Simple, Fast and Widely Applicable Concurrent Memory Reclamation via Neutralization

Ajay Singh, Trevor Alexander Brown, and Ali José Mashtizadeh

***Abstract*—Reclaiming memory in non-blocking dynamic data structures in unmanaged languages like C/C++ presents a unique challenge due to the risk of use-after-free errors caused by concurrent accesses. Existing safe memory reclamation (SMR) algorithms fall short of satisfying five key properties: high performance, bounded garbage, usability, consistency, and applicability. In particular, bounded garbage and high performance are quite difficult to achieve simultaneously. In this paper, we address this limitation by proposing a new, provably correct technique called neutralization based reclamation (NBR) that neutralizes threads using POSIX signals to provide the synchronization required for safe memory reclamation. NBR uses atomic reads and writes and achieves bounded garbage and high performance without imposing significant overhead on concurrent readers and writers. An extensive experimental evaluation serves to demonstrate the efficiency of our technique across various data structures, reclamation algorithms, and workloads. A detailed survey of popular concurrent data structures suggests NBR is applicable to a wide range of data structures, many of which could not be used with prior SMR algorithms that guarantee bounded garbage.**

***Index Terms*—Non-blocking data structures and algorithms, safe memory reclamation, shared memory synchronization.**

I. Introduction

In UNMANAGED languages like C/C++, the reclamation of records in non-blocking dynamic data structures poses a distinctive challenge. Unlike their locking counterparts, these data structures allow individual nodes to be concurrently accessed by multiple threads, which greatly increases the risk of *use-after-free* errors. A classic example can be seen in a so-called *lazy* linked list [1], in which threads search without acquiring any locks, and then acquire fine-grained locks on only a few nodes at the end of its search. Suppose a *reader* thread reads a node's *next* pointer, saving the address it sees in a local variable x, and a (*reclaimer*) thread subsequently unlinks and reclaims the node pointed to by x. If the *reader* thread attempts to dereference x, a *use-after-free* error occurs. The problem of reclaiming memory safely in such data structures is widely referred to as the *safe memory reclamation* (SMR) problem [2], [3], [4], [5].

In the literature on SMR algorithms, researchers have proposed a wide range of techniques with diverse properties, peculiarities, and limitations. Through rigorous experimentation and an extensive survey of the current literature [2], [3], [4], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], we have identified a set of desirable properties that are highly sought after in SMR algorithms. Some similar properties were also discussed by Kang et al. [5]. Here, we refine those properties and reaffirm their significance. These properties have served as guiding principles for the development of the neutralization-based technique presented in this paper, as well as a few other techniques [23], [24], [25], [26] which appeared after NBR [27].

- P1 *Performance:* Reclamation operations should exhibit low latency and high throughput, ensuring efficient utilization of system resources.
- P2 *Bounded Garbage:* The number of unlinked but unreclaimed records should be finite, even in the presence of thread failures or significant delays.
- P3 *Usability:* Intrusive changes to data structure layout, code modifications, and reliance on specialized hardware instructions and compilers should be minimized to enhance the ease of adoption and integration.
- P4 *Consistency:* The algorithm's performance should remain stable and consistent across different workloads, such as shifting between read-intensive and update-intensive scenarios, and when the system is oversubscribed with more threads than available cores.
- P5 *Applicability:* The algorithm should be applicable to a wide range of popular data structures, prioritizing usefulness over attempting to support any imaginable data structure [8].

All existing SMR algorithms exhibit significant limitations and shortcomings, and fall short of satisfying properties P1 to P5 simultaneously. We give a brief taxonomy of the popular families of SMR algorithms.

Perhaps the most popular family of SMR algorithms, Hazard Pointer Based (HP) algorithms [3], [13], [14], [15], [17], [28] ensure safe reclamation and bounded garbage by requiring readers to explicitly reserve records before accessing them. This allows a thread that is attempting to reclaim a set of records to identify which ones it can safely reclaim (the ones not reserved by any thread). Since a reader must reserve each record

Manuscript received 6 October 2022; revised 16 October 2023; accepted 18 November 2023. Date of publication 22 November 2023; date of current version 18 December 2023. This work was supported in part by the NSERC Collaborative Research and Development Grant, in part by the Canada Foundation for Innovation John R. Evans Leaders Fund with equal support from the Ontario Research Fund CFI Leaders Opportunity Fund, in part by Waterloo Huawei Joint Innovation Lab project, in part by NSERC Discovery Program, and in part by the University of Waterloo. Recommended for acceptance by F. Wolf. *(Corresponding author: Ajay Singh.)*

The authors are with the Cheriton School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada (e-mail: ajay.singh1@uwaterloo.ca; trevor.brown@uwaterloo.ca; mashti@uwaterloo.ca).

This article has supplementary downloadable material available at https://doi.org/10.1109/TPDS.2023.3335671, provided by the authors.

Digital Object Identifier 10.1109/TPDS.2023.3335671

before accessing it, HP-based SMR algorithms impose significant overhead, especially in data structures where threads repeatedly follow pointers (e.g., lists and trees). Reference counting based reclamation (RCBR) techniques (e.g., [6]) typically offer similar guarantees and overheads.

In contrast, Epoch-Based Reclamation (EBR) algorithms [2], [8], [29], [30] ensure safety in a highly efficient way, by dividing an execution into *epochs*, and reclaiming records in large batches. Intuitively, the epoch changes when all threads have "forgotten" pointers to records that were reclaimed in a previous epoch. Thus, whenever the epoch changes, all threads can reclaim a batch of records. The synchronization needed to track epoch changes imposes significantly less overhead than HP-based algorithms, but a stalled thread can prevent the epoch from advancing, so garbage is not bounded.

Hybrid SMR algorithms attempt to blend the best attributes of these approaches, combining ideas from each [2], [4], [7], [22], [31]. Such algorithms typically offer high performance and some notion of bounded garbage, but often compromise significantly on applicability, as we see in Section VI.

In summary, HP bounds garbage but incurs high overhead, EBR offers low overhead but allows unbounded garbage, and hybrid approaches compromise significantly on applicability.

The technique discussed in this paper [27], [32][1] offers an alternative solution. It avoids waiting for stalled threads to reclaim memory (unlike EBR), and does not require readers to repeatedly reserve records during searches (unlike HP). More specifically, readers do not reserve records, and writers reserve records exactly once, after they have finished gathering pointers to all of the records they intend to modify, and before they have begun modifying these records. Synchronization between reclaiming threads and threads accessing the data structure is mediated via *neutralizing*. As we will see, this allows NBR to guarantee bounded garbage with EBR-like speed, while remaining applicable to many data structures that are incompatible with existing HP-based and hybrid SMR algorithms.

Neutralization is achieved through the use of POSIX signals. During a neutralizing event, participating threads, based on their roles as readers or writers, take specific actions to ensure all accesses are safe. Readers discard all references they acquired before the event, and retry their operation from the beginning, effectively resetting their state. On the other hand, writers continue to operate unimpeded, as long as they promise only to access records that they reserved before the neutralization event. (If a writer has not yet gathered pointers to all of the records it intends to modify, it behaves like a reader.)

By neutralizing other threads, a reclaimer can effectively force readers (or writers that have not yet begun writing) to drop their pointers to any records that the reclaimer might be trying to reclaim. Crucially, this allows a reclaimer to reclaim records without having to wait for an epoch to advance (unlike EBR), and readers can avoid the overhead of reserving individual records (unlike HP). However, NBR retains the key benefit of HP: the set of reserved records, that cannot be reclaimed, is bounded. In NBR, only the records that active writers intend to modify are reserved. This allows NBR to guarantee bounded garbage without introducing per-record overheads for reservations.

NBR draws considerable inspiration from DEBRA+ [2], a hybrid algorithm combining EBR with a restricted form of HP. Similar to NBR, DEBRA+ utilizes neutralizing signals to guarantee bounded garbage. However, DEBRA+ suffers from limitations in terms of applicability and usability. Specifically, a thread receiving a neutralizing signal in DEBRA+ is required to restart, even if it has made modifications to shared memory. This necessitates the inclusion of data structure-specific recovery code, which can be complex and is not always feasible. Additionally, it is unlikely that DEBRA+ can be used with lock-based data structures, as neutralizing a thread holding locks can cause deadlock. NBR is simpler to implement, is compatible with data structures that use locks, and can be used to reclaim memory for many popular data structures for which it is unclear how DEBRA+ could be used.

In this work, we demonstrate the effectiveness of NBR by showcasing its ability to match or surpass the performance of existing SMR algorithms [P1]. Additionally, NBR achieves bounded garbage [P2], offers simplicity in its usage [P3] and consistent performance [P4], and is applicable to a wide range of data structures, including many that are not supported by popular SMR algorithms [P5].

Contributions: In this paper, we build on our prior work on NBR [27] and make the following **new** contributions:

- We provide a detailed theoretical proof demonstrating that, when utilized with compatible data structures, NBR preserves the correctness and progress guarantees of the original data structures.[2]
- We study the signal handling code of modern operating systems (Linux and FreeBSD) and design a microbenchmark to gain insight into the potential delays that can affect the delivery of these signals in the context of NBR. We analyze results obtained from the benchmark to better understand how long a thread should wait after sending neutralizing signals before it can safely proceed with reclamation.
- We conduct a extensive review of many existing concurrent data structures, examining their compatibility with NBR as well as other popular reclamation techniques. To our knowledge, this review represents the most comprehensive study of SMR algorithm compatibility to date, providing valuable insights into a metric that we think is under studied in the literature.
- Building upon the preliminary work's benchmark, we enhance the code of the original technique through manual optimizations. We include multiple new data structures and state-of-the-art reclamation algorithms, and evaluate performance across additional workloads. We also study the memory usage of different SMRs in greater detail. To

[1] A preliminary version of the technique appeared in the Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.

[2] Technically, for lock-free and wait-free data structures, the full progress guarantee is preserved only if the operating system kernel facilities offer a similar progress guarantee. This means, for example, the memory allocator, context switching mechanism, thread scheduler, and interprocess signaling mechanism should all be lock-free (or wait-free) in order for a data structure to be truly lock-free.

our knowledge, this is the most comprehensive evaluation of SMR algorithms to date.

The remainder of the paper is organized as follows. Section II discusses related work. Section III, introduces the model and assumptions on the format of a data structure operation. Section IV describes NBR and its implementation. In Section V, we present an optimized version called NBR+. Section VI, discusses the classes of data structures to which the technique applies and a few to which it does not, followed by a extensive comparison of the applicability of NBR with the other SMRs studied. In Section VII and Section VIII, we prove safety and progress property of NBR, and study the timing of signal delivery on Linux. Performance experiments appear in Section IX. Finally, we conclude in Section X.

II. Related Work

There is a rich literature of SMR algorithms, and the following survey does not characterize every known SMR algorithm. However, we have attempted to cover the state of the art fairly exhaustively, focusing especially on which of the properties P1 through P5 are satisfied by each algorithm (and to what degree they are satisfied).

A. Reference Counting

One of the earliest fairly general SMR techniques, lock-free reference counting [6] employs a count field within each record to track the number of active references across all threads. However, reference counting based techniques introduce significant overhead due to cache invalidations caused by frequent updates to reference counts. Updating reference counts is unfortunately quite slow. In modern optimistic data structures, in which threads search without locking, reference counting can completely negate any performance gains over traditional locking techniques (opposing P1). Recent advancements, such as deferred reference counting [31], have demonstrated improved performance in read-heavy workloads, but they still impose significant overhead in update-heavy scenarios (opposing P4). Reference counting also introduces other complications, such as the need to break pointer cycles, the need to invoke functions for many operations involving pointers, and the need to modify the memory layout of nodes (opposing P3).

B. Pointer Reservations

Pointer reservation based reclamation techniques require each thread to *reserve* pointers before dereferencing them. Reserving a pointer typically entails writing it to a shared memory address and subsequently performing some *validation* to ensure that any thread attempting to reclaim that record has observed the reservation [3], [14], [15]. Similar to RCBR, reserving a record introduces significant overhead to each record access (opposing P1). Additionally, users bear the responsibility of correctly releasing previously reserved pointers. Morevoer, in the event that validation of a reservation fails, threads need to take alternative actions, such as restarting an operation (since, e.g., the next node to be accessed cannot safely be accessed). This requires modifying the original data structure in potentially complex ways that may necessitate reproving correctness or progress (opposing P3, P5).

