hazard eras, interval-based reclamation, and no reclamation at all. As expected, speculative execution outperforms conservative approaches and so VBR yields both lock-freedom as well as high performance.

This paper is organized as follows. In Section 2 we provide an overview of the VBR scheme. In Section 3, we describe our shared memory model and specify some assumptions a data structure must satisfy in order to be correctly integrated with our reclamation mechanism. We describe the VBR scheme integration in Section 4. Experiments appear in Section 5. Related work is surveyed in Section 6. We prove that integrating the VBR scheme maintains linearizability and lock-freedom in Appendix A and B. Finally, an illustration of integrating VBR into a lock-free data-structure (Harris's linked-list [22]) appears in Appendix C.

## 2 Overview of VBR

The VBR memory reclamation scheme follows an optimistic approach where access to reclaimed objects is allowed. Optimistic approaches reduce the overhead but require care

to guarantee correctness. First, VBR allows immediate reclamation of each retired object. There is no need to wait for guards to be lowered to make sure an object is reclaimable as in other methods. This property ensures that stalled threads do not delay reclamation of any object. Second, on strongly-ordered systems (e.g., x86, SPARC TSO, etc.) VBR does not require a costly overhead on read or write accesses. No additional shared memory writes or memory synchronization fences are required with reads or writes to shared memory. This provides the high efficiency seen in the evaluation. However, on weakly-ordered systems (e.g., ARM, PowerPc, etc.), reads must be ordered using special CPU load or memory fence instructions [1]. VBR requires a type-preserving allocator. I.e., a memory space allocated for a specific type is used only for the same type, even when re-allocated. The assumption of type preserving (see also [13, 14, 55]) is necessary for applying our scheme, and is reasonable because data structure nodes are typically fixed-size nodes. Retired nodes are not returned to the operating system. Instead, they are returned to a shared pool of nodes, from which they can be re-allocated by any thread. As in [10, 48], a collection of local node pools (one per thread) is added to the shared pool. A thread accesses the shared pool only when it has no available nodes in its local pool.

Similarly to epoch-based reclamation, VBR maintains a global epoch counter, and as in [4, 41, 45, 55], VBR tracks the birth epoch and retire epoch of each allocated node. The birth epoch is determined upon allocation, and the retire epoch is set upon retirement. The reclamation of a retired node does not involve any action. Upon an allocation of a node, a thread makes sure that the retire epoch of the node is smaller than the current global epoch. If it is not, then the thread increments the global epoch. This ensures that an object is allocated at a global epoch that is strictly larger than its previous retire epoch. Next, the thread re-allocates the object by updating its birth epoch with the current global epoch. This method guarantees that the ABA problem [36] can only occur when the global epoch changes. Namely, when a thread encounters a node during a data-structure traversal, it is guaranteed that this node has not been re-allocated during the traversal if the global epoch has not changed.

VBR allows accessing reclaimed objects, while conservatively identifying reads that may access reclaimed nodes. To identify the access to a reclaimed node, each executing thread keeps track of the global epoch, by reading it upon most shared memory reads (as long as the epoch does not change, this read is likely to hit in the cache). When the thread observes an epoch change, it conservatively assumes that a value was read from a reclaimed memory and it applies a roll-back mechanism (described in Section 4.2), returning to a pre-defined checkpoint in its code. Since a node is always re-allocated at an epoch that is strictly larger than its former retirement, threads never rely on the content of stale values.

We now move on to handling optimistic writes. In addition to the birth epoch and retire epoch, each mutable field (e.g., node pointers) is associated with a version that resides next to it on the data structure node. During the execution, mutable fields are always updated atomically with their associated versions (using a wide CAS instruction). Throughout the life-cycle of a not-yet retired node, all of its versions remain greater than or equal to its birth epoch, and they never exceed its future retire epoch. Versions are decreased or increased during the execution in the following manner: when updating a pointer from a node $n$ to a node $m$, the pointer's version is set to the maximum birth epoch of the two nodes (either $n$'s or $m$'s). Notice that we assume that $n$'s pointer is never updated after its retirement (for more details, see Section 3.3), and therefore, none of its pointers are assigned a version that exceeds their retirement epoch.

Let us consider the ABA problem for this versioning scheme. The concern is that re-

allocations may result in an erroneous success of CAS executions. For example, suppose that a node $n$ points to another node, $m$, which in turn points to a third node, $k$. Now, suppose that a thread $T_1$ tries to remove $m$ by setting $n$'s pointer to point to $k$. Right before executing the removing CAS, $T_1$ is halted. While $T_1$ is idle, $T_2$ removes $m$ and then reallocates $m$'s space as a new node $d$. Next, $T_2$ inserts $d$ as a new node between $n$ and $k$. In the lack of versions, $T_1$'s CAS will now be erroneously successful. However, with versions it must fail. Since $d$'s birth epoch is necessarily bigger than $m$'s retire epoch, the version in the original pointer to $m$ must be smaller than the version assigned to the pointer when $d$ becomes its referent, and the CAS fails (for more details, see Appendix A).

# 3 Settings and Assumptions

In this section we describe our shared memory model and specify the assumptions a data structure must satisfy for integrating with our reclamation mechanism.

## 3.1 System Model

We use the basic asynchronous shared memory model, as described in [24]. In this model, a fixed set of threads communicate through memory access operations. Threads may be arbitrarily delayed or may crash in the middle of their execution (which immediately halts their execution). The shared memory is accessed via atomic instructions, provided by the hardware. Such instructions may be atomic reads and writes, the compare-and-swap (CAS) instruction and the wide-compare-and-swap (WCAS, which atomically updates two adjacent memory words, and is often supported in commodity hardware [56]) instruction. The CAS operation receives three input arguments: an address of a certain word in memory, an expected value and a new value (both of the size of a single word). It then atomically compares the memory address content to the expected value, and if they are equal, it replaces it with the new received value. Otherwise, it does nothing. The WCAS operation operates in the same manner, on two adjacent memory words.

Concurrent implementations provide different progress guarantees. *Lock-freedom* guarantees that as long as at least one thread executes its algorithm long enough, some thread will eventually make progress (e.g., complete an operation). This progress guarantee is not affected by the scheduler or even by the crash of all threads except for one. For a lock-free data structure to be truly lock-free, it must rely on an allocation method that is also lock-free, because otherwise a blocked allocation can prevent all threads from making progress.

A *data structure* represents a set of *items*, which are distinguished by unique keys, and are often arranged in some order. Each item is represented by a *node*, consisting of both mutable and immutable fields. In particular, each node has an immutable *key* field. The data-structure has a fixed set of *entry points* (e.g., the head of the linked-list in [22]), which are node pointers. A data structure provides the user with a set of operations for accessing it. Moreover, The user cannot access the data-structure in other ways, and the data structure operations never return a node reference. An item that belongs to the data structure set of items must be represented by a node which is reachable from an entry point, by following a finite set of pointers. In particular, the data-structure nodes are accessible only via the entry points. However, a reachable node does not necessarily represent an item in the data structure set of items. We denote the removal of an item from the set of items that the data structure represent by *logical deletion*, and we denote the unlinking of a node from the data structure (i.e., making the node unreachable from the entry points) as *physical deletion*. E.g., in [22], a special mechanism is used in order to mark reachable nodes as deleted. Once a

node is marked, it stops representing an item in the data structure set of items (i.e., it is logically deleted), even though it is reachable from an entry point.

## 3.2 Executions, Histories and Linearizability

A *step* can either be a shared-memory access by a thread (including the access input and output values), a local step that updates its own local variables, an invocation of an operation or the return from an operation (including the respective inputs and outputs). We assume each step is atomic, so an *execution* $E = s_1 \cdot s_2 \cdot \ldots$ consists of a sequence of steps, assumed to start after an initial state, in which all data-structures are initialized and empty. Given E, we further denote the finite sub-execution $s_1 \cdot s_2 \cdot \ldots \cdot s_i$ as $E_i$.

We follow [29], and model an execution E by its *history* H (and $E_i$ by $H_i$, respectively), which is the sub-sequence of operation invocation and response steps. A history is *sequential* if it begins with an invocation step, and all invocations (except possibly the last one) have immediate matching responses. We assume that a concurrent system is associated with a *sequential specification*, which is a prefix-closed set of all of its possible sequential histories. A sequential history is *legal* iff it belongs to the sequential specification. An invocation is *pending* in a given history if the history does not contain its matching response. Given a history H, its sub-sequence excluding all pending invocations is denoted as complete(H). An *extension* of a history H is a history constructed by appending responses to zero or more pending invocations in H. We further extend the notion of extensions, and say that an execution E' is an *extension* of an execution E if E is a prefix of E'. In addition, given an execution E, EXT(E) is the set of all histories H' such that (1) H' is an extension of E's respective history, and (2) H' is the respective history of an extension of E. Given a history H and a thread T, T's *sub-history*, denoted as H|T, is the sub-sequence of H consisting of all (and exactly) the steps executed by T. Two histories H and H' are *equivalent* if for every thread T, H|T and H'|T are equal. A history H is *well-formed* if for every executing thread T, H|T is a sequential history. A well-formed history H is linearizable if it has an extension H' for which there exists a legal sequential history S such that (1) complete(H') is equivalent to S, and (2) if a response step precedes an invocation step in H, then it also precedes it in S.

## 3.3 Implementation Assumptions

We focus on adding the VBR reclamation scheme to lock-free linearizable concurrent data-structure implementations. As in [54], we first assume that modifications are executed using the CAS instruction. No simple writes are used, and no other atomic instructions are supported. Consequently, our scheme does not support the use of other atomic primitives (such as $fetch\&add$ and *swap*).

▶ **Assumption 1.** *All updates occur only via CAS executions.*

In general, as in [45], we assume that all mutable fields of a removed node are invalidated, in order to prevent their future updates. It can be achieved either by marking them [22, 34] or by self-linking (in the case of pointers). More formally:

▶ **Assumption 2.** *Node fields are invalidated (and become immutable) using a designated* invalidate() *method. This method receives as input a node field and invalidates it. The invalidation succeeds iff the field is valid and is not concurrently being updated by another thread. In order to check whether a certain field is invalid, a thread calls a designated* isValid() *method. Finally, given a node field, a thread separates the value from the (possible) invalidation mark by calling a designated* getField() *method.*

Following the standard interface for manual reclamation [14, 31, 37, 45, 55], applying our reclamation scheme to an existing implementation includes allocating nodes using an *alloc* instruction and retiring nodes using a *retire* instruction. Nodes are always retired before they can be reclaimed by the reclamation scheme. We assume that it is possible to retire each node only once. To sum up, in a similar way to [37]:

▶ **Assumption 3.** *We assume the following life-cycle of a node $n$:*

1. ***Allocated**: $n$ is allocated by an executing thread, but is not yet reachable from the data-structure entry points. Once it is physically inserted into the data-structure, it becomes reachable.*
2. ***Reachable** (optional): $n$ is reachable from the data structure entry points, but is not yet necessarily logically inserted into the data-structure (e.g., [47, 51]). When it is made logically included in the data structure it becomes Reachable and valid.*
3. ***Reachable and valid**: $n$ is reachable from the entry points and is considered logically in the data-structure (i.e., valid). At the end of this phase, fields of $n$ are invalidated. We think of $n$ as invalid when at least one of its mutable fields is invalid.*
4. ***Invalid**: $n$ is logically deleted by a designated invalidation procedure, and all of its mutable fields are invalidated (e.g., by marking [22]). Once a field is invalidated, it becomes immutable. At the end of this phase, $n$ is unlinked (physically deleted) from the data structure.*
5. ***Unlinked**: At this point, $n$ is not reachable from the data-structure entry points, and therefore, it is not reachable from any other linked node. At the end of this phase it is retired by some thread. We assume a node is retired only once in an execution. After being retired the node is never linked back into the data structure.*
6. ***Retired**: $n$ has been retired, by a certain thread. We assume that only unlinked nodes can be retired.*

Notice that, as discussed in [13, 20], a node can be physically removed and re-inserted into the data-structure several times during stage 4. However, a *retire* instruction is issued on a node only after it is physically removed for the last time.

