RRR-SMR: Reduce, Reuse, Recycle - Better Methods for Practical Lock-Free Data Structures [draft, do NOT distribute]

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 recycling memory.

When using SMR, a new object needs to be allocated whenever adding something new to a data structure. However, when dealing with more complex situations, the same object has to be moved around different data structures (e.g., moving a node from one list to another one and then back to the original one) 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 pointer eventually ends up in the original data structure again; (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, memory reclamation, ABA

1 Introduction

One of the most critical problem 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 [5] that achieve a better level of parallelism are becoming increasingly popular.

Unfortunately, typical data structures that use non-blocking techniques 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 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

[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.

Authors' Contact Information: Md Amit Hasan Arovi, Pennsylvania State University, University Park, USA, mqa6105@psu.edu; Ruslan Nikolaev, Pennsylvania State University, University Park, USA, rnikola@psu.edu.

non-blocking wait queues to implement locks (mutexes) as in [2, 19]. 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 [7], even though some SMR literature clearly acknowledges [12] 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 trivially solved by attaching a monotonically increasing tag to every pointer, *all* tags generally have to be tracked from the root of the data structure to guarantee that a given object has not yet been moved from the original data structure. This is different from classical ABA-safe queue and stack algorithms [8], where such changes are trivially 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 clearly differentiate these two independent but often conflated problems, (2) we propose a general methodology for building RRR data structures using Harris' linked list [6] and Natarajan-Mittal tree [17] as examples; our idea is also directly applicable to hash tables [11], skip lists [5], 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 [13]; our approach is more memory efficient and provides better compatibility with ABA-safe stacks.

Our experimental results show practical benefits of using our model of memory management. In Section 5, we focus on two primary parameters – throughput and memory efficiency. In many cases, we demonstrate both significant 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.

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

[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, "freelist.h", which is dubbed as lock-free is a fairly limited list (and not even fully lock-free), which completely departs from the API of the former lists.

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 needed 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 [21], ARM [1], MIPS [16], and POWER [9].

Double-width LL/SC or CAS, which manipulates two *adjacent* words, is widely available: x86-64 [10] supports 128-bit CAS, and ARM64 [1] supports both 128-bit LL/SC and CAS.

2.3 The ABA Problem

It is not uncommon [14] 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 [8] is to pair a pointer with a tag by using double-width CAS. Since for a given variable, the tag value is unique (monotonically increments 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) [5, 7] and hazard pointers (HP) [12] are two common approaches for SMR, where the deallocation process is split into two phases: (1) retiring an object once it has been promised that future threads will no longer 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 for simple data structures is demonstrated with Michael and Scott's (M&S) lock-free queue algorithm [13]. 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 L25 and L54 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 `Recycle`, 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 L21-L22 as well as L48-49 is critical: the pointer load operation must be sandwiched between two tag load operations to detect any changes. L25 and L54, 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 [6] 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 allowed *lazy traversals*, i.e., logically deleted nodes are simply jumped over by the read-only search operation. However, this presents a serious obstacle when building RRR (recyclable) lists that we have previously discussed. At first sight, this is completely 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 over 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 [12] altered 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).

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

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

```
   // Recycle a node: was already previously used
34 node_t * Recycle(node_t * node, void * data)
35   node->Data = data;
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   void * data;
42   node_t * node = Deq(srcQ, &data);
43   if ( node != null )
44     Recycle(node, data);
45     Enq(dstQ, node);
46 node_t *Deq(q_t *q, void **data)
47   while true do
48     h.Tag = q->Head.Tag;
49     h.Ptr = q->Head.Ptr;
50     t.Tag = q->Tail.Tag;
51     t.Ptr = q->Tail.Ptr;
52     next.Tag = t.Ptr->Next.Tag;
53     next.Ptr = t.Ptr->Next.Ptr;
54     if ( h.Tag == q->Head.Tag )
55       if ( h.Ptr == t.Ptr )
56         if ( next.Ptr == null )
             // Queue is empty
57           return null;
58         new = {next.Ptr, t.Tag+1};
59         CAS(&q->Tail, t, new);
60       else
61         *data = next.Ptr->Data;
62         n = { next.Ptr, h.Tag+1 };
63         if ( CAS(&q.Head, h, n )
             // Delete the sentinel
64           return h.Ptr;
```

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

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 discussion to linked lists and queues. However, the technique presented in the paper is also directly suitable to hash tables [11], which is an array of linked lists, and skip lists [5], which, roughly speaking, represent multiple linked (sub)lists. Moreover, we used the same approach to implement Natarajan-Mittal tree [17]. In Section 3.2, we discuss two major changes to Natarajan-Mittal tree that are needed for its implementation with our method.

