NBR: Neutralization Based Reclamation

Ajay Singh
University of Waterloo
Canada
ajay.singh1@uwaterloo.ca

Trevor Brown
University of Waterloo
Canada
trevor.brown@uwaterloo.ca

Ali Mashtizadeh
University of Waterloo
Canada
mashti@uwaterloo.ca

Abstract

Safe memory reclamation (SMR) algorithms suffer from a trade-off between bounding unreclaimed memory and the speed of reclamation. Hazard pointer (HP) based algorithms bound unreclaimed memory at all times, but tend to be slower than other approaches. Epoch based reclamation (EBR) algorithms are faster, but do not bound memory reclamation. Other algorithms follow hybrid approaches, requiring special compiler or hardware support, changes to record layouts, and/or extensive code changes. Not all SMR algorithms can be used to reclaim memory for all data structures.

We propose a new neutralization based reclamation (NBR) algorithm that is faster than the best known EBR algorithms and achieves bounded unreclaimed memory. It is non-blocking when used with a non-blocking operating system (OS) kernel, and only requires atomic read, write and CAS. NBR is straightforward to use with many different data structures, and in most cases, require similar reasoning and programmer effort to two-phased locking. NBR is implemented using OS signals and a lightweight handshaking mechanism between participating threads to determine when it is safe to reclaim a record. Experiments on a lock-based binary search tree and a lazy linked list show that NBR significantly outperforms many state of the art reclamation algorithms. In the tree NBR is faster than next best algorithm, DEBRA by upto 38% and HP by upto 17%. And, in the list NBR is 15% and 243% faster than DEBRA and HP, respectively.

Keywords: safe memory reclamation, synchronization and concurrency control, concurrent data structures, algorithms.

1 Introduction

Fundamentally, safe memory reclamation (SMR) is about answering the question: When is it safe to free a record? Unlike garbage collection, which is automatic, SMR requires a program to invoke a retire operation on each record at some point after it becomes garbage (i.e., is unlinked from the data structure). The task of an SMR algorithm is to eventually free an unlinked record once no thread holds a pointer to it [7, 11, 30].

The challenge of SMR in concurrent data structures comes from write-after-read (or more apt term would be use-after-free) conflicts between threads, where one thread accesses a record that is concurrently freed by another. For example, consider a lazy-list where one thread is searching and another is deleting. The first thread acquires a reference to a record and stores it in a local variable. The other thread unlinks and frees. At this point the first thread’s reference is no longer valid as it has been freed. If this execution is not prevented it could lead to a crash of the program.

Researchers have developed a rich variety of SMR algorithms, with a diverse spectrum of desirable properties, idiosyncrasies and limitations. After experimenting with SMR algorithms and observing the state of art [2–7, 11–14, 16, 17, 20, 24, 26, 29, 30, 33, 36], we identified the following set of desirable properties. [P1] Performance: All reclamation operations should offer both low latency and high throughput. [P2] bounded garbage: The number of unreclaimed records should be bounded, even if threads experience halting failures or long delays. [P3] Usability: Intrusive changes to code, data, and the build environment, should be minimized. [P4] Consistency: Performance should not be drastically affected by changes in the workload (e.g., when shifting between read-intensive and update-intensive workloads). Additionally, there should be minimal performance degradation when the system is oversubscribed (with more threads than cores). [P5] Applicability: The algorithm should be usable with as many data structures as possible.

To set the stage for our contribution, we must first discuss other approaches. We broadly categorize existing work into: hazard pointer-based reclamation (HPBR), quiescent state-based reclamation (QSBR), epoch-based reclamation (EBR), reference counting based reclamation (RCBR), and hybrid algorithms that combine the aforementioned approaches [26].

In general, QSBR and EBR are fast but do not bound garbage, HPBR has bounded garbage but is not fast, and RCBR is neither fast nor does it bound garbage. Hybrid approaches...
have generally focused on achieving both of these properties simultaneously, usually by combining EBR (for its speed) with some variant of HPBR (to bound garbage), with varying levels of success.

The hybrid algorithm that most closely resembles our approach is DEBRA+ [11], a variant of EBR and limited HPs for lock-free data structures. It is fast as well as has bounded garbage property. DEBRA+ achieves bounded reclamation garbage via a neutralizing mechanism based on POSIX signals and data structure specific recovery code. A thread whose reclamation is delayed by a slow thread will send a neutralizing signal to the slow thread. Upon receipt of a neutralizing signal, a thread executes its recovery code and then restarts its data structure operation, allows reclamation to continue, ultimately guaranteeing a bound on the number of unclaimed records. However, this bound on garbage comes at the cost of both usability and applicability, as users need to write data structure specific recovery code which is not always straightforward, or even possible. Moreover, DEBRA+ cannot be used for lock-based data structures as neutralizing a thread that holds a lock could cause deadlock.

Contribution: Existing SMR algorithms all have significant shortcomings in their attempts at satisfying properties P1 through P5. This motivated us to propose a new Neutralization Based Reclamation algorithm (NBR) that is faster than existing SMR algorithms [P1], bounds garbage [P2], simple to use [P3], exhibits consistent performance, even on over-subscribed systems [P4], and is applicable to a large class of data structures, some of which are not supported by most SMR algorithms [P5].

NBR’s neutralization technique is similar to that of DEBRA+, with a few key differences. In NBR, each thread places unlinked objects in a thread-local buffer, and when the buffer’s size exceeds a predetermined threshold, the thread sends a neutralizing signal to all other threads. Upon receipt of such a signal, a thread checks whether its current data structure operation has already done any writes to shared memory, and if not, restarts its operation (using the C/C++ procedures sigsetjmp and siglongjmp). Otherwise, it finishes executing its operation. In contrast, to guarantee bounded garbage in DEBRA+, a thread must restart even if it has already written to shared memory—a design decision that limits DEBRA+’s applicability to specific lock-free data structures, and necessitates data structure specific recovery code. NBR does not require any recovery code, and can be used with nearly all structures that DEBRA+ supports and many others structures DEBRA+ does not, including some lock-free algorithms like Lock-based binary search tree with lock-free searches of Drachslar et al. [18] and DGT [15].

We also present an optimized version of NBR called NBR+ in which threads send fewer signals, and yet reclaim memory more often. This is accomplished by allowing threads to infer when memory can be freed simply by passively observing the signals sent in the system. Finally, as our experiments show, NBR+ is highly efficient, significantly outperforming the state of the art in SMR in various data structure workloads on a large-scale 4-socket Intel system.

The rest of the paper is structured as follows. Related work is surveyed in Section 2. In Section 3 we introduce the model. Section 4 describes our basic algorithm NBR, and characterizes its applicability. We describe an optimized version NBR+ in Section 5. Finally, experiments appear in Section 7, followed by conclusions in Section 8.

2 Related Work

Although detailed surveys of safe memory reclamation already exist in earlier works [11, 26], we would like to study existing techniques specifically through the lens of the desirable properties defined above.

RCBR involves explicitly counting the number of incoming pointers to a record, and typically storing this count alongside the record. The inclusion of this metadata in records complicates any advanced pointer arithmetic or implicit pointers, and can requires changes to record layouts (or the use of a custom allocator) as well as size. RCBR typically requires a programmer to invoke a deref operation to dereference a pointer (and sometimes to explicitly invoke operations for read, write and CAS) [6, 16, 29], adding significant overhead and programmer effort [opposing P1, P3]. Programmer intervention is also needed to identify and break pointer cycles in garbage records.

HPBR incurs significant overhead every time a new record is encountered, as a thread must first announce a hazard pointer (HP) to it in a shared location, then issue a memory fence and check whether the record has already been unlinked [17, 29, 30] [opposing P1, P3]. If the record has been unlinked, the data structure operation trying to access it must be restarted (a data structure specific action). Correctly dealing with such failure cases can require extensive code changes. This may also require the programmer to reprove the data structure’s progress guarantees [11]. Additionally, it is not clear how HPs could be used with data structures that allow threads to traverse pointers in unlinked records [11], and there are many examples of such data structures, e.g., [1, 8, 10, 19, 21, 23, 27, 31, 32, 34] [opposing P5]. (In such data structures, a search can potentially pass through many unlinked records, and yet end up back in the data structure, at the appropriate location.)

The latter limitation was addressed by Beware and Cleanup—a hybrid of RCBR and HPBR [24]. However, this algorithm requires a programmer to write a data structure specific cleanup procedure that changes all pointers in an unlinked record to point to current records in the data structure. This cleanup code ultimately ensures bound on garbage for data structures that allow traversing unlinked records, but the algorithm has higher overhead than either RCBR or HPBR.
and requires significant programmer effort [opposing P1, P3].