An important factor to consider when applying pointer reservation based techniques is their compatibility with data structures that involve traversing marked (logically deleted) nodes [1], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42]. In such cases, establishing the validity of reservations becomes challenging [2], limiting the applicability of these techniques [P5]. Some reservation based techniques sidestep these issues [7] at the cost of considerably higher programmer effort (opposing P3).

C. Epoch Based Reclamation

EBR algorithms [2], [8], [29], [30] offer a straightforward, high-performance approach. In these algorithms, the execution is divided into *epochs*, and each thread must regularly participate in the algorithm for the epoch to change (typically imposing a small amount of overhead on each data structure operation). Classically, each thread maintains three batches of records it is waiting to reclaim, one for each of the last three epochs, and whenever the epoch changes, each thread can reclaim its oldest batch. This approach incurs very little overhead, but a stalled thread can prevent the epoch from advancing, preventing all memory from being reclaimed (opposing P2).

Brown introduced an optimized variant of EBR called DEBRA, as well as an enhanced variant called DEBRA+[2] that bounds garbage in compatible non-blocking data structures. Like NBR, DEBRA+ uses neutralization. However, unlike NBR, when a thread is neutralized in DEBRA+, it restarts its operation, even if it had already begun modifying the data structure. Data structure specific recovery code is thus required to deal with the inconsistent states that can result. It is not clear how DEBRA+ could be applied to any lock based (or optimistic) algorithms [1], [43], since neutralization could cause deadlocks (opposing P3 and P5).

D. Epoch Reservations

Following DEBRA and DEBRA+, other techniques like Hazard Eras (HE) [17] and Interval Based Reclamation (IBR) [13] emerged that attempt to limit the amount of garbage that cannot be reclaimed to invidual epochs, or specific ranges of epochs. These techniques augment records with additional metadata, such as the epochs when a record was last allocated, and unlinked from the data structure. This information is then used to argue that a stalled thread cannot access particular records. Unfortunately, in certain scenarios involving stalled threads, memory consumption can still be extremely high, as we show in our experiments. Recent algorithms such as Wait Free Eras (WFE) [28], Hyaline (HY) [22], and Crystalline (CY) [44] have also attempted to bound garbage while achieving high performance. HY and CY employ an efficient form of reference counting, specifically on records that have *already* been unlinked from the data structure. These algorithms require per-record metadata, necessitating changes to the memory layout of nodes (opposing P3). Moreover, as we discuss in Section VI, it appears

that most epoch reservation techniques, including as HE, IBR and WFE cannot be used with a wide variety of data structures in which threads traverse chains of unlinked records (opposing P5). Additionally, as our experimental results demonstrate, these algorithms exhibit non-trivial overhead (opposing P1).

E. OS/Hardware Primitives

Popular hybrid memory reclamation algorithms combine various techniques, including leveraging operating system features like context switches [10] and POSIX signals [11], [21], as well as hardware primitives such as hardware transactional memory [19], [20]. One notable hybrid algorithm, Qsense [10], introduced by Balmau et al. optimizes for the common case by employing an EBR-like approach as a fast code path [8]. However, to ensure bounded garbage when threads experience delays, Qsense switches to a hazard pointer inspired slow path. Qsense provides bounded garbage, but it has been shown to be slower than both EBR [10] (opposing P1).

Another hybrid algorithm, StackTrack [19], focuses on automating memory reclamation through compiler assistance. It utilizes hardware transactional memory for common cases and incorporates a hazard pointer-like fallback mode to ensure progress [19]. However, StackTrack requires significant programmer intervention and assumes that programs will not hide pointers (via, e.g., special pointer arithmetic), which can limit its applicability (opposing P3 and P5). Due to their reliance on HP, StackTrack and Qsense have the same issues with respect to applicability as other HP and epoch reservation based approaches.

ForkScan [21] uses OS level copy-on-write support for memory pages and POSIX signaling to automatically reclaim memory for concurrent data structures. Like StackTrack, ForkScan does not work if users perform special pointer arithmetic (opposing P3). Due to the way that ForkScan identifies records that are safe to reclaim, it has the same applicability issues as StackTrack and Qsense (opposing P5). ForkScan's successor, ThreadScan [11], offers numerous improvements over the original, but suffers from the same applicability issue (opposing P5).

The recent PEBR [5] is another hybrid of HP and EBR. The key idea is to *eject* a stalled thread from the epoch advancement mechanism and use hazard pointers to protect the references of the ejected thread. Every read needs to reserve records as in HP, but the reservation is made more efficient using techniques from [14]. The publicly available implementation of PEBR is in Rust, which makes a direct experimental comparison with NBR's C++ implementation difficult. PEBR has been shown to be slower than Rust's Crossbeam implementation of EBR (opposing P1).

F. Optimistic Access

Cohen et. al. [18] introduced an interesting approach which allows threads to access potentially deleted records optimistically and roll back to a safe point if validation (after the access) indicates the record was deleted. These techniques involve instrumenting reads, writes, and CAS instructions and require the data structure to be in a normalized form, somewhat similar to (but not the same as) the form assumed by NBR. Aiming to enhance programmability, AOA [9] (Automatic Optimistic Access) was introduced, automating the transformation into the normalized form. Subsequently, the authors proposed the FreeAccess (FA) [12] technique, which eliminated the requirement for data structures to be presented in a normalized form.

It is important to note that these optimistic access based techniques first *perform unsafe accesses* and then check whether reclaimed memory was accessed. Thus, a programmer must either trap and ignore segmentation faults that occur (and lose a valuable debugging tool) or use type-stable allocator that never unmaps pages (and lose the ability to ever shrink the data structure's memory usage in a long running program). Both options oppose P3. The experimental results have shown that these techniques achieve performance comparable to hazard pointers [12]. However, NBR targets applications that can benefit from even higher performance (so P1 is not satisfied).

Recently, Sheffi et al. introduced VBR [23], which extends OA to allow threads to access reclaimed records for write accesses, as well. Each mutable field in a record also stores corresponding version numbers to facilitate optimistic writes, necessitating use of double-wide CAS for atomicity. VBR requires programmers to define checkpoints to rollback when reclaimed records are accessed, and it requires similar code instrumentation to HE and IBR (opposing P3). Additionally, as it performs unsafe accesses, it imposes the same requirements as OA, AOA and FreeAccess.

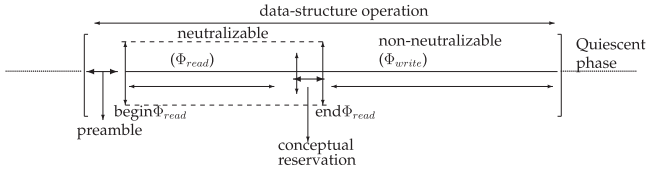


Fig. 1. Typical NBR compatible data structure operation with preamble (optional), read-phase (Φ_{read}) that ends with a conceptual reservation phase followed by write-phase (Φ_{write}). Just before beginΦ_{read} a checkpoint is set so that threads in Φ_{read} could be neutralized (discard all local references to shared memory) and restart.

III. System Model and Assumptions

We assume the standard asynchronous shared memory model (as in [27], [32]). A data structure consists of *records* which are accessible from an immutable entry point. Each record can be in one of five states: *allocated*, *reachable*, *retired*, *safe*, or *reclaimed*. Once a record is *allocated*, it can be made *reachable*, and once it is unlinked it becomes *retired*. It subsequently becomes *safe* when no thread will access it again, at which point it can be *reclaimed* (and then allocated again).

A. Assumptions on the Data Structure

NBR can be applied to data structure operations that follow a simple template. Each operation should consist of a sequence of *phases:* preamble, read-phase (Φ_{read}), (conceptual) reservation phase, and write-phase (Φ_{write}), as illustrated in Fig. 1. As we will see, this template ensures that it is safe for a thread to restart its Φ_{read} from any arbitrary point in the Φ_{read}. In this section,

we reproduce the rules governing each phase of a *data structure* operation, as originally presented in [27]. The template and the rules are crucial for the correct integration of NBR with a data structure. Note, outside a data structure operation threads are in *quiescent phase*, which is like preamble phase.

1) *Preamble:* Allows system calls and accesses to global variables. *No* access to shared records. This phase can be omitted in some data structure implementations.
2) *Read phase (Φ_{read}):* Permits reading from global variables and shared records, but only if pointers to them were obtained during this phase or the preamble. Prohibits system calls, writes/CASs to shared records or globals, and writes to thread-local data structures.[3]
3) *Reservation: Conceptual stage* where shared records to be modified in the next phase are identified and reserved. These records are referred to as *reserved* records.
4) *Write phase (Φ_{write}):* Permits accesses to global variables and system calls. Accesses to shared records (including modifications) are permitted only if they have been *reserved*.

IV. NBR

The key idea in NBR [27] is as follows. To bound garbage, whenever a reclaimer accumulates a certain number of retired records, it sends *neutralizing signals* to all threads. Threads receiving these signals either (a) have already explicitly reserved all records they will access, in which case the reclaimer can safely avoid reclaiming any of those records, or (b) will abandon their pointers and restart their operations, allowing the reclaimer to safely ignore them.

More specifically, threads in a preamble phase (or quiescent phase) can simply ignore the neutralizing signal since they do not access shared records. However, threads in a Φ_{read} (readers) or Φ_{write} (writers) do access shared records, and thus must synchronize carefully with the reclaimer.

The readers does that by discarding their pointers to shared records by simply restarting the current Φ_{read} from the entry point of the data structure. It is straightforward to see from the rules for the Φ_{read} that restarting the Φ_{read} does not have side effects. However, writers cannot simply restart because doing so could leave the data structure in an inconsistent state, as the writer could be in the middle of an update. Instead, at the very beginning of the Φ_{write} (in the conceptual reservation phase), the writer reserves all of the records it will access in its Φ_{write}. This enables the reclaimer to scan and avoid reclaiming those records, in turn allowing a writer who receives a neutralization signal to ignore it and proceed as usual. In summary, readers guarantee that they will discard pointers to all records which were retired before they received the neutralizing signal, writers guarantee that they will only access reserved records, and reclaimers guarantee that they will not free any reserved record.

[3]These restrictions exist because NBR uses `sigsetjmp` to implement a checkpoint from which a thread restarts and neutralization fires a POSIX signal that (sometimes) causes a thread to `siglongjmp` to its last checkpoint. For readers not familiar with the caveats of using these subroutines, building upon the original rules provided in [32], we further explain the rules in the accompanying supplementary material.

Algorithm 1: Neutralization Based Reclamation (NBR).

```
thread local variable:
1: int tid; ▷current thread id
2: record *limboBag; ▷retired records.
   Maxsize:S
3: atomic<bool> restartable;
   ▷tracks Φ_read / Φ_write
4: record *tail; ▷last record in limboBag
shared variable: n:#threads, r:max reserved
 records.
5: atomic<record*> reservations[n][r];
   ▷Assume:r << S
6: procedure CHECKPOINT( ) ▷must be INLINED.
7:   while sigsetjmp(...)) do ▷signal mask
     not saved.
8:     unblock neutralizing signal. ▷post
       siglongjmp.
9:   end while
10: end procedure
11: procedure SIGNALHANDLER( )
12:   if !restartable then
13:     return; ▷in Φ_write, ignore signal and
        return.
14:   end if
15:   siglongjmp(...); ▷ in Φ_read , jump to
      checkpoint.
16: end procedure
17: procedure BEGIN Φ_read( )
18:   reservations[tid].clear();
19:   restartable = True;
20: end procedure
21: procedure END Φ_read(rec = {rec_1, ···, rec_R})
22:   reservations[tid]=rec;
23:   restartable = False;
24: end procedure
25: procedure RETIRE(rec)
26:   if isLimboBagTooLarge() then
27:     signalAll ( );
28:     reclaimFreeable (tail);
29:   end if
30:   limboBag[tid].append(rec);
31: end procedure
32: procedure RECLAIMFREEABLE(tail)
33:    A= collectReservations ();
34:    R=limboBag[tid].remove(A, tail);
35:   free({R});
36: end procedure
```

A. Implementation

Pseudocode for NBR appears in Algorithm 1. We assume the C++ memory model. In particular, this means `atomic` accesses are, by default, sequentially consistent.

