RRR-SMR: Reduce, Reuse, Recycle: Better Methods for Practical Lock-Free Data Structures

MD AMIT HASAN AROVI, Pennsylvania State University, USA
RUSLAN NIKOLAEV, Pennsylvania State University, USA

Traditionally, most concurrent algorithms rely on safe memory reclamation (SMR) schemes for manual memory management. SMR schemes such as epoch-based reclamation (EBR) and hazard pointers (HP) are *typically* viewed as the *only* solution for memory recycling.

When using SMR, a new object needs to be allocated whenever something new is added to a data structure. However, in more complex scenarios, the same object may need to be moved between different data structures (e.g., moving a node from one list to another, and then back to the original list) in a copy-free manner, i.e., without deallocating and allocating the node again. It is typically impossible for two reasons: (1) the ABA problem would still arise even when using SMR since the same pointer can reappear (without going through the full SMR cycle) if the same node eventually ends up back in the original data structure; (2) while in simple queues and stacks, nodes can immediately be recycled, it is unclear how to adapt data structures which use non-trivial traversal and two-phase deletion strategies, e.g., linked lists, skip lists, hash tables, trees, etc., where it is *seemingly* impossible to always immediately move (logically) deleted objects since they might still be accessed by other threads.

We propose a general method of creating RRR (Reduce, Reuse, Recycle) data structures to allow safe memory recycling when using SMR which addresses the above-mentioned problems. Our method is applicable to linked lists, skip lists, hash tables, Natarajan-Mittal tree, and other data structures. We also discuss and propose a specialized approach – a more efficient version of Michael-and-Scott's (recycling) queue. Our evaluation on x86-64 shows promising results when using our methods for different data structures and SMR schemes.

CCS Concepts: • **Theory of computation → Concurrent algorithms**.

Additional Key Words and Phrases: non-blocking, ABA, hazard pointers, epoch-based reclamation

ACM Reference Format:
Md Amit Hasan Arovi and Ruslan Nikolaev. 2025. RRR-SMR: Reduce, Reuse, Recycle: Better Methods for Practical Lock-Free Data Structures. *Proc. ACM Program. Lang.* 9, PLDI, Article 234 (June 2025), 24 pages. https://doi.org/10.1145/3729337

1 Introduction

One of the most critical problems that any modern system must deal with is concurrency. In the past, blocking synchronization based on mutual exclusion sufficed. But even very basic systems today are becoming increasingly multi-core. Thus, *non-blocking* methods [10] that achieve a better level of parallelism are becoming increasingly popular.

Unfortunately, typical non-blocking data structures are breaking critical assumptions related to the classical memory management: (1) memory cannot be immediately reclaimed after an object is deleted from a data structure as there may exist stale pointers to the data structures for in-progress operations, and (2) memory objects cannot be moved from one data structure to another one as all deleted objects must be safely reclaimed, which can take an indefinite amount of time, before being

Authors' Contact Information: Md Amit Hasan Arovi, Pennsylvania State University, University Park, USA, arovi@psu.edu; Ruslan Nikolaev, Pennsylvania State University, University Park, USA, rnikola@psu.edu.

ACM 2475-1421/2025/6-ART234
https://doi.org/10.1145/3729337

reused again. The latter restriction is especially problematic – e.g., an object cannot be removed from one list and inserted into a different one directly, without reallocating and copying the object.

Note that the copy-free transfers are not that uncommon with the lock-based programming model, e.g., when processing requests and moving requests across different work queues (eventually returning into the original queue).[1] In some cases, copying is not even an option, e.g., when using non-blocking wait queues to implement locks (mutexes) as in [29]. Moving to or from the wait queue requires allocating a new object that should contain a copy of the previous object. However, memory allocators typically use locks, which creates a circular dependency here: to implement locks we need to allocate and copy an object, and to allocate an object, we first need to acquire locks. Moreover, "wait queues" may represent fairly complex data structures in practice, e.g., to support blocking timeouts, we may need sorted linked lists or even trees rather than simple queues. This makes transition to concurrent data structures error-prone, inconvenient, and costly in terms of copying overheads to the point when ordinary locks might be preferred instead.[2]

Additionally, the second problem would trigger the *ABA problem* (see Section 2) in typical concurrent algorithms due to false-positive pointer value matches when recycling memory objects. It is often erroneously assumed that SMR also fully solves the ABA problem [12], even though some SMR literature clearly acknowledges [20] that these two problems are independent. The only reason why the ABA problem does not typically occur with SMR is because all objects are deleted and reallocated again in typical implementations, as we further discuss in the paper.

Although the ABA problem itself can be easily solved by attaching a monotonically increasing tag to every pointer, ensuring that a given object has not been moved from the original data structure typically requires tracking *all* tags from the root. This contrasts with classical ABA-safe queue and stack algorithms [13], where such changes are easily detected at head or tail pointers.

To aggravate the problem further, linked lists, skip lists, hash tables, and trees use two-phase deletion, i.e., when the remove operation can merely mark an object "logically" deleted, while the actual "physical" deletion may take place in another thread. This creates serious obstacles for recycling after removal since "logically" deleted nodes cannot be immediately moved to another data structure, one of the central problem that we address in our paper.

We make several **contributions**: (1) we discuss the ABA problem and SMR in depth and differentiate these two independent but often conflated problems, (2) we propose a general method for building RRR data structures using Harris' linked list [11] and Natarajan-Mittal tree [25] as examples; our idea is also directly applicable to hash tables [19], skip lists [10], and other data structures that use an intermediate stage – "logical" deletion of their objects – which makes direct reuse of objects *seemingly* impossible, and (3) we propose an improved version of ABA-safe Michael-Scott's queue [21]; our approach is more memory efficient and more compatible with ABA-safe stacks.

In Section 5, we show practical benefits of using our model of memory management by focusing on two primary parameters – throughput and memory efficiency. In many cases, we demonstrate both memory efficiency improvement (up to 5x) as well as the throughput boost (up to 10x).

2 Background

We want to address above-mentioned memory management challenges for common non-blocking data structures such as queues, linked lists, hash tables, skip lists, and trees.

[1]This is a typical pattern when using *lock-based* lists in the Linux kernel. Consider a case when a task waits for a resource. The task is initially in the run queue, then moved to a wait queue until the resource becomes available, and finally moved back to the run queue once the resource is available.

[2]Linux provides a copy-free API for a lock-based list in "list.h" and an RCU-based list in "rculist.h", widely used in many device drivers. Conversely, "llist.h", which is dubbed as a lock-less list, is fairly limited (e.g., can only delete the very first node or all nodes) and still requires locks in some cases; it completely departs from the API of the former lists.

2.1 Lock-freedom

Among all non-blocking approaches, *lock-free* methods, which allow other threads to interfere at any point while still guaranteeing that *at least* one thread makes progress in a finite number of steps, received a particular interest in the literature. (It should be noted that good lock-free algorithms allow multiple threads to make concurrent progress in practice.) These must not be confused with simple spin locks or loosely-named "lockless" approaches that simply avoid explicit locks, where one preempted thread can still block all other threads.

2.2 Hardware Primitives

Special atomic *read-modify-write* instructions are typically used to implement non-blocking algorithms. Compare-and-swap (CAS), an instruction which atomically reads a memory word, compares it with the expected value, and exchanges it with the desired value is used almost universally in most algorithms. An alternative pair of instructions, load-link (LL) / store-conditional (SC), which splits the loading and writing phases but still guarantees atomicity, is preferred by many RISC architectures due to their lightweightness, e.g., RISC-V [31], ARM [3], MIPS [24], and POWER [14].

Double-width LL/SC or CAS, which manipulates two *adjacent* words, is widely available: x86-64 [15] and a recent RISC-V's "Zacas" extension [31] support 128-bit CAS, while ARM64 [3] supports both 128-bit LL/SC and CAS.

2.3 The ABA Problem

It is not uncommon [22] to use specialized lock-free data structures (e.g., queues or stacks) that avoid memory management issues altogether. Memory cannot be safely returned to the OS but can simply be recycled for new data entries in the data structure. Because entries are recycled and can return to the original data structure, CAS operations may erroneously succeed (*the ABA problem*) if the comparison is based solely on pointer values.

A widely acknowledged solution [13] is to pair a pointer with a tag by using double-width CAS. Since for a given variable, the tag value is unique (monotonically increasing upon every change), this ensures that even if the pointer value matches, CAS will only succeed if the variable has not yet been changed. LL/SC can also *theoretically* avoid the ABA problem by catching *any* alterations on the memory locations rather than just comparing values, but its usability is currently fairly limited due to lack of LL/SC nesting support, which is typically required by more complex ABA-safe algorithms. Thus, LL/SC typically implements CAS.

2.4 Safe Memory Reclamation (SMR)

If memory needs to be returned to an OS, a more complex approach is needed. Epoch-based reclamation (EBR) [10, 12] and hazard pointers (HP) [20] are two common approaches for SMR, where deallocation is split into two phases: (1) retiring an object once it has been promised that future threads would not access this object again by deleting it from a data structure, and (2) freeing the object once all in-progress threads have indicated that they no longer access stale pointers.

While it might be tempting to think that SMR fully addresses the ABA problem, it is not actually true in general. Consider a case when an object (O) is deleted from a data structure (A), then inserted to B, and is finally brought back to A. Unlike the simple ABA problem described above, memory is *occasionally* returned to the OS via SMR, but objects are also allowed to be moved around without triggering SMR. Unless we allocate a new copy when moving an object, sooner or later A may get a false positive match for the same object pointer value, which will corrupt the data structure.

2.5 ABA-safe Michael & Scott's Queue

One of the classical methods to solve the ABA problem is shown with Michael and Scott's (M&S) lock-free queue algorithm [21]. Figure 1 presents M&S queue with small changes: (1) we read the ABA tag and pointer components separately, using two 64-bit atomic load operations rather than a single 128-bit atomic load operation, which is not universally available and typically implemented via more expensive 128-bit CAS instruction on x86-64 and AArch64 (or, sometimes, 128-bit floating point instructions), (2) we only compare the tag component in lines L26 and L53 when verifying that the corresponding pointer has not changed, and (3) we show how a node can be safely re-initialized when moving from `src` to `dst` by using `RecycleNode`, which atomically increments the tag.

The tag comparison alone suffices to detect *any* pointer changes. However, we must also make sure that the pointer value is consistent with the tag value (as if it were fetched by the same 128-bit atomic load operation). This is why the order of operations at L22-L23 as well as L47-L48 is critical: the pointer load operation must be sandwiched between two tag load operations to detect any changes. L26 and L53, thus, serve an extra purpose, not envisioned in the original algorithm – emulating full atomicity of load operations.

At first sight, the above comes as a simple optimization to the original algorithm. However, the ability to read and *later* compare tags independently from pointers is an important property, which helps us to build a more general method.

2.6 Harris' Linked List

Timothy L. Harris [11] introduced the first practical CAS-based non-blocking implementation of lock-free linked lists. The fundamental problem in lock-free linked lists is that a deleting thread needs to simultaneously indicate that a node is deleted and change the preceding node's `next` pointer. Harris' key idea is that other threads can help with the deletion: the deleting thread first "logically" deletes the node by marking its `next` pointer, and then the node is "physically" deleted (i.e., unlinked) by either the originating thread or helper thread.