**QSBR and EBR** both leverage an observation that, in many data structures, threads do not carry pointers obtained in one data structure operation forward for use in a subsequent operation. QSBR and EBR each have a simple interface in which the programmer needs only invoke a specific operation at the start and end of a data structure operation. Unlike the approaches above, QSBR and EBR avoid all per-record and per-access overheads. A thread can reclaim its garbage records whenever it detects that all other threads have started a new data structure operation (and hence forgotten all pointers to said garbage records). However, in the event that a thread halts or is delayed, the amount of unreclaimed garbage can grow unboundedly (eventually exhausting system memory) [opposing P2].

The algorithm DEBRA+ that we described above was introduced by Brown in 2015 [11]. In the same paper, an algorithm called DEBRA was proposed which, to best of our knowledge, is the fastest EBR algorithm. DEBRA does not bound number of unreclaimed records (garbage), but was shown to be faster than DEBRA+. Note that our experiments show NBR+ outperforms DEBRA.

After DEBRA, numerous hybrid algorithms having the bounded garbage property appeared. For example, Hazard Eras (HE) [33], Interval Based Reclamation (IBR), which was inspired by HE [36], and Wait Free Eras (WFE), which built on HE. All of these algorithms use per-record metadata to encode the times at which a record is allocated and unlinked, and similar instrumentation is required in HPs [opposing P3]. HPs require per-record reserve and unreserve calls and fallback code in case the reservation fails. It is unclear how HE, IBR and WFE could be used with data structures that allow traversing unlinked records [opposing P5]. As we will see in our experiments, these algorithms also incur non-trivial overhead [opposing P1].

Various other algorithms utilize operating system features such as forced context switches [5], POSIX signals [3, 4], and hardware transactional memory [2, 20]. Qsense [5] is a hybrid algorithm that uses QSBR as a fast code path, and HPs with forced context switches as a slow code path to guarantee the bounded garbage property. However, in the event of long thread delays, reclamation can only proceed on the slow path, which is as slow as HPBR. It has been shown to be slower than EBR [5] [opposing P1]. None of [2, 4, 5] can be used with data structures that allow threads to traverse unlinked records [opposing P5]. Forkscan [3] addressed this issue in ThreadScan, but (like ThreadScan) assumes the programmer will not use advanced pointer arithmetic techniques (or implicit pointers) [opposing P3]. Forkscan has been shown to be slower than HPs in several workloads [3] [opposing P1].

Optimistic Access (OA) and Automatic Optimistic Access (AOA) [13, 14] proposed a particularly interesting approach: optimistically allow threads to *accesses reclaimed nodes*, and verify *after the fact* that the access was safe. This requires an assumption that either (a) memory will not be freed to the OS, or (B) any resulting trap/exception (such as a segmentation fault) will be caught and handled [opposing P3]. Additionally, OA requires the programmer to transform data structures into a *normalized form* [35] (which is similar to, but not the same as, the form we assume in this paper), and instrument every read/write/CAS [opposing P3]. OA automates this transformation with compiler support (for data structures that have a normalized form). Unfortunately, it doesn’t appear that AOA has been ported to modern compilers. The need for a normalized form was eliminated in Free Access (FA) [12], which proposed another compiler extension to perform automatic instrumentation surrounding writes, and blocks of consecutive independent reads. FA is a general technique that has been shown to be comparable to HPBR [12] [opposing P1]. In contrast, our work targets applications that can benefit from the higher performance handcrafted SMR.

### 3 Model

We consider an n-thread asynchronous shared memory system. Threads can perform atomic read, write, compare-and-swap (CAS) and fetch-and-add (FAA). A data structure consists of a set of records which are accessible from a root (e.g., the head of a list). A record can be viewed as a set of fields. Each record can be in one of the five states throughout its lifecycle: (1) *allocated*: record allocated from heap but not accessible through the root, (2) *reachable*: all threads can reach the record by following references from the root, (3) *unlinked*: is not reachable from root but threads may still have references to it, (4) *safe*: a record is unlinked and no thread has a reference to it and (5) *reclaimed*: a record is returned to OS for reuse. In states 3 and 4, a record is *garbage*, but may be accessed by other concurrent threads.

### 4 NBR

#### 4.1 Assumptions on the data structure

NBR requires that a data structure should offer operations that have (or can be restructured into) the form as shown in Figure 1. This is mainly due to its neutralization mechanism where a thread that hasn’t written to shared memory yet is...
required to restart. In order to facilitate correct integration of NBR in a data structure we discuss general rules/pitfalls that a programmer should follow/avoid in the ensuing text.

**Phase 0:** preamble. Accesses (reads/writes/CASs) to global variables are permitted. System calls (heap allocation/deallocation, file I/O, network I/O etc.) are permitted. Access to shared records, for example, nodes of a shared data structure, is not permitted.

**Phase 1:** \( \Phi_{read} \) (read phase). Reading global variables is permitted and reading shared records is permitted if pointers to them were obtained during this phase (i.e., traversing a sequence of shared objects by following pointers starting from a global variable). Writes/CASs to shared records, writes/CASs to shared globals, and system calls, are not permitted.

For example, if a thread allocates a node, say using `malloc`, within the \( \Phi_{read} \) and before it could even use or free the node, it gets neutralized. This would cause memory leak. Alternatively, if the thread invokes an arbitrary call to `free` and subsequently gets neutralized to execute the \( \Phi_{read} \) again then it would cause the double free error.

Similarly, lets say, a thread attempts two consecutive writes on shared records of some data structure, for instance, changing pointers while un-linking a node in a list. If this thread gets neutralized after just completing the first write (partial update) then it could corrupt the shared data structure.

Additionally, writes to thread local variables are also not permitted. To see why, again assume, some thread uses a thread local list to maintain some meta data. now, neutralization of the thread in middle of an update on its list could corrupt the structure of the list.

**Phase 2:** reservation. This is a conceptual stage that does not necessarily correspond to any data structure code. However, this is where a key NBR operation will be invoked. At this point, one must be able to identify all shared objects that will be modified by the operation in the next phase, so they can be provided to NBR. We call these reserved records.

**Phase 3:** \( \Phi_{write} \) (write phase). Accesses (reads/writes/CASs) to global variables, and system calls, are permitted. Accesses (including write/CAS) to shared records are permitted only if the records are reserved. To understand what could go wrong if the last rule is violated we need to understand NBR, so we will return to this rule with an example in Section 4.4.

Finally, threads not executing a data structure operation are said to be in a quiescent phase.

After explaining form of a data structure operation required by NBR and general rules that one should be aware of, we now move to explain our methodology in the ensuing sections.

---

Note, in some cases it is okay to do writes to a thread local variable or data structure. For example, if a thread wants to count how many times it gets neutralized while executing a \( \Phi_{read} \) then neutralization of this thread would not impact correctness of writes to this thread local count variable and doing this is correct.

### 4.2 Overview of NBR

In NBR, each thread accumulates records that it has unlinked in a private buffer. When the size of a thread \( T \)'s buffer exceeds a predetermined threshold, the thread sends a neutralizing signal to all other threads. Upon receipt of such a signal, the behavior of a thread \( T' \) depends on which phase it is executing in.

If \( T' \) is in a quiescent phase, or preamble (Phase 0), it holds no pointers to shared records, and does not prevent \( T \) from reclaiming records in its buffer. \( T' \) simply continues executing and can safely ignore the signal.

On the other hand, if \( T' \) is in \( \Phi_{read} \), it may hold pointers to records in \( T \)'s buffer (also called limboBag). Thus, preventing \( T \) from reclaiming memory. Note, however, that \( T' \) has not yet performed any modifications to any shared records (since it is still in \( \Phi_{read} \)). So, \( T' \) can simply discard all of its privately held pointers, jump back to the start of its \( \Phi_{read} \), without leaving any shared records in an inconsistent state. To implement this jump, every data structure operation invokes `sigsetjmp` at the start of its \( \Phi_{read} \), which creates a checkpoint (saving the values of all stack variables). A thread can subsequently invoke `siglongjmp` to return to the last place it performed `sigsetjmp` (and restore the values of all stack variables). It can then retry executing its \( \Phi_{read} \), traversing a new sequence of records, starting from the root, without any risk of accessing any records `free`ed by \( T \) (since those are no longer reachable).

The subtlety in NBR arises when \( T' \) is in \( \Phi_{write} \). In this case, \( T' \) may hold pointers to records in \( T \)'s buffer. Thus, preventing \( T \) from reclaiming memory. However, since \( T' \) may have modified some shared records (but not completed its operation yet), we cannot simply restart its data structure operation, or we may leave the data structure in an inconsistent state. So, \( T' \) will not restart its operation. Instead, it will simply continue executing wherever it was when it received the signal. At this point, the reader might wonder how we simultaneously avoid:

- **a.** Blocking the reclamation of \( T \), and
- **b.** The possibility that \( T' \) continues executing and one of the records it is about to access is concurrently `free`ed by \( T \).