Finally, we assume that a thread does not use data on nodes without occasionally checking that the nodes are valid. For our scheme to work, we require this check after modifying the data structure. We assume that if a thread performs a successful modification of the data structure, and if it has some locally saved pointers that were read prior to the modification, then the thread makes limited use of these pointers. Actually, we do not even need to impose the restriction on all modifications. Restrictions are needed only for "important" modifications that cannot be rolled back. Such modifications are called *rollback-unsafe* and they are formally defined in Section 4.2.1 below. In particular:

▶ **Assumption 4.** *If thread $T$ executes any rollback-unsafe modification after updating a local pointer $p$, then a future use of $p$ is limited. Suppose $p$ references a node $n$, then a future (i.e., after the rollback-unsafe modification) read of one of $n$'s mutable fields by $T$ is allowed only if the read is followed by an* isValid() *call, and if it returns FALSE, the field content is not used by $T$.*

While "not using" the content of a read field is intuitively clear, let us also formally say that the content of a read field is not used by a thread $T$, if $T$'s behavior is indistinguishable from its behavior when reading the $\perp$ sign instead of the actual value read. Note that even after a rollback-unsafe update, $T$ is allowed to use the content of fields that were read before the modification. However, after the modification, $T$ is not allowed to dereference a local

pointer and read values from the referenced node without checking the validity of the node. For example, $T$ is allowed to use previously read pointers as expected values of a CAS, or as the target of a write operation. $T$ can traverse a list in a wait free manner (since there is no modification involved). $T$ can trim all invalid nodes along a traversal. This is allowed since trimming includes checking the validity of the traversed nodes. All known lock-free data structures that we are aware of (e.g., [19, 20, 22, 27, 32, 34, 38, 40, 46]) satisfy Assumption 4.

# 4 VBR: Version Based Reclamation

In this section we present the VBR mechanism: a lock-free recycling support for lock-free linearizable [29] data-structures. We start by describing the reclamation scheme and the modifications applied to the nodes' representation in Section 4.1, and continue with the modifications applied to the data-structure operations in Section 4.2. In Section 4.2.1 we define the notion of code checkpoints and show how to insert them into an existing linearizable implementation. In Section 4.2.2 we go over the necessary adjustments to read operations (from shared variables). Handling update operations is described in Section 4.2.3. A full example API appears in Figure 1. For ease of presentation, we refer to data-structures for which each node has a single immutable field (the node's key) and a single mutable field (the node's next pointer), and nodes' invalidation is executed via the marking mechanism [22]. However, this interface can be easily extended to handle multiple immutable and mutable fields, and other invalidation schemes. We present a full correctness proof for Theorem 1 in Appendix A.

▶ **Theorem 1.** *Given a lock-free linearizable data-structure implementation, satisfying all of the assumptions presented in Section 3.3, it remains lock-free and linearizable after integrating it with VBR according to the modifications described in Sections 4.1-4.2.*

## 4.1 The Reclamation Mechanism

VBR uses a shared epoch counter, denoted $e$, incremented periodically by the executing threads. In addition, each executing thread keeps track of the global epoch using a local *my_e* variable. Each node is associated with *birth_epoch* and *retire_epoch* fields. Its birth epoch contains the epoch seen by the allocating thread upon its allocation, and its retire epoch contains the epoch seen by the thread which removed this node from the data structure, right before its retirement. We add a version field adjacent to each mutable field (e.g., node pointers). The field's version is guaranteed to always be equal to or greater than the node's birth epoch, and equal to or smaller than its eventual retire epoch (if there exists any). The field's data and its associated version are always updated together. E.g., see lines 9, 33 in Figure 1.

Handling reclamation at the operating system level often requires using locks (unless it is configured to ignore certain traps). Therefore, for maintaining lock-freedom, VBR uses a user-level allocator. Retired nodes are inserted into manually-managed node pools [25, 53] for future re-allocation. We use a type-preserving allocator. I.e., a memory space allocated for a specific type is used only for the same type, even when re-allocated.

Each thread's allocation and reclamation mechanism works as follows. Besides sharing a global nodes pool [25, 53], each executing thread maintains a local pool of retired nodes, from which it retrieves reclaimed nodes for re-allocations. When the local pool becomes large enough, retired nodes may be moved to a global pool of retired nodes, allowing re-distribution of reclaimed nodes between the threads. When retiring a node, it is possible to re-allocate

this node immediately. However, we use a local retired list to stall its re-allocation for a while. This allows infrequent increments to the global epoch counter, which improves performance. A retired node is therefore added to the private list of retired nodes. When the size of the retired list exceeds a pre-defined threshold, it is appended as a whole to the thread's local allocation pool, becoming available for allocation.

The full allocation method appears in lines 1-11 of Figure 1. First, the thread reads the retire epoch of the next available node in its allocation pool. If it is equal to the shared epoch counter, then the thread increments the shared epoch counter using CAS (line 4) and executes a rollback to the previous checkpoint (for more details, see Section 4.2.1). This makes sure that the birth epoch of a new node is larger than the retire epoch of the node that was previously allocated on the same memory space. If the CAS is unsuccessful, then another thread has incremented the global epoch value and there is no need to try incrementing it again. If $e$ is bigger than the retired node's retire epoch, the thread sets the new node's birth epoch to its current value. After setting the node's birth epoch, its next pointer version is set to this value, along with an initialization of its data to NULL in line 9. Due to Assumption 3 (the mutable fields of a node become immutable before it is retired), the WCAS executed in line 9 is always successful. Finally, the key field is set to the key received as input.

The *retire* method appears in lines 12-16. First, the retiring thread makes sure that the node is not already retired in line 13 (for more details, see Appendix A). Then, it sets the node's retire epoch to be the current global epoch, and appends the retired node to its local retired nodes list. In case its local copy of the global epoch counter is not up to date, it performs a checkpoint rollback in line 16 (for more details, see Section 4.2.1).

## 4.2 Code Modifications

Unlike former reclamation methods, VBR allows both optimistic reads and optimistic writes. Namely, the executing threads may sometimes access a previously reclaimed node, and either read its stale values or try to update it. To the best of our knowledge, there exists no other scheme which allows optimistic writes, and optimistic reads are allowed only in [12–14]. Our versioning mechanism ensures that a write to a previously reclaimed node always fails, and that stale values that are read from reclaimed nodes are always ignored. This gives rise to an additional problem – when failing to read a fresh value due to an access to a reclaimed node, the program needs to move control to an adequate location. This problem does not arise with epoch based reclamation because a thread never fails due to a test that the memory reclamation scheme imposes. Failures that arise due to optimistic access are not part of the original lock-free concurrent data structure. Interestingly, deciding how to treat failed reads or writes is very easy in practice. We could easily modify lock-free data structures that satisfy the assumptions presented in Section 3.3 (e.g., [19, 20, 22, 32, 34, 38, 40, 46]), at a minimal performance cost. However, while presenting this scheme, we would also like to propose a general manner to handle failed accesses. We are going to define the notion of execution checkpoints. Upon accessing an allegedly stale value, the code just rolls back to the appropriate checkpoint. This problem is given general treatment in the format of a normalized form assumption in [13, 14], and in a total separation between read and write phases during the execution in [48]. Although the VBR scheme can be applied to implementations that adhere to both models, both of them require extensive modifications to the original program's structure. Therefore, we propose a new method, which is more general and makes less assumptions on the given implementation. Our method is to carefully define program checkpoints.

```
 1: alloc(int key)
 2:     n := alloc_list → next
 3:     if (n → retire_epoch ≥ my_e)
 4:         CAS(&e, my_e, my_e + 1)
 5:         alloc_list → next := n
 6:         return to checkpoint                                  ▷ Checkpoint rollback
 7:     n → birth_epoch := my_e
 8:     n → retire_epoch := ⊥
 9:     WCAS(&(n → next), ⟨n → next.data, n → next.version⟩, ⟨NULL, my_e⟩)
10:     n → key := key
11:     return n

12: retire(Node* n, long n_b)
13:     if (n → birth_epoch > n_b || n → retire_epoch ≠ ⊥) return     ▷ Avoiding double retirements
14:     n → retire_epoch := e.get()
15:     retired_list → next := n
16:     if (n → retire_epoch > my_e) return to checkpoint             ▷ Checkpoint rollback

17: getNext(Node* n)
18:     n_next := unmark(n → next.data)
19:     n_next_b := n_next → birth_epoch
20:     if (my_e ≠ e.get()) return to checkpoint                      ▷ Checkpoint rollback
21:     return n_next, n_next_b

22: getKey(Node* n)
23:     n_key := n → key
24:     if (my_e ≠ e.get())  return to checkpoint                     ▷ Checkpoint rollback
25:     return n_key

26: isMarked(Node* n, long n_b)
27:     res := isMarked(n → next.data)
28:     if (n → birth_epoch ≠ n_b) return TRUE                        ▷ The node is already removed
29:     return res

30: update(Node* n, long n_b, Node* exp, long exp_b, Node* new, long new_b)
31:     exp_v := max { n_b, exp_b }
32:     new_v := max { n_b, new_b }
33:     return WCAS(&(n → next), ⟨ exp, exp_v ⟩, ⟨ new, new_v ⟩)

34: mark(Node* n, long n_b)
35:     exp := unmark(n → next.data)
36:     exp_v := max { n_b, exp → birth_epoch }
37:     if (n → birth_epoch ≠ n_b) return FALSE                       ▷ The node is already removed
38:     new := mark(exp)
39:     return WCAS(&(n → next), ⟨ exp, exp_v ⟩, ⟨ new, exp_v ⟩)
```

**Figure 1** An example VBR interface

### 4.2.1 Defining Checkpoints

VBR occasionally requires a rollback to a predefined checkpoint. In order to install checkpoints in a given code in an efficient manner, one needs to be able to distinguish important shared-memory accesses that cannot be rolled back from non-important accesses that allow rolling back. The notion of important shared-memory accesses is similar to the definition of an *owner CAS* in [52], and shares some mutual concepts with the *capsules* definition, given in [6, 7]. Informally, non-important shared-memory accesses (that can be rollbacked) either do not affect the shared memory view (e.g., reading from the shared-memory) or do not have any meaningful impact on the execution flow. For example, consider Hariss's implementation of a linked-list [22]. The physical removal of a node, i.e., trimming the node from the list

after it has been marked, can be safely rollbacked. If we try the same trim again, it will simply fail, and if we rollback further, this trim will not even be attempted. However, the (successful) insertion of a new node into the list and the (successful) marking of a node for logical deletion are both important and are not rollback-safe. In both cases, performing a rollback right after the successful update would result in a non-linearizable history (as the inserter or remover would not return TRUE after successfully inserting or removing the node, respectively). We now define the notion of *rollback-safe steps* in a given execution $E$ with a respective history $H$.

▶ **Definition 2** (Rollback-Safe Steps). *We say that $s_i$ is a rollback-safe step in an execution $E$ if $EXT(E_i)$=$EXT(E_{i-1})$.*

If $s_i$ is not a rollback-safe step, then we say that it is a *rollback-unsafe step*. According to Definition 2, if $s_i$ is a rollback-safe step, executed by a thread T during $E$, then T can safely perform a *local rollback step* after $s_i$. I.e., right after executing $s_i$ by T, T can restore the contents of all of its local variables and program counter (assuming they were saved before $s_i$), and the obtained execution would have the same set of corresponding history extensions. Note that, by Definition 2, local steps, shared memory reads and unsuccessful memory updates (CAS executions returning FALSE) are considered as rollback-safe steps. We extend Definition 2 in the following manner: a thread T can rollback to any previously saved set of local variables (including its program counter), as long as it has not performed any rollback-unsafe steps since they had been saved. I.e., it can safely rollback to its last visited checkpoint.

Given a code for a concurrent data-structure, checkpoints are first installed after some shared memory update instructions, in the following manner: let $l$ be a shared-memory update instruction (i.e., a CAS instruction). If there exists an execution $E = s_1 \cdot \ldots$ such that $l$ is executed successfully during a step $s_i$ (i.e., the CAS execution returns TRUE), and $s_i$ is a rollback-unsafe step in $E$, then a checkpoint is installed right after $l$. The installation of a checkpoint includes a check that the update is indeed successful (the CAS execution returns TRUE). If it is, then the checkpoint reference is updated (to the current value of the program counter), and all local variables are saved for a future restoration[1]. If the update is not successful (the CAS execution returns FALSE), then nothing is done. Checkpoints are also installed in the beginning of each data-structure operation. As opposed to the first type of checkpoint triggers, the installation does not depend on anything when done upon an operation invocation. Recall that, by Definition 2, an operation invocation is always a rollback-unsafe step. After rolling back to a checkpoint, the thread updates its local copy of the global epoch, recovers its set of local variables, and continues its execution[2].

### 4.2.2 Read Methods

As threads may access stale values, the only way to avoid relying on a stale value is to constantly check that the node from which the value was read has not been re-allocated. In a conservative way, we think of a read instruction as potentially reading a stale value if the global epoch number changed since the last checkpoint. A read of a stale value must imply a change of the global epoch number because the birth epoch of a node is strictly larger than

[1] It is unnecessary to save uninitialized variables and variables that are not used anymore

[2] Right before a thread rolls-back to its previous checkpoint, it handles some unlinked nodes for guaranteeing VBR's robustness. As this issue does not affect correctness, we move this discussion to Appendix B.

the retirement epoch of a previous node that resides on the same memory space. Therefore, the shared epoch counter $e$ is read upon each operation invocation (see Section 4.2.1), each node retirement, and after certain allocations and reads from the shared memory. Since a node cannot be allocated during an epoch in which a node, previously allocated from the same memory address, is not yet removed from the data-structure (see lines 3-6 in Figure 1), as long as the global epoch, read before the read of a node, is equal to the one read after the read of the node, it is guaranteed that the node's value is not stale. In general, when reading a node pointer into a local variable, it is always saved together with the node's birth epoch, as the node is represented by its birth epoch as well.

