NBR: Neutralization Based Reclamation

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

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

Ali Mashtizadeh
University of Waterloo
Canada
mashti@uwaterloo.ca

Abstract

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

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

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

ACM Reference Format:
Ajay Singh, Trevor Brown, and Ali Mashtizadeh. 2021. NBR: Neutralization Based Reclamation. In *26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21), February 27-March 3, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3437801.3441625

1 Introduction

Fundamentally, *safe memory reclamation* (SMR) is about answering the question: When is it safe to free a record? Unlike garbage collection, which is automatic, SMR requires a program to invoke a *retire* operation on each record at some point after it becomes *garbage* (i.e., is *unlinked* from the data structure). The task of an SMR algorithm is to eventually *free* an unlinked record once no thread holds a pointer to it [8, 14, 36].

The challenge of SMR in concurrent data structures comes from `use-after-free` conflicts between threads, where one thread accesses a record that is concurrently freed by another. For example, consider a lazy-list where one thread is searching and another is deleting. The first thread obtains a reference to a record and stores it in a local variable. The other thread unlinks and frees. At this point the first thread's reference is no longer safe as the record it points to has been freed.

Researchers have developed a rich variety of SMR algorithms, with a diverse spectrum of desirable properties, idiosyncrasies and limitations. After experimenting with SMR algorithms and observing the state of art [2–4, 6–8, 14–17, 19, 20, 22, 28, 30, 33, 36, 38, 43, 47], we identified the following set of desirable properties. [P1] *Performance*: reclamation operations should ideally offer both low latency and high throughput. [P2] *Bounded Garbage*: The number of records that are unlinked but not yet reclaimed should be bounded, even if threads experience halting failures or long delays. [P3] *Usability*: Intrusive changes to code, data, and the build environment, should be minimized. [P4] *Consistency*: Performance should not be drastically affected by changes in the workload (e.g., when shifting between read-intensive and update-intensive workloads). Additionally, there should be minimal performance degradation when the system is *oversubscribed* (with more threads than cores). [P5] *Applicability*: The algorithm should be usable with as many data structures as possible.

To set the stage for our contribution, we must first discuss other approaches. We broadly categorize existing work into: hazard pointer-based reclamation (HPBR), quiescent state-based reclamation (QSBR), epoch-based reclamation (EBR), reference counting based reclamation (RCBR), and hybrid algorithms that combine the aforementioned approaches [30]. In general, QSBR and EBR are fast but do not bound garbage, HPBR has bounded garbage but is not fast, and RCBR is neither fast nor does it bound garbage (in case retired nodes have pointer cycles [19]). Hybrid approaches have generally focused on achieving P1 and P2 simultaneously, usually by combining EBR (for its speed) with some variant of HPBR (to bound garbage), with varying levels of success.

The hybrid algorithm that most closely resembles our approach is DEBRA+ [14], a variant of EBR (with a restricted form of HPBR) that is designed for lock-free data structures. DEBRA+ is fast, and it achieves bounded garbage via a *neutralizing* mechanism based on POSIX signals and data structure specific recovery code. A thread whose reclamation is delayed by a slow thread will send a *neutralizing signal* to the slow thread. Upon receipt of a neutralizing signal, a thread executes its recovery code and then restarts its data structure operation, allowing reclamation to continue, ultimately guaranteeing a bound on the number of unreclaimed records. However, this bound on garbage comes at the cost of both usability and applicability, as users need to write data structure specific recovery code that is not always straightforward, or even possible. Moreover, it is not clear how DEBRA+ could be used for lock-based data structures, since neutralizing a thread that holds a lock could cause deadlock.

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

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

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

The rest of the paper is structured as follows. Related work is surveyed in Section 2. In Section 3, we introduce the model. Section 4 describes our basic algorithm *NBR*, and characterizes its applicability. We describe an optimized version *NBR+* in Section 5, followed by a brief discussion on *NBR*'s correctness in Section 6. Finally, experiments appear in Section 7, followed by conclusions in Section 8.

2 Related Work

Although detailed surveys of *safe memory reclamation* already exist in earlier works [14, 30], we would like to study existing techniques specifically through the lens of the desirable properties defined above.

RCBR involves explicitly counting the number of incoming pointers to a record, and typically storing this count alongside the record. The inclusion of this metadata in records complicates any advanced pointer arithmetic or implicit pointers, and can require changes to record layouts (or the use of a custom allocator) as well as size. RCBR typically requires a programmer to invoke a *deref* operation to dereference a pointer (and sometimes to explicitly invoke operations for read, write and CAS) [7, 19, 33, 38], adding significant overhead and programmer effort [opposing P1, P3]. Programmer intervention is also needed to identify and break pointer *cycles* in garbage records.

HPBR incurs significant overhead every time a new record is encountered, as a thread must first *announce* a hazard pointer (HP) to it in a shared location, then issue a memory fence (or use an atomic exchange instruction to announce the hazard pointer) and then check whether the record has already been unlinked [20, 33, 36] [opposing P1, P3]. If the record has been unlinked, the data structure operation trying to access it must be *restarted* (a data structure specific action). Correctly dealing with such failure cases can require extensive code changes. This may also require the programmer to *reprove* the data structure's progress guarantees [14]. Additionally, it is not clear how HPs could be used with data structures that allow threads to *traverse pointers in unlinked records* [14], and there are **many** examples of such data structures, e.g., [1, 9, 11, 21, 24, 26, 32, 37, 40, 44] [opposing P5]. (In such data structures, a search can potentially pass through many unlinked records, and yet end up back in the data structure, at the appropriate location.)

The latter limitation was addressed by *Beware and Cleanup*—a hybrid of RCBR and HPBR [28]. However, this algorithm requires a programmer to write a data structure specific *cleanup* procedure that changes all pointers in an unlinked record to point to *current* records *in the data structure*. This cleanup code ensures bounded garbage for data structures that allow traversing unlinked records, but the algorithm has higher overhead than either RCBR or HPBR and requires significant programmer effort [opposing P1, P3].

QSBR and EBR [27, 30, 35] both use the observation that, in many data structures, threads do *not* carry pointers obtained in one data structure operation forward for use in a subsequent operation. QSBR and EBR each have a simple interface in which the programmer need only invoke a specific operation at the start and end of a data structure operation. Unlike the approaches above, QSBR and EBR avoid all per-record and per-access overheads. A thread can reclaim its garbage records whenever it detects that all other threads have started a new data structure operation (and hence *forgotten* all pointers to said garbage records). However, in the event that a thread halts or is delayed, the amount of unreclaimed garbage can grow unboundedly [opposing P2].

DEBRA+ (described above) was introduced by Brown in 2015 [14]. In the same paper, an algorithm called DEBRA was proposed which, to the best of our knowledge, is the fastest EBR algorithm. DEBRA does not bound the number of unreclaimed records (garbage), but was shown to be faster than DEBRA+. Note that our experiments show NBR+ often *outperforms* DEBRA.

Since DEBRA, numerous hybrid algorithms offering bounded garbage have appeared, for example, Hyaline (HY) [38], Hazard Eras (HE) [43], Interval Based Reclamation (IBR) [47], and Wait Free Eras (WFE)[39]. All of these algorithms use per-record metadata to encode the times at which a record is allocated and unlinked, and the code instrumentation needed is similar to HPs [opposing P3]. HPs require per-record reserve and unreserve calls and fallback code (to restart the operation) in case the reservation fails (because the record is, or might be, unlinked). It is unclear how HE, IBR and WFE could be used with data structures that allow traversing unlinked records [opposing P5]. As we will see in our experiments, these algorithms also incur non-trivial overhead [opposing P1].

Various other algorithms utilize operating system features such as forced context switches [6], POSIX signals [3, 4], and hardware transactional memory [2, 22]. QSense [6] is a hybrid algorithm that uses QSBR as a *fast* code path, and HPs with forced context switches as a *slow* code path to bound garbage. However, in the event of long thread delays, reclamation can only proceed on the *slow* path, which is as slow as HPBR. QSense has been shown to be slower than EBR [6] [opposing P1]. None of [2, 4, 6] can be used with data structures that allow threads to traverse unlinked records [opposing P5]. Forkscan (FS) [3] was succeeded by ThreadScan (TS) [4], which addressed this issue, but FS assumes the programmer will not use advanced pointer arithmetic techniques (or implicit pointers) [opposing P3]. Additionally, FS has been shown to be slower than HPs in several workloads [3] [opposing P1].

Optimistic Access (OA) and Automatic Optimistic Access (AOA) [16, 17] proposed a particularly interesting approach: they optimistically allow threads to *accesses reclaimed nodes*, and verify *after the fact* that the access was safe. This requires an assumption that either (a) memory will not be `freed` to the OS, or (B) any resulting trap/exception (such as a segmentation fault) will be caught and handled [opposing P3]. Additionally, OA requires the programmer to transform data structures into a *normalized form* [46] (which is similar to, but not the same as, the form we assume in this paper), and instrument every read/write/CAS [opposing P3]. AOA automates this transformation with compiler support (for data structures that *have* a normalized form). Unfortunately, it doesn't appear that AOA has been ported to modern compilers. The need for a normalized form was eliminated in Free Access (FA) [15], which used a compiler extension to perform automatic instrumentation of writes and blocks of consecutive independent reads. FA is a general technique that has been shown to have comparable performance to HPBR [15] [opposing P1]. In contrast, our work targets applications that can benefit from the high performance handcrafted SMR.