The solution lies in the reservation phase (Phase 2) of \( T' \). During the reservation phase of \( T' \), just before it begins its \( \Phi_{write} \), \( T' \) reserves all of the shared records it will access in its \( \Phi_{write} \) by announcing pointers to them in a shared array. These reservations serve a similar purpose to hazard pointers, but still different than HP in terms of performance and safety guarantees which is discussed in Section 5.3. By the time \( T' \) is in its \( \Phi_{write} \) (so it will not `siglongjmp`), its reservations are visible to all threads, and \( T \) can refer to this information to avoid reclaiming any of those records.

In short, operations in the \( \Phi_{read} \) discard their pointers and restart, and operations in the \( \Phi_{write} \) must have reserved
Algorithm 1 NBR. Assumes, max number of reservations are less than the limboBag size.

| thread local variable: |
|------------------------|
| 1: int tid \( \triangleright \) current thread id |
| 2: record *limboBag \( \triangleright \) per-thread list of unlinked records. Maxsize:S |
| 3: bool restartable \( \triangleright \) local var to track \( \Phi_{\text{read}}/\Phi_{\text{write}} \) |
| 4: record *tail \( \triangleright \) Pointer to last record in limboBag |

| shared variable: |
|------------------|
| 5: atomic<record*> reservations[N][R] \( \triangleright \) N#threads, R:max reserved records. [R] \( \ll \) [S]. |

6: procedure BEGINS( \( \Phi_{\text{read}}( ) \) |
7: reservations[tid].clear(); |
8: CAS(&restartable, 0, 1); |
9: end procedure |

10: procedure END\( \Phi_{\text{read}}(r_
1 rec_1, r_
1 rec_2 \cdots r_
1 rec_R) \) |
11: reservations[tid] = \{r_
1 rec_1, r_
1 rec_2 \cdots r_
1 rec_R\}; |
12: CAS(&restartable, 1, 0); |
13: end procedure |

14: procedure RETIRE(rec) |
15: if isLimboBagTooLarge() then |
16: signalAll( ); |
17: reclaimFReable(tail); |
18: end if |
19: limboBag[tid].append(rec); |
20: end procedure |

21: procedure RECLAIMFREEABLE(tail) |
22: \( A = \text{collectReservations}( ); \) |
23: \( R = \text{limboBag[tid].remove}(A, \text{tail}); \) |
24: free({\( R \)}); |
25: end procedure |

them. This empowers the reclaimers to assume that readers lose all of their pointers in response to neutralizing, and the writers lose all pointers that are not reserved. As a result, we get safe memory reclamation.

4.3 Implementation of NBR

Algorithm 1 shows the pseudocode for NBR. Each thread collects unlinked records in its limboBag (line 2), and maintains a local restartable variable that indicates whether the thread should jump back to the start of its \( \Phi_{\text{read}} \) in the event that it receives a neutralization signal (line 3). We say the thread is restartable if restartable is true (1), and non-restartable otherwise. Additionally, each thread, before entering the \( \Phi_{\text{write}} \), reserves maximum number of records it could access at a \( \text{swmr} \) (single write multi read) reservations array (line 5). We assume number of such reserved records \( R \) are strictly less than maximum size of limbo bags.

A thread in \( \Phi_{\text{read}} \) clears its reservations (if any), and then changes restartable to true using a CAS (Line 8). This CAS might initially look very strange to the astute reader, since it is performed on a single-writer variable and cannot fail. The CAS prevents instruction reordering on the x86-64 architecture (additional fences may be needed for more relaxed memory models). More specifically, the goal of CAS at line 8 is to ensure that a thread \( T \) becomes restartable before any subsequent reads of shared records. If this CAS were simply an atomic write (rather than a read-modify-write instruction), it would be possible for \( T \)'s reads of shared records to be reordered before this write. In other words some reads of shared records in \( \Phi_{\text{read}} \) may appear to occur in \text{preamble} (or previous \( \Phi_{\text{write}} \)) due to instruction reordering. Note, this would break the rule that says access to shared records is not permitted in \text{preamble} (phase 0) as discussed in Section 4.1.

As a result, the thread, which is not yet restartable, might ignore a neutralization signal and access a \( \text{read} \) record. A thread \( T \) in \( \Phi_{\text{write}} \) announces a set of reservations, and then changes restartable to false using CAS (Line 12). This CAS is used to broadcast the reservations to other threads. More specifically, the CAS, by \( T \), at line 12 implies a memory fence, which ensures that all of the reservations announced in the previous line 11 are visible to other threads before \( T \) changes restartable to false. If this CAS were a simple write, it would be possible that a reclaime may miss observing some reservations of \( T \), and erroneously free those records. In other words, the following wrong execution may occur if CAS is not used: a thread \( T \) reserves record \( r \) and writes (note, not doing a CAS) restartable to 0. Concurrently, another thread \( T' \) sends a neutralizing signal to \( T \) and tries to free \( r \). Note, that it’s possible that the reservation of \( r \) by \( T \) may not be visible to \( T' \) yet. Thus \( T \) after changing restartable to 0 (i.e entering the \( \Phi_{\text{write}} \)) may access freed record \( r \).

The retire operation (line 14) begins by checking (line 15) whether the size of the limbo bag is above a predetermined threshold (32k in our experiments). If so, it sends a neutralizing signal to all threads using signalAll (Line 16), and then proceeds to reclaim all safe records (line 17). Otherwise, it simply adds \( r \) to \( \text{limboBag} \).

The reclaimFreeable procedure frees all records (up to the last record pointed by thread local pointer, \( \text{tail} \)) in the \( \text{limboBag} \) that are not reserved (line 21). It first scans reservations array of all other threads and collects the reserved records in set \( A \) (line 22). Then removes the retired records, which are not in \( A \) (set of reserved records), up to the \( \text{tail} \) of \( \text{limboBag} \) using remove(\( A \), \( \text{tail} \)) method at line 23. And, finally frees the safe set of records \( R \) at line 24.

After discussing the implementation of NBR we can now elaborate on how readers, writers and reclaimers collaborate to achieve safe memory reclamation.

\(^{3}\) Instead of doing a CAS, which has an implicit mfence, we can also use more efficient _sync_lock_test_and_set() which has an implicit xchg instruction. Please refer to section 11.5.1 of the manual: https://www.amd.com/system/files/TechDocs/47414_15h_sw_opt_guide.pdf
4.3.1 Reader-reclaimer handshake. Each thread $T'$ at the time of BEGIN, saves its execution state using sigsetjmp so that a restartable thread can jump back to this state upon receiving a neutralizing signal. When a reclaimer $T$ sends a neutralization signal to thread $T'$, the operating system causes the control flow of $T'$ to be interrupted, so that $T'$ will immediately execute a signal handler if $T'$ is currently running. (Otherwise, if $T'$ is not currently running, the next time it is scheduled to run it will execute the signal handler before any other steps.) The signal handler determines whether $T'$ is restartable by reading the local restartable variable. If the thread is restartable, then the signal handler will invoke siglongjmp and jump back to the start of the $\Phi_{\text{read}}$ (so it is as if $T'$ never started the $\Phi_{\text{read}}$).

This behaviour represents a sort of two-step handshake between readers (threads in $\Phi_{\text{read}}$) and reclaimers (threads executing lines 16 and 17 in retire() to avoid scenarios where a reader might access a freed record. A reclaimer guarantees that before reclaiming any of its unlinked records it will signal all threads, and all readers guarantee that they will relinquish any reference to unsafe records when they receive a neutralization signal.

4.3.2 Writers handshake. (1) Each reclaimer signals all threads before starting to reclaim any records. When a writer receives a signal, it executes a signalHandler that determines the thread is non-restartable, and immediately returns. The reclaimer then goes on to reclaim its limboBag (line 17), independent of the actions of the writer.

This is safe because a writer, before entering into the $\Phi_{\text{write}}$, reserves all of the shared records it will access in its $\Phi_{\text{write}}$ (line 11). Thus, (2) the writer guarantees to the reclaimer that, although it will not restart its data structure operation, it will only access reserved records. The (3) reclaimer, in turn, guarantees it will scan all announcements after signaling and before reclaiming the contents of its limboBag, and will consequently avoid reclaiming any records that will be accessed by the writer in its $\Phi_{\text{write}}$.

This three-step handshake formed by (1), (2) and (3) avoid scenarios where a writer might access a freed record. Crucially, all writers atomically ensure that their reserved records are visible to the reclaimer at the moment they become non-restartable. In turn, reclaimers scan reservations after sending neutralization signals (at which point any thread that does not restart has already made its reservations visible).

4.4 Revisiting rule in $\Phi_{\text{write}}$

In this section, we will trace an incorrect execution that could occur if a thread accesses records not reserved before entering $\Phi_{\text{write}}$.