W.l.o.g. and for simplifying our presentation, we assume each node originally consists of an immutable key field and a mutable next pointer field, and that a node is invalidated using the *mark()* method [22]. Therefore, there are roughly three types of read-only accesses in the original reclamation-free algorithm. The first type is the read of a node via the next pointer of its predecessor, the second one is the read of a node's key, and the third one is the read of a node's mark bit. We install the *getNext()* method instead of each pointer read in the original code, the *getKey()* method instead of each key read, and a new *isMarked()* method instead of the original one. Accesses to other (mutable or immutable) node fields should be very similar, and therefore require the same treatment.

The code for the *getNext()* method appears in lines 17-21, and the code for the *getKey()* method appears in lines 22-25. Both methods receive a pointer to the target node (assumed to be given as an unmarked pointer). First, the next node and its birth epoch (or the key, respectively) are saved in local variables. Then, the global epoch is read and compared to the previous recorded epoch. If the epoch has changed since the previous read, then the values may be stale, and the execution returns to the last checkpoint. Otherwise, the data is returned in line 21 (or line 25, respectively).

The code for the VBR-integrated *isMarked()* method appears in lines 26-29. It also receives an unmarked pointer to the target node, and additionally, its birth epoch. It first checks whether the node's next pointer is indeed marked (line 27), using the original *isMarked()* method, which receives the actual allegedly marked pointer and checks if it is marked. Then the node's birth epoch is read, for guaranteeing that the given node is the correct one (and not another one, allocated from the same memory space). If it is not, then the target node has certainly been marked in the past, and the method returns TRUE in line 28. Otherwise, it returns the answer received in line 27. As this method returns a correct answer regardless of epoch changes or the retirement of the target node, it does not read the global epoch nor returns to the last checkpoint.

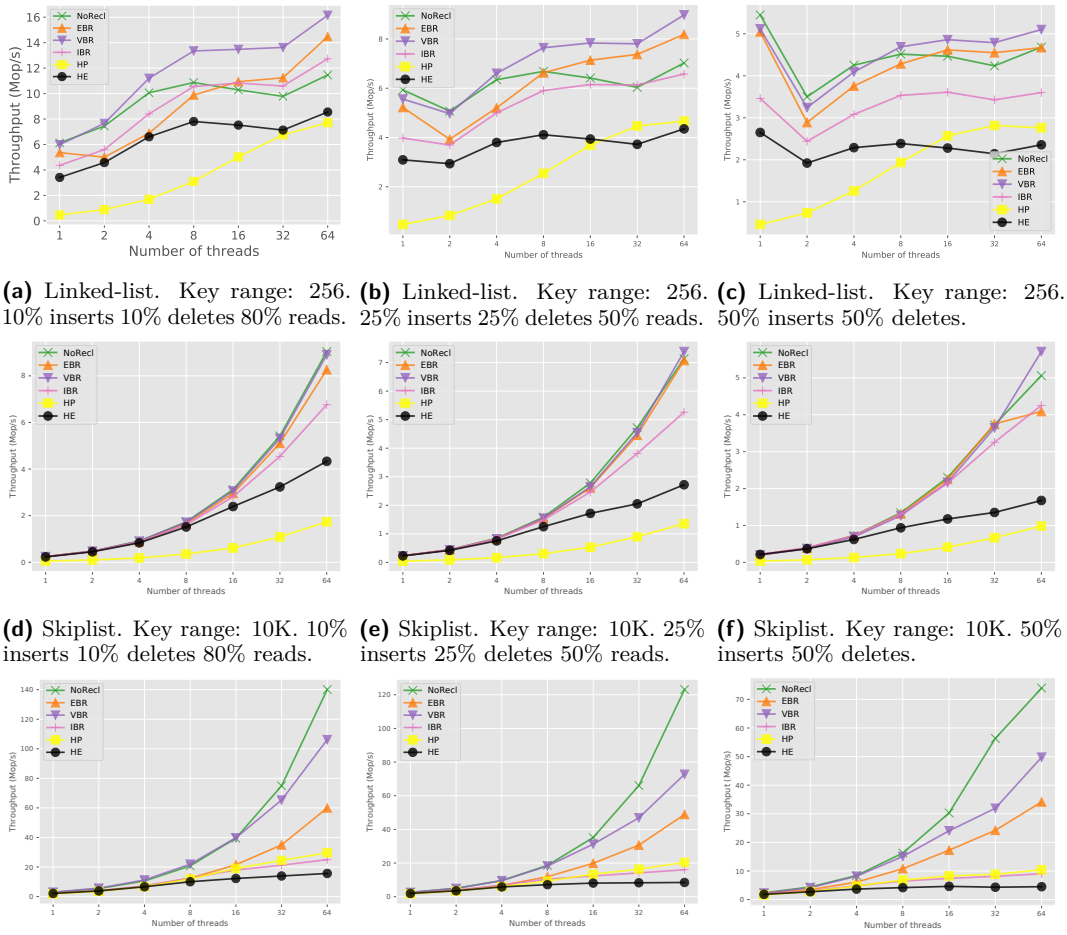
### 4.2.3 Update Methods

By Assumption 1, all data structure updates are executed using CAS instructions. We consider two types of pointer updates. The first type, depicted in lines 30-33 of Figure 1, is the update of an unmarked pointer. The second type (lines 34-39) is the marking of an unmarked pointer. The update of other mutable fields can be similarly implemented. In particular, the version of non-pointer mutable fields should always be equal to the node's birth epoch (which makes such fields much easier to handle).

The *update()* method replaces the original pointer update via a single CAS instruction. It receives pointers to the target node, its expected successor and the new successor, together with their respective birth epochs. All three pointers are assumed to be unmarked. The expected and new pointer versions are calculated in the same manner (lines 31-32): the maximum birth epoch of the target node and successor node. The next field is either

successfully updated or remains unchanged in line 33. In Appendix A we prove that the target node's next pointer is updated iff (1) it has not been reclaimed yet, (2) it is not marked, and (3) it indeed points to the expected node (including the given birth epoch).

The *mark()* method marks an unmarked *next* pointer, without changing its pointed node. It receives the target node and its birth epoch. The actual marking is executed in line 39. It uses the unmarked and marked variants of the pointer, and does not change the pointer's version (calculated in line 36). In Appendix A we prove that the target node is marked iff (1) it has not been reclaimed yet, (2) it is not marked, and (3) it indeed points to the expected node (read in line 35), just before the marking.




**(a)** Linked-list. Key range: 256. 10% inserts 10% deletes 80% reads. **(b)** Linked-list. Key range: 256. 25% inserts 25% deletes 50% reads. **(c)** Linked-list. Key range: 256. 50% inserts 50% deletes.

**(d)** Skiplist. Key range: 10K. 10% inserts 10% deletes 80% reads. **(e)** Skiplist. Key range: 10K. 25% inserts 25% deletes 50% reads. **(f)** Skiplist. Key range: 10K. 50% inserts 50% deletes.

**(g)** Hash table. Key range: 10M. 10% inserts 10% deletes 80% reads. **(h)** Hash table. Key range: 10M. 25% inserts 25% deletes 50% reads. **(i)** Hash table. Key range: 10M. 50% inserts 50% deletes.

**Figure 2** Throughput evaluation. Y axis: throughput in million operations per second. X axis: #threads.

## 5 Evaluation

For evaluating throughput of VBR we implemented lock-free variants of a linked-list from [34], a hash table (implemented using the same list), and a skip list. We implemented Herlihy and Shavit's lock-free skiplist [27] with the amendment suggested in [20] for lock-free reclamation.

VBR was integrated into all data-structures according to the guidelines presented in Section 4.

We evaluated VBR against a baseline execution in which memory is never reclaimed (denoted NoRecl), an optimized implementation of the epoch-based reclamation method [22] (denoted EBR), the traditional hazard pointers scheme [37] (denoted HP), the hazard eras scheme [45] (denoted HE), and the 2GEIBR variant of the interval-based scheme [55] (denoted IBR). For all reclamation schemes, we implemented optimized local allocation pools. Objects were reclaimed once the retire list is full, and were allocated from the shared pool if there were no objects available in the local pool. As retired objects cannot be automatically reclaimed in EBR, IBR, HE and HP, we tuned their retire list sizes in order to achieve high performance. We further tuned the global epoch update rate in EBR, HE and IBR (in VBR it seldom happens and does not require any tuning).

Pointer-based methods require that it would not be possible to reach a reclaimed node by traversing the data structure from a protected node, even if the protected node has been unlinked and retired. This prevents schemes like HP, HE, IBR, etc. from being used with some data structures such as Harris's original linked-list [22] or the lock-free binary tree of [11, 40]. We did not implement binary search trees, because some of the measured competing schemes cannot support it.

## 5.1 Setup

Our experimental evaluation was performed on an Ubuntu 14.04 (kernel version 4.15.0) OS. The machine featured 4 AMD Opteron(TM) 6376 2.3GHz processors, each with 16 cores (64 threads overall). The machine used 128GB RAM, an L1 cache of 16KB per core, an L2 cache of 2MB for every two cores and an L3 cache of 6MB per processor. The code was compiled using the GCC compiler version 7.5.0 with the -O3 optimization flag.

We implemented object pools in a similar way to [13], to avoid returning reclaimed objects to the OS. All schemes used that implementation, in which all pools are pre-allocated before the test. Each test was a fixed-time micro benchmark in which threads randomly call the *Insert()*, *Delete()* and *Search()* operations according to three workload profiles: (1) a search-intensive workload (80% searches, 10% inserts and 10% deletes), (2) a balanced workload (50% searches, 25% inserts and 25% deletes), and (3) an update-intensive workload (50% inserts and 50% deletes). Each execution started by filling the data-structure to half of its range size. For the hash-table, the load factor was 1. We measured the throughput of the above schemes. Each experiment lasted 1 second (as longer executions showed similar results) and was run with a varying number of executing threads. Each experiment was executed 10 times, and the average throughput across all executions was calculated.

## 5.2 Discussion

Figure 2 shows that VBR is faster than other manual reclamation schemes, even when contention is high (Figures 2a-2c), and in update-intensive workloads (Figures 2c, 2f, 2i). VBR outperforms epoch-based competitors (EBR, IBR and HE) due to its infrequent epoch updates. In order to avoid allocation bottlenecks, EBR, IBR and HE require frequent epoch updates. I.e., many global epoch accesses result in cache misses and slow down the allocation process, the reclamation process and the operations executions. In contrast to these methods, VBR requires infrequent epoch updates. An increment is triggered when the next node to be allocated has a retire epoch equal to the current global epoch. Most global epoch accesses during VBR hit the cache, and are negligible in terms of performance. In addition, VBR outperforms its pointer-based competitors (IBR, HE and HP) since it

requires neither read nor write fences. Specifically, in the hash table implementation, it surpasses the next best algorithm, EBR, by up to 60% in the search-intensive workload (Figure 2g), by up to 50% in the balanced workload (Figure 2h), and by up to 40% in the update-intensive workload (Figure 2i). In the skiplist implementation, VBR is comparable to the baseline and EBR for the search-intensive and balanced workloads (Figures 2d-2e). For the update-intensive workload, it outperforms the next best algorithm, IBR, by up to 35% (Figure 2f). In the linked-list implementation, VBR outperforms the next best algorithm, EBR, by up to 10%, 11% and 8%, respectively (Figures 2a-2c). For cache locality reasons, VBR outperforms the baseline execution for all linked-list and skiplist workloads and for all key ranges (Figures 2a-2f). As cache locality plays no role in the hash table implementation, VBR has no advantage against the baseline for this data-structure. VBR's throughput is around 75% of the baseline for the search-intensive workload, around 60% of the baseline for the balanced workload and around 65% of the baseline for the update-intensive workload.

## 6 Related Work

Much related work was already discussed in the introduction or in the evaluation section (Section 5). There are many memory management schemes, and in the evaluation we compared VBR against highly efficient schemes whose code is available (We could not compare against all). Previous works [31,48] defined a set of desirable reclamation properties. Safe reclamation algorithms should be fast (show low latency and high throughput), robust (the number of unreclaimed objects should be bounded), widely applicable and self-contained (not relying on external features).