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    node=malloc(sizeof(node_t));
10   node->Next = null;
11   l->Head = node;
12 bool Add(list_t * l, key_t key)
13   new = malloc(sizeof(node_t));
14   new->Key = key;
15   node_t **prev, *curr, *next;
16   while true do
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 *curr, *next;
24   while true do
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 do
35     if ( curr == null ) break;
36     next = curr->Next;
37     if ( *prev != curr ) goto 32 ;
38     if ( !isMarked(next) )
39       if ( curr->Key ≥ 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

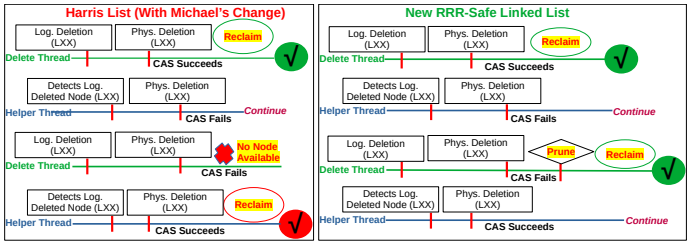
There are two fundamental issues to be resolved for the original Harris' linked list: (1) there should be an additional safety mechanism that validates that a node is not moved (for ABA safety as well as to avoid jumping over to a different list while traversing), and (2) there should be a mechanism 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 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.

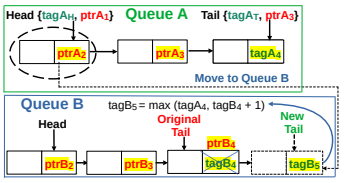
To address the **second problem**, we observe that the physical removal encompasses two separate tasks: (1) executing CAS to unlink the node; and (2) taking possession of the node for future memory

```
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;
   // An ABA-safe linked list
12 void InitList(list_t * l)
13   node = malloc(sizeof(node_t));
14   node->Next = { null, 0 };
15   l->Head = { node, 0 };
16 bool Do_Add(list_t *l, node_t *new, key_t key)
17   tagged<node_t> *prev;
18   tagged<node_t> curr, next;
19   while true do
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> curr, next;
30   while true do
31     if ( !Do_Find(l, key, &prev, &curr, &next) )
32       return null;
33     pair = { getMarked(next), next.Tag+1 };
34     if ( CAS(&curr->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   new = malloc(sizeof(node_t));
40   new->Key = key;
41   if ( !Do_Add(l, new) )
42     free(new);
43     return false;
44   return true;
45 bool Remove(list_t * l, key_t k)
46   node_t *node = Do_Rem(l, k);
47   if ( node != null ) smr_reclaim(node);
48   return node != null;
49 bool Contains(list_t * l, key_t k)
50   tagged<node_t> curr, next;
51   tagged<node_t> *prev;
52   return Do_Find(l, k, &prev, &curr, &next);
53 bool Do_Find(list_t *l, key_t k, 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 do
59     if ( curr.Ptr == null ) break;
60     next.Tag = curr->Next.Tag;
61     next.Ptr = curr->Next.Ptr;
62     curr_key = curr->Key;
63     if ( prev->Tag != curr.Tag ) goto 55 ;
64     if ( !isMarked(next.Ptr) )
65       if ( curr_key ≥ k )
66         *p_curr = curr;
67         *p_prev = prev;
68         *p_next = next;
69         return curr->Key == k;
70       prev = &curr.Ptr->Next;
71     else                                  // No reclamation here!
72       pair={getUnmarked(next), 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 do
85     if ( !curr.Ptr ) break;
86     next.Tag = curr->Next.Tag;
87     next.Ptr = curr->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), curr.Tag+1};
93       if ( !CAS(&prev, curr, pair ) goto 81 ;
94       if ( curr.Ptr == node ) break;
95     curr = next;
96 bool Move(list_t *dst, list_t *src, key_t k)
97   node_t *n = Do_Rem(src, k);
98   if ( n == null ) return null ;
99   if ( !Do_Add(dst, n) )
100    smr_reclaim(n);
101    return false;
102  return true;
```

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



(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.

reclamation. In our design, we, effectively, propose three phases: logical deletion, physical removal, 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.