Suppose, a thread $T$, in $\Phi_{\text{write}}$, is about to access a shared record $rec$, which was not reserved by it, and sleeps. Simultaneously, another thread $T'$ sends a neutralization signal to $T$ using signalAl11. Then, $T'$ scans the reservations array of the thread $T$. Note, $T$ did not reserve $rec$ so $T'$ will not find $rec$ in $T$'s reserved records (breach of writers handshake, Section 4.3.2). Therefore, $T'$ will assume that $rec$ could be freed safely. Eventually, $T'$ frees $rec$. Now, if $T$ wakes up and proceeds with its access of $rec$ it may crash.

5 NBR+

Now, we explain performance issue with NBR which motivated us to design NBR+.

Performance bottleneck in NBR. Signals on linux trigger page-fault routines and a switch from user to kernel mode incurs significant cost to speedup. Therefore, it is desirable to send as few signals as possible (while maintaining high reclamation throughput). However, every time a thread reclaims records from its limboBag, NBR requires the thread to send signals to all other threads. We call this a reclamation event. As a result, for every thread to go through exactly one reclamation event, $n(n-1)$ signals must be sent, where $n$ is the number of threads. This incurs significant overhead of $O(n^2)$ signals and severely limits the performance.

This motivated us to investigate how can we reduce the total number of threads that are required to be sent. We observed that in NBR, at any time $t$, a thread sends $n-1$ signals to reclaim its limboBag. This causes all other $n-1$ threads to discard any un-reserved references to shared records. Meaning, at time $t$, the retired records in limbo bags of all threads are either safe to free or reserved. Therefore, if somehow we could propagate this information that a reclamation event has occurred due to some thread $T$, then all other threads could piggyback on $T$ to partially or completely reclaim there own limbo bags without sending signals of their own. In other words, in the best case, all $n$ participating threads could carry out exactly one reclamation event in just $n-1$ signals. However, in the worst case $O(n^2)$ signals may be required to be sent for each thread to reclaim exactly once.

We formalize this event when each record is either safe to be freed or is reserved at a reclamation event using the notion of relaxed grace period. This also serves as crux of our next algorithm NBR+.

**Definition 1.** Relaxed grace period (RGP): A time interval $[t, t']$ during which each thread is neutralized due to a reclamation event triggered by some reclaimer thread.

**Overview of NBR+** The key insight in NBR+ is the observation that at each reclamation event, a reclaimer sends neutralization signals to all threads, and the resulting relaxed grace period causes all threads to discard all of their pointers to unreserved records. So, hypothetically, a single signal broadcast could empower all threads to reclaim some of their records, as long as they can reason about which of their records were unlinked before that relaxed grace period began. As a result, often we are able to allow threads to reclaim memory without sending any signals.
We explain the design of $\text{NBR}^+$ when the limboBag is full, we say that thread is at the HiWatermark. If a thread’s limboBag keeps growing without reclamation it will first cross the LoWatermark and then hit the HiWatermark. As shown in Algorithm 2, a thread determines whether it has passed the HiWatermark or LoWatermark using procedures $\text{isAtHiWm}$ (line 6) and $\text{isAtLoWm}$ (line 12).

To tackle (C2), the reclaimer at the LoWatermark (who wants to detect an RGP) must perform a handshake with another reclaimer at the HiWatermark (who triggers an RGP). $\text{NBR}^+$ implements this handshake using per-thread single-writer multi-reader timestamps (similar to vector clocks).

Whenever a reclaimer hits the HiWatermark, it first increments its timestamp (to an odd value) to indicate that it is currently broadcasting signals (line 7). This depicts beginning of an RGP. It then sends signals to all threads, and increments its timestamp again (to an even value) to indicate that it has finished broadcasting signals (line 9). This indicates the end of the RGP.

On the other hand, whenever a reclaimer $T$ passes the LoWatermark, it collects and saves the current timestamps of all threads (line 15), as well as the current tail pointer of its limboBag (line 14), so it can remember precisely which records it had unlinked before it reached the LoWatermark. $T$ then periodically collects the timestamps of all threads, comparing the new values it sees to the original values it saw when it passed the LoWatermark (line 17 - line 23). It continues to do this until it either detects a relaxed grace period or hits the HiWatermark itself (and triggers its own relaxed grace period). Observe that if the timestamp of any thread changes from one even number to another, then that thread has both begun and finished sending signals to all threads since $T$ hit the LoWatermark. Thus, $T$ could identify that an RGP has occurred, solving (C2).

Finally, to tackle (C3), observe that $T$ saves the last record (tail of its limboBag) it had retired before entering the LoWatermark at line 14. If $T$ successfully observes an RGP as explained in the solution to (C2), then all threads would either have discarded or reserved all their private references to the records in $T$’s limboBag up to the saved bookmarkTail making these records safe to reclaim. Thus, $T$ can invoke reclaimFreeable to free all un-reserved records up to the bookmarkTail (line 19), solving (C3).

The $\text{firstLoWmEntryFlag}$ (line 27) method is used to set firstLoWmEntryFlag after a thread reclaims either at LoWatermark (line 20) or at HiWatermark (line 11) to prepare it for subsequent reclamation.

Note, scanning $\text{announceTS[]}$ would incur overhead in terms of cache misses. This cost could be amortized over multiple $\text{retire}$ operations by scanning $\text{announceTS[]}$ after a fixed number of calls to $\text{retire}$ have been made.

---

Algorithm 2 $\text{NBR}^+$: Only variables that differ from $\text{NBR}$ are shown here. $\text{NBR}^+$ includes all variables and procedures of Algorithm 1. The retire operation is different in $\text{NBR}^+$.

The helper procedures required by $\text{NBR}^+$ are self explanatory.

5.1 Implementation of $\text{NBR}^+$

We explain the design of $\text{NBR}^+$ by building our exposition around three challenges that should be tackled in the order they are written.

(C1) When should a thread start tracking other threads for occurrence of an RGP event?

(C2) How can a thread recognize that an RGP event has occurred?

(C3) Once a thread recognizes that an RGP event has occurred how should it determine which records in its limboBag are safe to reclaim?

As a solution to (C1), each thread in $\text{NBR}^+$, in addition to watching the limboBag size to determine when it becomes too large (triggering a reclamation event), also determines when the limboBag size crosses a predetermined threshold called the LoWatermark (e.g., one half full or one quarter full). If a thread’s limboBag is full, we say that thread is at the HiWatermark.

As a challenge to (C2), the reclaimer at the LoWatermark (who wants to detect an RGP) must perform a handshake with another reclaimer at the HiWatermark (who triggers an RGP). $\text{NBR}^+$ implements this handshake using per-thread single-writer multi-reader timestamps (similar to vector clocks).

Whenever a reclaimer hits the HiWatermark, it first increments its timestamp (to an odd value) to indicate that it is currently broadcasting signals (line 7). This depicts beginning of an RGP. It then sends signals to all threads, and increments its timestamp again (to an even value) to indicate that it has finished broadcasting signals (line 9). This indicates the end of the RGP.

On the other hand, whenever a reclaimer $T$ passes the LoWatermark, it collects and saves the current timestamps of all threads (line 15), as well as the current tail pointer of its limboBag (line 14), so it can remember precisely which records it had unlinked before it reached the LoWatermark. $T$ then periodically collects the timestamps of all threads, comparing the new values it sees to the original values it saw when it passed the LoWatermark (line 17 - line 23). It continues to do this until it either detects a relaxed grace period or hits the HiWatermark itself (and triggers its own relaxed grace period). Observe that if the timestamp of any thread changes from one even number to another, then that thread has both begun and finished sending signals to all threads since $T$ hit the LoWatermark. Thus, $T$ could identify that an RGP has occurred, solving (C2).

Finally, to tackle (C3), observe that $T$ saves the last record (tail of its limboBag) it had retired before entering the LoWatermark at line 14. If $T$ successfully observes an RGP as explained in the solution to (C2), then all threads would either have discarded or reserved all their private references to the records in $T$’s limboBag up to the saved bookmarkTail making these records safe to reclaim. Thus, $T$ can invoke reclaimFreeable to free all un-reserved records up to the bookmarkTail (line 19), solving (C3).

The firstLoWmEntryFlag (line 27) method is used to set firstLoWmEntryFlag after a thread reclaims either at LoWatermark (line 20) or at HiWatermark (line 11) to prepare it for subsequent reclamation.
The thread which hasn’t reached either at LoWatermark or HiWatermark simply keeps on appending the retired record in its limboBag (line 25).

At first it may appear that a thread \( T \) can reclaim its limboBag as soon as it receives a neutralizing signal from a reclamer thread \( T' \). However, this alone is not enough for \( T \). Precisely, \( T \) needs to know whether all other threads have also been neutralized or not and thus all threads have discarded any unreserved private reference in \( T' \)’s limboBag. This information is provided when a thread observes an RGP.