Two novel methods initiated the study of memory reclamation for concurrent data structures. Pointer-based schemes [17, 26, 37] protect objects that are currently accessed by placing a hazard pointer referencing them. These methods are often slow and not always applicable. Epoch-based schemes [10, 22] (and quiescent state based schemes [23]) wait until all threads move to the next operation to make sure that an unlinked node cannot be further accessed. Such methods are sometimes not robust, and most hybrids of the two approaches [5, 10, 41, 45, 55] are either not always applicable, or rely on special hardware support. Drop-the-anchor [9] extends HP by protecting only some of the traversed nodes and reclaiming carefully, yet it is not easily applicable and has only been applied to linked-lists.

Another approach, which is neither fast nor robust, is reference counting based reclamation [8, 16, 21, 26]. This scheme keeps an explicitly count of the number of pointers to each object, and reclaims an object with a zero count. Such schemes require a way to break cyclic structures of retired objects, and are often slow or rely on hardware assumptions. This scheme has a wait-free (and in particular, robust) variant [50] and a lock-free variant [15], but they are not fast, since they require multiple expensive synchronization fences.

Many schemes rely on Hardware-specific or OS features, or affect the execution environment. ThreadScan [3], StackTrack [2], and Dragojević et al. [18] rely on transactional memory for the reclamation, which is not always available in hardware or may be slow in a software implementation. DEBRA+ [10] and NBR [48] use OS signals in order to wake unresponsive threads and allow lock-free progress even for EBR-based methods, if the OS signal implementration is lock-free. Morrison and Afek [39] avoid memory fences by waiting for a short while. This relies on specific hardware properties that might not always be available. Dice et. al. [17] and PEBR [31] avoid costly fences by relying on the existence of process-wide memory fences. QSense [5] requires control of the OS scheduler. In particular, to make hazard pointers visible, threads are periodically swapped out.

## References


1 std::memory_order. https://en.cppreference.com/w/cpp/atomic/memory_order. Accessed: 2021-04-19.

2 Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

3 Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. Threadscan: Automatic and scalable memory reclamation. *ACM Transactions on Parallel Computing (TOPC)*, 4(4):1–18, 2018.

4 Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. *ACM SIGPLAN Notices*, 53(1):14–27, 2018.

5 Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. Fast and robust memory reclamation for concurrent data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 349–359, 2016.

6 Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–264, 2019.

7 Guy E Blelloch, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. The parallel persistent memory model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 247–258, 2018.

8 Guy E Blelloch and Yuanhao Wei. Concurrent reference counting and resource management in wait-free constant time. *arXiv preprint arXiv:2002.07053*, 2020.

9 Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the anchor: lightweight memory management for non-blocking data structures. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 33–42, 2013.

10 Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 261–270, 2015.

11 Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using rcu balanced trees. *ACM SIGPLAN Notices*, 47(4):199–210, 2012.

12 Nachshon Cohen. Every data structure deserves lock-free memory reclamation. *Proc. ACM Program. Lang.*, 2(OOPSLA):143:1–143:24, 2018. doi:10.1145/3276513.

13 Nachshon Cohen and Erez Petrank. Automatic memory reclamation for lock-free data structures. *ACM SIGPLAN Notices*, 50(10):260–279, 2015.

14 Nachshon Cohen and Erez Petrank. Efficient memory management for lock-free data structures with optimistic access. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 254–263, 2015.

15 Andreia Correia, Pedro Ramalhete, and Pascal Felber. Orcgc: automatic lock-free memory reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 205–218, 2021.

16 David L Detlefs, Paul A Martin, Mark Moir, and Guy L Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

17 Dave Dice, Maurice Herlihy, and Alex Kogan. Fast non-intrusive memory reclamation for highly-concurrent data structures. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, pages 36–45, 2016.

18 Aleksandar Dragojević, Maurice Herlihy, Yossi Lev, and Mark Moir. On the power of hardware transactional memory to simplify memory management. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 99–108, 2011.

19 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 131–140, 2010.

**20** Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

**21** Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems*, 20(8):1173–1187, 2008.

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

**23** Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

**24** Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

**25** Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

**26** Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems (TOCS)*, 23(2):146–196, 2005.

**27** Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The art of multiprocessor programming*. Newnes, 2020.

**28** Maurice P Herlihy and J Eliot B Moss. Lock-free garbage collection for multiprocessors. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 229–236, 1991.

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

**30** Richard L Hudson and J Eliot B Moss. Sapphire: Copying gc without stopping the world. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 48–57, 2001.

**31** Jeehoon Kang and Jaehwang Jung. A marriage of pointer-and epoch-based reclamation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 314–328, 2020.

**32** Jonatan Lindén and Bengt Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *International Conference On Principles Of Distributed Systems*, pages 206–220. Springer, 2013.

**33** Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, volume 509518, 1998.

**34** Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82, 2002.

**35** Maged M Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 21–30, 2002.

**36** Maged M Michael. Aba prevention using single-word instructions. *IBM Research Division, RC23089 (W0401-136), Tech. Rep*, 2004.

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

**38** Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996.

**39** Adam Morrison and Yehuda Afek. Temporally bounding tso for fence-free asymmetric synchronization. *ACM SIGARCH Computer Architecture News*, 43(1):45–58, 2015.

40 Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 317–328, 2014.

41 Ruslan Nikolaev and Binoy Ravindran. Universal wait-free memory reclamation. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 130–143, 2020.

42 Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *Proceedings of the 6th international symposium on Memory management*, pages 159–172, 2007.

43 Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. *ACM SIGPLAN Notices*, 43(6):33–44, 2008.

44 Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L Hosking, Ethan Blanton, and Jan Vitek. Schism: fragmentation-tolerant real-time garbage collection. *ACM Sigplan Notices*, 45(6):146–159, 2010.

45 Pedro Ramalhete and Andreia Correia. Brief announcement: Hazard eras-non-blocking memory reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 367–369, 2017.

46 Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM (JACM)*, 53(3):379–405, 2006.

47 Gali Sheffi, Guy Golan-Gueta, and Erez Petrank. A scalable linearizable multi-index table. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 200–211. IEEE, 2018.

48 Ajay Singh, Trevor Brown, and Ali Mashtizadeh. Nbr: neutralization based reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–190, 2021.

49 Daniel Solomon and Adam Morrison. Efficiently reclaiming memory in concurrent search data structures while bounding wasted memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 191–204, 2021.

50 Håkan Sundell. Wait-free reference counting and memory management. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 10–pp. IEEE, 2005.

51 Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *International Conference On Principles Of Distributed Systems*, pages 330–344. Springer, 2012.

52 Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. *ACM SIGPLAN Notices*, 49(8):357–368, 2014.

53 R Kent Treiber. *Systems programming: Coping with parallelism.* International Business Machines Incorporated, Thomas J. Watson Research . . . , 1986.

54 Yuanhao Wei, Naama Ben-David, Guy E Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 31–46, 2021.

55 Haosen Wen, Joseph Izraelevitz, Wentao Cai, H Alan Beadle, and Michael L Scott. Interval-based memory reclamation. *ACM SIGPLAN Notices*, 53(1):1–13, 2018.

56 Pavel Yosifovich, David A Solomon, and Alex Ionescu. *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more.* Microsoft Press, 2017.


## A Correctness

Safe memory reclamation often requires that before reclaiming a retired node, the reclaimer ensures that no other threads hold local pointers to this retired node, as they may be later dereferenced. However, VBR does not return freed space to the operating system, and this requirement can be relaxed. In this section we prove Theorem 1. I.e., we prove that VBR

maintains the original program's linearizability and lock-freedom guarantee. In Section A.1, we prove that linearizability is maintained (see Lemma 3), and in Section A.2, we prove that lock-freedom is maintained as well (see Lemma 20).

Our correctness proof relates to applying the interface introduced in Section 4. Consequently, it applies to specific data-structures. However, it can be easily shown that the same invariants hold for every lock-free linearizable data-structure implementation, for which our assumptions hold (see Section 3.3).

## A.1 VBR Maintains Linearizability

In this section we prove Lemma 3:

▶ **Lemma 3.** *Given a lock-free linearizable data-structure implementation, that satisfy all of the assumptions presented in Section 3.3, the implementation remains linearizable after integrating it with VBR, according to the modifications described in Sections 4.1-4.2.*

Our motivation is that linearizability is a *local* property [29]. In our context, this means the following: Given a linearizable data-structure implementation (i.e., all possible executions are represented by linearizable respective histories), if each atomic instruction is replaced by its linearizable implementation, then the overall data-structure implementation remains linearizable.

Using this concept, in Section A.1.1 we prove some basic claims regarding the effect of each method from Figure 1. We do not set actual linearization points, since methods that terminate by a rollback step actually do not take effect at all. In Section A.1.2 we show that integrating a linearizable reclamation-free implementation with checkpoints and rollback steps maintains linearizability. Using the claims proved in Sections A.1.1-A.1.2, in Sections A.1.3 we prove that every VBR-integrated execution has the same history as some linearizable reclamation-free execution, by constructing the respective execution inductively. As every VBR-integrated execution has a linearizable history, Lemma 3 derives.

### A.1.1 Basic Linearizability-Related Claims

Before diving into our linearizability proof, note that the *getKey()* method gives rise to a certain issue. In a similar way to Assumption 4, if a thread $T$ installs a pointer to a node $n$, then $T$ cannot read $n$'s immutable fields after executing a rollback-unsafe step. However, when dealing with immutable fields, as opposed to Assumption 4, we do not assume that the original reclamation-free implementation satisfies this assumption. Instead, we treat immutable fields in the following manner: after $T$ installs a local pointer to $n$, and before its next attempt to execute a rollback-unsafe update, it must call the *getKey()* method (or the respective read of another immutable field) and save its output along with its local pointer to $n$ (unless $n$'s key has already been read into one of $T$'s local variables). Future reads of $n$'s key are replaced with reads of the saved value.

We first prove that our allocation mechanism is safe. I.e., a node is never allocated from a memory address before the previous node, allocated from the same address, is retired. In order to show that our allocation mechanism is safe, we prove Claim 6. We first make some basic observations:

▶ **Observation 4.** *The global epoch e never decreases.*

▶ **Observation 5.** *The node's birth and retire epochs only change during allocation and retirement.*

▷ Claim 6. Let $E$ be a VBR-integrated execution, and suppose that two different *alloc()* calls return nodes $n_1, n_2$, allocated from the same memory address. W.l.o.g., assume that $n_1$ is returned first. Then:

1. If $n_1$ never becomes reachable, then it is appended to an allocation list before $n_2$ is allocated.
2. If $n_1$ ever becomes reachable, then $n_1$ is retired exactly once.
3. If $n_1$ ever becomes reachable, then $n_1$'s retirement completes before $n_2$ is allocated.
4. $n_1$'s birth and retire epochs are smaller than $n_2$'s birth epoch.

Proof. W.l.o.g., let's assume that $n_2$ is the first node allocated from the memory address $n_1$ has been previously allocated from. By transitivity, the claim holds for any future allocation from this address.

1. Assume that $n_1$ never becomes reachable. By Assumption 3, it is never retired and therefore, is never appended to a retired nodes list. Since $n_2$ is allocated from the same address, $n_1$ must have been re-appended to an allocation list (for more details, see Appendix B), before $n_2$ is allocated.
2. Since the rolling-back mechanism may result in calling the *retire()* method more than once per node, extra precautions are added. After a new node is allocated, its retire epoch is initialized to $\perp$ (see line 8). A node is retired (i.e., added to a thread's retired nodes list in line 15) only if its retire epoch contains $\perp$ and its birth epoch is the one received as input (otherwise, the *retire()* call returns in line 13), and it does not contain $\perp$ after being retired (see line 14). In addition, according to Assumption 3, a single thread is in charge of retiring every node. Notice that, as can be shown inductively, $n_1$ cannot be accidentally retired by a thread trying to retire a node previously allocated from the same memory address. Therefore, $n_1$ is retired at most once.
3. When $n_2$ is allocated, it is popped out of the thread's allocation list (line 2). Therefore, the previous node, allocated from the same memory address (i.e., $n_1$) must have been previously retired and added to this allocation list.
4. If $n_1$ is never retired, then it is appended to an allocation list (for more details, see Appendix B), right before a rollback step. Since its birth epoch is set before this rollback step, by Observation 5, its birth epoch is at most the epoch value, as saved by the executing thread. Since this value is updated after rolling-back (see Section 4.2.1), $n_2$, allocated by the same thread, has a bigger birth epoch. Since $n_1$'s retire epoch is $\perp$ in this case, $n_2$'s birth epoch is bigger than $n_1$'s retire epoch as well.
   Otherwise, if $n_1$ is retired, then $n_1$'s retire epoch is set during its retirement (in line 14), which completes before $n_2$'s allocation. When $n_2$ is allocated, if $n_1$'s retire epoch equals the thread's copy of the global epoch, then $n_2$ is not allocated (see lines 3- 6). Since $n_2$'s birth epoch is set to the global epoch only after making sure that it is no longer equal to $n_1$'s retire epoch, $n_2$'s birth epoch is strictly bigger than $n_1$'s retire epoch (and by transitivity, bigger than its birth epoch).