1) Description of Variables: Each thread has an ID denoted by `tid`. The thread collects retired records in its `limboBag`, which is implemented as an array. The `tail` variable points to the last retired record added to the `limboBag`. Each thread

also has a local `restartable` flag, which determines how the thread will behave if it receives a neutralizing signal. This flag is set when a thread enters its Φ_{read}, and reset when the thread exits the Φ_{read}. Prior to exiting the Φ_{read}, the thread enter a conceptual reservation phase. During this phase, it announces all the records it will access in its subsequent write-phase (if any), using a single-writer multi-reader (SWMR) `reservations` array.

2) CHECKPOINT() and SIGNALHANDLER(): To enable thread restarts, CHECKPOINT() (line 6) is invoked just before entering the Φ_{read}. It creates a `checkpoint` from which the executing thread can safely restart after it receives a neutralizing signal. When `sigsetjmp` (line 7) is executed, it saves the current execution context (stack frame pointer, program counter and register contents) and returns false when executed for the first time, effectively creating a checkpoint from which the thread can restart.

A thread invokes BEGINΦ_{read}() to initiate a Φ_{read}. While in the Φ_{read}, if a neutralizing signal is received, the thread executes a custom signal handler (line 11). From within the signal handler, it uses `siglongjmp` (line 15) to jump back to the `checkpoint` (restoring the saved context) and re-executes the `sigsetjmp`, which this time returns true. It also unblocks the neutralizing signal (which was automatically blocked by `siglongjmp` to avoid recursive signal handler invocations). This effectively restarts the Φ_{read} from scratch with a new checkpoint.

3) BEGINΦ_{read} (): As mentioned above, a thread invokes BEGINΦ_{read}() to start its Φ_{read} (line 17). The thread first clears its previous reservations so that it can reserve new records in the reservation phase. It then sets `restartable` to true (using a sequentially consistent store; line 19). This ensures that the thread becomes restartable before it accesses any shared records in its Φ_{read}.

4) ENDΦ_{read} (): In order to enter a Φ_{write}, a thread invokes ENDΦ_{write}() (line 21). The thread begins by reserving a sequence of records (line 22) in its designated slots within the `reservations` array (implementing the conceptual reservation phase). Then, it resets the `restartable` flag (with a sequentially consistent store; line 23). This store ensures that when this thread, in its Φ_{write}, executes a signal handler and sees restartable is false, it has already reserved all of the records it will need in its Φ_{write}, so any reclaimers that read these reservations will see all of the records the thread will access. Note that these reservations are different from classic hazard pointers based reservations, both because records are reserved *after* the records have been accessed, and because no special validation is required after the stores to `reservations`.

5) RETIRE(): Whenever a thread unlinks a record from a data structure, it invokes RETIRE(,) which adds the record (*rec*) to the thread's `limboBag` (line 30). If the size of the `limboBag` exceeds a predefined threshold (line 26), 16 k in our experiments, neutralizing signals are sent to all other threads using the SIGNALALL() function (line 27). In our implementation, SIGNALALL() uses *pthread_kill()* to send *SIGQUIT*.

Upon receipt of the signal, a thread immediately executes SIGNALHANDLER() (line 11). If a thread is restartable (i.e, in a Φ_{read}), it restarts from its previous `checkpoint`. Otherwise, it returns and continues where it left off.

After, a reclaiming thread has sent neutralizing signals to all other threads (line 27), it executes RECLAIMFREEABLE() (line 28). Within RECLAIMFREEABLE() (line 32), the thread collects all reservations in a set A. These reservations are used to determine which records are safe to free (line 34), and these records are then deallocated (line 35).

B. Optimization: Relaxing the Memory Model

In the previous section, we presented the theoretical algorithm (Algorithm 1) assuming sequential consistency. However, in practice, there is considerable interest in relaxing memory models to obtain higher performance.

An implementation of NBR on a relaxed memory model should provide two essential guarantees.

Guarantee 1: When a thread T_1 executing a signal handler reads its `restartable` variable and sees $false$, T_1 must have already written its reservations, and any other thread T_2 that subsequently reads $T_1's$ reservations must see those reservations that were written by T_1.

Guarantee 2: A thread T must store true to its `restartable` variable before it discovers any new records in its Φ_{read}.

On modern Intel and AMD x86/64 systems, which implement total store order (TSO), one possible optimization is to set the C++ `atomic` memory order of all loads and stores in the algorithm to `memory_order_relaxed`, except for the stores to the `restartable` variable.

As long as the stores to `restartable` are sequentially consistent, if thread T_1 executes the signal handler and sees that its `restartable` variable is $false$, then it has already performed its stores to `reservations`, and any other thread loading T_1's reservations will see the records T_1 reserved before setting its `restartable` variable to $false$, satisfying Guarantee 1.

Similarly, sequential consistency ensures that `restartable` will be set to $true$ before T_1 discovers any new records in its Φ_{read}, satisfying Guarantee 2.

Note that on modern Intel and AMD x86/64 systems, these sequentially consistent stores to `restartable` typically compile to a `mov` instruction followed by an `mfence`, which is much slower than an `xchg` instruction (`atomic_exchange` in C++) which would also offer the required semantics. (In fact, some modern compilers will emit `xchg` for these stores with high optimization compiler flags.)

V. NBR+

The NBR algorithm, discussed in the preceding section, presents a simple solution to the safe memory reclamation problem. However, it has been noted that the use of POSIX signals in NBR introduces significant overhead [27]. In fact, it was noted that in a system with n threads, $O(n^2)$ signals are required for all threads to reclaim their `limboBags` at least once. Each thread needs to send a neutralizing signal to every other thread in order to initiate the reclamation of its `limboBag`, referred to as the *reclamation event*.

A key observation is that when one thread reclaims its `limboBag` upon reaching the threshold size, it effectively neutralizes the other $n-1$ threads by causing them to discard all unreserved references to shared records. Consequently, all records retired before this *reclamation event* become safe to be freed. If all threads could tap into this information, they could reclaim their `limboBags` without the need for sending signals themselves, thus reducing the signal overhead from $O(n^2)$ to $O(n)$ in an ideal scenario.

NBR+ [27] proposes a mechanism to leverage this information and significantly reduce the signal overhead, making the neutralization technique highly efficient. In NBR+, each thread passively monitors for a *reclamation event* starting from a predetermined lower `limboBag` size known as the `LoWatermark`, before reaching the maximum `limboBag` size (referred to as the `HiWatermark`) and triggering a reclamation event. When a thread reaches the `HiWatermark`, it announces the *reclamation event* by updating its slot in the SWMR (single-writer, multiple-reader) `announceTS[]` (Algorithm 2, line 7). Other threads, which have reached the `LoWatermark` but not yet reached the `HiWatermark`, observe these announcements to infer that a *reclamation event* has occurred. They can then safely reclaim all records in their `limboBags` retired before reaching the `LoWatermark`. If a thread has not reached the *LoWatermark* or the *HiWatermark*, it simply adds the retired record to its `limboBag` (line 29) and returns.

The Algorithm 2 provided in this manuscript serves as a reproduction of the original algorithm, aiming to provide self-containedness [27]. In the RETIRE() procedure, when a thread reaches the `HiWatermark`, it announces the timestamps corresponding to the start (line 10) and finishing (line 12) of the *reclamation event* by incrementing its slot in the `announceTS[]`. Other threads, which have reached the `LoWatermark` but not the `HiWatermark`, observe the *reclamation event* using the `scanTS` array (line 3). Upon reaching the `LoWatermark`, these threads save the current announced timestamps of other threads in their `scanTS` slots (line 18). They occasionally scan (line 21) whether a thread at the `HiWatermark` has started and finished a *reclamation event* (by incrementing its timestamp twice). If such an event is observed by a thread at the `LoWatermark` (line 22), it piggybacks on this observed `reclamation event` and reclaims all records in its `limboBag` that were retired before reaching the `LoWatermark` (line 23). Specifically, all records up to the `bookmarkTail` pointer saved from the record pointed to by `tail` at the time when thread initially hit the `LoWatermark` are considered safe for freeing and are freed.

After a thread reclaims at either the *LoWatermark* (line 13) or the *HiWatermark* (line 23), the CLEANUP( ) method (line 31) is used to set *firstLoWmEntryFlag*. This prepares the thread for future reclamation events.

Algorithm 2: NBR+ [27]: NBR+ Incorporates all Variables and Procedures From Algorithm 1 While Introducing a Modified `RETIRE()` and Additional Helper Procedures With Self Explanatory Names.

```
1: thread local variable: n: #threads
2: int otid; ▷other thread's ids, excluding tid.
3: int scanTS[n];
4: bool firstLoWmEntryFlag=true;
5: record* bookmarkTail;
6: shared variable:
7: atomic<int> announceTS[n];
8: procedure RETIRE(rec)
9:   if isAtHiWm() then
10:    FAA(&announceTS[tid],1);▷rec. event begin
11:    signalAll()
12:    FAA(&announceTS[tid],1);▷rec. event end
13:    reclaimFreeable(tail);
14:    cleanUp();
15:  else if isAtLoWm()
16:    if firstLoWmEntryFlag then
17:      bookmarkedTail = tail;
18:      scanTS[tid] =scanAnnounceTS()()
19:      firstLoWmEntryFlag =0; ▷ correction [27]
20:    end if
21:    for each otid do
22:      if announceTS[otid]≥scanTS[tid][otid]+2 then
23:        reclaimFreeable(bookmarkTail);
24:        cleanUp();
25:        break;
26:      end if
27:    end for
28:  end if
29:  limboBag[tid].append(rec);
30: end procedure
31: procedure CLEANUP()
32:  firstLoWmEntryFlag = 1;
33: end procedure
```

VI. APPLICABILITY

NBR categorizes data structures into three broad categories: compatible, semi-compatible, and incompatible, based on its applicability to these data structures. Compatible data structures can be further classified into two types based on the pattern of read and write phases.

A. Compatible Data Structures.

The first type of compatible data structures exhibit a single Φ_{read} followed by a single Φ_{write}. Examples include optimistic data structures such as lazylist [1] and DGT [43], where a sequence of reads identifies target nodes (Φ_{read}), followed by a few writes on those nodes (Φ_{write}). Applying NBR to these data structures is straightforward, involving invoking `checkpoint` and BEGINΦ_{read}() before the sequence of reads, and invoking ENDΦ_{read}() when the target nodes are identified and updates are executed. For read-only operations, ENDΦ_{read}() is invoked

before the operation returns to reset the `restartable` flag and mark the start of the quiescent phase.

The second type of compatible data structures involve a pattern of alternating *read-write phases*. For example, Harris's lock-free list(HL) [42] shown in Algorithm 3 follows this pattern. By examining our exposition on the HL, readers can gain insights into how NBR can be applied to more sophisticated data structures that share the similar design pattern [36], [38], [45], [46], [47].

In HL, during an update or contains operation, a sequence of reads is performed to identify target nodes (line 22-30), followed by auxiliary updates to unlink marked nodes from the list(line 40). The sequence of reads is then restarted from the *head* of the list, potentially repeating multiple times if marked nodes are encountered, resulting in a sequence of *read-write phases*.

To apply NBR to data structures with alternating *read-write phases*, the following steps are followed. First, the `checkpoint` and BEGINΦ_{read}() operations are invoked at the beginning of the sequence of reads. Then, the ENDΦ_{read}() operation is called when auxiliary updates begin. After that, the `checkpoint` and BEGINΦ_{read}() operations are invoked again when the sequence of reads restarts from the head of the list. Finally, the ENDΦ_{read}() operation is invoked when the final intended update is executed. In the provided example, line 19 and line 32 indicate the start and end of the Φ_{read} and Φ_{write}, respectively. For more detailed examples and discussions on incorrect placements of BEGINΦ_{read}() and ENDΦ_{read}() functions a programmer should be aware of, please refer to the preceding conference version [27].

If a neutralization signal is received during the first Φ_{read}, all threads will discard their shared node references and restart from the `checkpoint` corresponding to the first BEGINΦ_{read}() invocation. Similarly, if the neutralization signal is received during the second Φ_{read}, after an auxiliary update, the thread will discard all references acquired during this second Φ_{read} and jump back to the `checkpoint` corresponding to the second Φ_{read}.

It is crucial to highlight that each Φ_{read}, including the second Φ_{read}, starts from the head node of the data structure. As a result, there are no shared record references carried forward from the previous Φ_{read} or Φ_{write} (auxiliary update phase). This behavior ensures that each Φ_{read} is treated as a new operation, disregarding any previously acquired node references. Thus, the requirement of acquiring all references to shared records during the current phase (refer to Φ_{read} rules in Section III-A) and discarding them (Section IV) when restarting from the corresponding checkpoint is met, ensuring the safety of reclamation.