The original Harris' list allows *lazy traversals*, i.e., logically deleted nodes are simply jumped over by the read-only search operation. This presents a serious obstacle when building RRR (recyclable) lists that we have previously discussed. At first sight, this is infeasible: "logically" deleted nodes may get "physically" deleted by other threads in parallel and then moved to a different list. The search operation would then erroneously jump to a node located in an entirely different list. However, this issue could have been avoided if there were extra safety checks in the search operation.

Maged M. Michael altered [19] the algorithm such that upon encountering "logically" deleted nodes, they are immediately attempted to be "physically" deleted even in the search operation. This change is required when using HP rather than EBR due to differences in SMR mechanisms. However, we note that this also makes our problem more tractable, as we further discuss in this paper. That said, this change is still insufficient to build an RRR (recyclable) list since "logically" deleted nodes may still be accessed by other threads.

In Figure 2, we demonstrate Harris' list with Michael's change. The `Add` method inserts a new node into the list. Before inserting a new node, `Add` checks if the key of the to-be-inserted node exists via `Do_Find`. The new node gets inserted unless its key is already present in the list. `Remove` removes a node from the list. If the key is found in the list, the node is logically deleted at L27, and then one attempt is made to unlink it from the list at L28.

The biggest issue for *recyclability* arises when L28 fails, i.e., when memory cannot be immediately reclaimed. Contending threads cannot wait until L28 is complete: their `Do_Find` invocation helps to physically delete nodes that were previously marked as logically deleted by other threads (L46-L47). Consequently, contending threads will end up reclaiming the node in the original algorithm (L48).

```
     // ABA-tagged pointers
1    template <T> struct {
2      T *Ptr;
3      uint Tag;
4    } tagged;
     // Using tagged node pointers
5    struct {
6      tagged<node_t> Next;
7      data_t Data;
8    } node_t;
9    struct {
10     tagged<node_t> Head, Tail;
11   } q_t;
     // ABA-safe M&S queue
12   void InitQueue(q_t *q)
13     node_t *node = malloc(sizeof(node_t));
14     node->Next = { null, 0 };
15     q->Head = q->Tail = { node, 0 };
     // Allocate a brand new node
16   node_t *AllocNode(data_t data)
17     node_t *node = malloc(sizeof(node_t));
18     node->Data = data;
19     node->Next = { null, 0 };
20   void Enq(q_t *q, node_t *node)
21     while true
22       t.Tag = q->Tail.Tag;
23       t.Ptr = q->Tail.Ptr;
24       next.Tag = t.Ptr->Next.Tag;
25       next.Ptr = t.Ptr->Next.Ptr;
26       if ( t.Tag == q->Tail.Tag )
27         if ( next.Ptr == null )
28           new = { node, next.Tag+1 };
29           if ( CAS(&t.Ptr->Next, next, new) )  break;
30         else
31           new = { next.Ptr, t.Tag+1 };
32           CAS(&q->Tail, t, new);
33     new = { node, t.Tag+1 };
34     CAS(&q->Tail, t, new);

     // Recycle a node that was already previously used
35   void RecycleNode(node_t *node)
36     do                                  // Set to null and tag=tag+1
37       old = node->Next;
38       new = { null, old.Tag+1 };
39     while !CAS(&node->Next, old, new);
     // Moving nodes between queues
40   void Move(q_t *dstQ, q_t *srcQ)
41     node_t *node = Deq(srcQ);
42     if ( node != null )
43       RecycleNode(node);
44       Enq(dstQ, node);
45   node_t *Deq(q_t *q)
46     while true
47       h.Tag = q->Head.Tag;
48       h.Ptr = q->Head.Ptr;
49       t.Tag = q->Tail.Tag;
50       t.Ptr = q->Tail.Ptr;
51       next.Tag = h.Ptr->Next.Tag;
52       next.Ptr = h.Ptr->Next.Ptr;
53       if ( h.Tag == q->Head.Tag )
54         if ( h.Ptr == t.Ptr )
55           if ( next.Ptr == null )
               // Queue is empty
56             return null;
57           new = { next.Ptr, t.Tag+1 };
58           CAS(&q->Tail, t, new);
59         else
             // Retrieve data prior to CAS
60           data_t data = next.Ptr->Data;
61           new = { next.Ptr, h.Tag+1 };
62           if ( CAS(&q->Head, h, new) )
               // Move data to the sentinel node
               // since we are deleting the sentinel node
63             h.Ptr->Data = data;
64             return h.Ptr;
```

Fig. 1. Michael & Scott's (M&S) queue with ABA tagging.

Although Michael's modification, i.e., always calling L46-L47 even for `Contains`, is a good starting point to avoid jumping over to wrong lists, the issues of safe traversals still need to be properly addressed. Moreover, ABA safety does not mean much unless the recyclability problem is also fully resolved. We address both of these issues in our paper.

2.7 Other Data Structures

For simplicity, we limit our extensive background discussion to linked lists and queues. However, the technique presented in the paper is also directly suitable to hash tables [19], which is an array of linked lists, and skip lists [10], which, roughly speaking, represent multiple linked (sub)lists. In Section 3.2, we discuss our approach for Natarajan-Mittal tree [25] in detail.

3 Design

Our pseudo-code ignores memory reclamation for simplicity and generality. However, our implementation (evaluated in Section 5) properly integrates EBR and HP reclamation.

```
    // Node structure
1   struct {
2     node_t *Next;
3     key_t Key;                    // key is arbitrary
4   } node_t;
    // List structure
5   struct {
6     node_t *Head;
7   } list_t;
    // non-ABA safe Harris & Michael's linked list
8   void InitList(list_t *l)
9     l->Head = malloc(sizeof(node_t));
10    l->Head->Next = null;
11    l->Head->Key = $-\infty$;

12  bool Add(list_t *l, key_t key)
13    node_t *new = malloc(sizeof(node_t));
14    new->Key = key;
15    node_t **prev, *curr, *next;
16    while true
17      if ( Do_Find(l, key, &prev, &curr, &next) )
18        free(new);
19        return false;
20      new->Next = curr;
21      if ( CAS(prev, curr, new) )  return true;

22  bool Remove(list_t *l, key_t key)
23    node_t **prev, *curr, *next;
24    while true
25      if ( !Do_Find(l, key, &prev, &curr, &next) )
26        return false;
27      if ( !CAS(&curr->Next, next, getMarked(next)) )
          continue;
28      if ( CAS(prev, curr, next) )  smr_reclaim(curr);
29      return true;

30  bool Do_Find(list_t *l, key_t key, node_t ***p_prev,
      node_t **p_curr, node_t **p_next)
31    node_t **prev, *curr, *next;
32    prev = &l->Head;
33    curr = l->Head;
34    while true
35      if ( curr == null )  break;
36      next = curr->Next;
37      if ( *prev != curr )  goto 32 ;
38      if ( !isMarked(next) )
39        if ( curr->Key $\geq$ key )
40          *p_curr = curr;
41          *p_prev = prev;
42          *p_next = next;
43          return curr->Key == key;
44        prev = &curr->Next;
45      else
46        if ( !CAS(prev, curr, getUnmarked(next)) )
47          goto 32;
48        smr_reclaim(curr);
49      curr = next;
50    *p_curr = curr;
51    *p_prev = prev;
52    *p_next = next;
53    return false;

54  bool Contains(list_t *l, key_t key)
55    node_t **prev, *curr, *next;
56    return Do_Find(l, key, &prev, &curr, &next);

57  bool Move(list_t *dst, list_t *src, key_t key)
58    if ( Remove(src, key) )
59      return Add(dst, key);
60    return false;
```

Fig. 2. Linked list by Harris (w/ Michael's change).

3.1 New RRR-safe Linked List

There are two fundamental issues to be resolved for Harris' linked list: (1) there should be a safety mechanism that validates that a node is not moved (for ABA safety as well as to avoid jumping over to another list while traversing), and (2) there should be a mechanism which would make nodes immediately available after deletion (i.e., not delegating reclamation to a different thread).

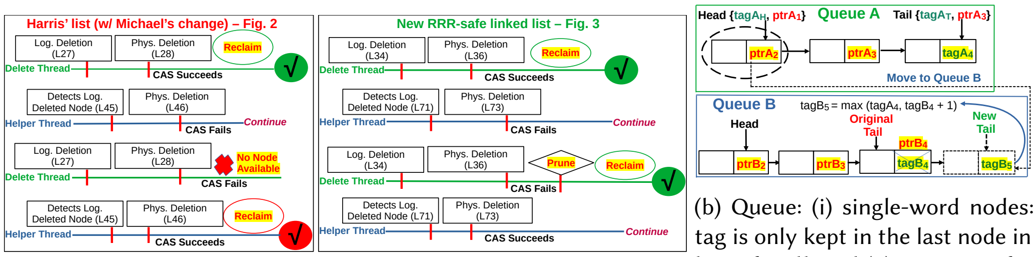
With respect to the **first problem**, we adopted an approach similar to that of M&S queue by using tags that are adjacent to pointers. The key idea here is that whenever a node is unlinked (physically removed), the preceding node's tag for the `next` pointer will also change. Let us consider a case when we reach $Node_{curr}$. Prior to jumping to this node, we have retrieved the next's tag from $Node_{prev}$, and we are about to go further. Before jumping to $Node_{next}$, we need to validate if the tag stored in the preceding node is still intact. In the original algorithm shown in Figure 2, L37 is conveniently checking that a node pointer has not changed after retrieving the next pointer. However, this validation is not ABA safe. Instead, we validate tags (L63) in Figure 3. Moreover, we must also retrieve the key (L62) prior to validating the tag.