Let us trace an example explaining what could go wrong if a thread reclaims its limboBag when it receives a signal. Assume, a system with three threads \( T_1, T_2 \) and \( T_3 \). \( T_1 \) is at HiWatermark, \( T_2 \) is at LoWatermark and \( T_3 \) holds a private reference to a record \( r_c \) in \( T_2 \)’s limboBag. \( T_1 \) being at HiWatermark starts neutralizing all threads one by one. First, it sends neutralizing signal to \( T_2 \) starting an RGP. \( T_2 \) upon receiving the signal reclaims its limboBag including \( r_c \). Note, that \( T_1 \) hasn’t neutralized \( T_3 \) yet, meaning RGP hasn’t occurred yet. Now, if \( T_3 \) accesses the \( r_c \) an use-after-free would occur. To prevent this, \( T_2 \) should not reclaim its limboBag unless \( T_1 \) ends the RGP by neutralizing \( T_3 \), marking the end of the RGP, after which \( T_2 \) scanning for a grace period at LoWatermark would successfully pass the check at line 18 to reclaim opportunistically. Thus, a thread can reclaim its limboBag iff a relaxed grace period has occurred, meaning all threads have been neutralized.

### 5.2 Applicability

In this section we will discuss general patterns of lock-free algorithms in context of our proposed reclamer to help readers appreciate the breadth of the class of data-structures that NBR can be used with, as well as some of the nuances in its applicability.

**A compatible pattern:** Herlihy [28], for the first time, showed that most of the data structures could be transformed into lock-free counterparts using optimistic concurrency control. Meaning they exhibit a synchronization-free search phase followed by an update phase that includes an atomic validation and update (Note: we deliberately use search/update phase instead of read/write phase in order to avoid confusion between NBR’s \( \Phi_{read} \) and \( \Phi_{write} \)). In such data structure it can be easily seen that synchronization-free searches correspond to \( \Phi_{read} \) and validation followed by updates corresponds to \( \Phi_{write} \). Thus, to use NBR one just needs to invoke \( \text{BEGIN} \Phi_{read} \) and \( \text{END} \Phi_{read} \) (marks beginning of a \( \Phi_{write} \)) at corresponding phases identified in such data structure.

For example, consider a lazy-list, it has a \( \Phi_{read} \) to find target records ((\( \text{pred}, \text{curr} \)) followed by a \( \Phi_{write} \) where a thread acquires locks on \( \text{pred} \) and \( \text{curr} \), validates if \( \text{pred} \) is not marked and its reference field still points to \( \text{curr} \) and updates the data-structure before releasing locks. This algorithmic structure maps trivially onto the phases used by NBR.

It is appropriate to point here that certain fast and bounded garbage based SMR algorithms such as DEBRA+ do not support such optimistic lock-based algorithms. If DEBRA+ is used with such an algorithm, it may neutralize a thread that holds a lock, causing it to restart its operation while holding locks, resulting in deadlock or corrupting the data structure. In contrast, such optimistic lock-based data structures can leverage NBR as by definition threads holding locks are in the \( \Phi_{write} \) and thus will not be restarted. Infact, NBR is applicable to any data structure that could be implemented using Herlihy’s universal methodology to write lock-free algorithms [28].

**An incompatible pattern:** Certain other lock-free data structures exhibit a pattern where search phase performs auxiliary updates while it is traversing towards some target record(s). Then, once it reaches its target record(s), it performs a final update phase. For example, in Harris’s lock-free list [25], a thread in the search phase modifies the list by unlinking any marked (logically deleted) records that it encounters while traversing towards its target location. Then, once it arrives at the target location, it performs an update phase in which the operation’s intended modification is performed.

For this kind of data structure, let us see why NBR and NBR+ cannot easily be used. Consider a list insert operation. To use NBR(+) we must be able to express the insert operation as a single \( \Phi_{read} \) and a single \( \Phi_{write} \). If we insert a \( \Phi_{write} \) to perform an auxiliary update, then after performing the auxiliary update we are stuck in a \( \Phi_{write} \) and cannot obtain pointers to any new nodes without exiting the \( \Phi_{write} \) and beginning a new data structure operation. If not, threads doing auxiliary updates might be neutralized in the middle of said updates and corrupt the data structure. To deal with data structures in such form, we discuss how NBR (+) could be easily applied by visualizing the data structure operation consisting of multiple NBR(+) read/write phases. Henceforth, we would refer to such application of NBR (+) as k-NBR (+) and would use the terms interchangeably.

**k-NBR (+):** We explain k-NBR (+) by using the aforementioned lock-free list as an example. Intuitively, an update operation in the list begins its first \( \Phi_{read} \), then subsequently encounters a marked node while traversing towards a pair of target nodes. This update must perform an auxiliary update, then after performing the auxiliary update we are stuck in a \( \Phi_{write} \) and cannot obtain pointers to any new nodes without exiting the \( \Phi_{write} \) and beginning a new data structure operation. If not, threads doing auxiliary updates might be neutralized in the middle of said updates and corrupt the data structure. To deal with data structures in such form, we discuss how NBR (+) could be easily applied by visualizing the data structure operation consisting of multiple NBR(+) read/write phases. Henceforth, we would refer to such application of NBR (+) as k-NBR (+) and would use the terms interchangeably.

Algorithm 3 sketches an example implementation of the insert operation of the list with k-NBR (+). Consider the following execution of this list algorithm. Suppose the initial configuration is \( L: 1_f \Rightarrow 2_f \Rightarrow 3_f \Rightarrow 4_f \Rightarrow 6_f \Rightarrow 10_f \), where each node is represented as keymarked (where
Algorithm 3 Example showing integration of NBR with harris list [25] having multiple read/write phases, $Φ_{read}(Φ_{read}Φ_{write})Φ_{write}$.

```plaintext
bool insert(key) {
    Node *right_node, *left_node;
    do{
        beginΦ_{read}();
        right_node = search (key, &left_node);
        if(((right_node != tail) && (right_node.key==key))
            return false;
        endΦ_{read}(left_node, right_node);
        Node *new_node = new Node(key);
        new_node.next = right_node;
        if(CAS(&(left_node.next), right_node, new_node))
            return true;
    } while (true);
}
beginΦ_{read}();
Node* = head;
Node *t_next = head.next;
beginΦ_{read}();
Node *t = head;
Node *t_next = head.next;
do{
    if(is_marked_reference(t_next)) {
        t = get_unmarked_reference(t_next);
        if(t == tail)
            break;
        t_next = t.next;
    }
    while(is_marked_reference(t.next) or (t.key < search_key));
    right_node = t;
    if((left_node.next == right_node)
        return right_node;
    else
        goto search_again;
}
endΦ_{read}(left_node, right_node);
if(CAS(&(left_node.next), left_node.next, right_node))
    goto search_again;
else
    return right_node;
} while(true);
```

marked is [true or false]. Now, suppose an operation \texttt{Ins.insert(9)} on $L$ begins a $Φ_{read}$ (line 4) for the \texttt{Ins} which invokes a search on $L$ which begins first $Φ_{read}$ of the search() (line 20) starting from $\langle \text{pred, curr} \rangle = (1_f, 2_f)$ and observes $\langle \text{pred, curr} \rangle = (2_f, 3_s)$, where $\text{curr} = 3_s$ is marked. To remove marked node $3_s$, search enters a $Φ_{write}$ (line 39) and changes the next pointer of $2_f$ to $4_f$, yielding list configuration: $1_f \Rightarrow 2_f \Rightarrow 4_f \Rightarrow 6_f \Rightarrow 10_f$. Moving forward, the search() will enter a second $Φ_{read}$ (line 20), and traverse the list again, starting from the root. As search() now obtains pointers to new nodes (which would be impossible in the regular NBR(+), algorithm, where you have only one $Φ_{read}$), we must argue that it doesn’t access any freed nodes. This is straightforward, since search() is again traversing from the root. Now, suppose \texttt{Ins} is neutralized by a concurrent thread, while it is in this second $Φ_{read}$ of its search(). Upon receipt of a neutralization signal, the thread performing search() will jump back to the beginning of its second $Φ_{read}$, and restart its search, once again, from the root. Suppose, search() eventually performs a $Φ_{read}$ that reaches its desired nodes $(\text{pred, curr}) = (6_f, 10_f)$ (line 38). It will then enter a final $Φ_{write}$ (line 8) and insert $9_f$ yielding $L$: $1_f \Rightarrow 2_f \Rightarrow 4_f \Rightarrow 6_f \Rightarrow 9_f \Rightarrow 10_f$.