1) Limitation: Restarting From the Root.: If instead of restarting from head (or entry point), threads were to continue searching from a midpoint in the list, such as resuming from a shared node R that was reserved during the previous Φ_{write}, would introduce a risk of dereferencing a freed node. Although R itself cannot be freed due to reservation, the nodes it points to may not be reserved and thus could be freed. Consequently, following any pointer from R could lead to accessing a freed node, resulting in a crash.

Algorithm 3: Demonstration That NBR is Simple to Use With Harris List [42] With Multiple Read/Write Phases $(\Phi_{read}\ \Phi_{write})^+$ and Other Compatible Data Structures. Reproduced From the Conference Version [27].

```
1  bool insert(key) {
2    Node *right_node, *left_node;
3    do{
4        right_node = search (key, &left_node);
5        if((right_node!=tail) && (right_node.key==key))
6          return false;
7        Node *new_node = new Node(key);
8        new_node.next = right_node;
9        if (CAS(&(left_node.next), right_node, new_node))
10         return true;
11   }while (true)
12 }
13
14 Node* search(key, Node** left_node) {
15   Node *left_node_next, *right_node;
16   search_again:
17   do {
18       CHECKPOINT();
19       beginΦread();
20       Node *t = head;
21       Node *t_next = head.next;
22       do{
23           if(!is_marked_reference(t_next)){
24             (*left_node) = t;
25             left_node_next = t_next;
26           }
27           t = get_unmarked_reference(t_next);
28           if (t == tail) break;
29           t_next = t.next;
30       }while(is_marked_reference(t_next) or (t.key<
             search_key));
31       right_node = t;
32       endΦread(left_node, right_node);
33
34       if (left_node_next == right_node)
35         if ((right_node != tail) && is_marked_reference(
               right_node.next))
36           goto search_again;
37         else
38           return right_node;
39       // unlink and retire one or more marked nodes.
40       if (CAS(&(left_node.next), left_node_next,
             right_node))
41         if ((right_node != tail) && is_marked_reference(
               right_node.next))
42           goto search_again;
43         else
44           return right_node;
45   } while(true);
46 }
```

Fortunately, there are many concurrent data structures in the literature that naturally restart from the head after auxiliary updates and exhibit alternating *read-write phases*. These data structures, are suitable for integration with NBR. Examples of such data structures include Harris' list [42], Brown's lock-free ABTree, chromatic tree, AVL tree (B17) [45], Natarajan et al.'s lock-free binary search tree [37], and several others [36], [38], [46], [47], including the recently proposed elimination (a,b) Tree [48]. In our experiments, we utilized the Harris list and ABTree among these options.

B. Semi-Compatible Data Structures.

The need to restart from the entry point (head in lists or root in trees) at the beginning of each read phase presents a challenge in directly applying NBR to data structures with patterns of alternating *read-write phases* that do not restart from the root

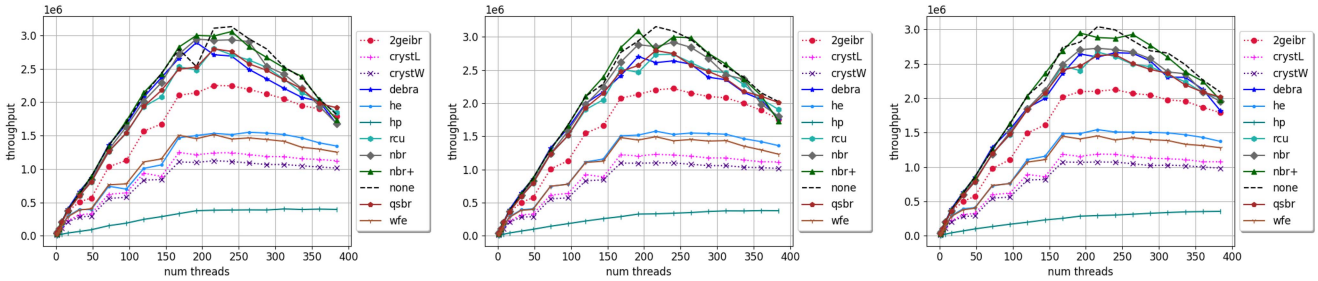


Fig. 2. Harris-Michael list. Figure showing the impact of transforming Harris-Michael list to restart from root to apply NBR is negligible. In NBR and NBR+ the HMlist restarts from root while for others it does not. Left: 10% updates. Middle: 50% updates. Right: 100% updates. Max size:20000.

node. For example, the Harris-Michael list [3] and certain search trees [35], [49], [50], [51]. However, it is possible to adapt NBR for use with these data structures by modifying their operations to restart from the root after auxiliary updates. This adaptation, requires careful consideration as it may introduce trade-offs, potentially affecting progress guarantees, necessitating a new amortized complexity analysis, or introducing additional overhead.

Nevertheless, we observe that modifying data structures to restart from the root after auxiliary updates is often a viable solution that does not significantly increase overhead. In scenarios with high contention, such as the Harris-Michael list, where multiple threads contend to unlink a same marked node, the majority of threads $(n-1)$ would already restart from the root [3]. By enforcing a restart from the root for all threads, the number of threads requiring a restart increases by just one, resulting in n threads restarting instead of $n-1$. This behavior aligns with the Harris list [42], where all threads contending on the auxiliary CAS already restart from the root. In low contention scenarios, the impact of restarting from the root on performance is minimal.

Experimental results further support the notion that the cost of restarting from the root is negligible when adapting the Harris-Michael list to work with NBR compared to using other reclamation techniques which use the list in its original form. Fig. 2 illustrates the analysis of the impact of restarting due to contention induced by varying workloads. Additionally, Fig. 6(a) and (b) (left column) provide an analysis of the impact of restarts resulting from contention induced by different data structures. These findings demonstrate that the overhead associated with restarting from the root in practical scenarios is generally low. Moreover, in search trees, given uniform node access distribution, contention is low, the performance difference between restarting from the root and continuing traversal from an ancestor is expected to be minimal due to shorter search paths.

By considering these factors, it is evident that modifying data structures to restart from the root after auxiliary updates allows for effective integration with NBR while maintaining reasonable performance characteristics.

C. Incompatible Data Structures.

Certain data structures, such as two concurrent relaxed-balance AVL trees [39], require rotations after update operations to maintain height balance. These rotations may discover new nodes that were not accessed during the previous search phase, making it impractical to reserve nodes in advance for subsequent write phases. As a result, the NBR technique does not apply to these data structures. Further, modifying their implementation to initiate rotations from the root would require extensive code changes which may require reproving progress or correctness.

Similarly, a recently introduced lock-free interpolation tree [50] periodically rebuilds its subtrees by repeatedly visiting and marking all nodes in old subtree which were not reserved beforehand to maintain balance. And, in order to visit and mark the nodes, it does not restart from the root, which violates the requirement of NBR. Consequently, NBR doesn't apply, also neither the DEBRA+ technique nor Hazard Pointers can be applied to this tree. At present, we are unaware of any SMR technique with bounded garbage that is compatible with the interpolation tree.

D. Compatibility of NBR Vs. Other SMR Algorithms

This section surveys a carefully curated list of data structures, summarized in Table I, and reports whether it is compatible with NBR and other SMR algorithms including EBR, DEBRA+ and HP (and variants of HP, including HE, IBR, WFE, ThreadScan, HY and QSense).

To safely apply NBR the following requirements should be satisfied, as was discussed in Section VI-A1 and Section III-A.

Requirement 1: Each Φ_{read} in a data structure using NBR should start from the root.

Requirement 2: All references to shared pointers that could be accessed within a Φ_{write} should be reserved before entering it.

Safely applying HPs to a data structure is more subtle. Typically, accessing shared records in a data structure operation using the original form of HPs is a three step process where pointers to records are reserved in a hand-over-hand manner [3]. **(1) announcing a hazard pointer:** requires a thread to save the shared record in a single-write multi-reader (SWMR) memory location. **(2) store-load memory order fence:** is necessary (on TSO) to ensure the reclaimers timely collect and skip freeing the announced hazard pointers. **(3) reachability validation:** is required to ensure the record was reachable from the root at the time it was announced to avoid announcing an already un-linked

TABLE I
APPLICABILITY OF SMR ALGORITHMS

Source	Data structure	Sync. type	NBR/NBR+	EBR	DEBRA+	HP/TS/IBR/HE/WFE/HY/QSense
LL05 [1]	linked list	opt. locks	Yes	Yes	No	No (reason, similar to [2])
HL01 [42]	linked list	lock-free	Yes	Yes	*	Yes
HM04 [3]	linked list	lock-free	No	Yes	*	Yes
DVY14a [35]	partially external BST	locks	**	Yes	No	No [2]
EFRB10 [36]	external BST	lock-free	No	Yes	*	No [2]
NM14 [37]	external BST	lock-free	Yes	Yes	*	No [2]
EFRB14 [49]	external BST	lock-free	No	Yes	*	No [2]
DGT15 [43]	external BST	ticket locks	Yes	Yes	No	No (no marking, cannot validate HP)
HJ12 [46]	internal BST	lock-free	Yes	Yes	*	No (similar to [2])
RM15 [52]	internal BST	lock-free	No	Yes	No	No (similar to [2])
BCCO10 [39]	partially external AVL	opt. locks	No	Yes	No	Yes
DVY14b [35]	partially external AVL	locks	No	Yes	No	No [2]
HL17 [47]	external relaxed AVL tree	lock-free	Yes	Yes	Yes	No (similar to [2])
B17b [45]	external AVL	lock-free	Yes	Yes	Yes	No [2]
S13 [38]	patricia trie	lock-free	Yes	Yes	*	No [2]
BER14 [53]	external chromatic tree	lock-free	Yes	Yes	Yes	No [2]
B17a [45]	external (a,b)-tree	lock-free	Yes	Yes	Yes	No [2]
BPA20 [50]	external interpolation tree	lock-free	No	Yes	No	No (similar to [2])

*compatible, but requires non-trivial data structure specific recovery code. **This is likely possible if code is restructured to reserve all relevant nodes before acquiring any locks.

record that could be freed simultaneously. When these three steps execute successfully, it can be said that the thread has *acquired* a hazard pointer.

Table I presents a survey of eighteen data structures, wherein NBR applies to eleven, DEBRA + applies to four and HP applies only to three. However this does not necessarily mean NBR is strictly more applicable than HP.

Qualitatively, NBR does not apply to data structure implementations where read and write phases interleave in a way that leads to violation of shared record access requirements (referred to as Requirement 1 and Requirement 2). On the other hand, it is not clear how HP should be applied to data structures where a sequence of logically deleted (or marked) nodes can be traversed [2]. This scenario arises in an important and substantial class of well known data structures, including unbalanced binary trees [35], [36], [37], [43], [46], [49], [52], relaxed AVL trees [47], Chromatic trees [53], B+trees [54], ab-trees [45], [48], linked lists [1], skip lists [55], and Euler tour trees [46]. We have not studied the applicability of NBR to *all* of these data structures, however several of them do appear in our survey.

Motivated in part by NBR, Petrank et al. [24] formalized NBR's data structure template and requirements (Requirement 1 and Requirement 2), and gave a name to data structures that satisfy them: *access-aware data structures*. This class of algorithms is the basis for the definition of what it means for an SMR algorithm to be *widely applicable* in that paper. More precisely, an SMR algorithm that can be applied to all access-aware data structure implementations is said to be widely applicable.

NBR is of course applicable to all access aware data structure implementations and is therefore widely applicable according to this definition. On the other hand, HP and its many variants are not applicable to all access-aware data structures, and are thus not widely applicable. It appears that DEBRA + is also not widely applicable as it does not support lock-based access-aware data structure implementations.

Details of HP applicability: Having discussed the requirements for ensuring safety and progress in a data structure, we next discuss the applicability of HP to the data structures in Table I.

In LL05 [1] searches are wait-free and updates use an optimistic locking pattern. If HP is applied to LL05 it is possible that a thread could repeatedly fail to acquire a HP on a node that is marked but not yet unlinked (because the thread that marked it is stalled or crashed). This breaks wait freedom for searches. DGT2015 [43], on the other hand, traverses the nodes in a synchronization-free manner (as in the lazy list) and uses version numbers and ticket locks to perform updates. However, since DGT15 doesn't use marking, it is not clear how HPs could be acquired safely.