To address the **second problem**, we observe that the physical removal encompasses two separate tasks: (1) executing CAS to unlink the node; and (2) taking possession of the node for future memory reclamation. In our design, we, effectively, propose three phases: logical deletion, physical removal,

```
1   template <T> struct {
2     T *Ptr;
3     uint Tag;
4   } tagged;
5   struct {
6     tagged<node_t> Next;
7     key_t Key;
8   } node_t;
9   struct {
10    tagged<node_t> Head;
11  } list_t;
12  void InitList(list_t *l)
13    node_t *node = malloc(sizeof(node_t));
14    node->Next = { null, 0 };
15    node->Key = −∞;
16    l->Head = { node, 0 };
17  bool Do_Add(list_t *l, node_t *new, key_t key)
18    tagged<node_t> *prev, curr, next;
19    while true
20      if ( Do_Find(l, key, &prev, &curr, &next) )
21        return false;
22      do
23        old_pair = { new->Next.Ptr, new->Next.Tag };
24        new_pair = { curr.Ptr, old_pair.Tag+1 };
25      while !CAS(&new->Next, old_pair, new_pair);
26      new_pair = { new, curr.Tag+1 };
27      if ( CAS(prev, curr, new_pair) ) return true;
28  node_t *Do_Rem(list_t *l, key_t key)
29    tagged<node_t> *prev, curr, next;
30    while true
31      if ( !Do_Find(l, key, &prev, &curr, &next) )
32        return null;
33      pair = { getMarked(next.Ptr), next.Tag+1 };
34      if ( !CAS(&curr.Ptr->Next, next, pair) ) continue;
35      pair = { next.Ptr, curr.Tag+1 };
36      if ( !CAS(prev, curr, pair) ) Do_Prune(l, curr.Ptr);
37      return curr.Ptr;
38  bool Add(list_t *l, key_t key)
39    node_t *new = malloc(sizeof(node_t));
40    new->Key = key;
41    new->Next = { null, 0 };
42    if ( !Do_Add(l, new) )
43      free(new);
44      return false;
45    return true;
46  bool Remove(list_t *l, key_t k)
47    node_t *node = Do_Rem(l, k);
48    if ( node != null ) smr_reclaim(node);
49    return node != null;
50  bool Contains(list_t *l, key_t k)
51    tagged<node_t> *prev, curr, next;
52    return Do_Find(l, k, &prev, &curr, &next);
53  bool Do_Find(list_t *l, key_t key, tagged<node_t> **p_prev,
      tagged<node_t> *p_curr, tagged<node_t> *p_next)
54    tagged<node_t> curr, next;
55    tagged<node_t> *prev = &l->Head;
56    curr.Tag = l->Head.Tag;
57    curr.Ptr = l->Head.Ptr;
58    while true
59      if ( curr.Ptr == null ) break;
60      next.Tag = curr.Ptr->Next.Tag;
61      next.Ptr = curr.Ptr->Next.Ptr;
62      curr_key = curr.Ptr->Key;
63      if ( prev->Tag != curr.Tag ) goto 55 ;
64      if ( !isMarked(next.Ptr) )
65        if ( curr_key ≥ key )
66          *p_curr = curr;
67          *p_prev = prev;
68          *p_next = next;
69          return curr_key == key;
70        prev = &curr.Ptr->Next;
71      else                                    // No reclamation here!
72        pair = { getUnmarked(next.Ptr), curr.Tag+1 };
73        if ( !CAS(prev, curr, pair) ) goto 55 ;
74      curr = next;
75    *p_curr = curr;
76    *p_prev = prev;
77    *p_next = next;
78    return false;
79  void Do_Prune(list_t *l, node_t *node)
80    tagged<node_t> curr, next;
81    tagged<node_t> *prev = &l->Head;
82    curr.Tag = l->Head.Tag;
83    curr.Ptr = l->Head.Ptr;
84    while true
85      if ( !curr.Ptr ) break;
86      next.Tag = curr.Ptr->Next.Tag;
87      next.Ptr = curr.Ptr->Next.Ptr;
88      if ( prev->Tag != curr.Tag ) goto 81 ;
89      if ( !isMarked(next.Ptr) )
90        prev = &curr.Ptr->Next;
91      else
92        pair = { getUnmarked(next.Ptr), curr.Tag+1 };
93        if ( !CAS(prev, curr, pair) ) goto 81 ;
94        if ( curr.Ptr == node ) break;         // Found the node!
95      curr = next;
96  bool Move(list_t *dst, list_t *src, key_t key)
97    node_t *node = Do_Rem(src, key);
98    if ( node == null ) return null ;
99    if ( !Do_Add(dst, node) )
100     smr_reclaim(node);
101     return false;
102   return true;
```

Fig. 3. A new RRR-safe linked list.

and taking possession of the node. The thread that *physically* removes the node, is not necessarily the thread that will eventually reclaim memory. Instead, the thread responsible for logical deletion will also be reclaiming memory.



(a) Linked List: ensuring that memory is reclaimed by the original thread.

(b) Queue: (i) single-word nodes: tag is only kept in the last node in lieu of **null**, and (ii) a new tag for recycled nodes is calculated using the max values across two queues.

Fig. 4. Original vs. RRR data structures.

In Figure 3, we demonstrate that helper threads no longer reclaim memory after physically deleting a node (L73). The original thread that calls `Do_Rem` takes possession of the node. If CAS in L36 succeeds, the node is physically deleted by `Do_Rem`, and no additional action is needed. However, if CAS fails, there are two possibilities: (1) the node is already physically deleted and we can now take possession of this node, or (2) the node is still in the list (logically deleted) but the state of the list has fundamentally changed causing CAS to fail. The first possibility is innocuous: no additional action is needed. However, for the second case, we must physically delete the node.

To that end, we introduce a special `Do_Prune` method, which is largely similar to `Do_Find`, except that it expects the node to be *not* present in the list. Note that only the original thread can take possession of the node and eventually recycle it. Thus, it suffices to search by the pointer value only. If the node with the provided pointer value is still in the list, it can only be a logically deleted node (L94), which is then immediately unlinked from the list inside `Do_Prune`. Upon `Do_Prune` completion, the node is no longer in the list.

Figure 4a shows an example with two threads for Harris' vs. proposed list. For each list, there are two examples: when CAS succeeds (two lines on top) and when CAS fails (two lines in the bottom). While the original thread in Harris' list cannot take possession of the node when CAS fails, the proposed list calls `Do_Prune`, after which the original thread is safe to recycle the deleted node.

The problem with the original Harris' list was not merely lack of ABA safety, but more importantly – lack of the clear ownership model. In the original algorithm, a thread that is *physically* deleting an object will reclaim memory. This is a serious obstacle when an object needs to be recycled by the deleting thread since the object might not be available for immediate recycling. We solved this problem by introducing the Do_Prune operation and experimentally validated that this change does not degrade performance materially. There is a trade-off to consider: the original Harris' list does not need `Do_Prune` when physical deletion fails in the original thread. `Do_Prune` may potentially scan the entire list to locate the logically deleted node or attest that it is no longer in the list. However, this only happens when the physical deletion fails in the deleting thread (L36), which does not affect performance much from our empirical observations. Moreover, the deletion operation already has the cost of `Do_Find` built in the operation. Therefore, the cost of traversals is simply doubled in the worst-case scenario, which does not change the overall asymptotic complexity of the algorithm.

3.2 Natarajan-Mittal Tree

We also implemented Natarajan-Mittal BST [25] via an indirection mechanism that we describe below. In Natarajan-Mittal BST, all *actual* keys are stored in leaf nodes, whereas keys in internal simply facilitate tree traversals in the proper direction. For the insertion operation, the original Natarajan-Mittal BST needs to allocate two (new internal and leaf) nodes. Removal, likewise, frees up two (internal and leaf) nodes. Thus, the copy-free move operation in the RRR-safe BST must move both the internal and the leaf node without allocating them again.[3]

A concept similar to Harris' "logical" deletion also exists in Natarajan-Mittal BST: *tagging*[4] (edges, sibling nodes) and *flagging* (leaf nodes), which are described in detail in [25]. Similar to Michael's modifications to linked lists, tagged edges must not be traversed lazily, as at every step, we need to validate that nodes are still in the original tree and have not been recycled. We use an approach similar to [5], which enables Michael-like modification for Natarajan-Mittal BST.

The main remaining obstacle is related to taking possession of removed nodes. Similar to linked lists, the original Natarajan-Mittal BST does not have a notion of strong ownership, as nodes are simply marked in `remove` but are potentially reclaimed by another thread when calling `cleanup`. There are three cases that need to be considered in the `remove` operation. In two cases, `remove` succeeds when the corresponding `cleanup` procedure succeeds: *parent* and *leaf* nodes are already known and can be immediately recycled. It is also possible that some other thread helps to complete `cleanup` after the "injection phase" of the `remove` operation [25], in which case it is not *always* possible to determine the *final* parent of the leaf node, as it may change during the `cleanup` process.

The parent node changes when two concurrent `remove` operations *simultaneously* flag two sibling leaf nodes. In such cases, one leaf node will be removed together with its parent while the other leaf node will be moved up to the previous location of the parent node. Consequently, the parent of the second leaf node will change. This is quite unfortunate as all knowledge about the new parent will be lost after `cleanup` finishes in the concurrent thread. We need to have a mechanism to prevent simultaneous flagging. However, it is impossible with the original data structure as flagging on the left and right leaf nodes are independent.

In Figure 5, we present a modification to address this problem by splitting simultaneous flagging. We use an indirection via a special *block* that contains both *left* (L) and *right* (R) pointers. Each internal node has an ABA-tagged pointer to the block, while each block contains ordinary (immutable) left and right pointers to the next level. Leaf nodes do not use blocks and simply point to **null** since they do not need to point to the next level. Whenever either left or right pointer changes, a new block have to be initialized and the ABA-tagged pointer has to be updated accordingly.

While allocating a new block together with a new leaf and internal nodes for the `insert` operation is fully expected, we **must avoid allocating new blocks** during the `move` or `remove` operations, which would defeat the purpose of our RRR approach. We, effectively, apply our recycling mechanism to the blocks. For this purpose, each thread keeps a pointer to exactly *one* spare block, which is allocated during tree initialization. Whenever, a thread calls an operation that needs to update left or right pointers, it initializes the spare block with new values accordingly. Then it performs the CAS operation to change the pointer to the block for the corresponding node. If the CAS operation succeeds, the previous block is recycled by the thread and will be used as its new spare block going forward. Otherwise, the spare block remains intact. While traversing, we expect that blocks move around (even across different trees), a problem that we already discussed in the context of linked lists. We use the same approach to reliably retrieve the node and its block contents by sandwiching data retrieval in the ABA tag check for the upper-level node.

We note that ABA-tag for the block pointer has a **dual** purpose to detect any changes to **both** block itself and next-level left/right pointers. The reason for that is that each block is *immutable*, once initialized, i.e., to change left or right pointers, we also have to change the block itself. Whenever either left or right node needs to be flagged, the block remains intact. However, the

[3]Note that while the leaf node is transferred without any changes, the key inside the internal node may need to be adjusted when moving. That said, neither the internal nor the leaf node needs to be reallocated.
[4]Not to be confused with the ABA tagging, an orthogonal concept used in this paper.

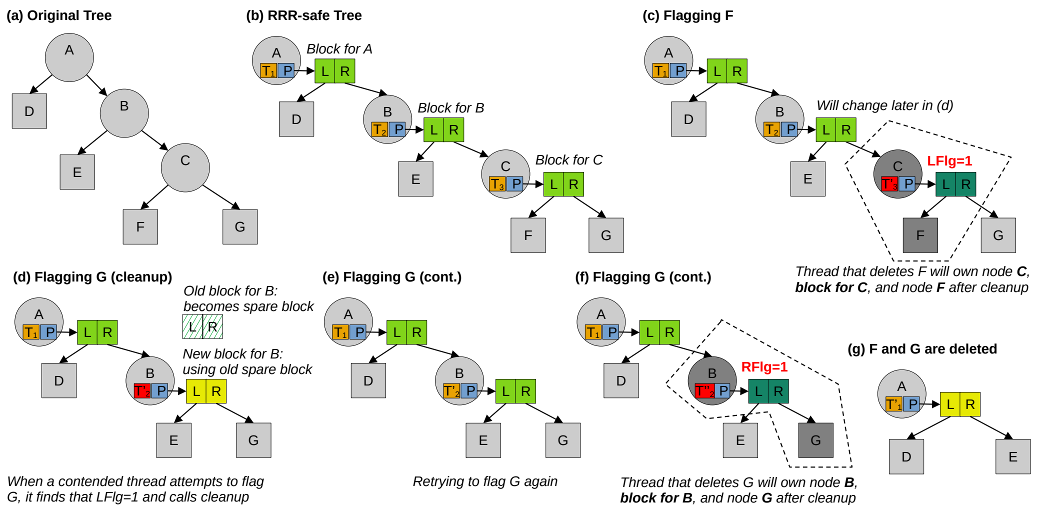

(a) Original Tree
(b) RRR-safe Tree
Block for A
Block for B
Block for C
(c) Flagging F
Will change later in (d)
LFlg=1
Thread that deletes F will own node **C**, **block for C**, and node **F** after cleanup
(d) Flagging G (cleanup)
Old block for B: becomes spare block
New block for B: using old spare block
When a contended thread attempts to flag G, it finds that LFlg=1 and calls cleanup
(e) Flagging G (cont.)
Retrying to flag G again
(f) Flagging G (cont.)
RFlg=1
Thread that deletes G will own node **B**, **block for B**, and node **G** after cleanup
(g) F and G are deleted

Fig. 5. Natarajan-Mittal BST change: two simultaneously deleted leaves (F, G) are being flagged separately.

block pointer is being flagged instead. We use two bits: *LFlg* for the left-node flagging and *RFlg* for the right-node flagging. By keeping two flags in the same block pointer, we can now fully prevent simultaneous flagging – a contended operation detects sibling flagging and calls `cleanup` before proceeding. Moreover, by packing two flags into the same pointer, we fully eliminate edge "tagging" from the original algorithm, e.g., once the left leaf node is flagged, it automatically means that the right edge is "tagged", which also allows to eliminate a separate edge-tagging CAS operation.