There is one extra case that still needs to be handled. Suppose that $n_1$ is retired right before the rollback step (see Appendix B). In this case, like in a standard retirement, if $n_1$'s retire epoch is $\perp$, then the retiring thread sets its retire epoch to be the current global epoch and appends it to its retired nodes list. Therefore, this case falls back to the standard case, and is already handled above.

◁

Given Claim 6, we are guaranteed that the life periods of any two nodes, allocated from the same memory address, do not overlap. Next, we are going to prove some claims for each method from Figure 1. This claims will later be used in Section A.1.3, when showing that any VBR-integrated execution has the same history as a linearizable reclamation-free one.

**The *isMarked()* method** Our *isMarked()* method always returns a correct answer, as it is not affected by global epoch changes, and never performs rollback steps. This concept is captured in Claim 7, that can actually be used as a linearization proof for this method.

▷ Claim 7. Assume that the input parameters to an *isMarked()* call by a thread T are a pointer to a node $n$, allocated from a memory address $a$, and its birth epoch $b$. Then:

1. If the method returns TRUE in line 28, then $n$ is already marked at this point.
2. If the method returns in line 29, then it returns TRUE iff $n$ is marked when its *next* pointer is read in line 27.

Proof. Recall that $b$ is $n$'s birth epoch, as originally read by T. If the method returns in line 28, then by Observation 5, $n$ is already retired when the birth epoch is read in line 28. In particular, by Assumption 3, $n$ is considered as marked (i.e., invalid) at this point, and the method indeed returns TRUE.

Otherwise, if the method returns in line 29, then by Claim 6, $n$ is not reclaimed before executing line 28. Therefore, the method returns TRUE iff the *isMarked()* call in line 27 returns TRUE. I.e., it returns TRUE iff the *next* field, read in line 27, is marked, and the claim holds. ◁

Recall that part 4 of Assumption 3 holds for every mutable field. Therefore, although Claim 7 relates to the *next* field specifically, it holds for any mutable node field.

**The *getKey()* method** Our *getKey()* method may result in a rollback step to the last checkpoint. Therefore, we only prove that it returns the correct answer when it returns in line 25. Recall that this method is always called before the next checkpoint installation. Therefore, it is safe to assume that the executing thread does not update its local copy of the global epoch between installing the pointer and calling the *getKey()* method.

▷ Claim 8. Assume that the input parameter to a *getKey()* call by a thread T is a pointer to a node $n$. Then the key returned in line 25 is equal to $n$'s key.

Proof. As described in Section 4.2.2, this method is called after T installs a pointer to $n$, and before installing the next checkpoint. Recall that the global epoch is always read in line 20, after installing pointers, and if it is different than the local copy, then a rollback step is performed. Therefore, In particular, it is guaranteed that T's local copy of the global epoch counter has not updated between installing the local pointer to $n$ and calling the *getKey()* method. Since the method does not return in line 24, it is guaranteed that the global epoch has not changed until executing line 24 as well. By Claim 6, $n$ was not retired before T executed line 24. Therefore, it was not retired when reading $n$'s key in line 23, and the claim holds. ◁

Recall that we treat any immutable field in the same way. I.e., threads must read it before installing the next checkpoint. Therefore, although Claim 8 relates to the *getKey()* method specifically, it holds for any read of an immutable node field.

**The *getNext()* method** We now turn to handle the *getNext()* method. Our *getNext()* method may result in a rollback step to the last checkpoint. Therefore, we only prove that it returns the correct answer when it returns in line 21, and when the output is indeed used by the calling thread. For more details, see Assumption 4.

▷ Claim 9. Assume that the input parameter to a *getNext()* call by a thread T is a pointer to a node $n$, allocated from a memory address $a$. Then either (1) the output from the *getNext()* call is not used by T, or (2) the output is a pointer to a node $m$, which is $n$'s successor when line 18 is executed, together with $m$'s birth epoch. In particular, $m$ is not yet reclaimed when line 18 is executed.

Proof. Assume that so far, all *getNext()* calls satisfy the claim conditions. Let $e_1$ be the global epoch value, right after T installs a local pointer to $n$. It is guaranteed that T's local copy of the global epoch is equal to $e_1$ at this point. Otherwise, it must perform a rollback step and discard its pointer to $n$, whether in line 6 or 20.

If the global epoch is no longer $e_1$ when T executes line 18, then since the method does not return in line 20, T must have already updated its local copy of the global epoch since installing the local pointer to $n$. T can only update its local copy of the global epoch after a checkpoint rollback. Therefore, T must have performed a rollback-unsafe step since installing the local pointer to $n$. By Assumption 4, there are two possibilities in this case:

1. $n$ is already marked at this point, and by Claim 7, a call to the *isMarked()* method returns the correct answer and T does not use the *getNext()* output.
2. $n$ is not yet marked. By assumption 3, it is still reachable. Thus, $m$ is also reachable when line 18 is executed. In particular, it is not yet retired. Since the global epoch does not change between the executions of lines 18 and 20, by Claim 6, $m$ is guaranteed to not be reclaimed before the execution of line 19. Therefore, by Observation 5, the birth epoch read in line 19 is indeed $m$'s birth epoch.

It still remains to handle the case in which the global epoch is still $e_1$ when T executes line 18. By Claim 6, $n$ is not yet reclaimed at this point. Since the method does not return in line 20, the global epoch is also $e_1$ when T executes line 19. By Claim 6, $n$'s successor cannot be reclaimed between the execution of line 18 and the execution of line 19. By Observation 5, its birth epoch is indeed the birth epoch read in line 19, which the method eventually returns. ◁

Recall that Assumption 2, Assumption 4 and Claim 7 hold for every mutable field. Therefore, although Claim 9 relates to the *getNext()* method specifically, it holds for any read of a mutable node field. Before handling our two update methods, we prove the following claim regarding our versioning mechanism.

▷ Claim 10. Let $n_1, n_2$ be two nodes, let $b_1, b_2$ be their birth epochs, and let $a_1, a_2$ be the memory addresses they were allocated from, respectively. In addition, assume that $n_1$'s *next* field points to $n_2$, with a version $v$. Then:

1. $v$ is the maximum between $b_1$ and $b_2$.
2. If either $n_1$ or $n_2$ are retired, then their retire epochs are at least $v$.

Proof. Assume by contradiction that at some point during the execution, the claim does not hold for the first time. Then it must happen upon an update of a node's *next* pointer. A node's *next* pointer is only updated in line 9, 33 and 39. In line 9, the node's pointer is set to NULL and therefore, the claim still vacuously holds.

Assume that the update is executed in line 33. Let $n_1, b_1, n_2, b_2, n_3, b_3$ be the *update()* method input parameters, respectively. In addition, for every $i \in \{1, 2, 3\}$, let $a_i$ be the memory address $n_i$ is allocated from. By the claim assumption, if $n_1$ points to $n_2$, then the pointer's version is the maximum between $b_1$ and $b_2$, as calculated in line 31. If $n_1$ is already retired when line 33 is executed, then by Claim 6, the birth epoch of the node, allocated from $a_1$ when the WCAS is executed in line 33, is bigger than $n_1$'s retire epoch. By the claim assumption, it is bigger then the maximum between $b_1$ and $b_2$. Since the WCAS is successful, the version calculated in line 31 is indeed the right version. I.e., $n_1$ is not yet retired when the WCAS is executed. By Observation 5, its birth epoch is still $b_1$ at this point. Moreover, as mentioned in Section 4.2.3, the WCAS success implies that $n_1$ is not marked, and by Assumption 3, it is reachable. Since $n_3$ is also reachable after the successful WCAS, by Assumption 3, it cannot be retired at this point. Therefore, by Observation 5, its birth epoch is still $b_3$. Therefore:

1. After the successful WCAS, $n_1$ points to $n_3$ with a version that is the maximum between $b_1$ and $b_3$, as calculated in line 32.
2. Since both $n_1$ and $n_3$ are not yet retired, by Observation 4, their retire epochs are going to be at least the current epoch, which is at least the maximum between $b_1$ and $b_3$.

Therefore, the claim necessarily does not hold for the first time, after an execution of line 39. Let $n_1$ and $b_1$ be the two input parameters to the *mark()* call, let $n_2$ be the node read from $n_1$'s *next* pointer in line 35, and let $b_2$ be its birth epoch, read in line 36. Since the method does not return in line 37, by Observation 5, $n_1$'s birth epoch is necessarily still $b_1$ when line 35 is executed. By the claim assumption, $n_1$ points to $n_2$ with a version which is the maximum between $b_1$ and $b_2$, as calculated in line 36.

Since the WCAS is successful, the version calculated in line 36 is indeed the right version. I.e., $n_1$ is not yet retired when the WCAS is executed. By Observation 5, its birth epoch is still $b_1$ at this point. Moreover, the WCAS success implies that $n_1$ is not marked (as the expected value, calculated in line 35, is not marked), and by Assumption 3, it is reachable. Since $n_2$ is also reachable after the successful WCAS, by Assumption 3, it cannot be retired at this point. Therefore, by Observation 5, its birth epoch is still $b_2$. Therefore:

1. After the successful WCAS, $n_1$ points to $n_2$ with a version that is the maximum between $b_1$ and $b_2$, as calculated in line 36.
2. Since both $n_1$ and $n_2$ are not yet retired, by Observation 4, their retire epochs are going to be at least the current epoch, which is at least the maximum between $b_1$ and $b_2$.

Since both claim assumptions still hold, we derive a contradiction. I.e., the claim assumptions always hold. ◁

### The *update()* method

▷ Claim 11. Assume that the input parameters to an *update()* call by a thread T are pointers to the nodes $n_1, n_2, n_3$, allocated from the memory addresses $a_1, a_2, a_3$, and their birth epochs, $b_1, b_2, b_3$, respectively. The WCAS executed in line 33 is successful iff $n_1$'s next pointer points to $n_2$ with an unmarked pointer, right before executing the CAS.

Proof. Recall that all input parameters are assumed to be unmarked. First, assume that the WCAS is successful. Then a node allocated from $a_1$ points to a node allocated from $a_2$, with a version which is the maximum between $b_1$ and $b_2$, as calculated in line 31. Assume by contradiction that either $n_1$ or $n_2$ are retired at this stage. By Claim 10, the retire epoch of each one of them is at least the value calculated in line 31. By Claim 6, the birth epoch

of a new node, allocated either from $a_1$ or $a_2$, must be strictly bigger than this value. By Claim 10, in this case, the pointer version must be bigger then the value calculated in line 31 – a contradiction. Therefore, if the WCAS is successful then $n_1$'s next pointer points to $n_2$ right before the linearization point.

Now, assume that the WCAS is unsuccessful, and assume by contradiction that $n_1$ points to $n_2$ with an unmarked pointer. By Claim 10, the pointer's version must be the one calculated in line 31 – a contradiction to the WCAS failure, and the claim follows. ◁

▷ Claim 12. Assume that the input parameters to an *update()* call by a thread T are pointers to the nodes $n_1, n_2, n_3$, allocated from the memory addresses $a_1, a_2, a_3$, and their birth epochs, $b_1, b_2, b_3$, respectively. If the WCAS executed in line 33 is successful, then after the CAS, $n_1$'s next pointer points to $n_3$ with an unmarked pointer.

Proof. Assume that the WCAS executed in line 33 is successful. Recall that all input node pointers are assumed to be received as unmarked pointers. Since the WCAS is successful, after it is executed, a node allocated from $a_1$ points (with an unmarked node) to a node allocated from $a_3$, with a version which is the maximum between $b_1$ and $b_3$, as calculated in line 32. Assume by contradiction that either $n_1$ or $n_3$ are retired at this stage. By Claim 10, the retire epoch of each one of them is at least the value calculated in line 32. By Claim 6, the birth epoch of a new node, allocated either from $a_1$ or $a_3$, must be strictly bigger than this value. By Claim 10, in this case, the pointer version must be bigger then the value calculated in line 32 – a contradiction. Therefore, if the WCAS is successful, then $n_1$'s next pointer points to $n_3$ right after the linearization point. ◁

Recall that Assumption 2 holds for every mutable field. Moreover, the version of non-pointer mutable fields should always be equal to the node's birth epoch. Therefore, Claims 11 and 12 can be easily adjusted to fit other mutable fields.

**The *mark()* method** Let $n_1, b_1$ be the two input parameters to a *mark()* call, executed by a thread T. I.e., $b_1$ is $n_1$'s birth epoch, as previously saved by T. Let $n_2$ be the node $n_1$ points to when its *next* pointer is read in line 35, and let $b_2$ be the birth epoch, read in line 36. In addition, let $a_1, a_2$ be the addresses $n_1$ and $n_2$ are allocated from, respectively. If the method returns in line 37, then its linearization point is set to be the read of the birth epoch in line 37. Otherwise, if the method returns in line 39, then its linearization point is set to be the WCAS execution in line 39. We prove that the method indeed takes effect in its linearization point using the following claims:

▷ Claim 13. Assume that the input parameters to a *mark()* call by a thread T are a pointer to a node $n_1$, allocated from the memory addresses $a_1$, and its birth epoch, $b_1$. If the method returns FALSE in line 37 then $n_1$ is already marked when this line is executed.