In HJ12 [46], hazard pointer acquisition could fail indefinitely if a thread that marks a node fails before unlinking it. More specifically, threads that would try to help this failed thread to finish unlinking the node would need to acquire an HP on the marked node, and it is not clear how this HP could be acquired (without acquiring HPs to all nodes on the search path from the root). This could prevent all threads from making progress. Such issues were described in [2]. Similarly, in HL17 [47] and the concurrent interpolation search tree of Brown et al. [50], it is not clear how a thread trying to acquire an HP on a node could verify that it is still reachable from the root. BCCO10 [39] uses lazy deletion in the sense that to delete a node with 2 children it converts it to a routing node (treating this as if the AVL tree is an external tree) and this routing node is deleted lazily at the time of next rebalancing step that involves it. Thus, it appears that we can use HP as one can leverage version based validation to know when a node is definitely reachable and if such a validation fails one can simply restart from the root. Note this doesn't theoretically impact progress guarantee of searches (unlike lazylist or DGT15) as they are already blocking due to hand over hand optimistic validation. However, in practice, HPs would necessitate restarts from the root when validation fails, whereas BCCO nominally attempts to continue traversal

from the parent node in this case. The logical ordering tree of DVY14 [35] is based on BCCO10, but DVY14 does not use version numbers, so a search has no way to tell whether a node is currently in the tree. Thus, it is not clear how one could use HPs in DVY14.

Details of NBR applicability: Reasoning about the applicability of NBR is comparatively easy, as one just needs to confirm that every Φ_{read} restarts from root, and that it is possible to reserve all records before entering a Φ_{write} (Requirement 1 and 2, respectively). In other words, each thread should restart from root after *helping* in search phase and no new pointers to shared records should be accessed in a Φ_{write}.

NBR, at first may appear to be easily applicable to EFRB10 [36] as it has lockless searches which either end in an update or helping and operations restart searches from root after helping. However, one helping step could lead to recursively more helping updates which could discover new pointers that could not be known before the first helping update. As a result such pointers can not be reserved before the first helping update (write phase), (violating Requirement 2). Similarly, NBR could not be applied to EFRB14 [49] as, after helping, an operation restarts from nearby ancestors and not from the root (violating Requirement 1). Restarting from a nearby ancestor is crucial to achieve the amortized complexity result in that work.

In HJ12 [46], the first lock-free internal BST, searches are sometimes required to help an ongoing update to avoid missing a node that is concurrently moved upwards in the tree, a situation that can arise when a search is concurrent with a two-child delete operation. However, in HJ12, after helping, searches restart from the root (satisfying NBR 's requirement that a Φ_{read} should start from root), and all records required to do the helping update can be known beforehand through the descriptor (satisfying NBR 's requirement that all records to be accessed in a Φ_{write} should be reserved). Thus, NBR can be used in HJ12.

Similarly, data structures designed using the tree update template of Brown et al. [45], for example lock-free chromatic trees, relaxed (a,b)-trees, relaxed AVL trees, and weak AVL trees HL17 [47] are compatible with NBR. In the template, operations do typically synchronization-free searches to find target node(s) (similar to our standard Φ_{read}), then check whether they need to help (similar to entering Φ_{write} during a search), and in the event that an operation O helps another, O typically restarts from the root. Helping is implemented using descriptors which contains pointers to all nodes that would be required to execute the helping update, so NBR could reserve all node pointers in the descriptor and enter Φ_{write} for the helping update and then restart its search (Φ_{read}) from the root.

The lock-free Patricia Trie (S13) [38] too follows a pattern in which searches are synchronization-free, and updates may perform auxiliary helping using descriptors and then restart from the root. Thus, NBR applies to S13 as we can know all records to be accessed in Φ_{write} beforehand, and after helping, the operation can be restarted from the root, satisfying both Requirement 1 and 2.

DGT15 [43] has sync-free searches followed by locking predetermined nodes for updates, which is similar to the Φ_{read} followed by single Φ_{write} pattern of LL05 [1]. Thus, NBR applies to DGT15.

The unbalanced external BST of Natarajan and Mittal (NM14) [37] too has a pattern where each operation starts with synchronization-free search that returns a $SeekRecord$ object followed by possible modifications to the data structure. During updates, the operation may possibly help by accessing nodes only pointed by a $SearchRecord$ object and then subsequently restart from root. Deletion consists of two modes: injection and cleanup. Both involve data structure modification, therefore, both should be done in NBR 's Φ_{write}. Additionally, it is possible that injection mode may succeed and subsequent cleanup may fail, in that case the operation is required to start from the root, satisfying the requirements for NBR.

The unbalanced external BST in DVY14 [35] does a synchronization-free search and then executes updates using locks. But within the update phase it may obtain pointers to new nodes, for example, the successor, children or parent of a node, which may violate the Requirement 2 of NBR. So, in order to apply NBR to DVY14, one must modify DVY14 to perform all of the aforementioned reads of new record pointers in the update phase before the first lock is acquired, then validate after lock acquisitions that the values of those reads have not changed (and restart if validation fails). Note, NBR would not work with the balanced variant of the DVY14 tree, which does bottom-up rebalancing without restarting from the root between rotations.

BCCO [39] Uses optimistic concurrency control (OCC) techniques to avoid locking nodes in searches as much as possible but occasionally locks and immediately unlocks a node as part of traversal, which suggests NBR cannot be used. However, as described in [56], this locking is an optional part of the BCCO algorithm (intended to improve fairness under heavy contention) and can simply be removed. Unfortunately, this algorithm performs recursive bottom-up rebalancing without restarting from the root between rotations. To use NBR, one would need to restart from the root after each rotation, which would be a substantial algorithmic change.

In RM15 [52], an insert operation, does a sync-free search followed by a CAS on an appropriate child pointer. If the CAS fails, the pointer is re-read to find a descriptor, which helps complete the conflicting operation and the operation restarts from the root. For safe application of NBR, we must reserve both children of node, and if either child is actually a descriptor, we must reserve all pointers in the descriptor. At this point, if we see a descriptor, we might as well help it before attempting the CAS (if the CAS would be doomed to fail anyway). Unfortunately, this changes the progress argument (although we don't think it actually changes the progress property). Delete in RM15 are the real problem for applying NBR. after the inject part, the cleanup part requires obtaining and dereferencing new pointers without restarting from the root. modifying this algorithm to work with NBR would require sweeping changes.

E. Ease of Use

Using NBR in compatible data structures is simpler compared to most reclamation techniques, although it is not as straightforward as DEBRA. The process involves identifying the start of a Φ_{write} and all records that will be needed to execute

the Φ_{write} to place ENDΦ_{write}() in data structure operation. This is similar to using two-phase locking. Additionally, the programmer needs to identify when a Φ_{read} begins to place a checkpoint and BEGINΦ_{read}(). A quantitative comparison and detailed discussion on required programming effort required to use NBR, DEBRA and HP can be found in [27].

VII. Correctness

Assumption 3: If T_i in order to reclaim its retired records sends a signal to T_j, then by the time T_i finishes sending the signal, T_j is guaranteed to receive it and execute a signal handler before taking further steps in its program.

Property 4: A thread T_j in Φ_{read}

4.1 upon receiving a signal executes a signal handler and restarts from the entry point of a data structure (i.e gets neutralized).

4.2 is permitted to dereference reference fields of shared records to discover new shared records (unless it is neutralized).

Property 5: A thread T_j in Φ_{write}

5.1 reserves all records to be used in Φ_{write} before entering Φ_{write}.

5.2 upon receiving a neutralizing signal simply continues its execution as if no signal was received.

5.3 does not access any records it did not reserve prior to entering Φ_{write}.

Property 6: Every *reclaimer* thread T_r does the following in order:

6.1 sends signals to all participating threads.

6.2 scans all reserved records of each participating thread T_j.

6.3 reclaims records in its bag that are not reserved.

Lemma 7: A reclaiming thread T_r is guaranteed to scan all the reservations of a thread T_w if T_w enters its Φ_{write} before it is signalled by T_r.

Proof: A thread T_w reserves records by announcing them in a single writer multi reader array. On x86/64, this announcement must be followed by a memory ordering instruction to ensure that the reservations are visible when the thread enters its Φ_{write}. Specifically, NBR uses a CAS to reset a thread local variable *restartable* which simultaneously begins Φ_{write} and ensures that the preceding reservations are visible to any other thread that sees T_w is in Φ_{write}.

Let t_{res} be the time at which the last reservation was made and t_{wp} be the time at which Φ_{write} began. As we just explained, this means **(A)** $t_{res} < t_{wp}$.

To show that T_r observes all of T_w's reservations, we show that $t_{scan} < t_{res}$.

From, Property 6 we know that **(B)** $t_{sig} < t_{scan}$, where t_{sig} is when T_r sent its last signal, and t_{scan} is when it reads the first reservation slot of T_w. And, since T_w was in Φ_{write} when it received the signal, we have **(C)** $t_{wp} < t_{sig}$. By **(A)**, **(B)** and **(C)**, $t_{res} < t_{wp} < t_{sig} < t_{scan}$. □

Lemma 8 (NBR is Safe): No *reclaimer* thread in NBR reclaims an unsafe record.

Proof: Suppose to obtain a contradiction that some *reclaimer* T_r reclaims an unsafe record rec. This can occur in only two ways: (1) a *writer* accesses a record rec that it did not reserve, or (b) a *reader* accesses a record rec in the limboBag of T_r that is being reclaimed.

It is easy to argue that (1) does not happen. By Property 6, T_r must have sent a signal to the *writer* before reclaiming rec. By Property 5, if the *writer* accesses rec it must reserve rec before entering Φ_{write}.

Next we show (1) does not happen. As above, by Property 6, T_r must have sent signal to the *reader* before reclaiming rec. Once T_r has sent this signal, Assumption 3 and Property 4 imply that the *reader* will execute its signal handler as its next step in its execution, at which point it will discard any private reference to rec. Consequently, by the time T_r begins reclaiming records in its limboBag, the *reader* will no longer have access to rec. □

Lemma 9 (NBR+ is safe): No *reclaimer* thread in NBR+ reclaims an unsafe record.

Proof: In NBR+, threads can reclaim at the *LoWatermark* or at the *HiWatermark*. Reclamation at the *HiWatermark* is similar to reclamation in NBR, and the argument that reclaimers at the *HiWatermark* do not reclaim unsafe records is similar to Lemma 8.

It remains to prove that reclamation at the *LoWatermark* is safe. We argue that any record rec reclaimed by a thread T_{lw} at the *LoWatermark* must be safe. In other words, rec must not be reserved by any thread, and no thread should have a private reference to rec.

In NBR+, T_{lw} reclaims only up to its *bookmarkTail*, which means it only reclaims records that it had retired before the time t when T_{lw} reached the *LoWatermark* and scanned the timestamps of all threads. And, T_{lw} reclaims these records only at a later time t' when it sees that all threads' timestamps have been incremented at least twice. These timestamp increments indicate that, between times t and t', all threads received signals and discarded their pointers to unreserved records. Since rec is retired before time $t < t'$, it follows that at time t' any thread that has a reference to rec must have reserved rec before t'. If rec is still reserved when T_{lw} scans the reservations of all threads (after t') then rec will not be reclaimed. Otherwise, rec is safe to reclaim. □

Lemma 10 (Both NBR and NBR+ are robust): The number of records that are retired but not yet reclaimed is bounded.

Proof: Let k be an upper bound on the number of records a thread reserves per operation, p be the number of processes, and h be the maximum limboBag size at which a thread decides to reclaim (i.e., the *HiWatermark*). Let, T_r be a *reclaimer* and T_j be an arbitrary thread. If T_j is delayed (or crashes), it can reserve at most k records in T_r's limboBag. A retired record can be present in only one limboBag, so in the worst case a single thread can prevent only k records from being reclaimed (across all limboBags). It follows that, in a system where $p-1$ threads can crash, those $p-1$ threads can prevent at most $k(p-1)$ records in total from being reclaimed. Moreover, a *reclaimer* always reclaims when it hits the *HiWatermark*, so a limboBag contains at most h records. □

Corollary 11: A thread can prevent only the records it reserves in a single operation from being reclaimed.

In NBR, we assume the number of records that can be reserved by a single data structure operation is smaller than the limboBag size (otherwise a thread could prevent all records in a limboBag from being reclaimed). Data structures typically require only a small number of reservations per operation. For example in our experiments, the lazy linked list [1] required a maximum of two reservations per operation, and the harris list [42], DGT binary search tree [43], and relaxed (a,b) tree [45] needed to reserve a maximum of three records per operation.