In terms of **space efficiency**, we still keep only two words in each node: ABA tag and block pointer (compare that to the original algorithm which keeps left and right pointers). Moreover, leaf nodes, which make more than half of the tree do not use blocks (point to **null**). Only internal nodes use blocks, which contain two ordinary pointers without ABA tags. Thus, the overhead is smaller to any (potential) alternative solution that would ABA-tag both left and right pointers.

In Figure 5a, we show an example for the original Natarajan-Mittal tree, consisting of (A-C) internal nodes and (D-G) leaf nodes. In Figure 5b, we show the same example for our proposed RRR-safe tree. Each node has a tuple consisting of a block pointer (P) and ABA tag (T). Every internal node points to its own block. Each block consists of immutable left (L) and right (R) ordinary pointers, which refer to the next-level nodes.

Assume that we have two concurrent operations to remove F and G, which should result in their flagging. In Figure 5c, F is about to be flagged. The first step is to check if its sibling is already flagged. For that purpose, we steal 2 bits from the block pointer P (LFlg and RFlg). Since F is about to be flagged, LFlg is changed from 0 to 1 (left child). Then, in Figure 5d, G is attempted to be flagged by setting RFlg in P from 0 to 1. However, since LFlg is already set, the contended thread detects a conflict and calls `cleanup` to help complete sibling (F) removal. At that point, new left (L) and right pointers (R) are initialized in the spare block. The spare block is then swapped with the previous block for B. After this process is complete, in Figure 5e, G is attempted to be flagged again. In Figure 5f, RFlg is set to indicate G's pending removal. As the final step, in Figure 5g, a spare block is used to initialize new left (L) and right (R) pointers and is exchanged with the old block for A.

Due to the substantial complexity of Natarajan-Mittal BST, we only discuss high-level code changes with respect to RRR safety. We publish full source code of our RRR-safe BST implementation.

```
   // Block structure
1  struct {
2    node_t *Left;
3    node_t *Right;
4  } block_t;
   // Node structure
5  struct {
6    tagged<block_t> Blk;
7    key_t Key;                         // key is arbitrary
8  } node_t;
   // Tree structure
9  struct {
10   node_t *R;           // sentinel ancestor node (root)
11 } tree_t;
   // Per-thread traversal structure
12 struct {
13   tagged<block_t> ancestor_blk;
14   tagged<block_t> parent_blk;
15   node_t *ancestor;
16   key_t ancestor_key;
17   node_t *successor;
18   node_t *parent;
19   key_t parent_key;
20   node_t *leaf;                   // the actual leaf node
21   key_t leaf_key;
22   node_t *left; // leaf is either left or right, the other
23   node_t *right;     // node is sibling (keeping both)
24   block_t *spare_block; // a per-thread spare block
25 } seek_record_t;
26 seek_record_t seekRecords[NUM_THREADS];
```

```
27 void Seek(tree_t *t, key_t key)
28   ...
29   while current != null
30     if ( checkFlg(seekRecord->parent_blk.Ptr, LFlg | RFlg) )
31       seekRecord->ancestor = seekRecord->parent;
32       seekRecord->ancestor_key = seekRecord->parent_key;
33       seekRecord->ancestor_blk = seekRecord->parent_blk;
34       seekRecord->successor = seekRecord->leaf;
35     ...
36     seekRecord->parent = seekRecord->leaf;
37     seekRecord->parent_key = seekRecord->leaf_key;
38     seekRecord->parent_blk = current_blk;
39     seekRecord->leaf = current;
40     seekRecord->left = left;
41     seekRecord->right = right;
42     while true
43       current_blk.Tag = current->Blk.Tag;
44       current_blk.Ptr = current->Blk.Ptr;
45       seekRecord->leaf_key = current->Key;
46       parent_tag = seekRecord->parent->Blk.Tag;
47       if ( seekRecord->parent_blk.Tag != parent_tag )  goto 27 ;
         // Except when current_blk is the last (leaf) node
48       left = getUnmarked(current_blk.Ptr)->Left;
49       right = getUnmarked(current_blk.Ptr)->Right;
50       if ( current_blk.Tag == current->Blk.Tag ) break;
51     if ( key < seekRecord->leaf_key )  current = left ;
52     else  current = right ;
```

Fig. 6. RRR-safe Natarajan-Mittal BST's key internal structures and traversal.

As in the original implementation, each thread keeps a record of traversed nodes: the actual *leaf* node, its *parent* internal node as well as *successor* and *ancestor* nodes. The successor node is the last untagged node prior to the parent node (can also be parent itself), while the ancestor node is the node one level up to the successor. To ensure RRR-safety, we also keep track of ABA tags and keys for these nodes *at the moment of traversal*. In Figure 6, we demonstrate how block and node internal structures are connected. We also show how the corresponding per-thread record used for traversals is changed. We sandwich node field retrievals (L43-L45) and block field retrievals (L48-L49) using the corresponding tag checks (L47, L50).

3.3 New RRR Queue

M&S queue uses the most straightforward solution for the ABA problem: pairing every single pointer with a tag. However, a queue is a specialized data structure, where new elements are only added to the tail of the queue, which can help optimize the method. Aside from the head and tail pointers, the ABA problem only needs to be *directly* addressed for the very last node in the queue. In the original M&S queue, the last node points to **null**, and we propose to reuse this space for the ABA tag. To identify the last node (i.e., that it keeps a tag), we steal the least significant bit from the pointer. (This is feasible as pointers are word-aligned.) This way, assuming 64-bit words, the same field can keep a 63-bit ABA tag in the last node and also indicate the end of the list. Aside from better memory efficiency, this approach is also convenient as node layouts become more *compatible* with Treiber's ABA-safe stack [34], i.e., we can move nodes from queues to stacks and vice versa.

Figure 7 summarizes all changes. While for brand new nodes, the tag can be obtained by simply incrementing the preceding node's tag value (L49), this is insufficient in a more general case.

```
struct {
  node_t *Next;
  data_t Data;
} node_t;
// A new RRR queue
node_t *Deq(q_t *q)
  while true
    h.Tag = q->Head.Tag;
    h.Ptr = q->Head.Ptr;
    t.Tag = q->Tail.Tag;
    t.Ptr = q->Tail.Ptr;
    next = h.Ptr->Next;
    if ( h.Tag == q->Head.Tag )
      if ( h.Ptr == t.Ptr )
        if ( next:IsTag == 1 )
          return null;                  // Queue is empty
        new = { next:Value, t.Tag+1 };
        CAS(&q->Tail, t, new);
      else
        data_t data = next:Value->Data; // Load before CAS
        new = { next:Value, h.Tag+1 };
        if ( CAS(&q->Head, h, new) )
          h.Ptr->Data = data;   // Move data to the sentinel
          return h.Ptr;
// Allocate a brand new node
node_t *AllocNode(data_t data)
  node_t *node = malloc(sizeof(node_t));
  node->Data = data;
  node->Next = { :IsTag=1, :Value=0 };
// Recycle an existing node
void RecycleNode(q_t *srcQ, node_t *node)
  node->Next = { :IsTag=1, :Value=GetTailTag(srcQ) };
uint GetTailTag(q_t *q)
  while true
    t.Tag = q->Tail.Tag;
    t.Ptr = q->Tail.Ptr;
    next = t.Ptr->Next;
    if ( t.Tag == q->Tail.Tag )
      if ( next:IsTag == 1 )
        return next;
      else
        CAS(&q->Tail, t, { next:Value, t.Tag+1 });
  CAS(&q->Tail, t, { node, t.Tag+1 });
void Enq(q_t *q, node_t *node)
  while true
    t.Tag = q->Tail.Tag;
    t.Ptr = q->Tail.Ptr;
    next = t.Ptr->Next;
    if ( t.Tag == q->Tail.Tag )
      if ( next:IsTag == 1 )
        if ( node->Next:Value < next:Value )
          node->Next:Value = next:Value + 1;
        if ( CAS(&t.Ptr->Next, next, { :IsTag=0,
           :Value=node }) ) break;
      else
        CAS(&q->Tail, t, { next:Value, t.Tag+1 });
  CAS(&q->Tail, t, { node, t.Tag+1 });
void Move(q_t *dstQ, q_t *srcQ)
  node_t *node = Deq(srcQ);
  if ( node != null )
    RecycleNode(node);
    Enq(dstQ, node);
```

Fig. 7. A new RRR queue: , separates words, : separates bits in the *same* word.

Consider an example where queue Q_1 has its last node N_1 with tag 2, and queue Q_2 has its last node N_2 with tag 1. When N_1 is removed and inserted to Q_2 after N_2, its tag becomes 2, which is obtained by incrementing the preceding tag. However, this is wrong as N_1 has appeared with the same tag (2) in both queues.

We need to ensure that the tag of N_1 will also be larger than its previous value in Q_1. Recall that N_1 can only be deleted as a sentinel node from the front of the queue. For the deletion to succeed, there must be at least one more node after N_1 in Q_1, and its tag is *at least* 3. Consequently, we can pick the maximum of the preceding tag + 1 in Q_2 and the last node's tag in Q_1. For the above example, N_1 gets tag 3 when it is inserted in Q_2 which resolves the problem and also ensures that tags in Q_2 are incremented monotonically.

Figure 4b shows how we move nodes from one queue to another. To retrieve the last node's tag, we introduced `GetTailTag`, which helps advance the tail pointer and retrieve the last node's tag.