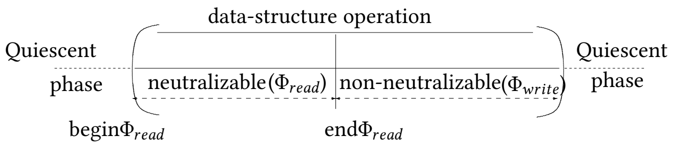


Figure 1. Visualizing the form of a *data-structure* operation for which *NBR* can be used. The thread performing this operation can be neutralized in the read-phase (Φ_{read}). However, it cannot be neutralized in the write-phase (Φ_{write}). endΦ_{read} marks the beginning of the operation's Φ_{write}.

3 Model

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

4 NBR

4.1 Assumptions on the data structure

NBR requires a data structure's operations to have (or be restructured into) the following form. This form is needed for the neutralization mechanism wherein an operation that has not yet written to shared memory is forced to restart.

The required form is described as a sequence of *phases* (illustrated in Figure 1), with specific rules in each phase. Along the way, we discuss potential pitfalls for readers unfamiliar with the use of `sigsetjmp` and `siglongjmp`.

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

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

To understand the latter restriction, suppose an operation allocates a node using `malloc` during its Φ_{read}, and before it uses the node, the thread performing the operation is neutralized. This would cause a memory leak.

Additionally, writes to thread local data structures are not recommended. To see why, suppose a thread maintains a thread local doubly-linked list, and also updates this list as part of the Φ_{read} of some operation on the shared data structure. If the thread is neutralized in middle of its update to this local list, it might corrupt the structure of the list. [a]

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

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

Finally, threads not executing a data structure operation are said to be in a *quiescent phase* (essentially the same as phase 0).

4.2 Overview of NBR

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

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

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

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

a. Blocking the reclamation of T, and
b. The possibility that T' continues executing and one of the records it is about to access is concurrently `free`d by T.

The solution lies in the *reservation* phase (Phase 2) of T'. During the reservation phase of T', just before it begins its Φ_{write}, T' *reserves* all of the shared records it will access in its Φ_{write} by *announcing* pointers to them in a shared array. These reservations serve a similar purpose to hazard pointers, but are quite different from HP in terms of performance and safety guarantees. This is discussed further in Section 5.3. By the time T' is in its Φ_{write} (so it will ignore any neutralization signals), its reservations are visible to all threads, and T can refer to these reservations to avoid reclaiming any of those records.

[a]Note, in some cases it is okay to write to a thread local variable or data structure. For example, if a thread wants to count how many times it gets neutralized while executing a Φ_{read} then neutralization of this thread is not a problem. More generally, any idempotent updates to thread local data structures should remain correct, even if the Φ_{read} is restarted. The programmer must simply proceed with caution.

[b]Technically `sigsetjmp` saves the current stack frame. Stack variables defined deeper on the stack will not necessarily be saved or restored.

Algorithm 1 *NBR.* Assumes, max number of reservations are less than the limboBag size.

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

    shared variable:
5:  atomic<record*> reservations[N][R]    ▷ N:#threads, R:max
    reserved records. |R| << |S|.

6:  procedure BEGIN$\Phi_{read}$( )
7:      reservations[tid].clear();
8:      CAS(&restartable, 0, 1);
9:  end procedure
10: procedure END$\Phi_{read}$($\{rec_1, rec_2 \cdots rec_R\}$)
11:     reservations[tid] = $\{rec_1, rec_2 \cdots rec_R\}$;
12:     CAS(&restartable, 1, 0);
13: end procedure
14: procedure RETIRE(rec)
15:     if isLimboBagTooLarge() then
16:         signalAll( );
17:         reclaimFreeable(tail);
18:     end if
19:     limboBag[tid].append(rec);
20: end procedure
21: procedure RECLAIMFREEABLE(tail)
22:     $A$ = collectReservations( );
23:     $R$ = limboBag[tid].remove(A, tail);
24:     free({$R$});
25: end procedure
```

In short, operations in the Φ_{read} discard their pointers and restart, and operations in the Φ_{write} must have reserved them. This empowers the *reclaimers* to assume that *reader*s lose all of their pointers in response to neutralizing, and the *writer*s lose all pointers that are not reserved. As a result, once a thread sends a neutralizing signal to all other threads, it can scan all reservations, and free any records in its buffer (limbo bag) that are not reserved.

4.3 Implementation of NBR

Algorithm 1 shows the pseudocode for *NBR*. Each thread collects unlinked records in its *limboBag* (line 2), and maintains a local *restartable* variable that indicates whether the thread should jump back to the start of its Φ_{read} in the event that it receives a neutralization signal (line 3). We say the thread is *restartable* if *restartable* is *true* (1), and *non-restartable* otherwise. Additionally, each thread, before entering the Φ_{write}, reserves all records it might access in a single-writer multi-reader (SWMR) *reservation* array, (line 5). We assume the maximum number R of reserved records is strictly less than maximum size S of a limbo bag.

A thread in Φ_{read} clears its reservations (if any), and then changes *restartable* to *true* using a CAS (Line 8). This CAS might initially seem strange, since it is performed on a single-writer variable and cannot fail. The CAS prevents instruction reordering on x86-64 architectures (additional fences may be needed for more relaxed memory models). More specifically, the goal of CAS at line 8 is to ensure that a thread T becomes restartable *before* any subsequent reads of shared records. If this CAS were simply an atomic write (rather than a read-modify-write instruction), it would be possible for T's reads of shared records to be reordered before this write. In other words some reads of shared records in Φ_{read} may appear to occur in *preamble* (or previous Φ_{write}) due to instruction reordering. This could end up breaking the rule that says access to shared records is not permitted in *preamble* (phase 0) as discussed in Section 4.1. As a result, the thread, which is not yet restartable, might ignore a neutralization signal and access a `freed` record.

Just before a thread T enters a Φ_{write}, it announces a set of reservations, and then changes *restartable* to *false* using CAS (Line 12). This CAS is used to broadcast the reservations to other threads. More specifically, a CAS by thread T at line 12 implies a memory fence, which ensures that all of the reservations (announced at the previous line 11) are visible to other threads *before* T changes *restartable* to false. If this CAS were a simple write, it would be possible for a *reclaimer* to miss some reservations of T, and erroneously `free` those records[c].

In other words, the following incorrect execution may occur on x86/64 if a write is used instead of CAS: a thread T reserves record *rec* and writes 0 to restartable. Suppose the reservations of thread T remain in the processor's store buffer, and are not visible to other threads yet. Then, another thread T' sends a neutralizing signal to T, scans the reservations and does not see *rec*, and consequently frees *rec*. Upon receiving the signal, T' will *not* restart since it has already written 0 to restartable.[d] Instead, it continues executing, and dereferences *rec* (accessing a freed record).

The `retire` operation (line 14) begins by checking whether the size of the limbo bag is above a predetermined threshold (32k in our experiments), at line 15. If so, it sends a neutralizing signal to all threads using `signalAll` (line 16), and then

[c] Instead of using CAS, on modern x86/64 machines we can use the more efficient `xchg` instruction. Please refer to section 11.5.1 of https://www.amd.com/system/files/TechDocs/47414_15h_sw_opt_guide.pdf for further details. In GCC, this means invoking func__sync_lock_test_and_set(), which is implemented using `xchg`.

[d] A detailed explanation of the behaviour of store buffers and serializing instructions on modern processor architectures is out of scope. But, briefly, if T and T' are executing on different processors, then T will not see the effects of any pending writes in the store buffer of T', but T' *will* see the effects its own pending writes in order to maintain sequential consistency.

proceeds to reclaim all *safe* (i.e., *unreserved*) records (line 17). Otherwise, it simply adds *rec* to *limboBag*.

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

After discussing the implementation of *NBR* we can now elaborate on how *reader*s, *writer*s and *reclaimer*s collaborate to achieve safe memory reclamation.

4.3.1 Reader-reclaimer handshake. Each thread T' at the time of BEGINΦ_{read} saves its execution state (program counter and stack frame) using *sigsetjmp* so that when it becomes restartable it can jump back to this state upon receiving a neutralizing signal. When a *reclaimer T* sends a neutralization signal to thread T', the operating system causes the control flow of T' to be interrupted, so that T' will immediately execute a *signal handler* if T' is currently running. (Otherwise, if T' is not currently running, the next time it is scheduled to run it will execute the signal handler before any other steps.) The signal handler determines whether T' is restartable by reading the local *restartable* variable. If the thread is restartable, then the signal handler will invoke `siglongjmp` and jump back to the start of the Φ_{read} (so it is as if T' never started the Φ_{read}).

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

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

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

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

4.4 Revisiting the Φ_{write} restriction

In this section, we will trace an incorrect execution that could occur if a thread accesses any record that is not reserved *before* entering the Φ_{write}.

Suppose a thread T is in a Φ_{write}, and sleeps just before it accesses a shared record *rec*, which it has *not* reserved. Then, another thread T' sends a neutralization signal to T using `signalAll`. Next, T' scans the *reservations* array of the thread T. T did not reserve *rec* so T' will not find *rec* in T's reserved records (which violates the writers handshake, Section 4.3.2). Therefore, T' will assume that *rec* can be freed safely, and will do so. Finally, T wakes up and proceeds with its *unsafe* access of rec.

5 NBR+

Next, we explain a performance issue with *NBR* which motivated us to design an improved version called *NBR+*.

Performance bottleneck in NBR. Signals on linux trigger page-fault routines and a switch from user to kernel mode incurs significant overhead. Therefore, it is desirable to send as few signals as possible (while maintaining high reclamation throughput). However, every time a thread reclaims records from its *limboBag*, NBR requires the thread to send signals to *all* other threads. This induces a *relaxed grace period* (RGP): A time interval [t, t'] during which each thread is neutralized due to a reclamation event triggered by some reclaimer thread. In NBR, every thread induces a RGP every time it tries to reclaim its *limboBag*. As a result, in order for all n threads to reclaim their *limboBag*s, $n(n-1)$ signals must be sent. The need to send $O(n^2)$ signals to allow all n threads to reclaim memory can severely limit performance. Naturally, we would like to improve this.

Suppose, in NBR, at some time t, a thread sends $n-1$ signals to other threads so it can reclaim its *limboBag*. This causes all of the other $n-1$ threads to discard any unreserved references to shared records. Meaning, at time t, the (unreserved) records in the limbo bags of **all threads** are safe to free. Therefore, if somehow we could propagate this information that a RGP has occurred due to some thread T, then all other threads could piggyback on T to partially or completely reclaim their own limbo bags without sending signals of their own. In other words, in the best case, all n participating threads could reclaim memory after detecting exactly one RGP, induced by sending a total of $n-1$ signals.

Overview of NBR+ The key insight in *NBR+* is that when a reclaimer sends neutralization signals to *all* threads, *all* threads discard their pointers to unreserved records, and thus *all* threads can potentially reclaim some records in their *limboBag*s. This suggests a design wherein each thread (1) passively detects a RGP by observing signals sent by another thread, and (2) determines which records in its *limboBag* were unlinked *before* the RGP (i.e., are safe to reclaim).

5.1 Implementation of NBR+

We explain the design of *NBR+* by building our exposition around three main design challenges.

(C1) When should a thread start tracking other threads' signals to detect a *RGP*?
(C2) How can a thread *recognize* that a *RGP* has occurred?
(C3) Once a thread recognises that a *RGP* has occurred how should it determine which records in its *limboBag* are safe to reclaim?

As a solution to **(C1)**, each thread in *NBR+*, in addition to watching the *limboBag* size to determine when it becomes *too large* (triggering neutralization), also determines when the *limboBag* size crosses a predetermined threshold called the *LoWatermark* (e.g., one half full or one quarter full). If a thread's *limboBag* is full, we say that thread is at the *HiWatermark*. If a thread's *limboBag* keeps growing without reclamation it will first cross the *LoWatermark* and then hit the *HiWatermark*. As shown in Algorithm 2, a thread determines whether it has passed the *HiWatermark* or *LoWatermark* using procedures `isAtHiWm` (line 6) and `isAtLoWm` (line 12). Once a thread has passed the *LoWatermark*, it begins recording and analyzing information about signals sent by other threads to detect RGPs.

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

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

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

Algorithm 2 *NBR+*: Only variables that differ from *NBR* are shown here. *NBR+* includes all variables and procedures of Algorithm 1. The `retire` operation is different in *NBR+*. Some helper procedures required by *NBR+* are self explanatory.