VIII. Immediacy of Signal Delivery in Practice

NBR assumes signal delivery is immediate for safety (Assumption 3), so that neutralizing signals are delivered to target threads before the sender initiates reclamation. We examine the signal handling code of two open source operating systems, FreeBSD and Linux, and find that signal delivery is immediate, unless the receiving thread has masked the signal or has exited. Additionally, our experiments, spanning over 10 hours, confirm low latency between signal generation and delivery, supporting Assumption 3.

To understand POSIX signal guarantees better, we analyze the steps involved in the signal delivery process. First, the signaling thread calls `pthread_kill`, which invokes a system call and transfers control to the operating system (kernel switch). Second, the kernel processes the call by marking the signal as pending in the target thread's per-thread structure (e.g., thread in FreeBSD). Third, if the thread is currently running, the kernel sends an Inter-Processor Interrupt (IPI) to asynchronously interrupt the target thread and transfer control to the operating system. Fourth, the destination core is interrupted (assuming an IPI is sent) through the local APIC, in case of x86 systems. Lastly, when the thread is resumed, the signal is delivered to the user space, followed by the execution of the user space signal handler.

For our purposes, we are primarily concerned with the point at which, after the signal is sent, we can guarantee that the target thread stops executing new user space instructions. For non-running target threads, this is true when the sender returns from its `pthread_kill` because the `pthread_kill` notifies the target thread that a signal is pending by setting a flag in a pending signal vector in the kernel thread structure. So, when the target thread is scheduled to run again, it checks its signal pending mask and immediately delivers the signal. In the case of running threads, the kernel employs an IPI to interrupt the core where the target thread is executing. This is an *asynchronous* process, which may need to wait for a certain amount of time to confirm the delivery of the IPI.

We have observed that, in practise, this waiting time for IPI delivery is reasonably low. To determine the upper bound of this waiting time, we design several benchmarks involving two threads placed on consecutive CPU cores within the same socket. The first thread sends a signal to the second thread by invoking `pthread_kill`. We measure the end-to-end times (CPU cycles required to send and deliver a signal) using the `rdtsc` instruction and further breakdown the measured times by using DTrace to analyze internal operating system functions which allows us to identify signals that induce an IPI.

TABLE II
Breakdown of the Approximate Signal Delivery Latencies in Cycles

Operation	CPU Cycles
end-to-end signal delivery	6527
sending ipi	393
local signal delivery	4324
`pthread_kill`	1920

We break down the pthread_kill into the call cost and the actual sending of the IPI. Signals are delivered using pthread_kill to a thread in another core in the same socket.

The latency is divided into two parts: the latency of transmitting the IPI to the target core and how the processor drains or aborts the remaining in-flight instructions. The IPI is sent at the end of the `pthread_kill` operation, and the local APIC completes the delivery (hardware portion) asynchronously. To establish an upper bound on the end-to-end signal delivery cost, we measured the local delivery cost using the int3 instruction, which triggers an exception that is delivered faster than an interrupt routed through the local APIC. This measurement represents the lower bound of the signal delivery cost on the destination core, as externally generated interrupts will take longer than an interrupt generated by the instruction stream.

It is worth noting that if multiple interrupts are pending, the delivery of the IPI may be delayed. However, for the purpose of our wait time analysis, once the first IPI is received and delivered at the target core, all other queued IPIs will be delivered immediately. Hence, it is safe to assume that no further instructions will be executed.

Table II presents the measurements for the number of CPU cycles required for signal sending and delivery, including end-to-end delivery, the cost of sending an IPI, the local signal delivery cost, and the time to execute a `pthread_kill` syscall. We have experimented with multiple hardware platforms, but we report these measurements on an Intel Xeon Gold 6342 CPU running at 2.80 GHz (Ice Lake) with FreeBSD 13.0.

We approximate the maximum waiting time for the IPI to complete by subtracting the local signal delivery time from the end-to-end time. This computation provides an upper bound on the moment when the processor stops executing user-space instructions. Additionally, we subtract the time it takes to complete `pthread_kill` since the source cannot initiate reclamation until control is returned from the operating system. This calculation results in unaccounted cycles that vary from zero to 283 cycles, on multiple machines we tested.

The machines used for evaluation of NBR in our experiments section had no unaccounted for cycles. This may be in part due to the extra overhead in returning from kernel mode when KPTI is enabled, which masks the signal delivery latency, giving the effect of immediate signal delivery. However, on the Intel's Icelake machine we calculated roughly 283 cycles that we can not account for. The unaccounted cycles represent the latency required for the two Local APICs to communicate the IPI request between the cores and any additional cost for interrupt delivery. Thus, on this machine threads should wait for 283 cycles before

initiating reclamation to ensure Assumption 3's validity. This latency is within a small factor of the cost of a cache line transfer between L2 caches.

In general, we have observed that signaling mechanism is designed to be fast and the signal delivery cost is low. However, it is important to acknowledge that for some specific processor designs the signal delivery cost may vary. While the exact variations are dependent on the specific architecture, we believe that it is still feasible to approximate the wait times by employing the profiling and benchmarking technique similar to ours. By carefully analyzing the characteristics of the architecture at hand, it is possible to infer reliable wait time for threads to initiate reclaiming after sending signals to ensure validity of Assumption 3, on such architectures. Alternatively, utilizing fast user space interrupts could also be explored to address this.

IX. Experimental Evaluation

We rigorously evaluate NBR and NBR+. To the best of our knowledge, this is the most extensive evaluation to date in terms of the number of reclamation schemes and variety of data structures evaluated.

Setup: We conducted our experiments on a quad-socket Intel Xeon Platinum 8160 machine with 192 threads, 384 GB of RAM, 33 MB of L3 cache per socket (total 132 MB). The machine ran Ubuntu 20.04 with kernel 5.8 with GCC 9.3.0. We implemented all algorithms in the Setbench [50] benchmark with `-O3` optimization flag and utilized *jemalloc* as the memory allocator [57]. Our evaluation consisted of three types of experiments:

(E1): Evaluating NBR (+) throughput with different workloads thread counts to understand scalability [P1].
(E2): Measuring throughput with varying data structure sizes to understand the impact of contention and cache misses[P4].
(E3): Evaluates peak memory usage of NBR+ with and without stalled threads to understand its memory behaviour [P2].

We studied multiple data structures with varying memory access patterns including of lists, hash tables, and trees. Specifically, we utilized the lazylist (LL) [1], Harris list (HL) [42] and Harris-Michael list (HMList) [3] for lists, HMList chaining based hashtable (HMHT), for hash table, and external binary search tree of David et al. (DGT) [43] and Brown's relaxed (a,b)-tree (BABT) [45] for trees.

LL and DGT belong to the class of data structures that are naturally compatible with NBR as they have a single Φ_{read} and Φ_{write} per operation. HL and BABT belong to the class of data structures with multiple Φ_{read} and Φ_{write} that start every Φ_{read} from root. Thus, they are suitable for NBR with careful separation of the phases as discussed in Section VI-A. Finally, HMList and HMHT represent the semi-compatible data structures discussed in Section VI-B. Each of these data structures is implemented with up to **twelve** different reclamation algorithms (as applicable), including NBR, NBR+ and *none* algorithm which is a leaky implementation. The reclamation algorithms are summarized in Table III.

TABLE III
SMRs Used in Benchmark

SMR Algorithm	Bounds Garbage?	Progress	Type
NBR+ [27]	Yes	cond. lock-free	OS + HP
NBR [27]	Yes	cond. lock-free	OS + HP
DEBRA [2]	No	blocking	EPOCH
HP [3]	Yes	lock-free	HP
QSBR & RCU [13]	No	blocking	EPOCH
2GEIBR [13]	No	lock-free	EPOCH + HP
HE [17]	No	lock-free	EPOCH + HP
WFE [28]	Yes	wait-free	EPOCH + HP
crystallineL [44]	Yes	lock-free	EPOCH + RC
crystallineW [44]	Yes	wait-free	EPOCH + RC

OS implies use of Operating System features, HP: hazard pointers like reservation. EPOCH: use of EBR. RC: reference counting.

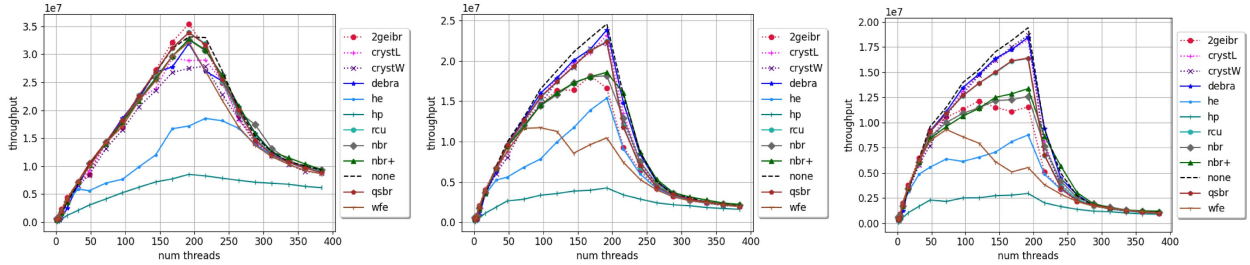
Trees do not include the lines for crystallineL and crystallineW. Although these algorithms worked correctly with lists, they occasionally crashed when used with trees in oversubscribed scenarios, and it is unclear how to use them correctly in this context. Therefore, we decided to exclude them from the tree plots. Additionally, since HP does not apply to HL [3], it was omitted from the HL plots. Furthermore, VBR [23] was not used in our experiments because it assumes a type-stable allocator that never frees memory to the OS, and a fair comparison would require forcing all algorithms to use memory pools. The reported results are obtained by averaging data from 5 timed trials, each lasting 5 seconds. The experiments were conducted with thread counts ranging from 1 to 384 threads on a system with 192 hardware threads and more than 91% of data points had less than 5% variance. Before each execution, the data structure was pre-filled to half of its maximum size.

For each of (E1), (E2), and (E3) we measure throughput for three workloads, (1) *update-intensive:* 100% updates where 50% of operations are inserts and the rest are deletes, (2) *balanced:* 25% of operations are inserts, 25% are deletes and rest are searches, and (3) *search-intensive:* 5% of the operations are inserts, 5% are deletes and the rest are searches.

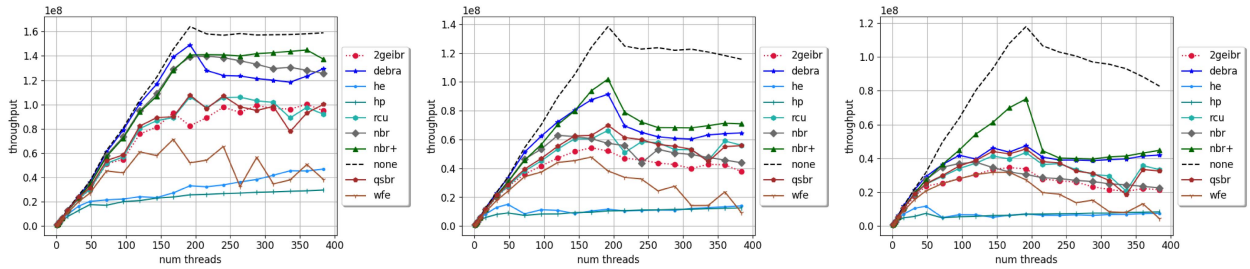
A. Discussion of E1

Figs. 3, 4, and 5 show the throughput of all the data structures with varying workloads. In general, it can be observed that especially in balanced and update intensive workloads, NBR+ is similar to other reclamation techniques at lower thread counts, i.e up to 48 threads. Then at higher thread counts, from 48 to 192 threads, NBR+ is better than other reclamation techniques, except in the LL and HMList data structures. In oversubscribed scenarios, i.e., for thread counts greater than 192, NBR+ is competitive and sometimes outperforms the competition.