4 Correctness

LEMMA 4.1. *Proposed Queue: The (implicit) ABA tag for any removed node is strictly smaller than the last node's tag in the queue.*

Proof. Tags are only preserved for the last node in the queue. Suppose there is just one node left in the queue. Then this node cannot be removed as this is the (only) remaining sentinel node. For any successful removals, we thus have at least two nodes in the queue. Suppose N_1, which is about to be removed, previously have had tag t_1 before it was erased with a pointer, i.e., before N_2 was inserted right after N_1. t_1 is no longer available, and its space is occupied by a pointer to N_2. When N_2 was inserted, at the very least, it had a value of $t_1 + 1$. Thus $t_2 > t_1$. □

Theorem 4.2. *Proposed Queue: The ABA tag for the node calculated as the maximum of the preceding node's tag + 1 from the destination queue and the last node's tag from the source queue is unique across all queues where the node has previously appeared.*

Proof. Recall that the first possibility is to simply make sure that ABA tags are incremented monotonically in the destination queue in a strictly increasing order. From Lemma 4.1, it follows that the node previously appeared with a smaller tag in the source queue. By induction, we can also say that the tag from the source queue was also greater than tags for this node in any other previous locations. Thus, the tag that will be used in the destination queue is unique across all other queues where the node has previously appeared. □

Theorem 4.3. *Linearizability for the proposed Queue: The proposed RRR-safe Queue achieves linearizability by ensuring that each operation (enqueue or dequeue) appears as an atomic event on a timeline T, in which each event conforms to the sequential specification of a queue.*

Proof. The proposed queue introduces a unique tagging function $f : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$, defined by $f(\text{src\_tail\_tag}, \text{dst\_tag}) = \max(\text{src\_tail\_tag}, \text{dst\_tag} + 1)$ to assign each node a tag that is strictly monotonic (see Theorem 4.2). This guarantees that each node version is unique, effectively preventing ABA scenarios. The tag t_i for node N_i satisfies $t_i > t'_i$ for every prior object instance (N'_i) in any queue, ensuring consistent comparisons. For each enqueue $E(x)$ and dequeue $D(y)$ operation, there exists a total order $<_T$ on $E(x) \cup D(y)$ such that each enqueue precedes the corresponding dequeue, if one exists. Operations on the queue are linearizable if $<_T$ preserves the order of actions according to this rule. The queue maintains uniqueness by ensuring t_i is strictly greater than all previous tags. Specifically, for nodes moved across queues, the tag $t_{\text{new}} = \max(t_{\text{tail\_src}}, t_{\text{prev\_dst}} + 1)$ ensures that no N_i reappears with an identical tag in two different queues. This invariant allows each enqueue and dequeue operation to be attributed to the correct queue and correct position in that queue in the timeline T. □

Theorem 4.4. *Proposed Linked List: After calling* `Do_Prune`*, the node is no longer present in the linked list and can be safely recycled.*

Proof. `Do_Prune` is only called when the physical deletion fails, for any reason, by the original (removal) thread. The method searches the node that was previously logically deleted. The first possibility is that `Do_Prune` fails to locate the node as it is already physically deleted by a concurrent thread, but this thread could not have taken possession of the node since the concurrent thread was not the originator of the deletion operation. The only remaining possibility is that the node is still in the list but marked as logically deleted. In that case, `Do_Prune` will unlink the node. □

Theorem 4.5. *Linearizability for the proposed RRR-safe Linked List: The proposed RRR-safe Linked List ensures that each operation (insertion, deletion, or find) can be modeled as an atomic event on a timeline T, consistent with the linearizable order for a linked list.*

Proof. Our change affects only the deletion operation. For each deletion operation $D(N_i)$ of node N_i, a two-phase deletion process – logical deletion $L_D(N_i)$ followed by physical removal $P_D(N_i)$ – ensures the node is removed consistently without interfering with ongoing operations.

Specifically, $D(N_i)$ appears atomic at a point between $L_D(N_i)$ and $P_D(N_i)$, guaranteeing a single instance in T. Using a prune function Do_Prune(N_i), the original deleting thread retains exclusive reclamation rights to N_i. This constraint provides that N_i can only reappear in the list if freshly recycled. Pruning further ensures that, once logically deleted, the node N_i is immediately unlinked and cannot re-enter the list without reinsertion. Each node N_i carries a unique tag t_i for the next pointer, which is verified at every traversal step. This tag-validation mechanism allows a traversal operation to detect if node N_i was altered by re-checking t_{i-1} from N_{i-1} before moving to N_{i+1}. The re-checking enforces consistent traversal order across concurrent threads, preventing nodes from inadvertently crossing list boundaries. □

Theorem 4.6. *Prevention of Simultaneous Sibling Deletion and Deterministic Parent Ownership in RRR-safe Natarajan-Mittal Tree: The proposed RRR-safe Natarajan-Mittal Tree ensures that, for every internal node n with two children L and R, at most one of the two deletion flags (*`LFLG`, `RFLG`*) can be set at any point in time. This serialization of logical deletions guarantees that only a single thread can successfully flag one of the siblings and deterministically acquires ownership of the parent node n.*

Proof. Let n be an internal node. Each internal node maintains a block pointer $\texttt{blk}(n) = \langle B, t, f \rangle$, where $B = (L, R)$ is an immutable block of child pointers, t is an ABA tag, and $f \in \{\texttt{LFLG}, \texttt{RFLG}\}$ encodes logical deletion flags. To flag a leaf (e.g., L), a thread first checks that the opposing flag (`RFLG`) is unset, then performs a CAS to atomically set `LFLG` and increment the tag. Because this update is atomic, only one flag can be set at any time and any concurrent attempt to flag the sibling will fail. This serialized flagging mechanism ensures that only one thread can take ownership of the parent node n, since a successful flag both logically deletes the target leaf and gives that thread the sole responsibility to reclaim n. Thus, simultaneous deletions are prevented. Formally, the invariant $\neg(f = \texttt{LFLG} \land f = \texttt{RFLG})$ holds, ensuring that siblings cannot be logically deleted concurrently. Furthermore, if a thread T_i successfully sets a flag via CAS on $\texttt{blk}(n)$, it becomes the sole owner of both the flagged child and the parent n and responsible for their recycling or retirement. □

Theorem 4.7. *Dual-Purpose ABA Tagging in RRR-safe Natarajan-Mittal Tree: Let n be an internal node in an RRR-safe Natarajan-Mittal Tree, and let* $\texttt{blk}(n) = \langle B, t, f \rangle$ *denote its atomic pointer, where* $B = (L, R)$ *is an immutable block of child pointers, t is an ABA tag, and* $f \in \{\texttt{LFLG}, \texttt{RFLG}\}$ *encodes the logical deletion state. The tag t serves a dual purpose: (i) it detects structural modifications to the left or right child* ***nodes****, and (ii) it distinguishes reused or recycled* ***blocks*** *that may have been moved to another part of the tree or to another tree. Furthermore, any traversal that observes* $\texttt{blk}_{before} = \langle B, t, f \rangle$*, reads the contents of block B, and subsequently rechecks* $\texttt{blk}_{after} = \langle B, t, f \rangle$*, is guaranteed to have observed a consistent view of both child pointers.*

Proof. In our design, blocks are immutable once initialized: their left and right pointers are fixed and cannot be changed in place. Any modification to the tree structure (e.g., during insert, remove, or move) must allocate (for insert) or recycle (for move or remove) a block and update the atomic tagged pointer in the parent node via a CAS operation. As a result, any valid structural change necessarily modifies either the pointer B, the tag t, or both. This guarantees that the tagged pointer reflects both logical and physical changes to the subtree. Because blocks may be recycled, the tag ensures uniqueness. During traversal, we apply the sandwich validation technique: the thread reads the tagged pointer before accessing block B, and re-reads it afterward to confirm that no change has occurred. If both tagged reads return $\langle B, t, f \rangle$, then the left and right pointers inside B are guaranteed stable for the duration of the read. If the tag changes, the contended thread detects a conflict and calls `cleanup` to help complete sibling removal or other concurrent operations. □

5 Evaluation

Aside from qualitative characteristics (ABA-safety, recyclability), the described memory management model has potential benefits in terms of performance and memory efficiency. This paper evaluates three data structures: a queue, a linked list, and a tree. We have used hazard pointers (HP) [20] and epoch-based reclamation (EBR) [10, 12], as these SMR schemes are commonly used. EBR is suitable for scenarios where memory reclamation can be postponed until all threads reach the synchronization point (the epoch boundary). However, HP requires more active tracking but gives immediate safety guarantees for each protected pointer and reduce latency in memory reclamation. Both EBR and HP have been actively studied and reported empirical success for many lock-free data structures. To ensure robustness under different memory management models, we integrated both EBR and HP into our queue and linked list implementations, and incorporated EBR into the tree algorithm. A comparison of our results against both SMR schemes shows the flexibility of our algorithm and provides performance guarantees.

Our benchmark is partially based on the test harness [30] and implements the following schemes:

- **Queue-HP:** Michael & Scott Queue for HP (not safe).
- **Queue-ABA-HP:** Michael & Scott Queue for HP (RRR and ABA safe).
- **ModQueue-ABA-HP (New):** Queue proposed in this paper for HP (RRR and ABA safe).
- **Queue-EBR:** Michael & Scott Queue for EBR (not safe).
- **Queue-ABA-EBR:** Michael & Scott Queue with EBR (RRR and ABA safe).
- **ModQueue-ABA-EBR (New):** Queue proposed in this paper for EBR (RRR and ABA safe).
- **List-HP:** Harris & Michael Linked List for HP (not safe).
- **List-ABA-HP (New):** Linked list proposed in this paper for HP (RRR and ABA safe).
- **List-EBR:** Harris & Michael Linked List for EBR (not safe).
- **List-ABA-EBR (New):** Linked list proposed in this paper for EBR (RRR and ABA safe).
- **NMTree-EBR:** Natarajan-Mittal Tree for EBR (not safe).
- **NMTree-ABA-EBR (New):** Tree proposed in this paper for EBR (RRR and ABA safe).

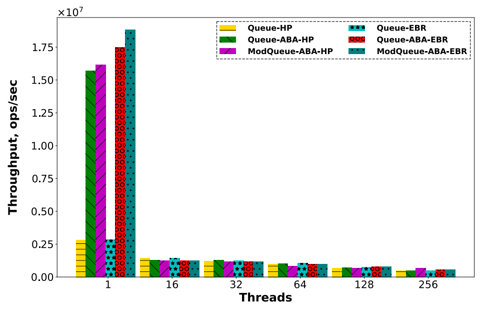
All ABA-safe versions are also RRR-safe, i.e., they deal with two-phase (logical) deletion properly.

Our testbed is the AMD EPYC 9754 128-core machine (up to 3.10 GHz), which is an x86-64 processor with an industry-leading core count on a single chip. We use all 256 hardware threads via hyperthreading. The system is equipped with 384 GiB of RAM. The benchmark is written in C++. We utilized clang++ 14.0.0 with -O3 optimizations as clang implements a more complete double-width CAS support for x86-64. We used **mimalloc** [23] as the memory allocator for the evaluation presented below due to its better performance. (For the sake of correctness verification, we also successfully ran all tests with standard malloc, which reclaims memory more aggressively.)

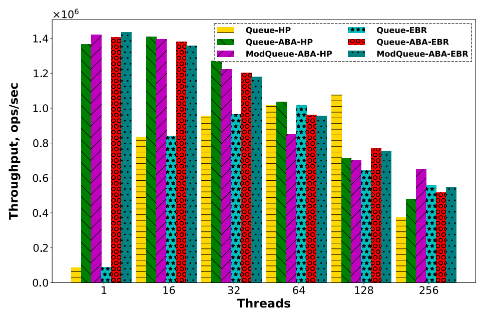
Since the differences may exist due to SMR mechanisms, we evaluate queue and list algorithms with both EBR and HP. No significant difference exists in overall relative trends. Due to complexity of NMTree and its compatibility issues with HP [1], we only implement and evaluate NMTree with EBR. For EBR, we use a more recent version with unconditional epoch increments [35].

We present results for both small and large payloads. Our small payloads already clearly demonstrate benefits. However, large payloads can also be useful. Even though we can always store a pointer, it comes with the cost of additional memory allocation. Moreover, the indirection cost would not only be an 8-byte pointer, but the entire cache line as nodes often need to be cache-aligned for performance. Let us say, we need to store 112 bytes (rounded to two 64-byte cache lines). When we store 112 bytes directly inside the node, we use 2 rather than 3 cache lines and avoid extra memory management operations. Avoiding indirection also helps to improve cache locality.