```
    thread local variable:
 1: int scanTS[N];                ▷ N = #threads. P = set of processes
 2: bool firstLoWmEntryFlag = true;
 3: record* bookmarkTail;

    shared variable:
 4: atomic<int> announceTS[N];

 5: procedure RETIRE(rec)
 6:     if isAtHiWm() then
 7:         FAA(&announceTS[tid],1);              ▷ RGP begin
 8:         signalAll()
 9:         FAA(&announceTS[tid],1);              ▷ RGP end
10:         reclaimFreeable(tail);
11:         cleanUp();
12:     else if isAtLoWm() then
13:         if firstLoWmEntryFlag then
14:             bookmarkedTail = tail;
15:             scanTS[tid] = scanAnnounceTS()
16:         end if
17:         for each otid ∈ P do   ▷ otid: other thread's id in P.
18:             if announceTS[otid]≥scanTS[tid][otid]+2 then
19:                 reclaimFreeable(bookmarkTail);
20:                 cleanUp();
21:                 break;
22:             end if
23:         end for
24:     end if
25:     limboBag[tid].append(rec);
26: end procedure
27: procedure CLEANUP
28:     firstLoWmEntryFlag = 1;
29: end procedure
```

[e]Note, scanning *announceTS[]* would incur overhead in terms of cache misses. This cost is amortized over multiple `retire` operations by scanning *announceTS[]* after a fixed number of calls to `retire` have been made.

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

`cleanUp()` (line 27) method is used to set *firstLoWmEntryFlag* after a thread reclaims either at *LoWatermark* (line 20) or at *HiWatermark* (line 11) to prepare it for subsequent reclamation.

A thread that has not reached the *LoWatermark* or the *HiWatermark* simply continues to append any retired records to its *limboBag* (line 25).

At first it may appear that a thread T can reclaim its *limboBag* as soon as it receives a neutralizing signal from a *reclaimer* thread T'. However, the receipt of a single signal is not enough for T to safely reclaim memory. To safely reclaim the set R of records in its *limboBag* up to its *bookmarkTail*, T needs to know that *all* threads have been neutralized *since* T *retired the records in* R. Otherwise, some other thread may still have a pointer to a record in R.

Let us discuss an example of what can go wrong if a thread reclaims its *limboBag* after it receives a single signal. Consider a system with three threads $T1$, $T2$ and $T3$. Suppose $T1$ is at its *HiWatermark*, $T2$ is at its *LoWatermark* and $T3$ holds a private reference to a record *rec* that is in $T2$'s *limboBag*. $T1$, being at its *HiWatermark*, begins neutralizing all threads one by one. First, it sends neutralizing signal to $T2$ (starting a *RGP*). $T2$, upon receiving the signal, reclaims its *limboBag* including *rec*. Note, that $T1$ hasn't neutralized $T3$ yet, meaning a *RGP* has not yet occurred. Now, if $T3$ accesses *rec*, a *use-after-free* error would occur. To prevent this, $T2$ should not reclaim the contents of its *limboBag* unless $T1$ *completes* the *RGP* by neutralizing $T3$ (preventing $T3$ from doing this unsafe access). The crucial point is that $T2$ must detect the *start* and *end* of a RGP to know that it can safely reclaim records in its *limboBag*.

5.2 Applicability

NBR (+)[f] naturally applies to many concurrent data structures that have synchronization-free searches followed by update(s) because in such data structures searches and updates correspond to the Φ_{read} and the Φ_{write} of *NBR*, respectively (as shown in Figure 1). Thus, to apply *NBR* one just needs to invoke BEGINΦ_{read} before the start of the search and ENDΦ_{read} before starting the update(s). For example, in the lazy-list of Heller et al. [32], the Φ_{read} of an operation would begin with the start of the search for target records and the Φ_{write} would consist of the locking and validation of target records followed by any modifications to them.

Certain other lock-free data structures exhibit a pattern where searches, in an operation, perform *auxiliary update(s)* followed by intended update(s). Such an operation has a *sequence* of *read-write phases*. For example, in Harris's lock-free list [29], while searching the list towards a target location, a thread may modify the list by unlinking any *marked* (logically deleted) records it encounters. Then, once it arrives at the target location, it performs the operation's intended modification.

Since *NBR* is designed for a single Φ_{read} and Φ_{write}, applying it carelessly to such a data structure could break the requirement that, after entering a Φ_{write}, no new records are discovered. (This would be unsafe, because it breaks the *writer*s handshake.) For instance, in such a data structure, if we enter a Φ_{write} to perform an *auxiliary update*, *NBR* would be stuck in the Φ_{write}, unable to obtain new pointers (that have not yet been reserved) to continue its traversal.

That said, *NBR can* be applied in some data structures that would require *multiple read/write* phases, provided that each consecutive pair of read and write phases obey the requirements set out in Section 4.1.

Example: Harris list. Algorithm 3 shows how *NBR* can be used with the Harris list [29], *despite* the fact that this list has auxiliary updates. We hope the reader can follow our exposition on the Harris list, and infer how *NBR* could be applied to more sophisticated data structures with similar design patterns (such as Brown's ABTree [10], which appears in our experiments).

To understand how *NBR* behaves when applied to the Harris list, suppose the initial list configuration is L: $1_f \implies 2_f \implies 3_t \implies 4_f \implies 6_f \implies 10_f$, where each node is represented as key_{marked} (where *marked* is $[t]$rue or $[f]$alse). Now, suppose a thread T performs *Ins:insert(9)*, starting with an invocation of `search()`. This invocation of `search()` starts a Φ_{read} (line 18) and begins traversing L. Starting from $\langle pred, curr\rangle = \langle 1_f, 2_f\rangle$ the thread observes $\langle pred, curr\rangle = \langle 2_f, 3_t\rangle$, where $curr = 3_t$ is marked. To remove marked node 3_t (an auxiliary *helping* update), T enters a Φ_{write} (line 31) and changes the next pointer of 2_f to 4_f, yielding the list configuration: $1_f \implies 2_f \implies 4_f \implies 6_f \implies 10_f$. Moving forward, T's `search()` will enter a *second* Φ_{read} (line 18), and traverse the list again, *starting from the root*. As T now obtain pointers to *new nodes* (which would be impossible with only a single Φ_{read} and Φ_{write}), we must argue that it doesn't access any `freed` nodes. However, this is straightforward, since it is again traversing *from the root* discarding any references from previous *read-write* phases. (From the perspective of SMR, it is as if T has simply started a new data structure operation.)

[f]We will simply write *NBR* in this section with the understanding that the entire discussion applies identically to *NBR+*.

Source	**Data structure**	**Sync. type**	**NBR+**	**EBR**	**DEBRA+**	**HP/TS/IBR/HE/WFE/HY/QSense**
LL05[32]	linked list	opt. locks	Yes	Yes	No	No (similar to [14])
HL01[29]	linked list	lock-free	Yes	Yes	*	Yes
HM04[36]	linked list	lock-free	No	Yes	*	Yes
DVY14a[21]	partially external BST	locks	**	Yes	No	No [14]
EFRB10[24]	external BST	lock-free	Yes	Yes	*	No [14]
NM14[37]	external BST	lock-free	Yes	Yes	*	No [14]
EFRB14[23]	external BST	lock-free	No	Yes	*	No [14]
DGT15[18]	external BST	ticket locks	Yes	Yes	No	No (no marks, cannot validate HP)
HJ12[34]	internal BST	lock-free	Yes	Yes	*	No (similar to [14])
RM15[42]	internal BST	lock-free	No	Yes	No	No (similar to [14])
BCCO10[9]	partially external AVL	opt. locks	No	Yes	No	Yes
DVY14b[21]	partially external AVL	locks	No	Yes	No	No [14]
HL17[31]	external relaxed AVL tree	lock-free	Yes	Yes	Yes	No (similar to [14])
B17b[10]	external AVL	lock-free	Yes	Yes	Yes	No [14]
S13[44]	patricia trie	lock-free	Yes	Yes	*	No [14]
BER14[12]	external chromatic tree	lock-free	Yes	Yes	Yes	No [14]
B17a[10]	external (a,b)-tree	lock-free	Yes	Yes	Yes	No [14]
BPA20[13]	external interpolation tree	lock-free	No	Yes	No	No (similar to [14])

Table 1. Applicability of SMR algorithms. Due to space constraints a detailed explanation of the contents of this table is relegated to [45]. *It appears likely that DEBRA+ is compatible, but one must design non-trivial data structure specific recovery code. **This is likely possible if code is restructured to reserve all relevant nodes before acquiring any locks.

Now, suppose T is *neutralized* by a concurrent *reclaimer* while it is in this second Φ_{read}. Upon receipt of a neutralization signal, T will jump back to the beginning of its *second* Φ_{read}, and restart its search, once again, from the *root*. Note that neutralizing does not affect the lock-free progress guarantee, since a thread sends neutralization signals only after performing many successful deletion operations. Suppose T eventually performs a Φ_{read} that reaches the nodes $\langle pred, curr\rangle = \langle 6_f, 10_f\rangle$ where it should perform its modification. T will then enter a final Φ_{write} and insert 9_f after returning (line 37) from the `search()`, yielding L: $1_f \Longrightarrow 2_f \Longrightarrow 4_f \Longrightarrow 6_f \Longrightarrow 9_f \Longrightarrow 10_f$.

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

Compatible data structures. There are numerous concurrent data structures in the literature with multiple *read-write phases* that *do* restart from the *root* after any *auxiliary updates*, and hence are natural candidates for pairing with *NBR*. For example, Harris' list [29], Brown's lock-free ABTree, chromatic tree and AVL tree (B17) [10], the lock-free binary search tree of Natarajan et al. [37], and many more [24, 31, 34, 44]. Among these, we used the Harris list and ABTree in our experiments.

Semi-compatible data structures. The need to restart from the root at the start of each Φ_{read} suggests that *NBR* cannot be used with the data structures like the Harris-Michael list [36], and some search trees [9, 13, 21, 23, 41], wherein the searches (Φ_{read}) after each *auxiliary update* (Φ_{write}) do not start from the *root*. However, we could potentially use *NBR* with such data structures if we were to modify the operations so they restart *from the root* after any auxiliary updates. Depending on the data structure, this might break the progress guarantee (for example changing a wait-free algorithm into a lock-free one, or necessitating a new amortized complexity analysis [23]), or it might simply add overhead.

For some data structures, the overhead of restarting from the *root* may be quite low in practice, and forcing operations to restart from the root may be a reasonable solution. (The cost of restarting from the root is studied in our experiments.) For example, in Harris-Michael list, in high contention scenarios where k threads all contend on an auxiliary CAS to unlink the same marked node, all threads except for the one that succeeds this CAS would already restart from the root [36] anyway! If we modify this list so threads always restart from the root, in this high contention scenario, k

Algorithm 3 Integration of *NBR* with Harris list[29] with multiple read/write phases $(\Phi_{read}\Phi_{write})^+$.