Proof. By Observation 5, $n_1$ is already retired when the birth epoch is read in line 37. In particular, by Assumption 3, $n_1$ is considered as marked (i.e., invalid) at this point. ◁

▷ Claim 14. Assume that the input parameters to a *mark()* call by a thread T are a pointer to a node $n_1$, allocated from the memory addresses $a_1$, and its birth epoch, $b_1$. Let $n_2$ be $n_1$'s successor when $n_1$'s *next* pointer is read in line 35. The WCAS executed in line 39 is successful iff prior to its execution, $n_1$ points to $n_2$ with an unmarked pointer.

Proof. Assume that the WCAS executed in line 39 is successful. Since the expected pointer is unmarked (as calculated in line 35), it remains to show that $n_1$ indeed points to $n_2$. Assume

by contradiction that either $n_1$ or $n_2$ are retired at this stage. By Claim 10, the retire epoch of each one of them is at least the value calculated in line 36. By Claim 6, the birth epoch of a new node, allocated either from $a_1$ or $a_2$, must be strictly bigger than this value. By Claim 10, in this case, the pointer version must be bigger then the value calculated in line 36 – a contradiction. Therefore, if the WCAS is successful then $n_1$'s next pointer points to $n_2$ right before to the WCAS.

Now, assume that the WCAS executed in line 39 is unsuccessful. If either $n_1$'s *next* pointer is marked, or the node allocated from $a_1$ does not point to the node allocated from $a_2$ at this stage, then we are done. Otherwise, assume by contradiction that $n_1$ points to $n_2$ with an unmarked pointer. By Claim 10, the pointer's version must be the one calculated in line 36 – a contradiction to the WCAS failure, and the claim follows. ◁

▷ Claim 15. Assume that the input parameters to a *mark()* call by a thread T are a pointer to a node $n_1$, allocated from the memory addresses $a_1$, and its birth epoch, $b_1$. Let $n_2$ be $n_1$'s successor when $n_1$'s *next* pointer is read in line 35. If the WCAS executed in line 39 is successful, then $n_1$ points to $n_2$ via a marked pointer, right after the WCAS execution.

Proof. If the WCAS is successful, then the updated pointer is indeed marked (as the new pointer value is marked in line 38). Since the WCAS is successful, after it is executed, a node allocated from $a_1$ points to a node allocated from $a_2$, with a version which is the maximum between $b_1$ and $b_2$, as calculated in line 36. Assume by contradiction that either $n_1$ or $n_2$ are retired at this stage. By Claim 10, the retire epoch of each one of them is at least the value calculated in line 36. By Claim 6, the birth epoch of a new node, allocated either from $a_1$ or $a_2$, must be strictly bigger than this value. By Claim 10, in this case, the pointer version must be bigger than the value calculated in line 36 – a contradiction. Therefore, if the WCAS is successful, then $n_1$'s next pointer points to $n_2$ right after the linearization point. ◁

Recall that Assumption 2 hold for every mutable field. Moreover, the version of non-pointer mutable fields should always be equal to the node's birth epoch. Therefore, Claims 13, 14 and 15 can be easily adjusted to fit other mutable fields.

### A.1.2 Inserting Checkpoints into Reclamation-Free Implementations

We first show that, given any (and in particular, a reclamation-free) execution, checkpoint rollbacks preserve its set of possible history extensions. Recall that rollback-safe steps are defined in Definition 2, and may be either local steps, shared-memory reads or shared memory writes. In addition, given a linearizable implementation, installing checkpoints is well-defined in Section 4.2.1. After a thread installs a checkpoint, a rollback to this checkpoint includes restoring the local variables, saved upon installing this checkpoint. Intuitively, by Definition 2, threads execute only rollback-safe steps between checkpoints and thus, a rollback to the last checkpoint is always safe. We prove this notion in the following claim:

▷ Claim 16. Assume that $s_j$ is a rollback-safe step, executed by a thread T. Let $0 < i < j$, and assume that for every $i < t < j$, if $s_t$ is a step executed by T, then $s_t$ is also a rollback-safe step. Let $E'$ be the execution obtained by removing from $E_j$ all steps $s_t$ (for every $i < t \leq j$), executed by T during $E_j$. Then EXT($E'$)=EXT($E_j$).

Proof. We are going to prove the claim by induction on $j - i$. For the base case, the claim holds by Definition 2. For the induction step, assume that the claim holds for $j - 1$. Let $E''$ be the execution obtained by removing all steps $s_t$ (for every $i < t \leq j - 1$), executed by T during $E_{j-1}$. Then $E'' = E'$, and by the induction hypothesis, $EXT(E'') = EXT(E_{j-1})$.

I.e., $EXT(E') = EXT(E_{j-1})$, and by Definition 2, $EXT(E') = EXT(E_j)$, and the claim holds. ◁

Our next goal is to show equivalence between any given linearizable implementation and the implementation obtained after inserting checkpoints and rollback instructions into the given code. Let $E$ be a linearizable execution, and assume that $s_j$ is a rollback step by a thread T. Since checkpoints are installed upon every operation invocation, there exists at least one step $s_i$ (for some $i < j$) which is T's last checkpoint visit. W.l.o.g., let $i$ be the maximal such index before $j$. Then by Claim 16, executing $s_j$ maintains the set of possible history extensions. In particular, since the original execution is linearizable, then integrating it with checkpoints and rollback steps maintains its linearizability. We conclude with the next Corollary to Claim 16, using its transitivity property:

▶ **Corollary 17.** *Given any linearizable implementation that satisfy the assumptions from Section 3.3, integrating it with checkpoints according to the guidelines from Section 4.2.1, and any number of rollback instructions, maintains its linearizability.*

### A.1.3 The Inverse Transformation

Let $I^{RF}$ be a linearizable reclamation-free implementation, satisfying the assumptions from Section 3.3. By Corollary 17, we can assume that it also contains checkpoint instructions, and that its linearizability property is not affected by inserting rollback steps. Let $I$ be the implementation obtained after applying the transformation from Section 4 to $I^{RF}$. Given a VBR-integrated execution $E \in I$, we construct a reclamation-free execution $E^{RF} \in I^{RF}$ by induction on $E$'s length[3].

First, we map node addresses as follows: if a node is allocated during $E$ from a memory address $a$ and with a birth epoch $b$, then during $E^{RF}$, it is mapped to a node, allocated from the memory address $\langle a, b\rangle$. This mapping is legal since the memory is considered to be unbounded in the reclamation-free setting. Moreover, by Claim 6, it is safe to assume that every two different nodes are allocated from different addresses in the reclamation-free setting.

In addition, any local variable from the original implementation $I^{RF}$ (not added after the VBR integration) is mapped to itself. Now, consider the finite VBR-integrated execution $E_{i-1} \in I$ and its respective reclamation-free execution $E^{RF}_{i-1} \in I^{RF}$, obtained so far.

1. If $s_i$ is an operation invocation, an operation response, or a local step from the original execution (not added after the VBR integration), then it is also appended to $E^{RF}_{i-1}$.
2. Assume that $s_i$ is a return from an *alloc(k)* call in line 11, let $a$ be the address $n$ is allocated from, and let $b$ be its birth epoch (as set in line 7). Then a respective allocation of a node from the memory address $\langle a, b\rangle$, by the same thread, and an initialization of its key to be $k$, are appended to $E^{RF}_{i-1}$.
3. Assume that $s_i$ is the read of a node's *next* pointer in line 18, by a thread T. Assume that the input parameter to this *getNext()* call is a pointer to a node, allocated from a memory address $a$ with a birth epoch $b$. Let $m$ be the node , allocated from $\langle a, b\rangle$ during $E^{RF}_{i-1}$. If T returns in line21, then a read of $m$'s *next* pointer and a respective *unmark()* call, both by T, are appended to $E^{RF}_{i-1}$.

[3] We ignore *retire()* calls, as they do not affect linearizability

4. Assume that $s_i$ is the read of a node's key in line 23, by a thread T. Assume that the input parameter to this *getKey()* call is a pointer to a node, allocated from a memory address $a$ with a birth epoch $b$. Let $m$ be the node , allocated from $\langle a, b\rangle$ during $E_{i-1}^{RF}$. If T returns in line25, then a read of $m$'s key is appended to $E_{i-1}^{RF}$.
5. Assume that $s_i$ is the read of a node's key, after this key has been previously saved in a local pointer by a thread T (for more details, see Section 4.2.2). Assume that this node is allocated from the address $a$ and with a birth epoch $b$. Then a read of the key of the node, allocated from $\langle a, b\rangle$, is appended to $E_{i-1}^{RF}$.
6. Assume that $s_i$ is the read of a node's *next* pointer in line 27, by a thread T. Assume that the input parameter to this *isMarked()* call is a pointer to a node, allocated from a memory address $a$ with a birth epoch $b$. Let $m$ be the node , allocated from $\langle a, b\rangle$ during $E_{i-1}^{RF}$. If T returns in line 29, then a respective *isMarked()* call is appended to $E_{i-1}^{RF}$.
7. Assume that $s_i$ is returning TRUE in line 28, by a thread T. Assume that the input parameter to this *isMarked()* call is a pointer to a node, allocated from a memory address $a$ with a birth epoch $b$. Let $m$ be the node , allocated from $\langle a, b\rangle$ during $E_{i-1}^{RF}$. Then a read of $m$'s *next* pointer and a respective *isMarked()* call, both by T, are appended to $E_{i-1}^{RF}$.
8. Assume that $s_i$ is the execution of line 33. Let $a_1, b_1, a_2, b_2, a_3, b_3$ be the input parameters to the respective *update()* call. For every $i \in \{1, 2, 3\}$, let $n_i$ be the node allocated from $\langle a_i, b_i\rangle$ during $E_{i-1}^{RF}$. Then a CAS on $n_1$'s *next* field, with $n_2$ as the expected value and $n_3$ as the new one, is appended to $E_{i-1}^{RF}$.
9. Assume that $s_i$ is the read of a node's *next* pointer in line 35, by a thread T. Assume that the input parameter to this *mark()* call is a pointer to a node, allocated from a memory address $a$ with a birth epoch $b$. Let $m$ be the node , allocated from $\langle a, b\rangle$ during $E_{i-1}^{RF}$. If T returns in line 39, then a read of $m$'s *next* pointer into a local variable *exp* is appended to $E_{i-1}^{RF}$.
10. Assume that $s_i$ is the execution of line 39, by a thread T. Assume that the input parameter to this *mark()* call is a pointer to a node, allocated from a memory address $a$ with a birth epoch $b$. Let $m$ be the node , allocated from $\langle a, b\rangle$ during $E_{i-1}^{RF}$, and let *exp* be the respective local variable from the previous transformation step. Then a mark of $m$'s *next* field, with *exp* as the expected value, is appended to $E_{i-1}^{RF}$.
11. Assume that $s_i$ is returning FALSE in line 37, by a thread T. Assume that the input parameter to this *mark()* call is a pointer to a node, allocated from a memory address $a$ with a birth epoch $b$. Let $m$ be the node , allocated from $\langle a, b\rangle$ during $E_{i-1}^{RF}$. Then a mark attempt of $m$'s *next* pointer is appended to $E_{i-1}^{RF}$.
12. If $s_i$ is a checkpoint installation, then a checkpoint installation by the same thread is appended to $E_{i-1}^{RF}$.
13. If $s_i$ is a rollback step, then a rollback step by the same thread is appended to $E_{i-1}^{RF}$.

Other steps are discarded when constructing $E^{RF}$. Recall that successful shared memory updates are only executed in lines 33 and 38, which are the respective last instructions in both update methods. Therefore, checkpoint instructions are installed in the same places in both implementation. Next, we prove the following equivalence between $E_i$ and its respective transformation output $E_i^{RF}$:

▷ Claim 18. For every $i > 0$, $E_i^{RF} \in I^{RF}$.

**Proof.** We prove the claim by induction on $i$. For the induction step, assume that $E_{i-1}^{RF} \in I^{RF}$. Note that we do not claim for correctness at this stage. We just prove that the execution $E^{RF}$ follows the original reclamation-free code.