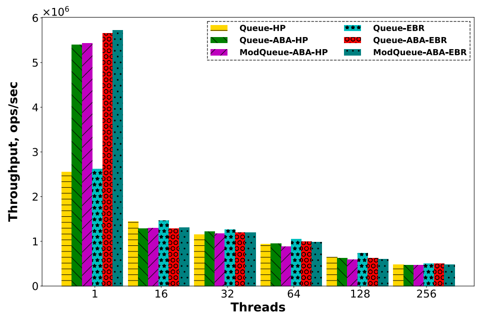
We chose a large payload size of **64 KiB** throughout our evaluation, as it is common for disk I/O operations and also not that far off from the maximum transmission unit (MTU) jumbo frame size



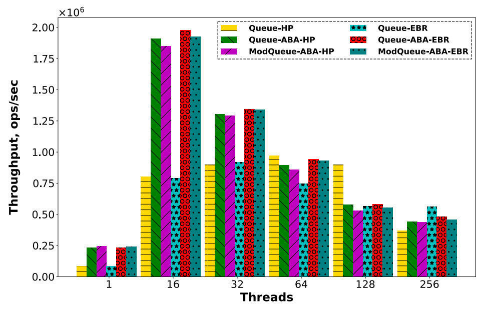
(a) No Payload, 90% recycle



(b) 64KiB Payload, 90% recycle



(c) No Payload, 50% recycle



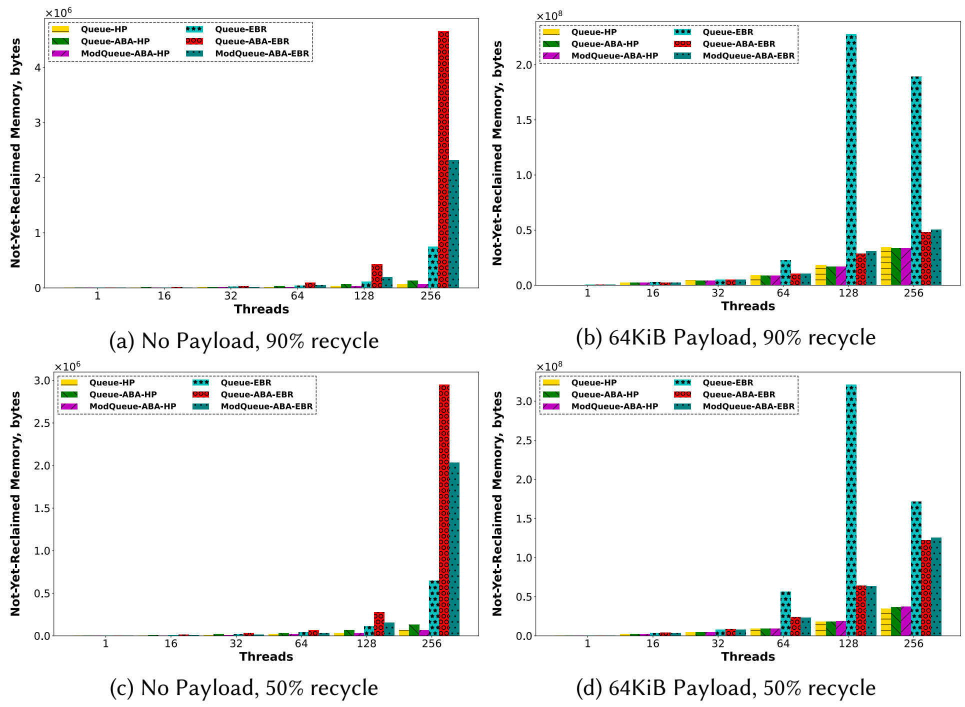
(d) 64KiB Payload, 50% recycle

Fig. 8. Queue [Random Operations], Throughput: *higher is better.*

for high-end (10GbE+) network adapters. We chose small 8-byte payloads for queues to represent a minimalist scenario where only an 8-byte value (e.g., pointer) is being kept, which is referred to as **No Payload** in our results. We chose a small payload size of **128 bytes** for lists and trees because these data structures typically keep keys of varying data types (strings, integers, etc.) and sizes. Additionally, whenever feasible, nodes can be aligned to 128 bytes on x86-64 to avoid false sharing.

For queues, we evaluated both **random** 50% enqueue and 50% dequeue operations and **pairwise** operations (one dequeue always follows one enqueue in every thread). Due to largely similar results, we present charts for **random** operations only. For linked lists and trees, we run a pair of deletion and insertion operations in every thread. We consider **two cases**: (1) 50% times recycling and 50% removal/insertion, and (2) 90% times recycling and 10% removal/insertion. 50%- and 90%-recycling refers to *directly* moving objects from one list, queue, or tree to another one, with the remaining operations involving insertion and deletion. Both cases are relevant depending on the actual usage scenario. The original (RRR-unsafe data structures) emulate recycling by allocating a new node, copying, and then disposing of the old node. This is the only way to move nodes across different data structures when no support other than basic SMR exists.

We demonstrate both throughput and memory consumption (waste). Memory consumption is measured via the number of not-yet reclaimed nodes. These are the nodes which are in limbo state due to the use of SMR. (To calculate memory waste, we capture snapshots of the average number of not-yet-reclaimed nodes periodically in each thread.) The memory consumption in our charts is the total size of each node with all its components, including ABA tags and pointers. This measurement ensures that any additional overhead brought in by our tagging scheme is fully included in the results. Although node sizes increase slightly due to ABA tags, our method decreases the number of nodes overall held in the limbo state, thus lowering overall memory consumption.



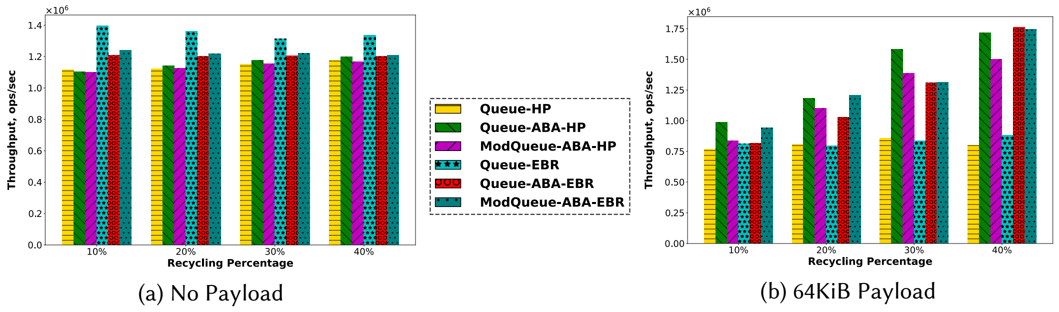
(a) No Payload, 90% recycle

(b) 64KiB Payload, 90% recycle

(c) No Payload, 50% recycle

(d) 64KiB Payload, 50% recycle

Fig. 9. Queue [Random Operations], Average Memory Consumption (waste): *lower is better.*



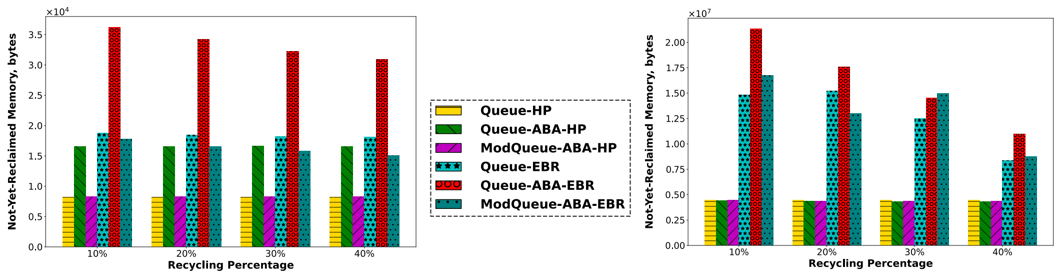
(a) No Payload

(b) 64KiB Payload

Fig. 10. Queue [Random Operations, 32 Threads], Throughput vs. Recycling Percentage: *higher is better.*

We calculate the median of all runs as the final result for throughput and memory consumption. The relative standard deviation is fairly negligible, mostly less than 1%.

For queues, we run the benchmark with 256 pre-filled elements, conducting 5 runs each lasting 20 seconds. Figure 8 illustrates that our queue algorithm and ABA-safe M&S queue outperform the ABA-unsafe M&S algorithm in terms of throughput in the peak throughput under both 50% and 90% recycling conditions (similar results are observed for both *random* and *pairwise* operations). Furthermore, our algorithm's throughput roughly matches to that of the (ABA-safe) M&S algorithm at various thread counts. Figure 9 reveals that the new algorithm is more memory efficient than (ABA-safe) M&S queue due to smaller nodes. The new queue often maintains comparable or better memory efficiency to the ABA-unsafe M&S queue in most threads while getting qualitative and



(a) No Payload (b) 64KiB Payload

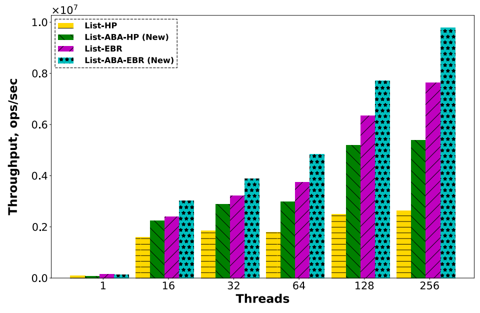
Fig. 11. Queue [Random Operations, 32 Threads], Memory Waste vs. Recycling Percentage: *lower is better.*

performance benefits of ABA-safe M&S queue. To further explore less frequent recycling, we evaluate throughput and memory consumption at 10%–40% recycling for 32 threads for random operations. In Figure 10, we see performance benefit for our new algorithm for both no payload and 64KiB payload scenarios. For 64KiB payload, the throughput gap with respect to (ABA-safe) M&S queue starts increasing at 10%. In Figure 11, we show memory consumption. For no extra payload, our algorithm shows excellent memory efficiency for both HP and EBR with respect to (ABA-safe) M&S queue. Even for 64KiB payload, our version continues to show more memory efficiency than (ABA-safe) M&S queue for 10% or more recycling percentage.

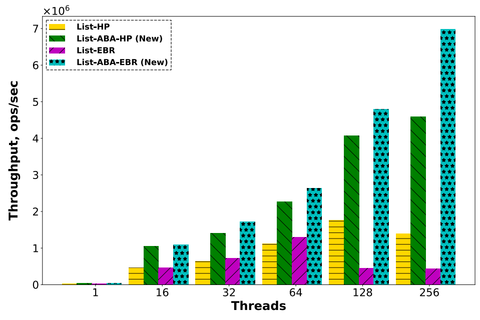
For some thread counts, our new queue has a higher throughput than (ABA-safe) M&S queue due to the use of single-width rather double-width CAS. On the other hand, under some scenarios, the throughput might also slightly drop due to the longer CAS loop which calculates a new tag in the last node. However, the differences in throughput are negligible. The major advantage of the new queue is its better compatibility with other ABA-safe data structures such as ABA-safe lock-free stacks [34], which do not use tags for the `next` pointers. Unlike M&S queue, the proposed queue does not keep tags in every node: they are only needed for the head and tail pointers.