In order for $k$-NBR to be safe, it is crucial that \texttt{Ins} forgets all pointers and restarts from the root every time it begins a new $Φ_{read}$. Intuitively, this is because each new read phase is effectively a new data structure operation—all pointers are forgotten when the new $Φ_{read}$ begins. If it attempts to continue searching from somewhere in the middle of the list, perhaps by restarting its search from a shared node $R$ that was reserved by the previous $Φ_{write}$, then \texttt{Ins} could easily dereference a freed node. To see why, note that, although $R$ cannot be freed (since it is reserved), the nodes that it points to are not necessarily reserved, and so they could be freed. Thus, as soon as \texttt{Ins} follows any pointer starting from $R$, it could access a freed node and crash.

Revisiting the incompatible pattern: Having seen how the lock-free list algorithm can be divided into multiple NBR (+) friendly phases, we can observe the following. Petrank et al. [35] introduced a lock-free normalized form that many prominent lock-free algorithms can be transformed into. This form expresses a lock-free data structure operation as a sequence of three procedures, namely, a CAS generator, a CAS executor and a Wrap-Up. The CAS generator represents a typical search phase in a lock-free data structure wherein a thread traverses towards some target record(s), perhaps occasionally helping other threads along the way by doing auxiliary CASes (similar to the removal of marked nodes in the lock-free list while searching), and finally returns a descriptor, which can be used as an input to the CAS executor. Crucially, the CAS generator can safely restart its entire search starting from the root of the data structure, every time it helps another thread. The CAS executor attempts to perform the desired modification to the data structure, and the Wrap-Up removes any temporary modifications to the data structure intended to facilitate helping of this operation.

From the perspective of applying $k$-NBR (+) to such a data structure, the normalized form’s CAS generator can be implemented as a sequence of pairs of $Φ_{read}$ and $Φ_{write}$, one pair for each auxiliary (helping) update, with each $Φ_{read}$ consisting of a new search starting from the root of the data structure. The CAS executor can be implemented as a $Φ_{write}$ at the end of the final $Φ_{read}$ started by the CAS generator. The Wrap-Up can be implemented as a sequence of pairs of $Φ_{read}$ and $Φ_{write}$ (if it must perform helping along the way), with each $Φ_{read}$ consisting of a new search starting from the root, followed by a terminal $Φ_{write}$ to perform the final cleanup. Thus, we can leverage the normalized form to
transform many complex lock-free algorithms into a form to which NBR can be applied.

We evaluate the performance of NBR+ with two prominent data structures having multiple read/write phases and results show that NBR+ is fast.

5.3 Usability

Figure 2 compares the complexity of using HP, NBR and DEBRA in the insert operation of a lazy list. As is evident in Figure 2c, HP is difficult to use because it requires a programmer to protect every record by announcing hazard pointers at a swmr location, using store/load fence or xchg to ensure that each announcement is observed timely by subsequent reads by other threads, validating that the announced record is still safe before dereferencing it, and restarting if validation fails. Programmers also need to unprotect records that they will no longer dereference, further increasing the need for intrusive code changes.

On the contrary, applying NBR to a data structure operation is, intuitively similar to performing two-phased locking, in the sense that the primary difficulty revolves around identifying where the Φ_write should begin, and which records it will access. As shown in Figure 2b, the programmer just needs to invoke beginΦ_read before the operation accesses its first shared record, for example, at the start of the traversal for target records in a lazy-list. And, invoke endΦ_read before modifying any shared record, for example, in the lazy-list the Φ_write begins just before the lock acquisition on pred. If there are no modifications to be performed in an operation, for example, in the contains operation of a lazy-list, the programmer can simply invoke endΦ_read before returning from the operation.

DEBRA is simplest as it requires programmers to invoke just two functions corresponding to the start and the end of a data structure operation without worrying about the logic in the operation (Figure 2a).

In conclusion, we believe that NBR would only require medium programming effort lying in between the complex HP and simplest DEBRA. However, even though DEBRA is simplest, we believe that the benefits due to NBR’s bounded garbage property and better performance outweigh the extra effort of identifying which shared records will be modified by the Φ_write and at what location to invoke endΦ_read, to mark the beginning of a Φ_write.

Just to give readers a quantitative view of the amount of programming effort needed to use HP and NBR we measured number of extra reclamation related lines of code needed to be written in our implementation of insert(), delete() and contains() methods for the lazylist and DGT. We observed that NBR required only 10 extra lines of code in comparison to 30 extra lines of code needed to use HP.

As mentioned earlier, before entering its Φ_write NBR needs to reserve all the records that could be accessed in the Φ_write. One may have noticed that there could be concurrent data structures where it may not be possible to know all the records beforehand. Thus, NBR may not apply to such data structures. However, in some cases the issue has a simpler form which could be solved by just reserving all the records that could be accessed in Φ_write. For example, in trees a thread might access left or right child but it may not be clear beforehand which one would be accessed in the Φ_write. To correctly apply NBR in this case one may simply reserve both left and right child records.

Furthermore, HP have two main demerits. First, (D1) high overhead of publishing HP after every read of a shared record, which usually involves using xchg or mfence on x86 architecture. Second, (D2) HPs are not safe with data structures that allow traversal of marked nodes [11]. Since NBR use reservations like HPs one may think that it may also inherit drawbacks of HPs. However, despite the use of HP like reservations, NBR has neither of these drawbacks. Firstly, NBR only requires to publish its reservations once per operation right before entering the Φ_write and not after every read of a shared record. Thus, it avoids high overhead of frequent publishing of reservations in HPs. Secondly, unlike HP, NBR(+) is safe with data structures that may require traversal over marked (or logically deleted) nodes due to reader-reclaimer and writers handshakes.

6 Correctness

We show that NBR and NBR+ is both safe and have bounded garbage. Proofs of these lemmas are provided in the appendix (Section A) accompanying this paper.

Lemma 1 (NBR is Safe). Reclaimer threads in NBR only reclaim records that are safe.

Lemma 2 (NBR+ is safe). Reclaimer threads in NBR+ only reclaim records that are safe.

Lemma 3 (Both NBR and NBR+ have bounded garbage). The number of unreclaimed records per thread is bounded.

Please note that NBR assumes that number of records that could be reserved per data structure operation are strictly less than than the limboBag size in order to be able to reclaim whenever the limboBag is full. Practically most of the data structures require only a small number of reservations per operation. For example in our experiments, we used lazylist [27] required maximum 2 reservations per operation and harris list [25], DGT [15], and (a,b) tree [9] needed to reserve maximum 3 records at a time.

7 Experimental Evaluation

Setup: We used a quad-socket Intel Xeon Platinum 8160 machine running at 2.1GHz with 192 hardware threads and 377 GiB memory having shared L3 cache (33.79 MiB) on Ubuntu 18.04 with gcc version 7.4.0.
we did not show it separately in Figure 3 even though it is using ABTree [9, chapter 8] and harris list [25].

NBR (HP) and a leaky implementation (none). Note, since uniform comparison.

pared k-

reclamation algorithms using DGT. Finally, for (E3) we compared peak memory usage of each of the aforementioned algorithms by max up to 10–50% (Figure 3a search-intensive workload).

Every trial is run for a thread range: 24, 48, 72, 96, 120, 144, 168, 192, 216, 240, 252. We used a maximum size of 2 M and 20 K for trees and lists (unless explicitly mentioned), respectively. Each execution starts by prefilling the data structure to half of its size, i.e 1 M for trees and 10 K for lists.

We subject NBR+ to exhaustive evaluation by running it under oversubscription on three workload profiles to establish P4:

1. update-intensive: 50% inserts and 50% deletes.
2. middle: 25% inserts, 25% deletes and 50% searches.
3. search-intensive: 90% searches, 5% inserts and 5% deletes.

Discussion Figure 3 shows that NBR+ is faster than all competitors. In the tree, it surpasses the next best algorithm, DEBRA, by up to ~38% and ~12% (Figure 3a, update-intensive and middle workloads, resp.) and is comparable to DEBRA in search-intensive workloads where it outperforms other algorithms by max up to 10–50% (Figure 3a search-intensive workload).

DEBRA performs better than NBR+ for low thread counts, but NBR+ outperforms it after 96 threads in update intensive workloads (Figure 3a, leftmost plot), after 120 threads in middle workload(Figure 3a, center), and is comparable in lookup intensive workload (Figure 3a, rightmost plot). The slow performance of DEBRA at higher number of threads could be attributed to infrequent advancement of epochs by slow threads which leads to halting of regular reclamation of limbo bags, also called delayed thread vulnerability. As a result limbo bags of all threads keeps on growing until the slow thread announces the required epoch.

The delayed thread vulnerability leads to accumulation of large number of retired records waiting to be reclaimed at

All algorithms used in the experiments were implemented in the Setbench [11] benchmark compiled with -O3 level optimization and use the jemalloc as the memory allocator [22].