One interesting observation is the cross-over between DEBRA and NBR+ at the higher thread counts, particularly evident in the DGT, BABT and HMHT data structures. The slowdown of DEBRA at higher thread counts could be attributed to the *delayed thread vulnerability*, where slow threads infrequently advance epochs, halting regular reclamation of limbo bags. This results in the accumulation of a large number of retired records waiting to be reclaimed. When the slow thread finally

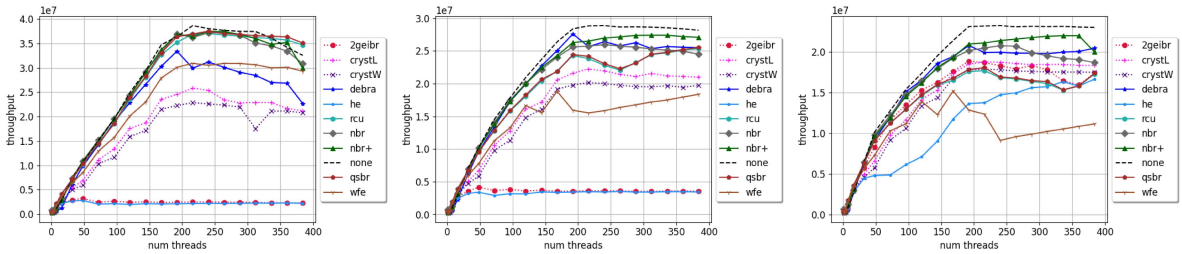


(a) Lazylist (LL) throughput. Updates: Left: 10%. Middle: 50%. Right: 100%. Max size:2000.

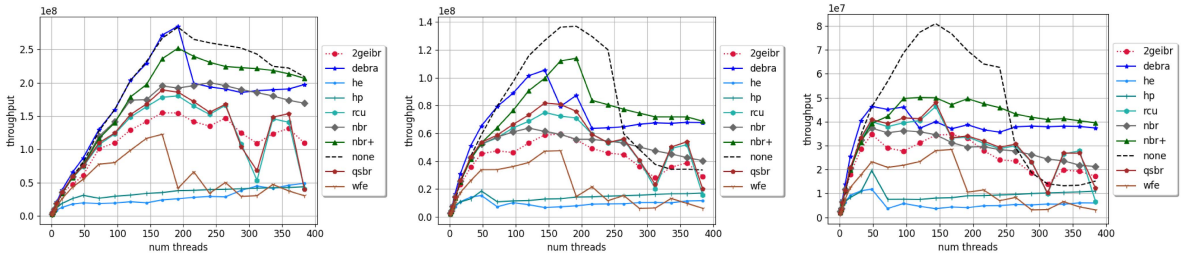


(b) DGT throughput. Updates: Left: 10%. Middle: 50%. Right: 100%. Max size:2000000.

Fig. 3. **E1:** Evaluation with data structures having a single read-write phase. Y axis: throughput in million operations per second. X-axis: #threads.



(a) Harris list (HL) throughput. Updates: Left: 10%. Middle: 50%. Right: 100%. Max size:2000.



(b) (a,b) tree (BABT) throughput. Updates: Left: 10%. Middle: 50%. Right: 100%. Max size:2000000.

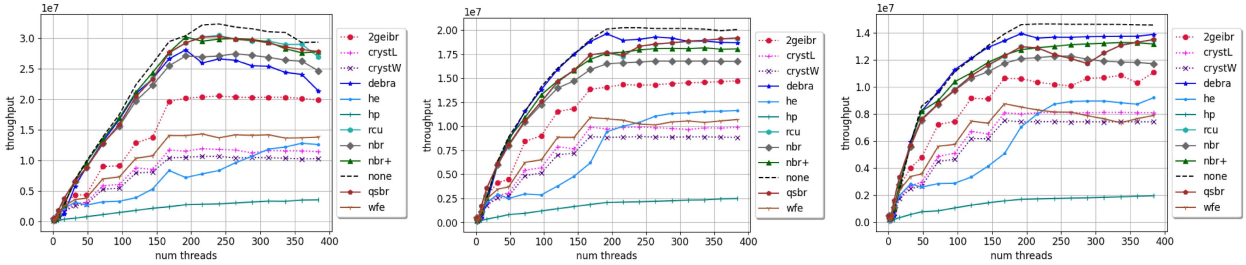
Fig. 4. **E1:** Evaluation with data structures having multiple read-write phases. Y axis: throughput in million operations per second. X-axis: #threads.

announces the latest epoch, all threads reclaim their large limbo bags, causing a *reclamation burst* (many records being freed at once). This burst harms overall throughput as reclamation bursts bottleneck the underlying allocator by increasing contention and triggering slow code paths as internal buffers in the allocator are quickly filled. The probability of threads getting delayed increases as more threads get involved in high inter-socket and update-intensive computations. Furthermore, the thread count where NBR+ overtakes DEBRA is lower in the *update-intensive* than the *search-intensive* workloads. This is because the overhead of burst reclamation sets in at lower thread counts for *update-intensive* workloads.

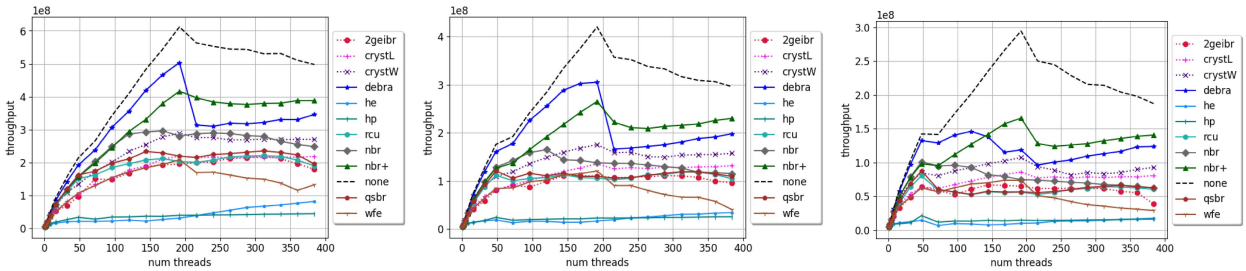
B. *Discussion of E2*

In the second experiment, we evaluate all the reclamation algorithms with data structures of small and very large sizes. This serves two purposes. First, this illustrates the behavior under high contention at small sizes and low contention at large sizes. Second, this experiment also studies the impact of varying cache miss rates on throughput when data fits in the LLC (last level cache), and when it does not.

Fig. 6, shows the HMList (left column), DGT (middle column), and BABT (right column). The first row (Fig. 6(a)) depicts these data structures with sizes that fit in the LLC. The HMList

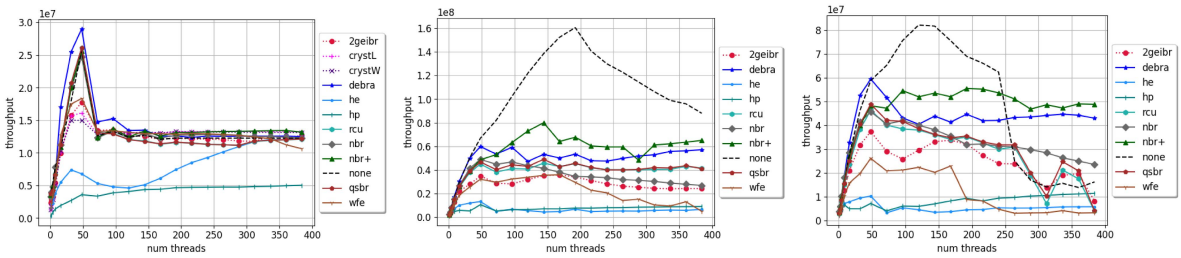


(a) Harris-Michael list (HMList) throughput. Updates: Left: 10%. Middle: 50%. Right: 100%. Max size:2000.

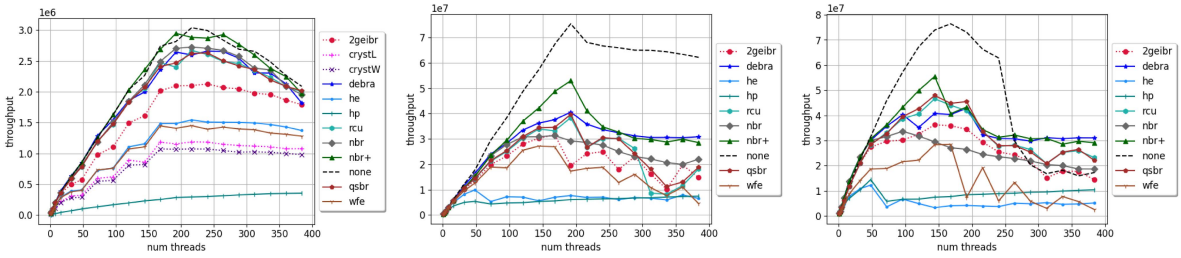


(b) HMlist-based hashtable (HMHT) throughput. Updates: Left: 10%. Middle: 50%. Right: 100%. load factor:6. Buckets:10K.

Fig. 5. **E1:** Evaluation with data structures that can be modified to restart from root/head. Y axis: throughput in million operations per second. X-axis: #threads. In NBR and NBR+ the HMlist restarts from root while for others it does not.



(a) Left: Harris-Michael list size 200. Middle: DGT size 200K. Right: BABT size 200K.



(b) Left: Harris-Michael list size 20K. Middle: DGT size 20M. Right: BABT size 20M.

Fig. 6. **E2:** Throughput across different data structure sizes. Workload: 100% updates. Y axis: throughput in million operations per second. X-axis: #threads. Trees size of 20 M exceeds LLC. Trees size of 200 K fits LLC. Lists always fit in LLC.

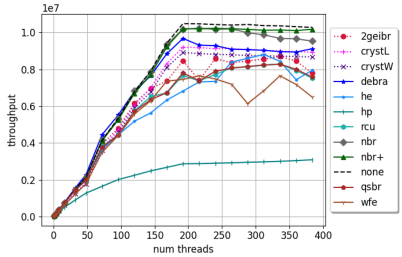


Fig. 7. **E2:** HMList-based Hash table with load factor 600 that does not fit in LLC.

is of size 200 and both the trees are of size 200 K. The second row (Fig. 6(b)) depicts these data structures with sizes that do not fit in LLC. The HMList is of size 20 K and both the trees are of size 20 M. Note, the list, even at 20 K size, fits in LLC and it is not possible to go beyond the size of 20 K and complete the experiments in a reasonable amount of time. So, to emulate the case of very low contention in lists we use the maximum size of 20 K. Additionally, in Fig. 5(b) and Fig. 7 we include results for chaining hash tables with 60 K buckets (fits in the LLC) and 6 M buckets (exceeds the LLC), with load factors 6 and 600, respectively.

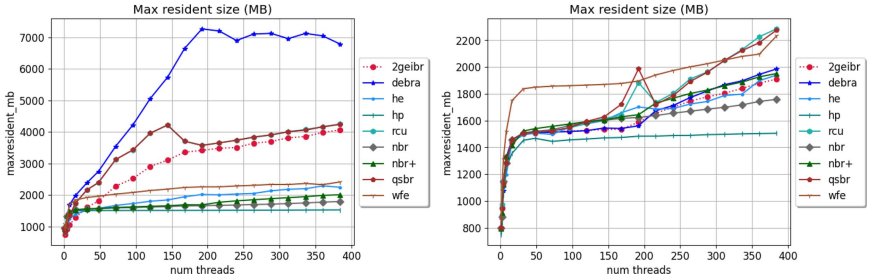


Fig. 8. **E3:** Left: DGT with stalled threads. Right: DGT with no stalled threads. Max Size: 20 M.

Interestingly, NBR+ outperforms the competition, especially at the high thread counts. When combined with the analysis of E1, it is clear that NBR+ is fast [P1] and consistent [P4]. Specifically, NBR+ is comparable to other fast epoch-based algorithms like DEBRA, QSBR, RCU for the HMList at both the small and large sizes (see the left column in Fig. 6(a), (b)). In DGT (see the middle column in Fig. 6(a), (b)), NBR+ is significantly faster at higher thread counts. Similarly, in BABT (see the right column in Fig. 6(a), (b)) NBR+ is faster at small sizes and comparable to other reclamation algorithms at large sizes. In the case of HMHT, NBR+ is comparable at small sizes (see the right column in Fig. 5(b)) as well as at large sizes (see Fig. 7).

Furthermore, experiment E2 also helps to explore two disparate usage scenarios for semicompatible data structures like HMList and the HMHT. First, at a large size, for example 20 K in HMList and a load factor of 600 in HMHT, we hypothesize restarts would be inexpensive due to low contention. Second, at a small size, i.e. 200 in HMList and the load factor of 6 in HMHT, we assume that restarting from the head node will occur frequently due to high contention, therefore it should have high overhead and slowdown the data structures.