```
bool insert(key) {
  Node *right_node, *left_node;
  do{
      right_node = search (key, &left_node);
      if((right_node!=tail) && (right_node.key==key))
        return false;
      Node *new_node = new Node(key);
      new_node.next = right_node;
      if (CAS(&(left_node.next), right_node, new_node))
        return true;
  }while (true)
}

Node* search(key, Node** left_node) {
  Node *left_node_next, *right_node;
  search_again:
  do {
      beginΦread();
      Node *t = head;
      Node *t_next = head.next;
      do{
          if(!is_marked_reference(t_next)){
            (*left_node) = t;
            left_node_next = t_next;
          }
          t = get_unmarked_reference(t_next);
          if (t == tail) break;
          t_next = t.next;
      }while(is_marked_reference(t_next) or (t.key<
           search_key));
      right_node = t;
      endΦread(left_node, right_node);

      if (left_node_next == right_node)
        if ((right_node != tail) && is_marked_reference(
              right_node.next))
          goto search_again;
        else
          return right_node;
      if (CAS(&(left_node.next), left_node_next,
            right_node))
        if ((right_node != tail) && is_marked_reference(
              right_node.next))
          goto search_again;
        else
          return right_node;
  } while(true);
}
```

threads must restart instead of $k-1$. Incidentally, by doing this, we essentially obtain the Harris list [29], in which all threads contending on the auxiliary CAS already restart from the *root*. (In low contention scenarios the way we restart should not affect performance significantly.)

Furthermore, in search trees, assuming a uniform distribution of accesses, threads tend to spread out in the tree, so average contention is quite low. This suggests that, when a thread encounters contention, the performance difference between restarting from the root and continuing a traversal from an ancestor will be small in many workloads.

Incompatible data structures. We are aware of a few data structures that are incompatible with (or would require extensive code changes to work with) *NBR*. Two concurrent implementations of a relaxed-balance AVL tree appear in [9, 21]. In each of these implementations, after a key is inserted, rotations must be performed to rebalance the tree. These rotations are performed starting at the bottom of the tree, possibly continuing all the way to the root (traversing upwards using parent pointers). In the process of performing these rotations, a thread may encounter many new nodes that were *not* traversed as part of the initial search in the insert operation. In order to use *NBR* with these algorithms, one would need to rewrite the implementations to perform a new search from the root *for each rotation*.[g]

A recent lock-free interpolation search tree [13] also appears to be incompatible. For example, in this algorithm, entire subtrees are periodically rebuilt to maintain balance, and during this process, threads *mark* all nodes in the subtree, one by one, alternating between steps that *mark* a node and *discover* a new node (without restarting from the root in between). It is not clear how one could transform this algorithm into the form required by *NBR*. (Note, however, that neither DEBRA+ nor HPs can be used with this data structure, either. We are not aware of *any* SMR algorithm with bounded garbage that is compatible with this tree.)

Comparing with other SMR algorithms *NBR* can be used with many data structures that other SMR algorithms like DEBRA+ and HP (and variants of HPs, including HE, IBR, WFE, ThreadScan, HY and QSense) are incompatible with [18, 24, 32, 34, 37]. There are also some data structures that are compatible with other SMR algorithms but not *NBR* [9, 36]. See Table 1 for an overview. Due to lack of space, a detailed analysis of the table's contents, and an exposition of how *NBR* can be applied to these data structures, is relegated to the full version of this paper [45].

5.3 Ease of use

Figure 2 compares the difficulty of using *HP*, *NBR* and *DEBRA* in the insert operation of the lazy list of Heller et al. [32]. As Figure 2c demonstrates, *HP* is cumbersome to use because it requires a programmer to protect every record by *announcing* hazard pointers, using a store/load fence or *xchg* instruction to ensure that each announcement is visible in a timely manner by other threads, validating that the announced record is still safe before dereferencing it, and restarting if validation fails. Programmers also need to unprotect records that they will no longer dereference, further increasing the need for intrusive code changes.

On the contrary, applying *NBR* to a data structure operation is, intuitively, similar to performing two-phased locking, in the sense that the primary difficulty revolves around identifying where the Φ_{write} should begin, and which records it will access. The programmer just needs to invoke `begin`Φ_{read} before the operation accesses its first shared record, in this example, at the start of the traversal for target records. Then s/he must invoke `end`Φ_{read} before modifying any shared records. In this example, the Φ_{write} begins just before the

[g]In addition to their AVL tree, Drachsler et al. [21] also present an *unbalanced* BST, which can more easily be modified to work with *NBR*.

```
|| void OP_DEBRA()
|| {
|| recl_start_op
|| RETRY:
||     pred=head; curr=pred.next;
||     while (key ≤ curr.key) {
||         pred=curr;
||         curr=cur.next;
||     }
||
||     if (key == curr.key) {
||         return false;
||     }
||
||     lock(pred);
||     lock(curr);
||     if (!validate()) {
||         unlock(pred); unlock(curr);
||         goto RETRY;
||     }
||
||     do update
||     unlock(pred); unlock(curr);
|| recl_end_op
|| }
```

(a) DEBRA

```
|| void OP_NBR+()
|| {
|| RETRY:
|| beginΦ_read
||     pred = head; curr = pred.next;
||     while(key ≤ curr.key) {
||         pred=curr;
||         curr=cur.next;
||     }
||
|| endΦ_read
||     if (key == curr.key) {
||         return false;
||     }
||
||     lock(pred); lock(curr);
||     if (!validate()) {
||         unlock(pred); unlock(curr);
||         goto RETRY;
||     }
||
||     do update
||     unlock(pred); unlock(curr);
|| }
```

(b) *NBR+*