We perform three kinds of experiments:

(E1): Evaluates scaling of NBR+ with the increasing number of threads to establish its speed [P1].

(E2): Evaluates peak memory usage of NBR+ to show that it has bounded garbage property [P2].

(E3): Evaluates throughput of k-NBR to show that even though it requires restarts from the root of a data-structure, k-NBR’s speed still compares reasonably to next best DEBRA, strengthening its case of applicability [P5].

For (E1) we picked the lazy-list [27] and DGT [15] as representative of lists and trees, the two popular categories of concurrent data structures to evaluate NBR+ against quiescent based reclamation (QSBR), RCU, DEBRA, 2geibr variant of interval based reclamation\(^4\) (IBR) [36], hazard pointers (HP) and a leaky implementation (none). Note, since NBR is the basic version we use to illustrate our methodology, we did not show it separately in Figure 3 even though it outperformed HP in our experiments. Similarly, for (E2) we compared peak memory usage of each of the aforementioned reclamation algorithms using DGT. Finally, for (E3) we compared k-NBR with DEBRA and the leaky implementation (none) using ABTree [9, chapter 8] and harris list [25].

Reported results are collected by averaging the output of each algorithm over 3 trials where each trial is of 5 seconds.

\(^4\)We adapt QSBR and RCU algorithms from the IBR benchmark along with their default configuration and compile them in Setbench’s framework for uniform comparison.
the announcement of the required epoch. Once the required epoch is announced by the slow thread all thread reclaim their large limbo bags leading to a reclamation burst. This harms the overall throughput as reclamation bursts could bottleneck the underlying allocator (jemalloc in our experiments) by triggering slow paths. The Probability of threads getting delayed increases as more threads get involved in high inter-socket and update-intensive computations, which becomes more dominant once hyper-threading also gets involved, which adds overhead of more context switches between two threads on same core.

Furthermore, one may notice that there is a shift in the threads at which NBR+ overtakes DEBRA from update intensive workload to search-intensive workloads. This could be attributed to the fact that the overhead of burst reclamation sets in early for update-intensive workloads than in workloads with infrequent updates.

HP outperforms the other EBR variants in update-intensive workloads (Figure 3a, leftmost plot) but they appear to be slow in the search-intensive workload (Figure 3a, rightmost plot). We believe this could be due to the interplay of two overheads: a) the overhead due to the memory fence in HP that is incurred on every read of a shared record and b) the overhead due to burst reclamation caused by the delayed thread vulnerability in EBR variants. For EBR based algorithms, the burst reclamation overhead is low for search-intensive workloads than that of the update-intensive workloads because limbo Bags in search intensive workloads remain small due to less reclamation activity. However, in HP, the per read overhead which remains more or less the same across the workload profiles appears to dominate the reduced overhead of burst reclamation in search-intensive workload in DEBRA+. Therefore, EBR variants appear to be faster than HP in search-intensive workloads.

Also, in lazylist as can be observed in Figure 3b, NBR+ is comparable to RCU, QSBR, and DEBRA and performs better than HP (by 2x) and IBR (by more than 50%) across all workloads and when running with over-subscription [P1, P4].

In our second type of experiment (E2), we validate the bounded garbage property of NBR+ [P2] by measuring the peak memory usage of all reclamation algorithms when a thread is stalled (Figure 4c) and when no thread is stalled (Figure 4d). Each trial is run for 25 seconds. During this whole length of the experiment (25 seconds) one thread is made to sleep within a data-structure operation, imitating a stalled thread. As expected, since DEBRA and RCU do not have bounded garbage property, they exhibit an increase in peak memory usage in presence of a stalled thread (Figure 4c) while NBR+, HP, IBR, and QSBR variants maintain approximately the same peak memory usage due to continuous reclamation, which is independent of progress of any other thread. Interestingly, NBR+ matches the peak memory

\[\text{We believe that HP's poor performance is due to high cost of mfence used to publish the reservations which could be reduced by using more efficient xchg instructions to broadcast the reservations (refer section 11.5.1 in https://www.amd.com/system/files/TechDocs/47414_15h_sw_opt_guide.pdf). We did that optimization separately in our experimentation, and noticed that it did increase the absolute throughput for HPs but relative trend of HPs with respect to NBR remained the same.}\]
usage of HP. Thus in terms of usage of memory also NBR+ fares well.

One may think, the requirement of restarting from the root every time a new $\Phi_{\text{read}}$ begins may degrade the performance of data structure using NBR and that may practically limit its applicability to different data structures, especially those having multiple read/write phases. To verify that this is not the case, in the third experiment (E3) we evaluate throughput of NBR with data structures which have multiple read/write phases, namely harris list[25] and ABTree[9, chapter 8]. We design our experiment for two scenarios. First, for reasonably large data structure size (2 M in the tree and 20K in the list) where restarts would be in-expensive due to low contention (leftmost in Figure 4). Second, for small data structure size (200 in both the tree and the list) where restarting from the head node could be relatively expensive due to high contention amongst threads (rightmost in Figure 4).

As expected, with ABTree NBR+ is faster for low contention scenario and is comparable to DEBRA for high contention scenario showing that NBR+ has little or no overhead due to restarts in $\Phi_{\text{read}}$ (Figure 4a). Surprisingly, as can be seen in Figure 4b, NBR+ is faster than DEBRA for higher number of threads in low contention scenario (leftmost plot) and in high contention scenarios (rightmost plot) where restarts are costly though NBR+ is slower than DEBRA the depreciation in throughput is still comparable.

In summary, our experiments reveal that our NBR methodology is fast[P1], bounds garbage[P2] and consistent[P4]. Additionally, we also show that NBR can be fairly easily integrated[P3] in multiple data structures[P5].

Code: The source code can be found on github (https://gitlab.com/aajayssingh/nbr_setbench) or on immutable open access repository at https://zenodo.org/record/4295722.

8 Conclusions

In this paper, we presented NBR, a safe memory reclamation algorithm that is a hybrid between EBR and a limited form of HPBR, and which uses POSIX signals to ensure bound on reclamation garbage. It is simpler to use than the most similar hybrid algorithm, DEBRA+, while supporting a large class of data structures, some of which are not supported by DEBRA+. We also developed an optimized version of NBR called NBR+ that achieves similar reclamation throughput with fewer signals by passively observing signals being sent in the system to optimistically detect relaxed grace periods. Our experiments demonstrate that NBR+ surpasses the performance of the state of the art in SMR algorithms in typical benchmark conditions, while minimizing performance degradation in oversubscribed workloads.

Acknowledgments

This work was supported by Natural Sciences and Engineering Research Council of Canada (NSERC) Collaborative Research and Development grant: CRDPJ 539431-19, Canada Foundation for Innovation John R. Evans Leaders Fund with
equal support from the Ontario Research Fund CFI Leaders Opportunity Fund: 38512, Waterloo Huawei Joint Innovation Lab project titled “Scalable Infrastructure for Next Generation Data Management Systems”, NSERC Discovery grant: RGPIN-2019-04227, NSERC Discovery Launch Supplement: DGECR-2019-00048, and the University of Waterloo. We would also like to thank anonymous reviewers for their insightful comments that helped us to improve the manuscript.