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` will have to take the 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 the 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 introduced 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 the 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' list and the proposed list. For each list, there are two examples: when CAS succeeds (two lines on the 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 can call `Do_Prune`, after which the original thread can take posession of 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. The `Do_Prune` operation 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 (L38), which is fairly infrequent 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

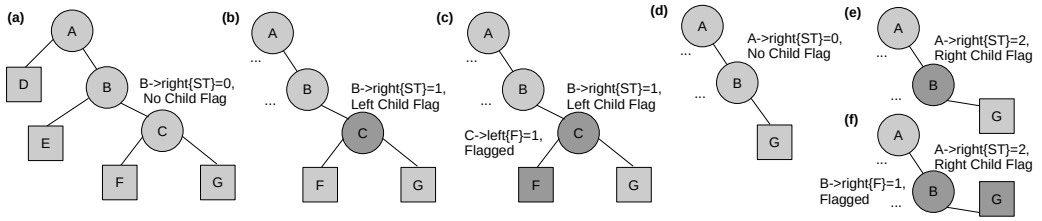


Fig. 5. Natarajan-Mittal tree change: two simultaneously deleted leaves (F, G) are being flagged separately.

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 [17], where ABA tags are attached to `left` and `right` pointers. Unlike linked lists, Natarajan-Mittal BST uses two (internal and leaf) nodes for the insertion operation and also frees up two (internal and leaf) nodes during removal. The move operation 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" (edges, sibling nodes) and "flagging" (leaf nodes), which are described in detail in [17]. Similar to Michael's modifications to linked lists, tagged edges should 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. Upon encountering a tagged edge, it is immediately unlinked from the corresponding parent by changing the link to the sibling of the flagged leaf node.

Similar to linked lists, there is a problem of taking possession of removed nodes. Removed nodes might not immediately be available due to being freed up by a concurrent helping thread (in the original algorithm). There are three cases that need to be considered in the delete operation. In two cases, the delete operation 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 delete operation [17], 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 delete 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 another thread.

In Figure 5, we present a modification to address this problem by splitting simultaneous flagging. In Figure 5(a), none of the nodes have been flagged yet, but we are in the process of removing F and G. In Figure 5(b), F is about to be flagged. The first step is to check if its sibling is already flagged. For that purpose, we add 2 additional bits in the preceding parent link to keep flagging state (ST). Since F is about to be flagged, ST is changed from 0 (nothing being flagged) to 1 (left child). Then, in Figure 5(c), F will be flagged. In the meantime, a concurrent thread in Figure 5(d) found that G cannot be flagged until the C-F edge is cleaned up. Only after that, the preceding grandparent link's ST is changed from 0 to 2 (Figure 5(e)) and G is flagged (Figure 5(f)).

[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 internal nor leaf node needs to be reallocated.

Due to the substantial complexity of Natarajan-Mittal BST and lack of space, we omit its pseudocode and evaluation. These will be included in an extended report after the paper is accepted.

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.[4] 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 [24], i.e., we can move nodes from queues to stacks and vice versa.

Figure 6 summarizes all changes. While for brand new nodes, the tag can be obtained by simply incrementing the preceding node's tag value (L48), this is insufficient in a more general case. 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 getting 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 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 modify and move nodes from one queue to another. To retrieve the last node's tag, we introduced the `GetTailTag` method, 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. Recall that tags are only preserved for the last node in the queue. Suppose there is just one node left in the queue. In that case, 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.*