For low contention, the NBR+ based implementation which enforces restarts is still faster than other reclaimers which do not restart, in HMlist and HMHT. Surprisingly, even for high contention, NBR+ outperforms other reclaimers. This suggests that, at least for semicompatible lists, NBR 's methodology is sufficiently fast and restarts in the data structure incur low overhead.

C. Discussion of E3

In our third type of experiment, we measure peak memory consumption of all reclaimers when a thread is stalled (see the middle column in Fig. 8) and when no thread is stalled (see the right column in Fig. 8) to establish that NBR (+) bounds garbage.

Each trial is run for 40 seconds. For DEBRA, QSBR, RCU, HP, HE, NBR and NBR+ one thread is made to sleep within a data structure operation, for whole 40 seconds, imitating a stalled thread. And, for 2GEIBR, one thread is made to stall to maximize the interval of reservations such that maximum number of nodes are prevented from getting reclaimed, imitating pathological scheduling.

As expected, DEBRA, 2GEIBR, QSBR and RCU do not have bounded garbage, show an increase in peak memory usage when a thread stalls (Fig. 8). In contrast, NBR, NBR+, HP exhibit similar memory usage regardless of thread stalls. In HE and WFE memory usage increases marginally as a result of the stalled thread, showing some minor sensitivity to stalled threads. In summary, our experiments reveal that NBR is fast[P1], bounds garbage [P2], and is consistent [P4]. Additionally, we also show that NBR can be fairly easily integrated [P3] in multiple data structures [P5].

Code: (https://gitlab.com/aajayssingh/nbr_setbench_plus).

X. Conclusion

We proposed two neutralization based reclamation techniques (NBR and NBR+) for safely reclaiming memory in dynamic concurrent data structures. Provided analysis of their correctness and progress guarantees, investigated their safe usage on modern operating systems, and conducted a survey to assess the applicability of these techniques. Neutralization based reclamation leverage the common pattern observed in many concurrent data structures, characterized by their read-write phases. By utilizing POSIX signals, this technique establishes a lightweight coordination mechanism among threads, preventing use-after-errors. Future exploration of user-space signals could further enhance these techniques' effectiveness. Overall, our experiments demonstrate that these techniques provide fast and consistent performance and bound garbage. Furthermore, our integration of these techniques with multiple data structure shows their simplicity and widely applicabililty when compared to other hybrid approaches.

References

[1] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit, "A lazy concurrent list-based set algorithm," in *Proc. Int. Conf. Princ. Distrib. Syst.*, 2005, pp. 3–16.

[2] T. A. Brown, "Reclaiming memory for lock-free data structures: There has to be a better way," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2015, pp. 261–270.

[3] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, Jun. 2004.

[4] A. Braginsky, A. Kogan, and E. Petrank, "Drop the anchor: Lightweight memory management for non-blocking data structures," in *Proc. 25th Annu. ACM Symp. Parallelism Algorithms Architecture*, 2013, pp. 33–42.

[5] J. Kang and J. Jung, "A marriage of pointer-and epoch-based reclamation," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2020, pp. 314–328.

[6] D. L. Detlefs, P. Martin, M. Moir, and G. Steele Jr., "Lock-free reference counting," *Distrib. Comput.*, vol. 15, no. 4, pp. 255–271, 2002.

[7] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas, "Efficient and reliable lock-free memory reclamation based on reference counting," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 8, pp. 1173–1187, Aug. 2009.

[8] T. Hart, P. McKenney, A. D. Brown, and J. Walpole, "Performance of memory reclamation for lockless synchronization," *J. Parallel Distrib. Comput.*, vol. 67, no. 12, pp. 1270–1285, 2007.

[9] N. Cohen and E. Petrank, "Automatic memory reclamation for lock-free data structures," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 260–279, 2015.

[10] O. Balmau, R. Guerraoui, M. Herlihy, and I. Zablotchi, "Fast and robust memory reclamation for concurrent data structures," in *Proc. 28th ACM Symp. Parallelism Algorithms Architecture*, 2016, pp. 349–359.

[11] D. Alistarh, W. Leiserson, A. Matveev, and N. Shavit, "Threadscan: Automatic and scalable memory reclamation," *ACM Trans. Parallel Comput.*, vol. 4, no. 4, pp. 1–18, 2018.

[12] N. Cohen, "Every data structure deserves lock-free memory reclamation," in *Proc. ACM Program. Lang.*, 2018, pp. 1–24.

[13] H. Wen, J. Izraelevitz, W. Cai, H. A. Beadle, and M. L. Scott, "Interval-based memory reclamation," *ACM SIGPLAN Notices*, vol. 53, no. 1, pp. 1–13, 2018.

[14] D. Dice, M. Herlihy, and A. Kogan, "Fast non-intrusive memory reclamation for highly-concurrent data structures," in *Proc. ACM SIGPLAN Int. Symp. Memory Manage.*, 2016, pp. 36–45.

[15] M. Herlihy, V. Luchangco, P. Martin, and M. Moir, "Nonblocking memory management support for dynamic-sized data structures," *ACM Trans. Comput. Syst.*, vol. 23, no. 2, pp. 146–196, 2005.

[16] G. E. Blelloch and Y. Wei, "Concurrent reference counting and resource management in wait-free constant time," 2020, *arXiv:2002.07053*.

[17] P. Ramalhete and A. Correia, "Brief announcement: Hazard eras-non-blocking memory reclamation," in *Proc. 29th ACM Symp. Parallelism Algorithms Architecture*, 2017, pp. 367–369.

[18] N. Cohen and E. Petrank, "Efficient memory management for lock-free data structures with optimistic access," in *Proc. 27th ACM Symp. Parallelism Algorithms Architecture*, 2015, pp. 254–263.

[19] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit, "Stacktrack: An automated transactional approach to concurrent memory reclamation," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 1–14.

[20] A. Dragojević, M. Herlihy, Y. Lev, and M. Moir, "On the power of hardware transactional memory to simplify memory management," in *Proc. 30th Annu. ACM SIGACT-SIGOPS Symp. Princ. Distrib. Comput.*, 2011, pp. 99–108.

[21] D. Alistarh, W. Leiserson, A. Matveev, and N. Shavit, "Forkscan: Conservative memory reclamation for modern operating systems," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 483–498.

[22] R. Nikolaev and B. Ravindran, "Hyaline: Fast and transparent lock-free memory reclamation," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2019, pp. 419–421.

[23] G. Sheffi, M. Herlihy, and E. Petrank, "VBR: Version based reclamation," in *Proc. 33rd ACM Symp. Parallelism Algorithms Architecture*, 2021, pp. 443–445.

[24] G. Sheffi and E. Petrank, "The era theorem for safe memory reclamation," in *Proc. 28th ACM SIGPLAN Annu. Symp. Princ. Pract. Parallel Program.*, 2023, pp. 435–437.

[25] J. Jung, J. Lee, J. Kim, and J. Kang, "Applying hazard pointers to more concurrent data structures," in *Proc. 35th ACM Symp. Parallelism Algorithms Archit.*, 2023, pp. 213–226.

[26] A. Singh, T. Brown, and M. Spear, "Efficient hardware primitives for immediate memory reclamation in optimistic data structures," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Petersburg, FL, USA, May 15–19, 2023, pp. 112–122, doi: 10.1109/IPDPS54959.2023.00021.

[27] A. Singh, T. Brown, and A. Mashtizadeh, "NBR: Neutralization based reclamation," in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2021, pp. 175–190.

[28] R. Nikolaev and B. Ravindran, "Universal wait-free memory reclamation," in *Proc. 25th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2020, pp. 130–143.

[29] K. Fraser, "Practical lock-freedom," Comput. Lab., Univ. Cambridge, Cambridge, U.K., Tech. Rep. UCAM-CL-TR-579, 2004.

[30] P. E. McKenney and J. D. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," in *Proc. Parallel Distrib. Comput. Syst.*, 1998, pp. 509–518.

[31] D. Anderson, G. E. Blelloch, and Y. Wei, "Concurrent deferred reference counting with constant-time overhead," in *Proc. 42nd ACM SIGPLAN Int. Conf. Program. Lang. Des. Implementation*, 2021, pp. 526–541.

[32] A. Singh, T. Brown, and A. Mashtizadeh, "NBR: Neutralization based reclamation," 2020, *arXiv:2012.14542*.

[33] T. Brown, F. Ellen, and E. Ruppert, "A general technique for non-blocking trees," in *Proc. 19th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2014, pp. 329–342.

[34] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. E. Tarjan, "The CB tree: A practical concurrent self-adjusting search tree," *Distrib. Comput.*, vol. 27, no. 6, pp. 393–417, 2014.

[35] D. Drachsler, M. Vechev, and E. Yahav, "Practical concurrent binary search trees via logical ordering," in *Proc. 19th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2014, pp. 343–356.

[36] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," in *Proc. 29th ACM SIGACT-SIGOPS Symp. Princ. Distrib. Comput.*, 2010, pp. 131–140.

[37] A. Natarajan and N. Mittal, "Fast concurrent lock-free binary search trees," in *Proc. 19th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2014, pp. 317–328.

[38] N. Shafiei, "Non-blocking Patricia tries with replace operations," in *Proc. IEEE 33rd Int. Conf. Distrib. Comput. Syst.*, 2013, pp. 216–225.

[39] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, "A practical concurrent binary search tree," *ACM Sigplan Notices*, vol. 45, no. 5, pp. 257–268, 2010.

[40] P. Fatourou, E. Papavasileiou, and E. Ruppert, "Persistent non-blocking binary search trees supporting wait-free range queries," in *The 31st ACM Symp. Parallelism Algorithms Architecture*, 2019, pp. 275–286.

[41] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, "Concurrent tries with efficient non-blocking snapshots," in *Proc. 17th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2012, pp. 151–160.

[42] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proc. Int. Symp. Distrib. Comput.*, 2001, pp. 300–314.

[43] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," *ACM SIGARCH Comput. Architecture News*, vol. 43, no. 1, pp. 631–644, 2015.

[44] R. Nikolaev and B. Ravindran, "Crystalline: Fast and memory efficient wait-free reclamation," 2021, *arXiv:2108.02763*.

[45] T. Brown, "Techniques for constructing efficient lock-free data structures," 2017, *arXiv:1712.05406*.

[46] S. V. Howley and J. Jones, "A non-blocking internal binary search tree," in *Proc. 24th Annu. ACM Symp. Parallelism Algorithms Architectures*, 2012, pp. 161–171.

[47] M. He and M. Li, "Deletion without rebalancing in non-blocking binary search trees," in *Proc. 20th Int. Conf. Princ. Distrib. Syst.*, Madrid, Spain, Dec. 13–16, 2016, pp. 34:1–34:17.

[48] A. Srivastava and T. Brown, "Elimination (a, b)-trees with fast, durable updates," in *Proc. 27th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2022, pp. 416–430.

[49] F. Ellen, P. Fatourou, J. Helga, and E. Ruppert, "The amortized complexity of non-blocking binary search trees," in *Proc. 2014 ACM Symp. Princ. Distrib. Comput.*, 2014, pp. 332–340.

[50] T. Brown, A. Prokopec, and D. Alistarh, "Non-blocking interpolation search trees with doubly-logarithmic running time," in *Proc. 25th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2020, pp. 276–291.

[51] A. Ramachandran and N. Mittal, "CASTLE: Fast concurrent internal binary search tree using edge-based locking," *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 281–282, 2015.

[52] A. Ramachandran and N. Mittal, "A fast lock-free internal binary search tree," in *Proc. Int. Conf. Distrib. Comput. Netw.*, 2015, pp. 1–10.

[53] T. Brown, F. Ellen, and E. Ruppert, "A general technique for non-blocking trees," in *Proc. 19th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2014, pp. 329–342.

[54] A. Braginsky and E. Petrank, "A lock-free B tree," in *Proc. 24th Annu. ACM Symp. Parallelism Algorithms Architectures*, 2012, pp. 58–67.

[55] V. Aksenov, D. Alistarh, A. Drozdova, and A. Mohtashami, "The splay-list: A distribution-adaptive concurrent skip-list," *Distrib. Comput.*, vol. 36, pp. 395–418, 2023.

[56] M. Arbel-Raviv, T. Brown, and A. Morrison, "Getting to the root of concurrent binary search tree performance," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2018, pp. 295–306.

[57] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," in *Proc. Bsdcan Conf.*, 2006.