References

[1] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E Tarjan. 2014. The CB tree: a practical concurrent self-adjusting search tree. Distributed computing 27, 6 (2014), 393–417.
[2] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. 2014. Stacktrack: An automated transactional approach to concurrent memory reclamation. In Proceedings of the Ninth European Conference on Computer Systems. 1–14.
[3] Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. 2017. Forkscan: Conservative memory reclamation for modern operating systems. In Proceedings of the Twelfth European Conference on Computer Systems. 483–498.
[4] Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. 2018. Threads: Automatic and scalable memory reclamation. ACM Transactions on Parallel Computing (TOPC) 4, 4 (2018), 1–18.
[5] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zabotchi. 2016. Fast and robust memory reclamation for concurrent data structures. In Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures. 349–359.
[6] Guy E Blelloch and Yuanhao Wei. 2020. Concurrent Reference Counting and Resource Management in Wait-free Constant Time. arXiv preprint arXiv:2002.07053 (2020).
[7] Anastasia Braginsky, Alex Kogan, and Erez Petrank. 2013. 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. 33–42.
[8] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. ACM Sigplan Notices 45, 5 (2010), 257–268.
[9] Trevor Brown. 2017. Techniques for Constructing Efficient Lock-free Data Structures. arXiv preprint arXiv:1712.05406 (2017).
[10] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A General Technique for Non-blocking Trees. In Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’14). 329–342. Full version available from http://tbrown.pro.
[11] Trevor Alexander Brown. 2015. 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. 261–270.
[12] Nachshon Cohen. 2018. Every data structure deserves lock-free memory reclamation. Proceedings of the ACM on Programming Languages 2, OOPSLA (2018), 1–24.
[13] Nachshon Cohen and Erez Petrank. 2015. Automatic memory reclamation for lock-free data structures. ACM SIGPLAN Notices 50, 10 (2015), 260–279.
[14] Nachshon Cohen and Erez Petrank. 2015. Efficient memory management for lock-free data structures with optimistic access. In Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures. 254–263.
[15] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronous concurrency: The secret to scaling concurrent search data structures. ACM SIGARCH Computer Architecture News 43, 1 (2015), 631–644.
[16] David L Deteufs, Paul A Martin, Mark Moir, and Guy L Steele Jr. 2002. Lock-free reference counting. Distributed Computing 15, 4 (2002), 255–271.
[17] Dave Dice, Maurice Herlihy, and Alex Kogan. 2016. Fast non-intrusive memory reclamation for highly-concurrent data structures. In Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management. 36–45.
[18] Dana Drachsler, Martin Veechev, and Eran Yahav. 2014. Practical concurrent binary search trees via logical ordering. In Proceedings of the 19th ACM SIGPLAN symposium on Principals and practice of parallel programming. 343–356.
[19] Dana Drachsler, Martin Veechev, and Eran Yahav. 2014. Practical concurrent binary search trees via logical ordering. In Proceedings of the 19th annual ACM symposium on Principles and practice of parallel programming. ACM, 343–356.
[20] Aleksandar Dragojević, Maurice Herlihy, Yossi Lev, and Mark Moir. 2011. 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. 99–108.
[21] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking binary search trees. In Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing. 131–140.
[22] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In Proc. of the BSDcan conference, ottawa, canada.
[23] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. 2019. Persistent non-blocking binary search trees supporting wait-free range queries. In The 31st ACM Symposium on Parallelism in Algorithms and Architectures. 275–286.
[24] Anders Gidenstam, Marina Papatriantafillou, Håkan Sundell, and Philipps Tsigas. 2008. Efficient and reliable lock-free memory reclamation based on reference counting. IEEE Transactions on Parallel and Distributed Systems 20, 8 (2008), 1173–1187.
[25] Timothy L Harris. 2001. A pragmatic implementation of non-blocking linked-lists. In International Symposium on Distributed Computing, Springer, 300–314.
[26] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of memory reclamation for lockless synchronization. J. Parallel and Distrib. Comput. 67, 12 (2007), 1270–1285.
[27] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer, and Nir Shavit. 2005. A lazy concurrent list-based set algorithm. In International Conference On Principles Of Distributed Systems. Springer, 3–16.
[28] Maurice Herlihy. 1993. A methodology for implementing highly concurrent data objects. ACM Transactions on Programming Languages and Systems (TOPLAS) 15, 5 (1993), 745–770.
[29] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. 2005. Nonblocking memory management support for dynamic-sized data structures. ACM Transactions on Computer Systems (TOCS) 23, 2 (2005), 146–196.
[30] Magid M Michael. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Transactions on Parallel and Distributed Systems 15, 6 (2004), 491–504.
[31] Aravind Natarajan and Neeraj Mittal. 2012. Concurrent tries with efficient non-blocking snapshots. In Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming. 317–328.
[32] Aleksandar Prokopc, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent tries with efficient non-blocking snapshots. In Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming. 151–160.
[33] Pedro Ramalhete and Andreia Correia. 2017. Brief announcement: Hazard era-non-blocking memory reclamation. In Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures.
A Correctness

Assumption 4. If \( T_i \) send a signal to \( T_j \), then \( T_j \) is guaranteed to receive it and execute a signal handler before dereferencing any reference field of a record.

Property 5. A thread \( T_j \) in \( \Phi_{\text{read}} \)

5.1 upon receiving a signal executes a signal handler and gets neutralized.

5.2 could dereference the reference field of a record to discover new records.

Property 6. A thread \( T_j \) in \( \Phi_{\text{write}} \)

6.1 protects all records used in \( \Phi_{\text{write}} \) and upon receiving a signal executes a signal handler.

6.2 could not dereference the reference field of a record to discover unprotected records.

Property 7. Every reclaimer thread \( T_r \) does the following in order:

7.1 sends signal to all participating threads.

7.2 scans all protected records of each participating thread \( T_j \).

7.3 reclaims records which are not protected.

Lemma 8 (NBR is Safe). No reclaimer thread in NBR would reclaim a record that is not safe.

Proof. Wlog, assume, the statement is false. Implying, there exists a reclaimer \( T_r \) which reclaims an unsafe record \( r \). This could occur only in two ways: (1) a reader dereferences a reference to \( r \) in limboBag of \( T_r \), or (2) a writer dereferences \( r \) which it did not protect.

We show that (1) is false: Due to Property 7.1 \( T_r \) must have sent signal to reader before reclaiming \( r \). From, Assumption 4 and Property 5.1, reader is guaranteed to execute signal handler as next step in its execution, hence will be neutralized relinquishing its private reference to \( r \), if any.

Next we show that (2) is also false: Again, Due to Property 7.1 \( T_r \) must have sent signal to writer before reclaiming \( r \). From Property 6 writer must have protected \( r \) and any reference which it could access through reference field of \( r \).

Thus, (1) and (2) are false, meaning \( r \) is safe which contradicts our assumption that \( r \) is unsafe.

\( \square \)

Lemma 9 (NBR+ is safe). No reclaimer thread in NBR+ would reclaim a record that is not safe.

Proof. NBR+ has two kind of threads that reclaim (1) threads at LoWatermark and (2) threads at HiWatermark. Reclamation at HiWatermark is similar to reclamation in NBR. Since NBR is safe(refer Lemma 8), reclamation at HiWatermark in NBR+ is safe. Therefore, our task reduces to prove that reclamation at LoWatermark in NBR+ is safe.

Assume, (A) reclamation by threads at LoWatermark is not safe. Let \( T_{lw} \) be such a thread. Then following case must be true. (C) Their is a record, say \( r \), in limboBag of \( T_{lw} \) to which another thread, say \( T_r \), holds an unsafe private reference.

But, in NBR+, \( T_{lw} \) only reclaims only up to the bookmarkTail, which depicts a time, say \( t \), at which \( T_{lw} \) entered LoWatermark. And, \( T_{lw} \) decides to reclaim only at later time \( t' \), such that \( t < t' \), which corresponds to a safe relaxed-quietness event. By definition, safe relaxed-quietness implies that any record unlinked before it would be safe. Thus, no thread could hold any private reference to \( r \) which was unlinked before \( t \) after time \( t' \), since \( t < t' \), implying, (C) is false. Consequently, our assumption (A) is false. Since, (A) cannot be both true and false. Therefore, by contradiction, it is established that NBR+ is safe.

\( \square \)

Lemma 10 (Both NBR and NBR+ are robust). Number of records that could stay un-reclaimed per thread are bounded.

Proof. Assuming data-structure using NBR is correct. Thus, a record is passed as an argument to \( \text{retire} \) only once. Consequently, an unlinked record would be present in exactly one thread’s limboBag. Say, \( k \) is number of records a thread could protect per operation, \( p \) is number of processes, and \( n \) is maximum limboBag size at which a thread decides to reclaim. Usually \( p << n \) and \( k << n \).

Let, \( T_r \) be a reclaimer and \( T_j \) be an arbitrary thread. Now, if \( T_j \) is delayed or crashed, it could only reserve atmost \( k \) records of \( T_r \)’s limboBag. Thus, in worst case a single thread could prevent only \( k \) records from being reclaimed. Inducting on number of threads, all \( p - 1 \) threads could prevent only \( k(p - 1) \) records from being reclaimed.

Since, \( p << n \), \( k(p - 1) << n \). Thus, number of records which could stay un-reclaimed per thread would be \( O(kp) \).

\( \square \)

Corollary 11. Total \( k(p-1)^2 \) records could stay un-reclaimed across all threads.

B Additional Experiments

This section contains additional results for different data structure sizes. Figure 5a and Figure 5b show throughput of DGT for the size of 20M and 20K, respectively. As can be observed, NBR+ is faster than other techniques across the size of 20K (high contention) and 20M (low contention).

For the lazy list (Figure 6a and Figure 6b) in extremely high contention (size 200 and update-intensive workload) NBR+ is slower than the EBR based variants but still comparable
Figure 5. Evaluation of throughput across different tree sizes. Y axis: throughput in million operations per second. X axis: #threads.

Figure 6. Evaluation of throughput across different list sizes. Y axis: throughput in million operations per second. X axis: #threads.

to HP. The degradation in throughput could be attributed to the overhead of the signal apparatus – frequent neutralizing signals, `siglongjmp` and `sigsetjmp` which becomes prominent due to high number of updates and small list size relative to other SMR algorithms.