[4]This is feasible as all pointers are word-aligned.

```
1  struct {
2    node_t * Next;
3  } node_t;
   // A new RRR queue
4  node_t *Deq(q_t *q, void **data)
5    while true do
6      h.Tag = q->Head.Tag;
7      h.Ptr = q->Head.Ptr;
8      t.Tag = q->Tail.Tag;
9      t.Ptr = q->Tail.Ptr;
10     next = t.Ptr->Next;
11     if ( h.Tag == q->Head.Tag )
12       if ( h.Ptr == t.Ptr )
13         if ( next:IsTag == 1 )
             // Queue is empty
14           return null;
15         new = { next:Value, t.Tag+1 };
16         CAS(&Q->Tail, t, new);
17       else
18         *data = next:Value->Data;
19         n = {next:Value, h.Tag+1};
20         if ( CAS(&Q.Head, h, n )
             // Delete sentinel
21           return h.Ptr;
   // Allocate a brand new node
22 node_t * AllocNode(void * data)
23   node = malloc(sizeof(node_t));
24   node->Data = data;
25   node->Next = { :IsTag=1, :Value=0 };
   // Recycle an existing node
26 void Recycle(q_t * srcQ, node_t * node, void * data)
27   node->Data = data;
28   node->Next = { :IsTag=1, :Value=GetTailTag(srcQ) };
```

```
29 uint GetTailTag(q_t * q)
30   while true do
31     t.Tag = q->Tail.Tag;
32     t.Ptr = q->Tail.Ptr;
33     next = t.Ptr->Next;
34     if ( t.Tag == q->Tail.Tag )
35       if ( next:IsTag == 1 )
36         return next;
37       else
38         CAS(&q->Tail, t, {next:Value, t.Tag+1});
39   CAS(&Q->Tail, t, { node, t.Tag+1 });
40 void Enq(q_t *q, node_t *node)
41   while true do
42     t.Tag = q->Tail.Tag;
43     t.Ptr = q->Tail.Ptr;
44     next = t.Ptr->Next;
45     if ( t.Tag == q->Tail.Tag )
46       if ( next:IsTag == 1 )
47         if ( node->next:Value < next:Value )
48           node->Next:Value = next:Value + 1;
49         if ( CAS(&t.Ptr->next, next, { :IsTag=0,
             :Value=node }) ) break;
50       else
51         CAS(&q->Tail, t, {next:Value, t.Tag+1});
52   CAS(&q->Tail, t, { node, t.Tag+1 });
53 void Move(q_t *dstQ, q_t *srcQ)
54   void * data;
55   node = Deq(srcQ, &data);
56   if ( node != null )
57     Recycle(node, data);
58     Enq(dstQ, node);
```

Fig. 6. A new RRR queue: . separates words, : separates bits in the *same* word.

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 O_i (enqueue or dequeue) appears as an instantaneous, atomic event on a linear timeline T, in which each event conforms to the sequential specification of a queue.*

Proof. The proposed queue introduces a unique tagging function $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, defined by $f(\text{src\_tag}, \text{dst\_tag}) = \max(\text{src\_tag}+1, \text{dst\_tag}+1)$ 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 $t_i > t_{i-1}$ for every prior instance of 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 shared across queues, the tag $t_{\text{new}} = \max(t_{\text{last\_src}} + 1, t_{\text{prev\_dst}} + 1)$ ensures that no node N_i reappears with an identical tag in different queues. This invariant allows each enqueue and dequeue operation to be uniquely identified in the linear timeline T. Thus, each operation is placed unambiguously on T, ensuring all queue operations are atomic and sequentially consistent. □

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

Proof. `Do_Prune` is only called when the physical deletion fails by the original (removal) thread. The method searches the node that was previously logically deleted. However, no other thread could have taken the possession of this node. Thus, the only possibility 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 L_i (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. For each deletion operation $D(x)$ of node N_i, a two-phase deletion process—logical deletion $L_D(x)$ followed by physical removal $P_D(x)$—ensures the node is removed consistently without interfering with ongoing operations. Specifically, $D(x)$ appears atomic at a point between $L_D(x)$ and $P_D(x)$, guaranteeing a single instance in T. Using a prune function Do_Prune(N_i), the original deleting thread T_{orig} 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 $F(x)$ to detect if a node was altered by comparing t_i from N_{i-1} before moving to N_{i+1}. This condition $t_{i-1} < t_i$ enforces consistent traversal order across concurrent threads, preventing nodes from inadvertently crossing list boundaries. Therefore, each list operation L_i is mapped to an atomic instant on timeline T, maintaining both sequential and ABA-safe properties for a linearizable linked list. □

5 Evaluation

Aside from qualitative characteristics (ABA-safety, recyclability), the described memory management model has potential benefits in terms of performance and memory efficiency. We evaluate two data structures presented in this paper: a queue and a linked list. We have used hazard pointers (HP) [12] and epoch-based reclamation (EBR) [5, 7], 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. So we integrated both into our algorithm to ensure the robustness of our algorithm under different memory management models. A comparison of our results against both SMR schemes shows the flexibility of our algorithm and provides performance guarantees.