- If $s_i$ is an operation invocation, an operation response, or a local step from the original execution (not added after the VBR integration), then it is also appended to $E^{RF}_{i-1}$, and obviously, $E^{RF}_i \in I^{RF}$.
- Assume that $s_i$ is a return from an *alloc()* call in line 11. Then a respective allocation appeared in $I_{RF}$. Since the return from the *alloc()* call is the only step resulting in appending an allocation to $E^{RF}_{i-1}$, the claim still holds for $E^{RF}_i$ in this case.
- Assume that $s_i$ is the read of a node's *next* pointer in line 18, by a thread T. Then a respective read of this *next* pointer, along with an *unmark* call, appeared in $I_{RF}$. Since the read of line 18 is the only step resulting in appending the respective read and unmark of a *next* pointer to $E^{RF}_{i-1}$, the claim still holds for $E^{RF}_i$.
- Assume that $s_i$ is the read of a node's key in line 23. Then a respective read of this key appeared in $I_{RF}$. Since the read of line 23 is the only step resulting in appending the respective read key to $E^{RF}_{i-1}$, the claim still holds for $E^{RF}_i$.
- Assume that $s_i$ is the read of a node's key, after it has been previously saved in a local variable. Then a respective read of this key appeared in $I_{RF}$, and the claim still holds for $E^{RF}_i$.
- Assume that $s_i$ is the read of a node's *next* pointer in line 27, by a thread T, that returns in line 29. Then a respective *isMarked()* call appeared in $I_{RF}$. Since a respective *isMarked()* call is appended to $E^{RF}_{i-1}$ only with respect to the execution of line 27 in this case, the claim still holds for $E^{RF}_i$.
- Assume that $s_i$ is returning TRUE in line 28, by a thread T. Then a respective *isMarked()* call appeared in $I_{RF}$. Since a respective *isMarked()* call is appended to $E^{RF}_{i-1}$ only with respect to the execution of line 28 in this case, the claim still holds for $E^{RF}_i$.
- Assume that $s_i$ is the execution of line 33. Then a respective pointer update appeared in $I_{RF}$. Since the WCAS execution in line 33 is the only step resulting in appending the respective update to $E^{RF}_{i-1}$, the claim still holds for $E^{RF}_i$.
- Assume that $s_i$ is the read of a node's *next* pointer in line 35, by a thread T. Then a respective pointer read, in the scope of a *mark()* execution, appeared in $I_{RF}$. Since the execution of line 35 is the only step resulting in appending the respective pointer read to $E^{RF}_{i-1}$, the claim still holds for $E^{RF}_i$.
- Assume that $s_i$ is the execution of line 39, by a thread T. Then a respective pointer mark, in the scope of a *mark()* execution, appeared in $I_{RF}$. Since the execution of line 39 is the only step resulting in appending the respective marking to $E^{RF}_{i-1}$, the claim still holds for $E^{RF}_i$.
- Assume that $s_i$ is returning FALSE in line 37, by a thread T. Then a respective pointer mark, in the scope of a *mark()* execution, appeared in $I_{RF}$. Since the execution of line 37 is the only step resulting in appending the respective marking to $E^{RF}_{i-1}$, the claim still holds for $E^{RF}_i$.
- Assume that $s_i$ is a checkpoint installation by a thread T. Then the previous step by T was either an operation invocation or a shared-memory update in line 33 or 39. I.e, by the induction hypothesis, the last step in $E^{RF}_{i-1}$ it the execution of a checkpoint-trigger as well, and the claim still holds for $E^{RF}_i$.
- Assume that $s_i$ is a rollback step. By Corollary 17, rollback steps can be installed at any stage, and the claim still holds for $E^{RF}_i$.

◀

After proving that $E^{RF}$ indeed follows the original code of the reclamation-free implementation (including checkpoints and rollback steps). It still remains to show that $E$ and $E^{RF}$ share the same history, as we prove in Claim 19 below.

▷ Claim 19. Let $i > 0$. Then the following hold:

1. After $E_i$ and after $E_i^{RF}$, the shared memory view is identical.
2. If a local variable $x$ contains a value $v$ after $E_i^{RF}$, then it either contains $v$ after $E_i$, or its content is indistinguishable to the executing thread from $\perp$ after $E_i$.
3. $E_i$ and $E_i^{RF}$ share the same history.

Proof. We are going to prove by induction that if the invariants hold for $E_{i-1}$ and $E_{i-1}^{RF}$, then they hold for $E_i$ and $E_i^{RF}$ as well. For the inductive step, assume that the claim holds for $E_{i-1}$ and $E_{i-1}^{RF}$.

1. If $s_i$ is either discarded or has no influence on shared memory, then the claim still holds by the induction hypothesis (invariant 1). Otherwise, it must either be the execution of line 33 or 38. By the induction hypothesis (invariants 1 and 2), and by Claims 11-15, the claim still holds for $E_i$ and $E_i^{RF}$.
2. If $s_i$ is discarded, then the claim still holds by the induction hypothesis (invariant 2). Otherwise, if $s_i$ is an operation invocation, an operation response, or a local step from the original execution (not added after the VBR integration), then the claim still holds by the induction hypothesis (invariants 1 and 2). Otherwise:
   - If $s_i$ is the return from an *alloc()*, resulting in the read of the allocated node into a thread's local pointer, then by our transformation, the respective node, with the same key, is read into the thread's same local variable in $E_i^{RF}$, and the claim still holds.
   - If $s_i$ is the read of a node's key into a local variable, then by the induction hypothesis (invariant 1) and Claim 8, the claim still holds.
   - If $s_i$ is the read of a node's *next* field into a local variable (that is not ignored by the executing thread), then by Assumption 4, the induction hypothesis (invariant 1), Claim 9 and Claim 14, then the claim still holds. If the executing thread ignores the read value, then it is indistinguishable to it from $\perp$, and the claim still holds.
   - If $s_i$ is an *isMarked()* call then by the induction hypothesis (invariant 1) and Claim 7, the claim still holds.
   - If $s_i$ is a read of a pointer in line 35, then it is indistinguishable to the executing thread from $\perp$, and the claim still holds.
   - If $s_i$ is a pointer update, then its response is read into a local variable. By the induction hypothesis (invariants 1 and 2) and Claims 11-15, the claim still holds.
   - If $s_i$ is a checkpoint installation then the same respective local variables are logged. By the induction hypothesis (invariants 1 and 2), the claim still holds.
   - If $s_i$ is a rollback step, then by the induction hypothesis (invariants 1 and 2), both executions rollback to the same checkpoint and restore the same local variables, and the claim still holds.
3. If $s_i$ is neither an invocation nor a response step, then the history of both executions remains unchanged. If $s_i$ is an invocation step, then by our construction, an identical invocation is appended to $E_{i-1}^{RF}$. If $s_i$ is a response step, then by our construction, a respective response step is appended to $E_{i-1}^{RF}$. By invariant 2, the response output is identical in both executions. Finally, by the induction hypothesis (invariant 3), the claim still holds.

◁

By Claims 18 and 19, every VBR-integrated execution $E \in I$ has the same history as a certain reclamation-free execution $E^{RF} \in I^{RF}$. By our assumption, $I^{RF}$ contains only linearizable executions, which derives Lemma 3.

## A.2 VBR Maintains Lock-Freedom

In this section we prove that VBR maintains the lock-freedom guarantee of the original linearizable and lock-free implementation:

▶ **Lemma 20.** *Given a lock-free linearizable data-structure implementation, that satisfy all of the assumptions presented in Section 3.3, the implementation remains lock-free after integrating it with VBR, according to the modifications described in Sections 4.1-4.2.*

We first prove that VBR is *robust*. I.e., a stalled thread may not prevent the reclamation of an unbounded number of retired objects and therefore, the system never blocks due to a possible exhaustion of the heap. We start by showing that the number of unused nodes that are not retired is bounded:

▷ Claim 21. Let $E$ be a VBR-integrated execution and let $C_i$ be a configuration. The number of unlinked nodes (see Assumption 3) that are not retired at $C_i$ is bounded.

Proof. During $E$, an unlinked node $n$ must be in one of the following statuses:

1. $n$ was allocated by a thread $T$, and has not become reachable since, and $T$ is now performing a rollback step. After $T$ performs its rollback step, no thread will have a local pointer to $n$, and by Assumption 3, $n$ will also not be reachable from shared-memory.
2. $n$ has gone through stages 1-5 of Assumption 3, and has not been retired yet.

By Assumption 3, there exist no other possibilities. The first case is handled as stated in Appendix B. Before $T$ rolls back, it appends $n$ to its allocation list. Although it is not a standard retirement, it makes sure that $n$ is recycled.

The second case is naturally handled, according to Assumption 3. If the retiring thread is given with enough scheduler time, then it should eventually retire $n$. If it is not given enough time, then the number of unlinked nodes that are still not retired is still bounded.

In addition, as described in Appendix B, if $T$ is forced to execute a roll-back step before it gets the chance to retire $n$, then it retires $n$ right before rolling back (i.e., if $n$'s retire epoch is $\perp$, it sets its retire epoch to be the current global epoch and appends it to its retired nodes list). Notice that if $n$'s retirement depends on following steps (i.e., there exists an extension of the current execution in which $T$ does not retire $n$), then $n$ should not be retired at this stage anyway. ◁

After showing that the number of unused nodes that are not retired is bounded, we show that the number of retired nodes that are not reclaimed is also bounded. Recall that the executing threads use local retired nodes list. Therefore, a stalled thread may prevent the reclamation of the nodes residing in its own retired nodes list. One can easily add a *stealing* mechanism, allowing other threads to reclaim nodes that were retired by other threads. However, we have not implemented such a mechanism, since the size of these lists is bounded (and pre-defined), and does not affect the system progress in practice. For showing that VBR is robust, it still remains to show that a stalled thread does not affect the reclamation of nodes, retired by other threads. We prove a stronger claim:

▷ Claim 22. Let $E$ be a VBR-integrated execution, and let $s_i$ and $s_j$ be two consecutive calls to the *alloc()* method by a thread T. Then at least one call returns a new allocated node in line 11.

Proof. If the first call to the *alloc()* method returns in line 11, then we are done. Otherwise, it returns in line 6, after T tries to increase the global epoch in line 4, and after the new node, $n$, is appended to the allocation list in line 5.

When the *alloc()* method is called again by T, If the node retrieved from the allocation list is not $n$, then it must be a node $m$, returned to T's allocation list before performing the rollback step. The node $m$ is guaranteed to not be reachable yet (for more details, see Section 4.2.1). By Assumption 3, $m$ is also guaranteed to not be retired. Therefore, $m$'s retire epoch is $\perp$, the condition in line 3 does not hold and $m$ is returned in line 11.

Otherwise, $n$ is node retrieved from the allocation list in line 2. By assumption 3, no other thread tries to meanwhile retire $n$, and thus, its retire epoch does not change before T calls the *alloc()* method for the second time. By Observation 4, the global epoch can only increase. Therefore, even if T's CAS in line 4 (during the first *alloc()* call) is not successful, the global epoch is guaranteed to be bigger than $n$'s retire epoch when T rolls-back in line 6. After executing line 6, T updates its local copy of the global epoch. Consequently, it is guaranteed to be bigger than $n$'s retire epoch during the second *alloc()* call, and the second call returns in line 11. ◁

After showing that VBR is robust, it still remains to show that it does not foil the original implementation's lock-freedom guaranty. To derive a contradiction, assume that there exists a VBR-integrated execution $E$ that does not satisfy the lock-freedom guaranty. Thus there is a suffix of $E$ in which no operations terminate and some operations take infinitely many steps. Recall that there is a finite number of executing threads, and that we assume a well-formed execution. Therefore, at every given point during $E$, there is a finite number of pending invocations.

▷ Claim 23. There is a finite number of logical node insertions, logical node removals and node retirements during $E$.

Proof. Assume by contradiction that there are infinitely many logical node insertions during $E$. In particular, there is a pending operation *op* during which there are infinitely many logical node insertions. By Assumption 3, only previously allocated nodes can be logically inserted into the data-structure. Therefore, there exist at least two different points after the invocation of *op*, that are considered as logical insertions by the thread executing *op*. By Lemma 3, $E$ is linearizable. By the *non-blocking* property of linearizability (see [29]), *op* has two different linearization points – a contradiction. Therefore, there is a finite number of logical node insertions during $E$.

By Assumption 3, a node can only be logically removed after it is logically inserted, and can only be retired after it is logically removed. In addition, by Claim 6, every node is retired exactly once. Therefore, there is also a finite number of logical node removals and retirements during $E$. ◁

▷ Claim 24. There is a suffix of $E$ in which the global epoch counter does not change.

Proof. By Claim 23, there is a finite number of node retirements during $E$. Therefore, there is a suffix $\beta$ of $E$ in which node retirements are not executed anymore. Let $e$ be the retire epoch of the last retired node during $E$. By Observation 4, the retire epoch of every previously retired node is at most $e$.

The global epoch counter can only be increased in line 4, and only if it equal to the retire epoch of a newly allocated node. During $\beta$, all nodes have a retire epoch which is either at most $e$, or $\perp$. Therefore, the global epoch does not increase after it is incremented to $e + 1$. ◁

▷ Claim 25. There is a suffix of $E$ in which rollback steps are never executed.