```
|| void OP_HP()
|| {
|| RETRY:
||     pred = head; curr = pred.next;
||     protect(curr) RETRY on fail ;
||     while (key ≤ curr.key) {
||         unprotect(pred) ;
||         pred=curr;
||         curr=cur.next;
||         protect(curr) RETRY on fail ;
||     }
||     if (key == curr.key) {
||         unprotect(pred) ; unprotect(curr) ;
||         return false;
||     }
||     lock(pred); lock(curr);
||     if (!validate()) {
||         unprotect(pred) ; unprotect(curr) ;
||         unlock(pred);unlock(curr);
||         goto RETRY;
||     }
||     do update
||     unprotect(pred) ; unprotect(curr) ;
||     unlock(pred);unlock(curr);
|| }
```

(c) HP

Figure 2. Complexity of using *DEBRA*, *NBR* and *HP* on a lazy list. *DEBRA* << *NBR* << *HP*.

lock acquisition on pred. If there are no modifications to be performed in an operation, for example, in the `contains` operation of the lazy-list, then the programmer can simply invoke endΦ_{read} before returning from the operation.

DEBRA is simplest as it requires programmers to invoke just two functions corresponding to the start and the end of a data structure operation (Figure 2a).

In terms of programmer effort, *NBR* finds a middleground between DEBRA and HPs. Although *NBR* is slightly more involved than DEBRA, we believe that the benefits due to *NBR*'s bounded garbage property and better performance outweigh the extra effort of identifying which shared records will be modified by the Φ_{write} and where in the code to invoke endΦ_{read}.

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

As mentioned earlier, in *NBR*, before a thread enters a Φ_{write}, it must reserve all the records that will be accessed in the Φ_{write}. In some data structures it might not be possible to determine *precisely* which records *will* be accessed in the Φ_{write}. For example, in a tree, an operation may decide *during* the write phase whether to modify the left or right child pointer. To apply *NBR* in such a tree, one can simply reserve *both* pointers. (Nevertheless there may be some data structures where it is infeasible to reserve *all* of the records that might be accessed in a Φ_{write}.)

6 Correctness

We show that *NBR* and *NBR+* are both safe and have bounded garbage. Proofs can be found in [45].

Lemma 1 (NBR is Safe). *Reclaimer threads in* NBR *only reclaim records that are safe (i.e., that no other thread has access to).*

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

Lemma 3 (Both *NBR* and *NBR+* have bounded garbage). *The number of unlinked, unreclaimed records is bounded.*

Please note that *NBR* assumes that number of records that could be reserved per data structure operation are strictly less than than the limboBag size in order to be able to reclaim whenever the limboBag is full. Practically most of the data structures require only a small number of reservations per operation. For example in our experiments, we used lazylist [32] required maximum 2 reservations per operation and harris list [29], DGT [18], and (a,b) tree [10] needed to reserve maximum 3 records at a time.

7 Experimental Evaluation

Setup: We used a quad-socket Intel Xeon Platinum 8160 machine running at 2.1GHz with 192 hardware threads and 384 GiB memory having shared L3 cache (33.79 MiB) on Ubuntu 18.04 with GCC/G++ version 7.4.0.

All algorithms used in the experiments were implemented in the Setbench [13] benchmark compiled with -O3 optimization, and used *jemalloc* as the memory allocator [25].

We perform four kinds of experiments:

(E1): Evaluates throughput over different thread counts and workloads to understand *NBR* (+)'s performance and scalability [P1].
(E2): Evaluates peak memory usage of *NBR+* to show the advantage of bounded garbage [P2].
(E3): Evaluates the impact of contention on performance [P4].
(E4): Evaluates the impact of modifying a data structure to restart from root to make it work with *NBR+*.

For (E1) we picked the lazy-list [32] and DGT [18] as representative of lists and trees, to evaluate *NBR+* against QSBR, RCU, DEBRA, and the 2geibr variant of interval based reclamation (IBR) [47], hazard pointers (HP) and a leaky implementation (none).[h] For (E2) we compared peak memory usage of each of the aforementioned reclamation algorithms using DGT. For (E3) we compared *NBR* with DEBRA and the leaky implementation (none) using the ABTree data structure [10, chapter 8] and the Harris list [29] with very large data structure sizes, and *extremely small* data structure sizes. Finally, for (E4), we modified Harris-Michael list (HMList) such that every Φ_{read} restarts from root which basically yields Harris list (HList). Thus, we compare *NBR+* based on HList against DEBRA implemented with original HMlist (with no enforced restarts, *debra-norestarts*) and DEBRA implemented with modified list (enforced restarts, *debra-restarts*) refer Figure 4b.

Reported results are obtained by averaging data over 3 timed trials, each lasting 5 seconds, for each thread count in {24, 48, 72, 96, 120, 144, 168, 192, 216, 240, 252}, and each data structure. We used a key range size of 2 M and 20 K for trees and lists, respectively. Each execution starts by prefilling the data structure to half of the key range size, i.e., 1 M for trees and 10 K for lists.

We subject *NBR+* to exhaustive evaluation by running it under oversubscription (i.e., with more threads than logical processors) on three workload profiles to establish P4:

For each of (E1), (E2), and (E3) we subject *NBR+* to exhaustive evaluation by running it on three workload profiles, (1) *update-intensive* 50% inserts and 50% deletes, (2) *balanced* 25% inserts, 25% deletes and 50% searches, and (3) *search-intensive* 90% searches, 5% inserts and 5% deletes.

Discussion (E1) results for the DGT tree and lazy linked list appear in Figure 3. *NBR+* matches or outperforms its competitors for nearly all data points. In the tree, it surpasses the next best algorithm, DEBRA, by up to 38% and 12% (Figure 3a, update-intensive and balanced workloads, resp.) and is comparable to DEBRA in search-intensive workloads where it outperforms the next best algorithm by up to 10% (Figure 3a search-intensive workload).

In these graphs, DEBRA performs better than NBR+ for low thread counts, but NBR+ outperforms it after 96 threads in update intensive workloads (Figure 3a, leftmost plot), after 120 threads in the balanced workload (Figure 3a, center), and the two algorithms are comparable in the search-intensive workload (Figure 3a, rightmost plot). The slow performance of DEBRA at higher thread counts could be attributed to the infrequent advancement of epochs by slow threads, which leads to halting of regular reclamation of limbo bags. We call this the *delayed thread vulnerability*. As a result, the limbo bags of all threads keeps on growing until the slow thread finally catches up and announces the required epoch.

The delayed thread vulnerability leads to accumulation of large number of retired records waiting to be reclaimed at the announcement of the required epoch. Once the required epoch is announced by the slow thread, all threads reclaim their large limbo bags, which leads to a *reclamation burst* (many, many records being freed at once). This harms the overall throughput, as reclamation bursts can bottleneck the underlying allocator (jemalloc in our experiments) by increasing contention and triggering slow/fallback code paths. The probability of threads getting delayed increases as more threads get involved in high intersocket and update-intensive computations.

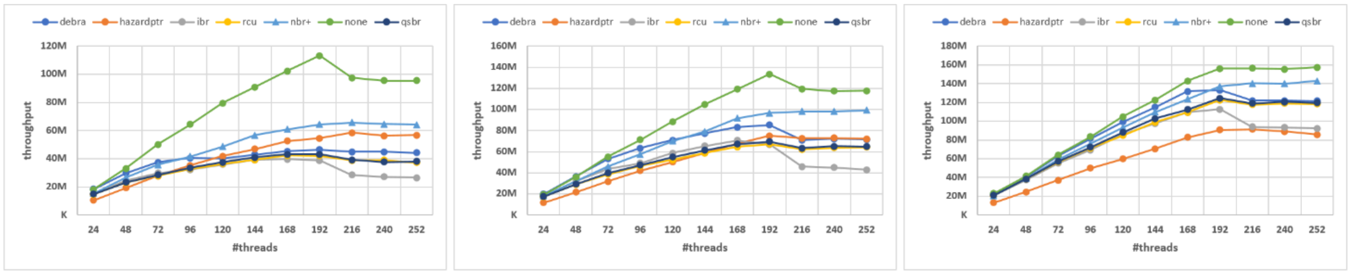
Furthermore, one may notice that the thread count where *NBR+* overtakes DEBRA is different in the *update intensive* and *search-intensive* workloads. This could be attributed to the fact that the overhead of burst reclamation sets in at lower thread counts for *update-intensive* workloads than in workloads with infrequent updates.

HP outperforms the other EBR variants in update-intensive workloads (Figure 3a, leftmost plot) but they appear to be slow in the search-intensive workload (Figure 3a, rightmost plot). This can be attributed to the fact that overhead due to delayed thread vulnerability dominates the over head of per read cost of fence in HP for update-intensive workloads. Whereas, for search-intensive workloads overhead due to delayed thread is lesser than that of per read fences in HP. Also, as one can observe in Figure 3b, in the lazylist *NBR+* is comparable to *RCU*, *QSBR*, and *DEBRA* and performs better than *HP* (by 2x) and *IBR* (by more than 50%) across all workloads and when oversubscribed [P1, P4].[i]

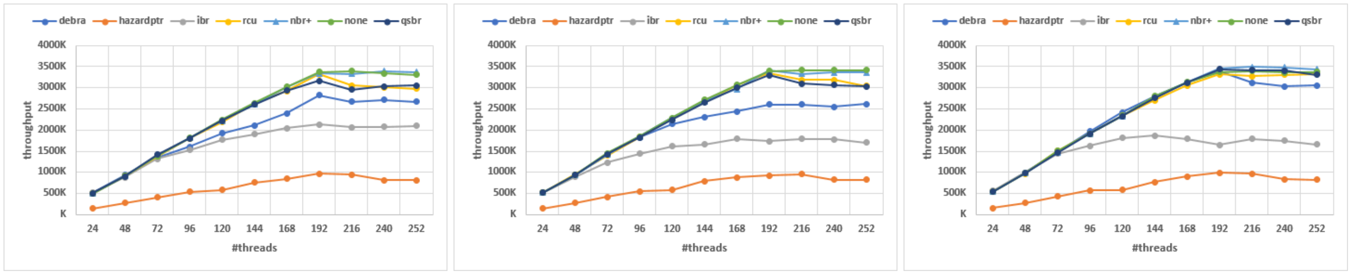
In our second type of experiment (E2), we validate the bounded garbage property of *NBR+* [P2] by measuring the peak memory usage of all reclamation algorithms when a thread is stalled (Figure 4c) and when no thread is stalled (Figure 4d). Each trial is run for 25 seconds. During this whole length of the experiment (25 seconds) one thread is made to sleep within a data-structure operation, imitating a