We based our benchmark partially on the previously published code [20] and implemented the following schemes:

- **Queue-HP:** Michael & Scott Queue for HP (not RRR safe).
- **Queue-ABA-HP:** Michael & Scott Queue for HP (RRR and ABA safe).

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

All ABA versions are not merely ABA safe, but 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. The system is equipped with 384 GiB of RAM. To ensure more consistent behavior, simultaneous multithreading (SMT) is disabled. 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** [15] 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.)

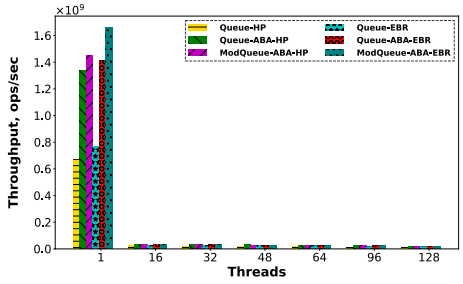
Since the differences may exist due to SMR schemes, we evaluated queue and list algorithms with both EBR and HP, and observed no significant differences in overall trends. For EBR, we used a more recent version with unconditional epoch increments [25].

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 (64 bytes) as nodes need to be cache-aligned for performance. Let us say, we need to store 112 bytes (rounded to 2 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.

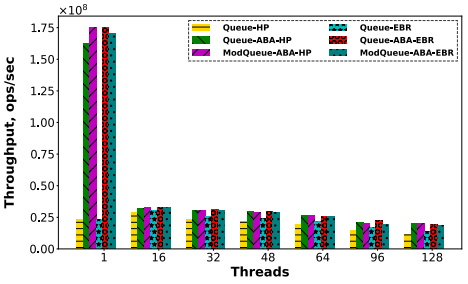
We chose a large payload size of **64 KiB** for both queues and lists as it is common for disk I/O operations and also not that far off from the maximum transmission unit (MTU) jumbo frame size for high-end (10GbE+) network adapters. We chose a small payload size of 8 bytes for queues to represent a minimalist scenario where only a data 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 because linked lists typically keep keys of varying data types (strings, integers, etc.) and sizes. Additionally, whenever feasible, list nodes can be aligned to 128 bytes on x86-64 to avoid false sharing.

For queues, we use **pairwise** enqueue-dequeue operations (one dequeue follows one enqueue in every thread) and also **random** 50% enqueue - 50% dequeue operations. For linked lists, we implement a pair of deletion and insertion operations in every thread. We consider **two cases**: (1) 50% times recycling and 50% deletion/insertion, and (2) 90% times recycling and 10% deletion/insertion. 50%/90% recycling refers to *directly* moving objects from one list or queue to another 50% or 90% of the time, 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.

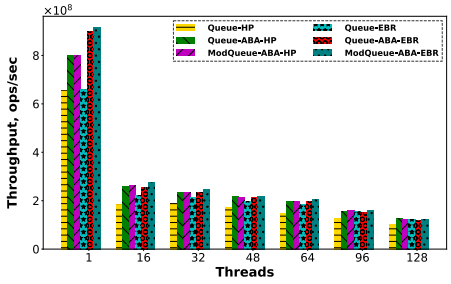
We demonstrate both throughput and memory consumption (waste). Memory consumption is measured via the number of not-yet reclaimed nodes. These are the nodes which are in limbo state due to the use of SMR (To calculate memory waste, we capture snapshots of the average number of