Proof. By Claim 24, there is a suffix $\alpha$ of $E$ in which the global epoch counter does not change. Let $T$ be a thread, executing a rollback step during $\alpha$. Upon rolling-back to its last checkpoint, $T$ updates its local copy of the global epoch counter. Starting from this update, $T$'s local copy of the global epoch counter remains equal to the global epoch counter, and $T$ never execute a rollback step again. The number of rollback steps during $\alpha$ is bounded by the number of executing threads (which is finite), and the claim holds. ◁

Let $s_i$ be the first step in the suffix of $E$, guaranteed by Claim 25. By Claim 18, the reclamation-free execution $E^{RF}$, obtained by our transformation, is a legal execution of the original reclamation-free implementation. Starting from the step which is equivalent to $s_i$, $E^{RF}$ contains no rollback steps as well. Therefore, by the lock-freedom guaranty of the original implementation, some pending operation terminates during $E^{RF}$. By Claim 19, it terminates during $E$ as well – a contradiction. VBR does not foil the original implementation's lock-freedom guaranty, which derives Lemma 20.

## B Handling Unlinked Nodes before Checkpoint Rollbacks

In this section we explain how we avoid losing access to unlinked nodes when rolling back to a checkpoint. This obviously does not affect linearizability, as by Assumption 3, such nodes cannot represent data-structure items anyway. Moreover, this does not directly affect robustness, as robustness relates to the number of unreclaimed retired nodes, which is bounded in VBR (for more details, see Appendix A.2). However, our goal is to show that the system never blocks as a result of an allocation block (due to an exhaustion of the heap). Therefore, we handle such unlinked nodes as follows.

Assume that a thread T is about to execute a rollback step to its previous checkpoint, Then there are two types of nodes that may become unreachable to all of the executing threads after T executes its rollback step:

1. Nodes that were allocated by T after its last checkpoint visit, and are not yet reachable from the data-structure entry points.
2. Unlinked nodes that should be retired by T.

By Assumption 3, all other nodes are either reachable or must be retired by other threads. The first type of unlinked nodes is handled easily: for every such node, T appends them to its allocation list. By Assumption 3, these nodes are not yet retired and therefore, their retire epoch is $\perp$. In Appendix A we show that this does not foil correctness.

Handling the other type is more complex. Recall that by Assumption 3, such nodes should already be unreachable. Therefore, T has direct access to them via its local pointers, or by following a sequence of *next* pointers, starting from a node referenced by one of its local pointers (e.g., retiring a sequence of unlinked nodes after a physical deletion in [22]). By Assumption 3, T should be able to distinguish between such nodes and nodes that should not be retired. Additionally, when T is in charge of retiring a certain node, and this node is only reachable by a sequence of node pointers, it is guaranteed that all of the nodes in the sequence are also retired by T. In particular, they are not yet reclaimed, and by Assumption 2, their pointer data does not change throughout this retirement procedure. Therefore, T can safely read the relevant node pointers, while ignoring pointer versions. For every such node $n$, if $n$'s retire epoch is $\perp$ then T updates $n$'s retire epoch to be the current global epoch, and appends $n$ to its retired nodes list. Note that T does not update its local copy of the global epoch, as its is going to be updated after the rollback step.

## C Integrating VBR: An Example

We are going to demonstrate VBR's integration using a standard implementation of a lock-free linked-list [22, 27, 34]. The linked-list interface suggests three operations: the *add()* operation adds a key to the list (and does nothing if the key is already present), the *remove()* operation removes an existing key (and does nothing if no such key is present), and the *contains()* operation returns a boolean indicating whether a key is present. In addition to these three operations, the implementation includes an auxiliary *find()* method, used by the *add()* and *remove()* operations for locating a given key. The *find()* method is also in charge of physically deleting marked nodes that has not been physically removed by their logical remover yet (logical and physical deletions are discussed in Section 3.3).

```
1:  find(int key)
2:  retry:
3:      pred := head
4:      pred_b := head → birth_epoch
5:      ⟨ curr, curr_b ⟩ := getNext(pred)
6:      curr_key := getKey(curr)
7:      while (TRUE)
8:          if (isMarked(curr, curr_b))
9:              ⟨ succ, succ_b ⟩ := getNext(curr)
10:             while (isMarked(succ, succ_b))
11:                 ⟨ succ, succ_b ⟩ := getNext(succ)
12:             if (update(pred, pred_b, curr, curr_b, succ, succ_b) == FALSE)
13:                 goto retry
14:             ⟨ curr, curr_b ⟩ := ⟨ succ, succ_b ⟩
15:             curr_key := getKey(curr)
16:         if (curr_key ≥ key)
17:             return pred, pred_b, curr, curr_b
18:         ⟨ pred, pred_b ⟩ := ⟨ curr, curr_b ⟩
19:         ⟨ curr, curr_b ⟩ := getNext(pred)
20:         curr_key := getKey(curr)
```

**Figure 3** The *find* method.

```
1:  add(int key)
2:      install checkpoint
3:      my_e := e.get()
4:      while (TRUE)
5:          ⟨ pred, pred_b, succ, succ_b ⟩ := find(key)
6:          succ_key := getKey(succ)
7:          if (succ_key == key) return FALSE
8:          n := alloc(key)
9:          n_b := n → birth_epoch
10:         res := update(pred, pred_b, succ, succ_b, n, n_b)
11:         if (res == TRUE)
12:             install checkpoint
13:             my_e := e.get()
14:             return TRUE
15:         else retire(n, n_b)
```

**Figure 4** The *add* operation.

As briefly discussed in Section 4.2.1, the standard linked-list implementation consists of several update instructions. Specifically, we chose to rely on an implementation with four update instructions. The first two are the insertion of a new node into the list (executed during

```
1:  remove(int key)
2:      install checkpoint
3:      my_e := e.get()
4:      ⟨ pred, pred_b, curr, curr_b ⟩ := find(key)
5:      curr_key := getKey(curr)
6:      if (curr_key ≠ key) return FALSE
7:      while (isMarked(curr, curr_b) == FALSE)
8:          ⟨ succ, succ_b ⟩ := getNext(curr)
9:          res := mark(curr, curr_b)
10:         if (res == TRUE)
11:             install checkpoint
12:             my_e := e.get()
13:             if (update(pred, pred_b, curr, curr_b, succ, succ_b) == FALSE)
14:                 find(key)
15:             retire(curr, curr_b)
16:             return TRUE
17:     return FALSE
```

**Figure 5** The *remove* operation.

```
1:  contains(int key)
2:      install checkpoint
3:      my_e := e.get()
4:      curr := head
5:      curr_b := head → birth_epoch
6:      curr_key := getKey(curr)
7:      while (curr_key < key)
8:          ⟨ curr, curr_b ⟩ := getNext(curr)
9:          curr_key := getKey(curr)
10:     return (isMarked(curr, curr_b) == FALSE)
```

**Figure 6** The *contains* operation.

the *add()* operation) and the marking of a node for logically deleting it (executed during the *remove()* operation). In both cases, the successful update determines the linearization point of the respective operation. I.e., the output of the respective CAS execution is crucial for linearizability. If the executing thread performs a rollback step, right after one of these updates, the obtained execution would have a different set of corresponding history extensions (and in particular, it would not be linearizable). Therefore, these two updates are considered as rollback-unsafe steps, and when are executed successfully, must be followed by a checkpoint installation.

The remaining two update instructions, which are the physical deletion of a node (either during a *remove()* operation or the *find()* auxiliary method), are both rollback-safe steps. Intuitively, the remover identity does not affect linearization, so performing a rollback step right after the physical deletion would not change the set of corresponding history extensions. Therefore, checkpoints are not installed after a successful execution of these update instructions.

The standard lock-free linked-list implementation is a natural candidate for demonstrating the VBR integration. Assumption 1 holds since all shared-memory updates occur via CAS executions. Assumption 2 holds since a node's *next* pointer is its only mutable field, and it is indeed invalidated via marking. Assumption 3 holds since list nodes indeed follow the life-cycle, presented in Section 3.3: they are first allocated, then become reachable and valid (a node's logical insertion into the list is its physical insertion), then invalid (via marking), physically removed, and retired – either by their logical remover (after making sure they are physically removed) or by some physical remover. Finally, Assumption 4 holds since, after a

rollback-unsafe update, the list traversal is never resumed from a previously referenced node. E.g., after marking a node for its logical deletion, the following list traversal (for physically deleting this node) is initiated from the list head, and from any other previously saved node reference.

The pseudo code for the VBR integration example appears in Figures 3-6. The methods from Figure 1 are marked with red and checkpoint installations are marked with blue. As described in Section 4.2.1, installing a checkpoint includes updating the checkpoint reference. The checkpoint reference is used when a thread performs a rollback step (after performing line 6, 16, 20 or 24 from Figure 1). It should also include saving the content of local variables for a later recovery. However, this part is unnecessary in the linked-list implementation, as local variables are never overwritten after a checkpoint. Finally, the local copy of the global epoch counter is always updated after installing (or rolling back to) a checkpoint.

## C.1 The *find()* method

The original *find()* auxiliary method receives a key as its input parameter, and returns pointers to two nodes, *pred* and *curr*. *pred* represents the node with the maximal key which is smaller than the input key in the list, and *curr* represents the node with the minimal key which is equal or greater than the input key in the list. In addition, it is guaranteed that at some point during the *find()* execution, both node were reachable (and in particular, logically in the list), and that *curr* was *pred*'s successor at this point. This method is also in charge of physically removing logically deleted nodes, while traversing the list.

The VBR-integrated pseudo code for the *find()* auxiliary method appears in Figure 3. This variant returns the two respective nodes, together with their birth epochs (see line 17). As this method is not an interface operation, a checkpoint is never installed upon its invocation. Additionally, as all updates performed in its scope (i.e., the physical removal in line 14) are rollback-safe steps, it dose not include installing checkpoints at all. Consequently, whenever a read method (either *getKey()* or *getNext()*) results in a rollback, the rollback is to a checkpoint, installed during the calling operation (either an *add()* or a *remove()* operation).

## C.2 The *add()* operation

The original *add()* operation receives a key as its input parameter, and inserts a node with the given key into the list (and returns TRUE upon a successful insertion). If there already exists a node with the given key, which is logically in the list, it does nothing (and returns FALSE). The successful physical insertion of a new node is also its logical insertion, and is done via a CAS instruction. The operation returns TRUE only after such a successful CAS, which means that the CAS result is crucial for maintaining linearizability. Therefore, the physical insertion of a node is considered as a rollback-unsafe step according to Definition 2.

The VBR-integrated pseudo code for the *add()* operation appears in Figure 4. As this is an interface operation, a checkpoint is installed immediately after invocation (see line 2). In addition, a second checkpoint is installed after the successful physical (and logical) insertion of a new node into the list (see line 12). This second checkpoint is obviously unnecessary, as there are no rollback steps between the installation and the return in line 14. However, we added it for the completeness of our example.

## C.3 The *remove()* operation

The original *remove()* operation receives a key as its input parameter, and removes a node with the given key from the list (and returns TRUE upon a successful removal). If there does

not exist a node with the given key, which is logically in the list, it does nothing (and returns FALSE). After traversing the list and locating the deletion candidate, it is first logically deleted via marking (the remover is the thread executing the successful respective CAS), then physically removed and retired. The physical removal is not considered as a crucial phase of the *remove()* operation. In certain implementations, the *remove()* operation returns right after the logical deletion, and the physical deletion is executed during a future traversal (and may not happen at all). In our example implementation, the removing thread is also in charge of the node's retirement. Therefore, a node's physical removal is validated via an extra call to the *find()* method, before retirement.

The VBR-integrated pseudo code for the *remove()* operation appears in Figure 5. As this is an interface operation, a checkpoint is installed immediately after invocation (see line 2). In addition, a second checkpoint is installed after the successful marking of the victim node (see line 11). This second checkpoint is necessary, as the remover thread may rollback during the *find()* execution (line 14) or when retiring the victim node (line 15). Note that the physical removal trial in line 13 is considered as a rollback-safe step according to Definition 2 and thus, it is not followed by a checkpoint installation. In addition, note that the *getNext()* call in line 8 does not violate Assumption 4, as no rollback-unsafe step is executed prior to this reference read.

## C.4 The *contains()* operation

The original *contains()* operation receives a key as its input parameter. It returns TRUE if the list contains a node with the given key, and FALSE otherwise. While the *contains()* operation may be implemented using the *find()* auxiliary method, it is traditionally implemented in a wait-free manner, with a single list traversal, while avoiding physical deletions (which may foil wait-freedom).

The VBR-integrated pseudo code for the *contains()* operation appears in Figure 6. Although it is built on the wait-free variant of the original *contains()* implementation, it is not wait-free (as checkpoint rollbacks do not maintain wait-freedom). As it contains no shared-memory updates, it includes a single checkpoint installation, upon invocation (line 2).