[h]We adapted QSBR, RCU and 2geibr (IBR) from the IBR benchmark, integarting them into Setbench to ensure a fair comparison.

[i]We believe that HP's poor performance is due to high cost of *mfence* used to publish the reservations which could be reduced by using more efficient xchg instructions to broadcast the reservations (refer section 11.5.1 in https://www.amd.com/system/files/TechDocs/47414_15h_sw_opt_guide.pdf). We did that optimization separately in our experimentation, and noticed that it did increase the absolute throughput for HPs, but HPs remained significantly slower than *NBR+*.

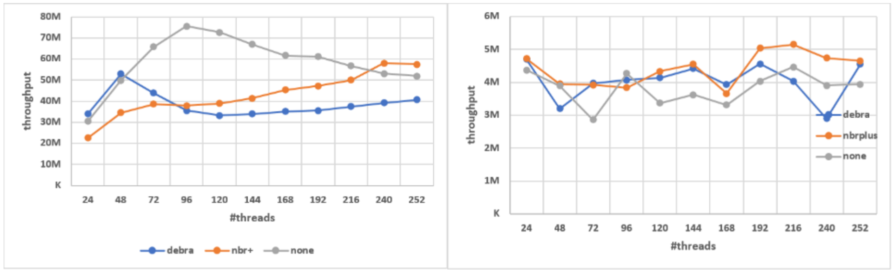


(a) DGT-tree. Left: 50i-50d. Middle: 25i-25d. Right: 5i-5d. Max size:2M.

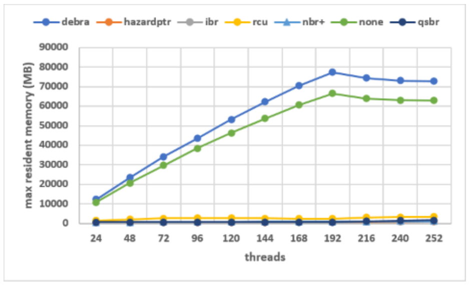


(b) Lazy linked-list. Left: 50i-50d. Middle: 25i-25d. Right: 5i-5d. Max size:20K.

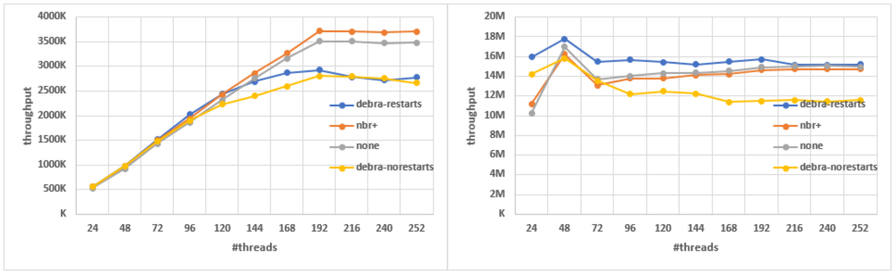
Figure 3. Evaluation of throughput. Y axis: throughput in million operations per second. X axis: #threads. As evident *NBR+* is fast [P1] and Consistent [P4].



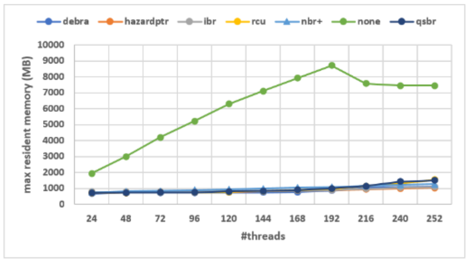
(a) Brown's ABTree. Left: 50i-50d. Max size: 2M. Right: 50i-50d. Max size:200.



(c) With stalled threads.



(b) Harris list. Left: 50i-50d. Max size:20K. Right: 50i-50d. Max size:200.



(d) NO stalled threads.

Figure 4. Fig. **(a), (b)**: Evaluation of throughput for data structures with multiple read-write phases. Y axis: throughput in million operations per second. X axis: #threads. Fig. **(c), (d)**: Y axis: Max Resident Memory, X axis: #threads, Workload: 50i-50d. DGT-tree. Max size: 2M. *NBR+* has bounded garbage (P2), i.e. have constant memory usage in presence of stalled threads along with *QSBR* and *IBR*.

stalled thread. As expected, since *DEBRA* and *RCU* do not have bounded garbage property, they exhibit an increase in peak memory usage in presence of a stalled thread (Figure 4c) while *NBR+*, *HP*, *IBR*, and *QSBR* variants maintain approximately the same peak memory usage due to continuous reclamation.

In the third experiment (E3) we evaluate throughput of *NBR* with data structures which have multiple read/write phases restarting from root, namely the Harris list[29] and the Brown ABTree[10, chapter 8]. We design our experiments to explore two disparate usage scenarios. First, we want to study reasonably large data structure size (2 M in the tree and 20K in the list) wherein we hypothesize restarts would be inexpensive due to low contention (leftmost in Figure 4). Second, we want to study extremely small data structures (200 in both the tree and the list) where restarting from the head node will occur as frequently as possible due to high contention (rightmost in Figure 4).

As we hypothesized, in the ABTree, *NBR+* is faster than the other SMR algorithms in the low contention scenario (especially at high thread counts). Moreover, in the high contention scenario, *NBR+* is comparable to DEBRA, suggesting that *NBR+* introduces relatively little overhead in practice due to restarting from the root in Φ_{read} (Figure 4a). Surprisingly, in the list, as can be seen in Figure 4b, *NBR+* is faster than DEBRA for very large thread counts in the low contention scenario (leftmost plot) and in high contention scenarios (rightmost plot) where restarts are costly *NBR+* is slower than DEBRA.

In (E4), for low contention, *NBR+* based on the modified HMList (HList) which enforces restarts is still faster than both *debra-restarts* (based on HList) and *debra-norestarts* (based on HMList). For high contention, *NBR*, surprisingly, is still faster than *debra-norestarts* but is slower than *debra-restarts*. This reveals that at least for lists *NBR*'s methodology is sufficiently faster such that restarts in data structure have low/negligible impact on its performance.

In summary, our experiments reveal that our *NBR* methodology is fast[P1], bounds garbage[P2] and consistent[P4]. Additionally, we also show that *NBR* can be fairly easily integrated [P3] in multiple data structures[P5].
Code: The latest version of our source code can be found on gitlab (https://gitlab.com/aajayssingh/nbr_setbench). The original submitted artifact is available at https://doi.org/10.5281/zenodo.4409185.

8 Conclusions

In this paper, we presented *NBR*, a *safe memory reclamation* algorithm that is a hybrid between EBR and a limited form of HPBR, and which uses POSIX signals to bound unreclaimed garbage. *NBR* is simpler to use than the most similar hybrid algorithm, DEBRA+, while supporting a large class of data structures, some of which are not supported by DEBRA+ (nor other popular SMR algorithms). We also developed an optimized version of *NBR* called *NBR+* that achieves similar throughput with fewer signals by passively observing signals being sent in the system to optimistically detect relaxed grace periods. Our experiments demonstrate that *NBR+* meets or exceeds the performance of the state of the art in SMR algorithms in typical benchmark conditions, while minimizing performance degradation in oversubscribed workloads.

Acknowledgments

This work was supported by: the Natural Sciences and Engineering Research Council of Canada (NSERC) Collaborative Research and Development grant: CRDPJ 539431-19, the Canada Foundation for Innovation John R. Evans Leaders Fund with equal support from the Ontario Research Fund CFI Leaders Opportunity Fund: 38512, Waterloo Huawei Joint Innovation Lab project "Scalable Infrastructure for Next Generation Data Management Systems", NSERC Discovery Launch Supplement: DGECR-2019-00048, NSERC Discovery Program under the grants: RGPIN-2019-04227 and RGPIN-04512-2018, and the University of Waterloo. We would also like to thank the reviewers for their insightful comments.

9 Artifact Description

This section provides a step by step guide to run our artifact (nbr_setbench) in a docker container. Other ways to setup a machine to use our artifact are provided in the *readme* file at the URL: https://doi.org/10.5281/zenodo.4409185.

To better reproduce the results described in our paper we recommend to run the nbr_setbench on a NUMA machine with atleast 2 NUMA nodes having a recent Linux distribution (we used Ubuntu 18.04 or 20.04) with ~ 188GB RAM and recent Docker installation (we used version 19.03.6, build 369ce74a3c).

Steps to load and run the provided Docker image: Sudo permission may be required to execute the following instructions.

1. Install the latest version of Docker on your system. We tested the artifact with the Docker version 19.03.6, build 369ce74a3c. Instructions to install Docker may be found at https://docs.docker.com/engine/install/ubuntu.
   ```
   $ docker -v
   ```
2. Download the artifact from Zenodo at URL: https://doi.org/10.5281/zenodo.4409185.
3. Extract the downloaded folder and move to *nbr_setbench/* directory using *cd* command.
4. Find docker image named *nbr_docker.tar.gz* in *nbr_setbench/* directory. And load the downloaded docker image with the following command:
   ```
   $ sudo docker load -i nbr_docker.tar.gz
   ```
5. Verify that image was loaded:
   ```
   $ sudo docker images
   ```
6. Start a docker container from the loaded image:
   ```
   $ sudo docker run --name nbr -i -t \
   --privileged nbr_setbench /bin/bash
   ```
7. Invoke *ls* to see several files/folders of the artifact: Dockerfile, README.md, common, ds, install.sh, lib, microbench, nbr_experiments, tools.

Steps to run the experiments: To compile, run and see results of the experiment follow these steps:

Input: Inputs to the experiment can be configured in corresponding input files at:
/nbr_setbench/nbr_experiments/inputs/

Output: Generated figures can be found in directory:
/nbr_setbench/nbr_experiments/plots/generated_plots/

1. Assuming you are currently in *nbr_setbench*, execute the following command:
   ```
   $ cd nbr_experiments/
   ```
2. Run the following command to generate plots for throughput evaluation:
   ```
   $ ./run.sh
   ```
3. Run the following command to generate plots for memory usage evaluation:
   ```
   $ ./run_memusage.sh
   ```

After the above scripts finish executing DO NOT exit the terminal as we would need to copy the generated figures on the host machine to be able to see them.

Steps to visualize the plots: Resultant figures could be found in *nbr_experiments/plots/generated_plots*.

To visualize the generated figures on your host machine copy the plots from the docker container to your host system by following these steps:

1. Verify the name of the docker container. Use the following command which would give us the name of the loaded docker container under NAMES column which is 'nbr'.
   ```
   $ sudo docker container ls
   ```
