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

Related Version *Full Version*: `https://arxiv.org/abs/2107.13843` [48]

Funding This work is supported by the United States - Israel BSF grant No. 2018655

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 [9, 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, 50] 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 [56]). 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 [9, 49] 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 [13] (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 [11, 12] 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 [11–13], 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 [9, 12, 13, 37, 49]. 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 [13] for data structures that are written in the normalized form of lock-free data structures [53]. Subsequently, a weaker version of normalized concurrent data structures was presented in [49], 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. A full correctness proof, along with an illustration of integrating VBR into a lock-free data-structure (Harris's linked-list [22]), appear in the full version of this paper [48].

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 [15]. 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 [12, 13, 56]) 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 [9, 49], 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 [3, 41, 45, 56], 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 reallocations 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 [48]).

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 [57]) 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 [55], 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 [13, 31, 37, 45, 56], 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, 52]). 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 [12, 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 VBR: 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 [48].

▶ **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.

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, 54] 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, 54], 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 [48]). 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 [11–13]. 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 [12, 13], and in a total separation between read and write phases during the execution in [49]. 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

```
 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

general and makes less assumptions on the given implementation. Our method is to carefully define program checkpoints.

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 [53], and shares some mutual concepts with the *capsules* definition, given in [5, 6]. 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

[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 [48].

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 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 [48] 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 [48] 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.

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.

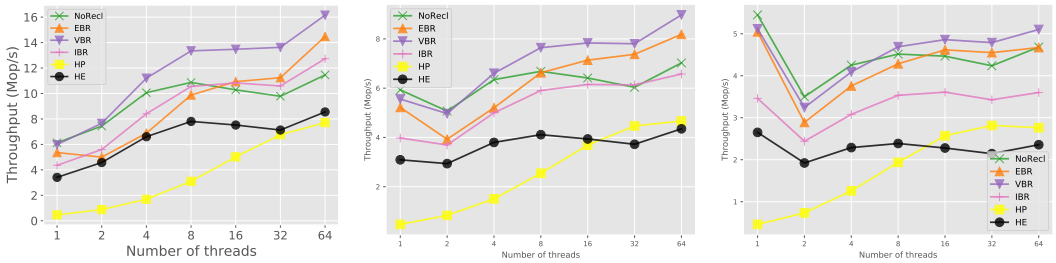
We evaluated VBR against a baseline execution in which memory is never reclaimed (denoted NoRecl), an optimized implementation of the epoch-based reclamation method [22] (denoted EBR), the traditional hazard pointers scheme [37] (denoted HP), the hazard eras scheme [45] (denoted HE), and the 2GEIBR variant of the interval-based scheme [56] (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 [10, 40]. We did not implement binary search trees, because some of the measured competing schemes cannot support it.

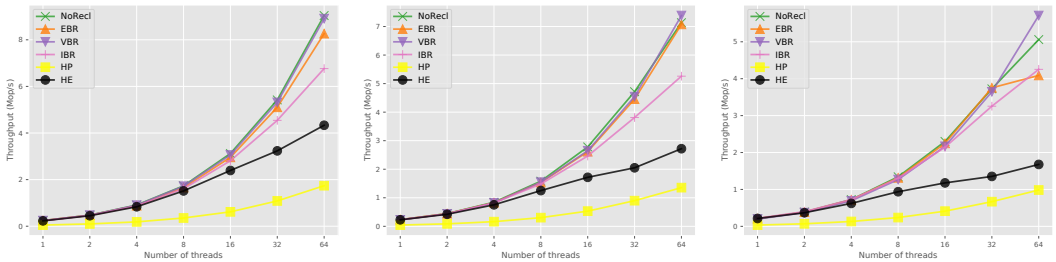
5.1 Setup

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.

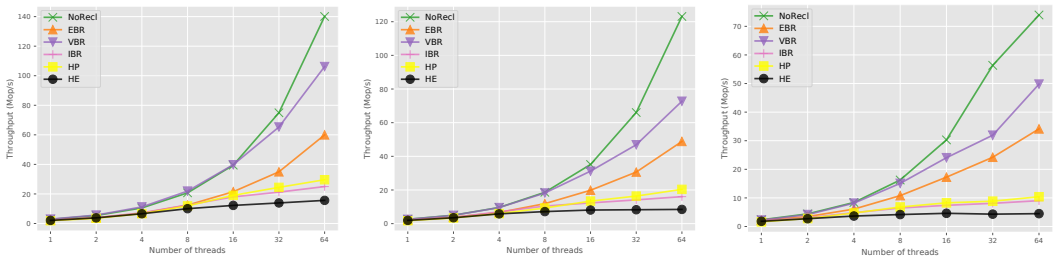
We implemented object pools in a similar way to [12], to avoid returning reclaimed objects to the OS. All schemes used that implementation, in which all pools are pre-allocated before the test. Each test was a fixed-time micro benchmark in which threads randomly call the *Insert()*, *Delete()* and *Search()* 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



(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.

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. 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, 49] 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 [9, 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 [4, 9, 41, 45, 56] are either not always applicable, or rely on special hardware support. Drop-the-anchor [8] 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 [7, 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 [51] and a lock-free variant [14], 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 [2], StackTrack [1], 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+ [9] and NBR [49] 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 [4] requires control of the OS scheduler. In particular, to make hazard pointers visible, threads are periodically swapped out.

References

1 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.

2 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.

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

4 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.

5 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.

6 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.

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

8 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.

9 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.

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

11 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.

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

13 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.

14 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.

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

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 Gali Sheffi, Maurice Herlihy, and Erez Petrank. Vbr: Version based reclamation, 2021. `arXiv:2107.13843`.

49 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.

50 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.

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

52 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.

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

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

55 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.

56 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.

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