For linked lists, we perform 5 runs with 4096 pre-filled elements, each running for 60 seconds. As illustrated in Figure 12, our ABA-safe linked list solution demonstrates highly competitive throughput compared to existing Harris & Michael solutions for both 128B and 64KiB payloads, under both 50% and 90% recycle scenarios. Specifically, for 64KiB payloads, our algorithm achieves notably higher throughput for both EBR and HP at 256 threads. While the gains are more modest for 128B payloads, our approach still consistently outperforms the baseline under both memory reclamation schemes. We see that for 128 thread and 256 threads for 64KiB payload, the original EBR-based algorithm experiences degraded throughput, largely due to excessive memory usage and sensitivity of EBR to even shortest system events. In terms of memory consumption, Figure 13 shows significant memory savings for 64KiB payloads, which are due to more memory being recycled directly and not held in the limbo state. Albeit smaller, memory savings also exist for 128-byte due to less memory being held in the limbo state. Further analysis (Figure 14) shows that our list design maintains superior throughput at lower recycling percentages (10%–40%) for both HP and EBR, across both payload sizes (128B and 64KiB). While Figure 15 shows that we incur slightly higher memory usage under low recycling rates, this difference drops at 50% and above. Note that even when memory usage is higher for our approach, throughput is also higher.

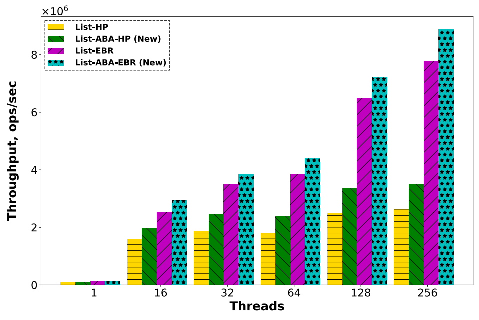
For trees, we run the benchmark with 65536 pre-filled elements, conducting 5 runs each lasting 60 seconds. As shown in Figure 16, for a 128B payload under 50% recycling, our algorithm performs comparably to the original Natarajan-Mittal (NM) tree at high thread counts (e.g., 128-256 threads). However, at 90% recycling, a notable performance gap emerges in favor of our algorithm, indicating



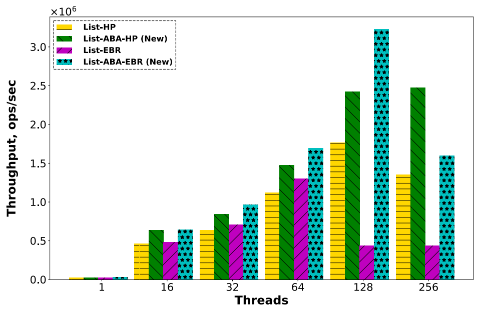
(a) 128B Payload, 90% recycle



(b) 64KiB Payload, 90% recycle

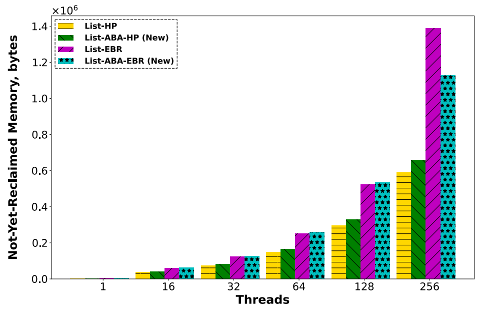


(c) 128B Payload, 50% recycle

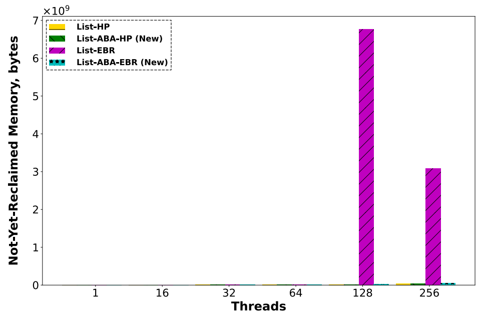


(d) 64KiB Payload, 50% recycle

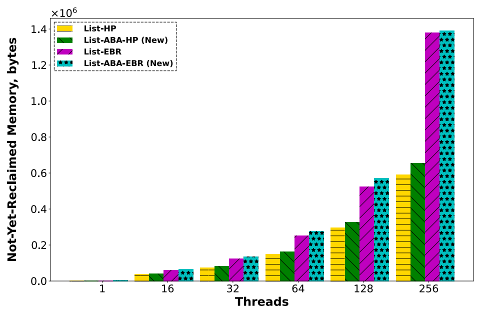
Fig. 12. Linked List Throughput: *higher is better.*



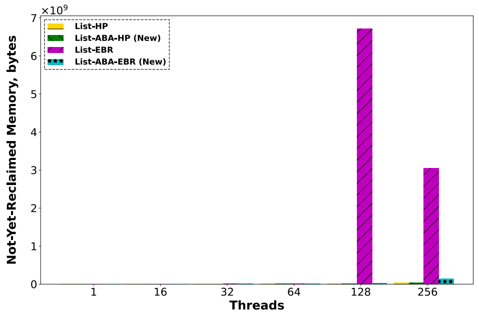
(a) 128B payload, 90% recycle



(b) 64KiB payload, 90% recycle

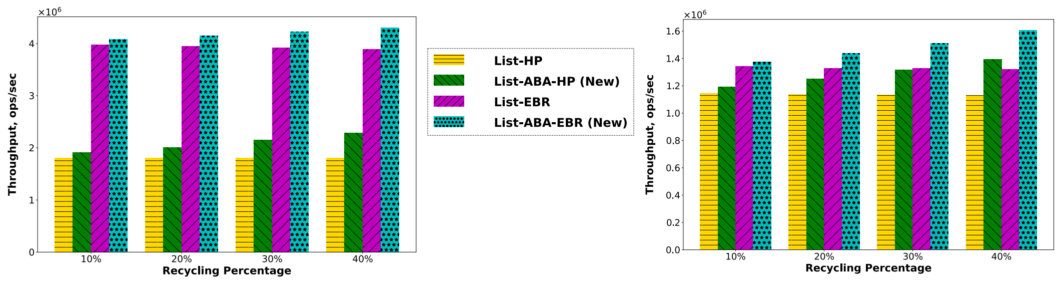


(c) 128B payload, 50% recycle



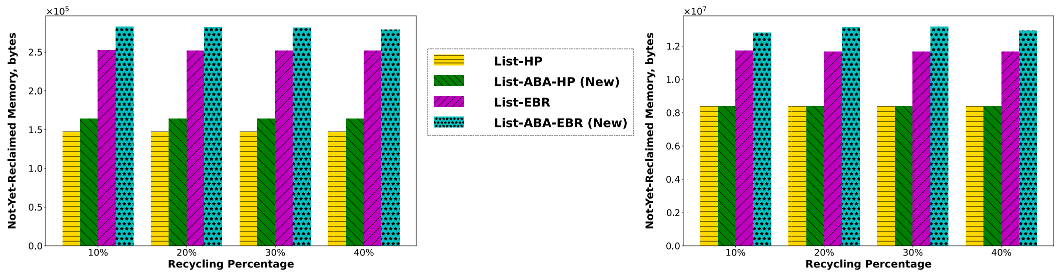
(d) 64KiB payload, 50% recycle

Fig. 13. Linked List Average Memory Consumption (waste): *lower is better.*



(a) 128B Payload (b) 64KiB Payload

Fig. 14. Linked List [64 Threads] Throughput vs. Recycling Percentage: *higher is better.*



(a) 128B Payload (b) 64KiB Payload

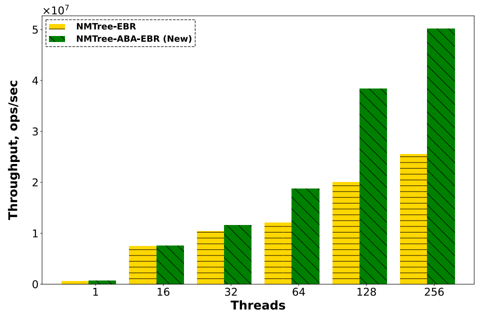
Fig. 15. Linked List [64 Threads] Memory Waste vs. Recycling Percentage: *lower is better.*

its benefits as recycling frequency increases. When the payload size increases to 64KiB, our approach consistently demonstrates significantly higher throughput across both 50% and 90% recycling scenarios. In particular, at 90% recycling with 32 threads (in peak throughput), our algorithm achieves nearly a 10x improvement compared to the original NM tree. Figure 17 presents the corresponding memory usage under EBR. Across all configurations, except one point, our algorithm consistently consumes less memory than the original NM tree. Overall, our design surpasses the original NM tree in both throughput and memory efficiency, especially under larger payloads.

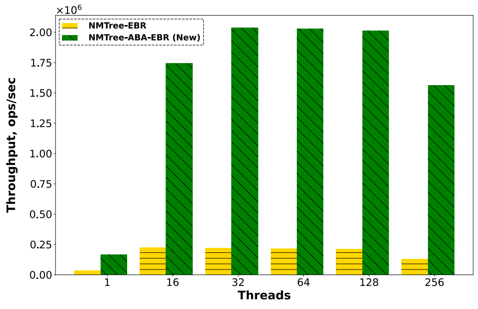
6 Related Work

Memory management for concurrent data structures has been studied for over three decades. We focus on past work that represents key ideas in the field. There is a cornucopia of both manual [7, 10, 12, 17, 18, 26–28, 30, 32, 33, 35] and automatic schemes [1, 2, 8, 9, 16]. In this work, without loss of generality, we consider manual SMR schemes, which are more suitable for languages such as C and C++. As discussed in [20], the ABA problem is orthogonal to the underlying SMR method and can still happen with automatic garbage collectors [13] under the circumstances considered in our paper. Moreover, the recyclability problem implies that memory objects can be *immediately* recycled, which is yet another problem independent from SMR and ABA problems.

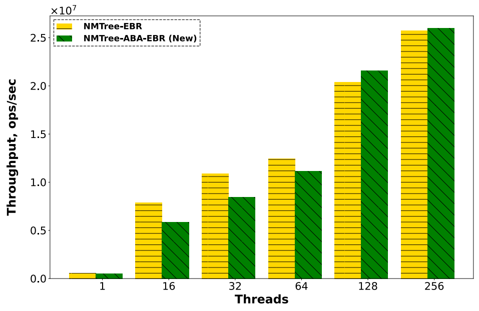
Most manual schemes can be categorized as epoch- or pointer-based reclamation schemes. In *epoch-based* reclamation (EBR) [10, 12], a thread uses the global epoch value at the beginning of a data structure operation as its reservation. Then, at the end of the data structure operation, it resets its reservation. A retired object is marked with the current epoch value at the time it is retired. It can only be safely reclaimed when its epoch is older than all current reservations. Unfortunately,



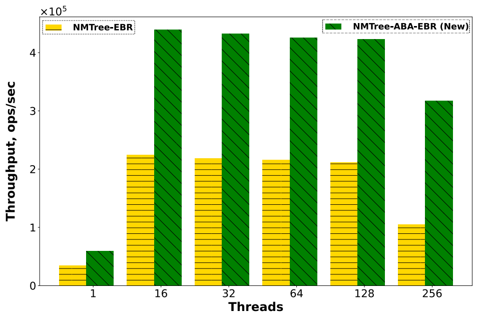
(a) 128B Payload, 90% recycle



(b) 64KiB Payload, 90% recycle

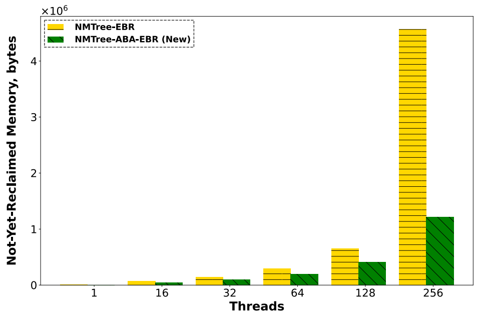


(c) 128B Payload, 50% recycle

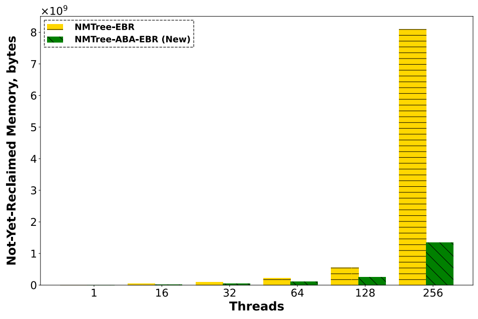


(d) 64KiB Payload, 50% recycle

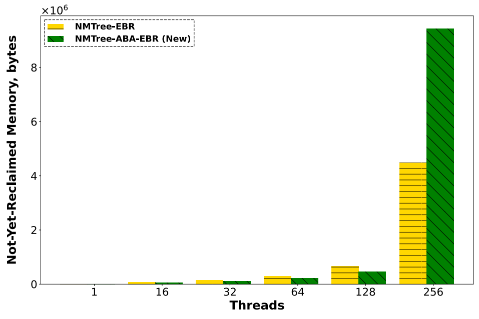
Fig. 16. NM Tree Throughput: *higher is better.*