2. Open a new terminal on the same machine. Move to any directory where you would want the generated plots to be copied (use cd). And execute the following command to copy the generated plots from the *nbr_experiments/plots/generated_plots* folder to your current directory.
   ```
   $ sudo docker cp nbr:/nbr_setbench/ \
    nbr_experiments/plots/generated_plots/ .
   ```

Now you can analyse the generated plots. Each plot follows a naming convention:

1. throughput-[data structure name]-[number of inserts]-[number of deletes].png. For example, a plot showing throughput of DGT with 50% inserts and 50% deletes is named as: throughput-guerraoui_ext_bst_ticket-i50-d50.png.
2. Similarly the plot for peak memory usage experiments follows a naming convention: mem_usage-[data structure name]-[number of inserts]-[number of deletes].png. For example, a plot showing mem_usage of DGT with 50% inserts and 50% deletes is named as: mem_usage-guerraoui_ext_bst_ticket-i50- d50.png.

References

[1] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E Tarjan. 2014. The CB tree: a practical concurrent self-adjusting search tree. *Distributed computing* 27, 6 (2014), 393–417.

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

[3] Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. 2017. Forkscan: Conservative memory reclamation for modern operating systems. In *Proceedings of the Twelfth European Conference on Computer Systems*. 483–498.

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

[5] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. 2018. Getting to the Root of Concurrent Binary Search Tree Performance. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 295–306.

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

[7] Guy E Blelloch and Yuanhao Wei. 2020. Concurrent Reference Counting and Resource Management in Wait-free Constant Time. *arXiv preprint arXiv:2002.07053* (2020).

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

[9] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. *ACM Sigplan Notices* 45, 5 (2010), 257–268.

[10] Trevor Brown. 2017. Techniques for Constructing Efficient Lock-free Data Structures. *arXiv preprint arXiv:1712.05406* (2017).

[11] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A General Technique for Non-blocking Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. 329–342. Full version available from http://tbrown.pro.

[12] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 329–342.

[13] Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. 2020. Non-blocking interpolation search trees with doubly-logarithmic running time. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 276–291.

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

[15] Nachshon Cohen. 2018. Every data structure deserves lock-free memory reclamation. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–24.

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

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

[18] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized concurrency: The secret to scaling concurrent search data structures. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 631–644.

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

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

[21] Dana Drachsler, Martin Vechev, and Eran Yahav. 2014. Practical concurrent binary search trees via logical ordering. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming.* 343–356.

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

[23] Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. 2014. The amortized complexity of non-blocking binary search trees. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing.* 332–340.

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

[25] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the bsdcan conference, ottawa, canada.*

[26] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. 2019. Persistent non-blocking binary search trees supporting wait-free range queries. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures.* 275–286.

[27] Keir Fraser. 2004. *Practical lock-freedom.* Technical Report. University of Cambridge, Computer Laboratory.

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

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

[30] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of memory reclamation for lockless synchronization. *J. Parallel and Distrib. Comput.* 67, 12 (2007), 1270–1285.

[31] Meng He and Mengdu Li. 2017. Deletion without rebalancing in non-blocking binary search trees. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016).* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[32] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer, and Nir Shavit. 2005. A lazy concurrent list-based set algorithm. In *International Conference On Principles Of Distributed Systems.* Springer, 3–16.

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

[34] Shane V Howley and Jeremy Jones. 2012. A non-blocking internal binary search tree. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures.* 161–171.

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

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

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

[38] Ruslan Nikolaev and Binoy Ravindran. 2019. Hyaline: fast and transparent lock-free memory reclamation. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing.* 419–421.

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

[40] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent tries with efficient non-blocking snapshots. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming.* 151–160.

[41] Arunmoezhi Ramachandran and Neeraj Mittal. 2015. CASTLE: fast concurrent internal binary search tree using edge-based locking. *ACM SIGPLAN Notices* 50, 8 (2015), 281–282.

[42] Arunmoezhi Ramachandran and Neeraj Mittal. 2015. A fast lock-free internal binary search tree. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking.* 1–10.

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

[44] Niloufar Shafiei. 2013. Non-blocking Patricia tries with replace operations. In *2013 IEEE 33rd International Conference on Distributed Computing Systems.* IEEE, 216–225.

[45] Ajay Singh, Trevor Brown, and Ali Mashtizadeh. 2020. NBR: Neutralization Based Reclamation. arXiv:2012.14542 [cs.DC]

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

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

A Correctness

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

Property 5. *A thread T_j in Φ_{read}*

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

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

Property 6. *A thread T_j in Φ_{write}*

6.1 protects all records used in Φ_{write} and upon receiving a signal executes a signal handler.

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

Property 7. *Every* reclaimer *thread T_r does the following in order:*

7.1 sends signal to all participating threads.

7.2 scans all protected records of each participating thread T_j.

7.3 reclaims records which are not protected.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

B Applicability of NBR

In this section we will reason about why HP and *NBR* are applicable to certain data structure and not to several others as enumerated in Table 1.

In order to safely apply *NBR* to a concurrent data structure either 1) the data structure should offer a single read and single write phase or 2) if it has multiple read-write phases then each read-write phase pair should appear as if it is a separate operation. To be more specific, the latter requirement says that each read phase of the data structure should restart from the *root*. Additionally, one should be able to reserve all pointers that would be needed in a Φ_{write} beforehand. This is to ensure correct handshakes between *reader*s, *reclaimer*s and *writer*s. We formally state this requirement as follows:

Requirement 12. *Each* Φ_{read} *in a data structure using* NBR *should restart from the root.*

Requirement 13. *All references to shared pointers that could be accessed within a* Φ_{write} *should be reserved before entering it.*

Reasoning about the requirement desirable for safe application of HP to a data structure is more subtle. Fundamentally, access of a shared record in data structure operations using HP is a three step process where pointer to records are protected in a hand over hand manner. Assume a thread traversing from $rec1$ to $rec2$ by following a pointer to $rec2$ in the member field of $rec1$. Thus, the thread has pointers to both the records as its local variable. Without loss of generality, assume that $rec1$ is already protected and the thread is attempting to protect $rec2$. In order to successfully protect $rec2$ the thread should follow these steps:

1. **Announcement step:** announce pointer to $rec2$ at an *swmr* memory location so that it is timely visible to all other participating threads. This often requires using `mfence` or `xchg`.
2. **Reachability validation step:** $rec2$ should be reachable from the root at the time it was announced. This is to avoid announcing a freed node. It involves verifying that $rec2$ is still reachable from $rec1$ and $rec1$ is not marked. More detailed discussion could be found in Brown's seminal paper which proposed a fast EBR based SMR algorithm, DEBRA[14].

3. **Acquisition step:** If the *Reachability validation step* is successful then thread is said to have *acquired* HP on $rec2$ and any subsequent dereference of $rec2$ would be safe. Otherwise, the thread has to retry protecting the $rec2$.

We formally state this requirement for safe application of HP to a data structure as follows:

Requirement 14. *A thread that reads a shared record should acquire HP on it before using it.*

Thus, in order to safely use *NBR* and HP with a data structure Requirement 12 and 14 should be satisfied such that safety and progress guarantees of the original data structure are not violated.

Keeping these requirements in mind we will first reason about the applicability of HP to data structures in Table 1 followed by the application of *NBR*.

In LL05[32] searches are wait-free and updates use optimistic locking pattern. If HP is applied to LL05 it may happen that a thread could repeatedly fail to acquire HP (due to Requirement 14) on a node marked but not yet unlinked by a failed/delayed thread. This breaks waitfreedom of searches in the original data structure.

DGT2015[18], on the other hand, traverse the nodes in a sync-free manner (as in lazylists) and uses version numbers and ticket locks to optimistically execute the updates. However, since DGT15 doesn't use marking, it is not clear how HP could be acquired safely.

In HJ12, protect calls 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 a HP on the marked node, which would be impossible. This would block all threads from making progress. Such issues were described in [14]. Similarly, in HL17 [31] and interpolation search tree [13] 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. Thus, making it complex to apply HP to such data structures without significant modifications which may break the progress guarantees.

BCCO10[9] 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. 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. DVY14[21] is based on BCCO10, but they don't 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.

Reasoning about applicability of *NBR* is comparatively easy as one just needs to confirm whether every Φ_{read} restarts from root and if it is possible to reserve all records before entering a Φ_{write}, Requirement 12 and 13, respectively. Or 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 can be applied to EFRB10[24] as it has lock-less searches which either ends in an update or helping. In former case *NBR* can simply reserve the nodes returned by the search and execute the update within a Φ_{write}. In latter case, after helping a thread, it restarts from root. Thus, in order to help, *NBR* could simply reserve all pointers in the descriptors and then do a helping update inside a Φ_{write} followed by a Φ_{read} that restarts from root. However, *NBR* could not be applied to EFRB14[23] as after helping the tree restarts from nearby ancestors and not the root, meaning that Φ_{read} doesn't restart from root (Requirement 12).

HJ12[34], first lock-free internal BST is loosely based on EFRB BST. Here, the searches are required to do an auxiliary update to avoid missing a node in the tree because some concurrent update may have moved the node up on its search path, a pathology that occurs when search of a node overlaps with a two-child delete operation. However, in HJ12, after each auxiliary help, searches restart from 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, HJ12 applies to *NBR*.

Similarly, data structures designed using Brown's tree update template [10], for example lock-free chromatic tree, ABTree, AVL trees and HL17[31] could be used with *NBR*. In the template, operations do sync-free searches to find target node(s) (similar to standard Φ_{read}), then check whether they need to help (auxiliary updates, similar to entering Φ_{write} during searches), and after helping operations restart from root. Now since, helping involves descriptors which contains pointers to all nodes that would be required to execute the helping update, *NBR* could reserve all node pointers in the descriptor and enter Φ_{write} for the auxiliary update and then start subsequent search (Φ_{read}) from root.

lock-free patricia trie S13[44] too follows a pattern in which searches are sync-free. And, updates may do auxiliary helping using descriptors and then restart from root. Thus, *NBR* applies to S13 as we can know all records to be accessed in Φ_{write} beforehand through the descriptors and every search (Φ_{read}) post the auxiliary update restart from the root, satisfying both Requirement 12 and 13.

DGT15[18] has sync-free searches followed by locking one node for inserts, two nodes for deletes, then modifying the nodes. It is similar to single Φ_{read} followed by a single Φ_{write} design pattern of LL05[32]. Thus, *NBR* applies to DGT15.

Unbalanced external BST of Natarajan and Mittal (NM14) [37] too has a pattern where each operation starts with sync-free search that returns a *SeekRecord* object followed by updates. During updates, the operation may possibly help by accessing nodes only pointed by the *SearchRecord* object and then subsequently restart from root. Which is along the lines of *NBR*'s Requirement 12 and 13. 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. Thus, satisfying the requirements for *NBR*.

The unbalanced external BST in DVY14[21] does a sync-free search and then executes updates using locks. But within the update phase it may obtain pointers to new nodes for example, successor, children or parent of a node which may violate the Requirement 13 of *NBR*. However, in order to make it work with *NBR*, 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 their balanced variant of this tree, which does bottom-up rebalancing without restarting from the root between rotations.

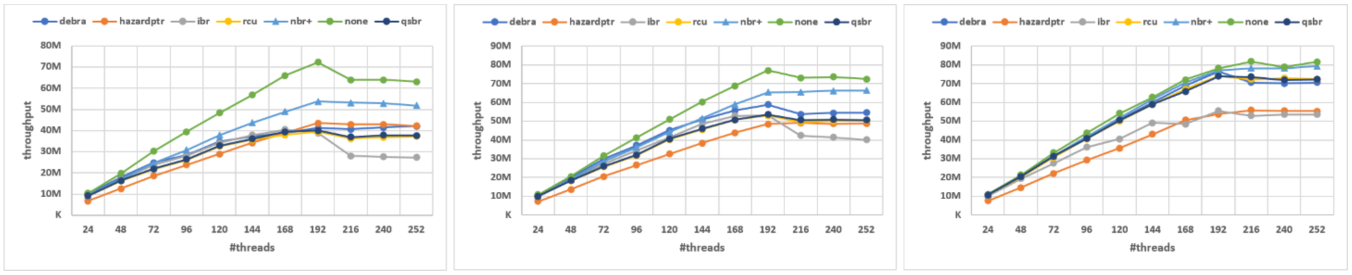
BCCO [9] Uses optimistic concurrency control (OCC) techniques to avoid locking nodes in searches as much as possible but occasionally locks and immediately unlocks a node as part of traversal, which suggests NBR cannot be used. however, as described in [5], this locking is an optional part of the BCCO 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.

Unbalanced internal BST of Ramachandran and Mittal (RM15)[42] ensures *search* operation for a key k does not miss k if it is moved upwards by having the search save a pointer to the successor of k. If the *search* does not find k, it checks the successor to see if k was moved. Whereas, an insert operation, first does a sync-free search, then does a CAS on appropriate child, and returns if it succeeds. If the CAS fails, insert re-reads the pointer it failed to change to check whether it points to a descriptor (which might need help). If the pointer is a descriptor, then insert helps it and subsequently restarts from the root. The fact that the operation re-reads the pointer after performing a CAS, and helps whatever descriptor it finds, is a problem for *NBR*. This would mean dereferencing a pointer obtained after entering the write phase without restarting from the root— violation of the Requirement 12. If we want to help safely, then before we do the aforementioned CAS, we must reserve both children of node, and if either child is actually descriptor, we must reserve all pointers in the descriptor (so we can safely dereference them if needed during helping). 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.

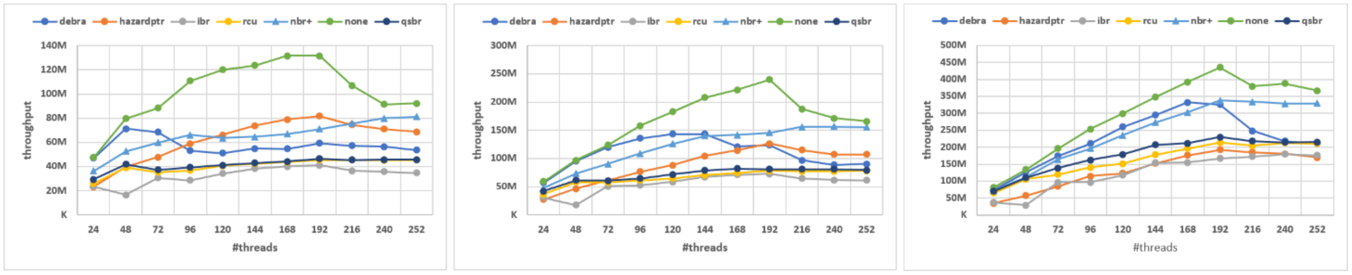
C Additional Experiments

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

For the lazy list (Figure 6a and Figure 6b) in extremely high contention (size 200 and update-intensive workload) *NBR+* is slower than the EBR based variants but still comparable to HP. The degradation in throughput could be attributed to the overhead of the signal apparatus –frequent neutralizing signals, `siglongjmp` and `sigsetjmp` which becomes prominent due to high number of updates and small list size relative to other SMR algorithms.

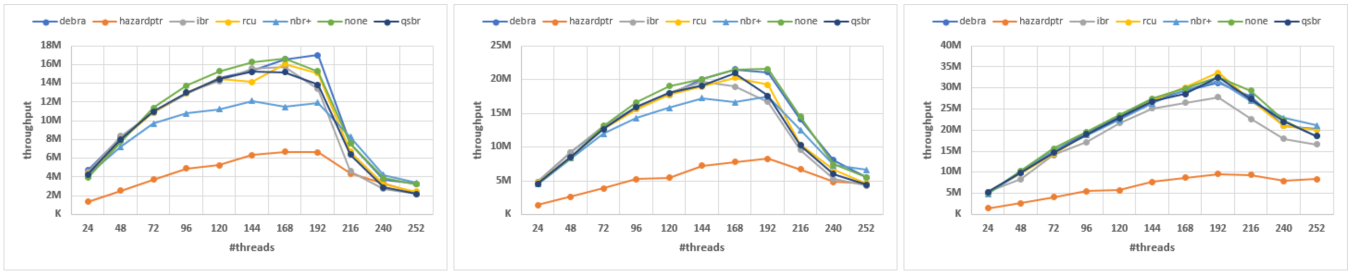


(a) DGT-tree. Left: 50i-50d. Middle: 25i-25d. Right: 5i-5d. Max size:20M.

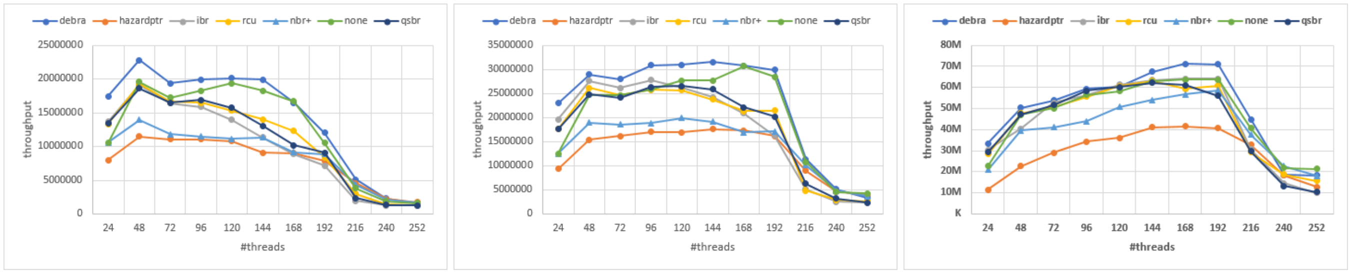


(b) DGT-tree. Left: 50i-50d. Middle: 25i-25d. Right: 5i-5d. Max size:20K.

Figure 5. Evaluation of throughput across different tree sizes. Y axis: throughput in million operations per second. X axis: #threads.



(a) lazy list. Left: 50i-50d. Middle: 25i-25d. Right: 5i-5d. Max size:2K.



(b) lazy list. Left: 50i-50d. Middle: 25i-25d. Right: 5i-5d. Max size:200.

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