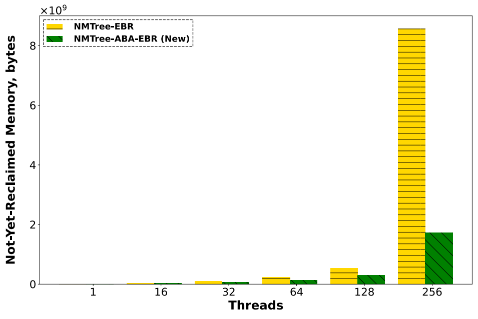
(a) 128B payload, 90% recycle



(b) 64KiB payload, 90% recycle



(c) 128B payload, 50% recycle



(d) 64KiB payload, 50% recycle

Fig. 17. NM Tree Average Memory Consumption (waste): *lower is better.*

EBR reserves an unbounded amount of memory if one thread stalls. This problem is addressed by more precise, *pointer-based* memory reclamation techniques, e.g., hazard pointers (HP) [20], are very precise as they track each object individually. HP bound memory usage even when threads are stalled but is more complex to use. Despite HP and EBR differences, our method applies to both.

ABA safety and recyclability have already been considered for a few *simple* lock-free data structures in the past. For example, Treiber's stack [34] can fully solve the ABA problem by placing an adjacent tag next to the stack top pointer. Michael and Scott's lock-free FIFO queue [21] is another example, where by using double-width CAS, we can add an ABA tag to *every* pointer. Thus, the queue would get around the ABA issues related to pointer changes. Both of these data structures can be used with the proposed approach. However, in this paper, we also present a new RRR-safe queue which is more memory efficient and more compatible with Treiber's stack.

Building more complex ABA-safe data structures is typically more challenging. Timothy L. Harris [11] pioneered the development of non-blocking linked lists, addressing the challenge of efficiently deleting nodes while ensuring thread safety. The approach allows other threads to assist in node deletion, marking nodes as "logically" deleted before physically unlinking them. However, the original Harris' implementation presented difficulties when ensuring RRR safety, particularly concerning lazy traversals and the risk of nodes being moved to different lists. Maged M. Michael [20] modified the algorithm to enable physical deletion of "logically" deleted nodes during search operations, which is crucial when using hazard pointers (HP).

Although Harris' list can be augmented with ABA tags, and Michael's modification enhances tractability of the problem, this still does not resolve the challenge of constructing an RRR-safe recyclable list, a central problem that we address in this paper.

Other well-known examples include skip lists and hash tables. A lock-free implementation of skip lists [10, 13], effectively, has to deal with multiple (internal) sublists. The same problem of logical and physical deletion needs to be addressed for skip lists in every sublist.

A hash table is a type of data structure that stores and retrieves key-value pairs quickly by utilizing a hashing mechanism. It offers quick access to values according to their keys and, on average, has constant-time complexity. A lock-free list-based set algorithm could be used as the building block of a lock-free hash table algorithm [19]. Thus, the RRR-safe solution solves the corresponding problem for hash tables.

Finally, lock-free locks [6] bring lock-free guarantees while using a more familiar lock-based API. However, the approach allocates new and reclaims old descriptors, which prevents direct recycling.

7 Conclusion

We introduced RRR (memory-recyclable) versions of the lock-free queue, linked list, and Natarajan-Mittal Tree. Our version of the queue provides a great alternative to the classical ABA-safe M&S queue since it only needs single-width CAS to manipulate node pointers. This makes nodes twice as smaller, and the overall algorithm becomes more memory efficient. This also makes queue nodes more compatible with other data structures such as ABA-safe Treiber's stack, which potentially allows intermixing nodes in these two data structures.

Our first-of-a-kind RRR-safe linked list demonstrates a more general method to solve the ABA safety and recyclability in data structures. Our method solves the fundamental problem – reclaiming memory after logical deletion, a technique used in many other data structures, not only linked lists. Thus, the same method can also be adopted for hash tables and skip lists that also rely on an intermediate step of logical node removal. Our evaluation for linked lists reveals that memory efficiency, better throughput, or both are achievable with our approach compared to the original Harris' algorithm. A similar method, which recycles two nodes rather than just one, is implemented with Natarajan-Mittal Tree, where the same practical benefits are observed.

Acknowledgments

We would like to thank the anonymous reviewers and shepherd for their insightful comments and suggestions, which helped greatly improve the paper.

Data-Availability Statement

The benchmark and data supporting this paper are available on Zenodo [4]. Our artifact includes: (1) a Linux VM image deployable with VirtualBox; (2) source code for the benchmark, along with all tested data structures and reclamation schemes; (3) benchmark scripts for running tests and generating charts. We include detailed instructions on how to replicate the results in this paper. The most up-to-date source code is available on GitHub as well: https://github.com/rusnikola/rrr-smr.

References

[1] Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. 2021. Concurrent deferred reference counting with constant-time overhead. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 526–541. doi:10.1145/3453483.3454060

[2] Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. 2022. Turning manual concurrent memory reclamation into automatic reference counting. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 61–75. doi:10.1145/3519939.3523730

[3] ARM. 2024. ARM Architecture Reference Manual. http://developer.arm.com/.

[4] Md Amit Hasan Arovi and Ruslan Nikolaev. 2025. Artifact for PLDI'25. https://doi.org/10.5281/zenodo.15258497

[5] Md Amit Hasan Arovi and Ruslan Nikolaev. 2025. Fixing non-blocking data structures for better compatibility with memory reclamation schemes. arXiv:2504.06254 [cs.DC] https://arxiv.org/abs/2504.06254

[6] Naama Ben-David, Guy E. Blelloch, and Yuanhao Wei. 2022. Lock-free locks revisited. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 278–293. doi:10.1145/3503221.3508433

[7] 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* (Donostia-San Sebastián, Spain) *(PODC '15)*. Association for Computing Machinery, New York, NY, USA, 261–270. doi:10.1145/2767386.2767436

[8] Nachshon Cohen. 2018. Every data structure deserves lock-free memory reclamation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 143 (oct 2018), 24 pages. doi:10.1145/3276513

[9] Andreia Correia, Pedro Ramalhete, and Pascal Felber. 2021. OrcGC: Automatic Lock-Free Memory Reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21)*. ACM, 205–218. doi:10.1145/3437801.3441596

[10] Keir Fraser. 2004. *Practical lock-freedom.* Technical Report. Univ. of Cambridge, Computer Laboratory. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf

[11] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*. Springer-Verlag, Berlin, Heidelberg, 300–314.

[12] 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. doi:10.1016/j.jpdc.2007.04.010

[13] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[14] IBM. 2005. PowerPC Architecture Book, Version 2.02. Book I: PowerPC User Instruction Set Architecture. http://www.ibm.com/developerworks/.

[15] Intel. 2024. Intel 64 and IA-32 Architectures Developer's Manual. http://www.intel.com/.

[16] Jaehwang Jung, Jeonghyeon Kim, Matthew J. Parkinson, and Jeehoon Kang. 2024. Concurrent Immediate Reference Counting. *Proc. ACM Program. Lang.* 8, PLDI, Article 153 (June 2024), 24 pages. doi:10.1145/3656383

[17] Jaehwang Jung, Janggun Lee, Jeonghyeon Kim, and Jeehoon Kang. 2023. Applying Hazard Pointers to More Concurrent Data Structures. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures* (Orlando, FL, USA) *(SPAA '23)*. Association for Computing Machinery, New York, NY, USA, 213–226. doi:10.1145/3558481.3591102

[18] Jeehoon Kang and Jaehwang Jung. 2020. A marriage of pointer- and epoch-based reclamation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association

for Computing Machinery, New York, NY, USA, 314–328. doi:10.1145/3385412.3385978

[19] Maged M. Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Winnipeg, Manitoba, Canada) *(SPAA '02)*. Association for Computing Machinery, New York, NY, USA, 73–82. doi:10.1145/564870.564881

[20] Maged M. Michael. 2004. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (June 2004), 491–504. doi:10.1109/TPDS.2004.8

[21] Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* (Philadelphia, Pennsylvania, USA) *(PODC '96)*. Association for Computing Machinery, New York, NY, USA, 267–275. doi:10.1145/248052.248106

[22] Microsoft. 2021. Windows App Development: Interlocked Singly Linked Lists. https://learn.microsoft.com/en-us/windows/win32/sync/interlocked-singly-linked-lists.

[23] Microsoft. 2024. Mimalloc allocator. https://github.com/microsoft/mimalloc.

[24] MIPS. 2016. MIPS32/MIPS64 Rev. 6.06. http://www.mips.com/products/architectures/.

[25] 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* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 317–328. doi:10.1145/2555243.2555256

[26] 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* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 130–143. doi:10.1145/3332466.3374540

[27] Ruslan Nikolaev and Binoy Ravindran. 2021. Snapshot-free, transparent, and robust memory reclamation for lock-free data structures. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 987–1002. doi:10.1145/3453483.3454090

[28] Ruslan Nikolaev and Binoy Ravindran. 2024. A Family of Fast and Memory Efficient Lock- and Wait-Free Reclamation. *Proc. ACM Program. Lang.* 8, PLDI, Article 235 (June 2024), 25 pages. doi:10.1145/3658851

[29] Ruslan Nikolaev, Mincheol Sung, and Binoy Ravindran. 2020. LibrettOS: a dynamically adaptable multiserver-library OS. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 114–128. doi:10.1145/3381052.3381316

[30] 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* (Washington, DC, USA) *(SPAA '17)*. Association for Computing Machinery, New York, NY, USA, 367–369. doi:10.1145/3087556.3087588

[31] RISC-V International. 2024. Ratified Specification. https://riscv.org/specifications/ratified/.

[32] Gali Sheffi, Maurice Herlihy, and Erez Petrank. 2021. VBR: Version Based Reclamation. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '21)*. Association for Computing Machinery, New York, NY, USA, 443–445. doi:10.1145/3409964.3461817

[33] Ajay Singh, Trevor Brown, and Ali Mashtizadeh. 2021. NBR: neutralization based reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 175–190. doi:10.1145/3437801.3441625

[34] R. K. Treiber. 1986. *Systems Programming: Coping with Parallelism.* Technical Report RJ 5118. IBM Almaden Research Center.

[35] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. 2018. Interval-based memory reclamation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3178487.3178488

Received 2024-11-15; accepted 2025-03-06