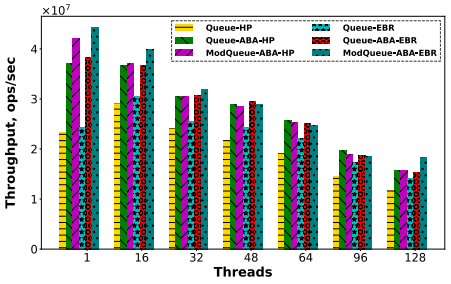
(a) No Payload, 90% recycle



(b) 64KiB Payload, 90% recycle

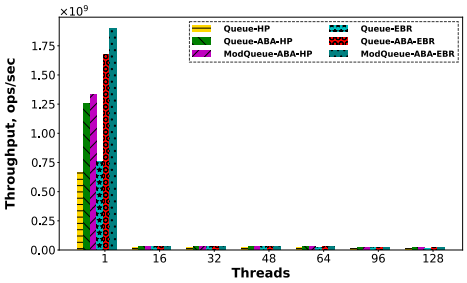


(c) No Payload, 50% recycle

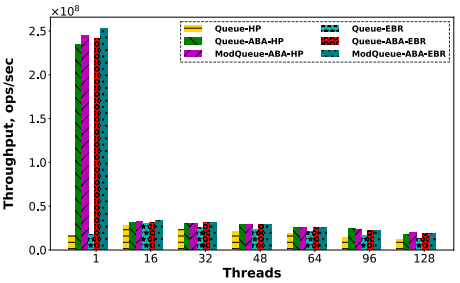


(d) 64KiB Payload, 50% recycle

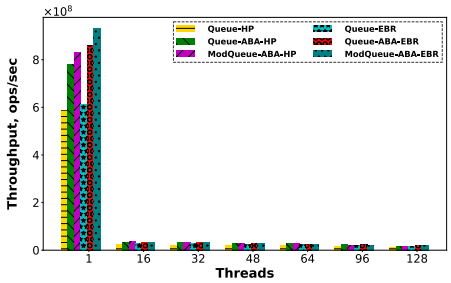
Fig. 7. Queue [Pairwise Operations], Throughput: *higher is better.*



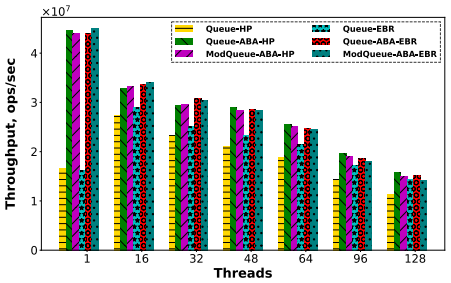
(a) No Payload, 90% recycle



(b) 64KiB Payload, 90% recycle

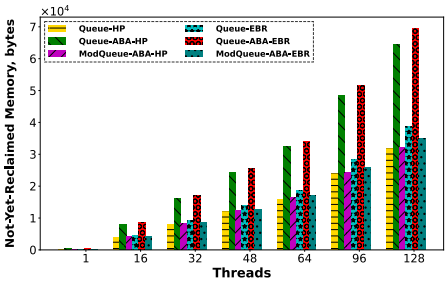


(c) No Payload, 50% recycle

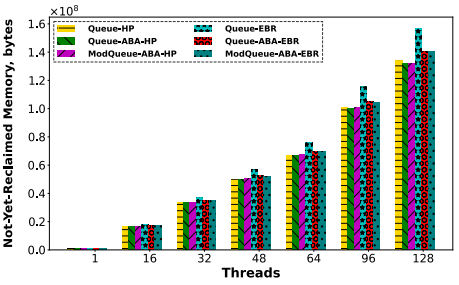


(d) 64KiB Payload, 50% recycle

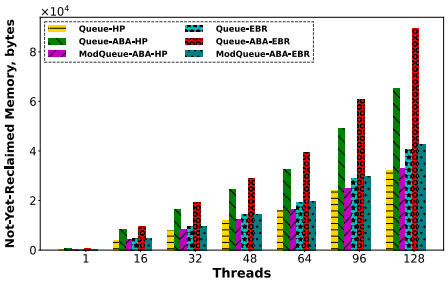
Fig. 8. Queue [Random Operations], Throughput: *higher is better.*



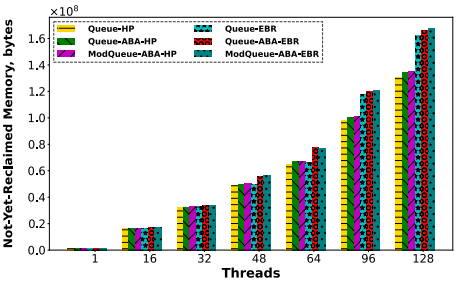
(a) No Payload, 90% recycle



(b) 64KiB Payload, 90% recycle

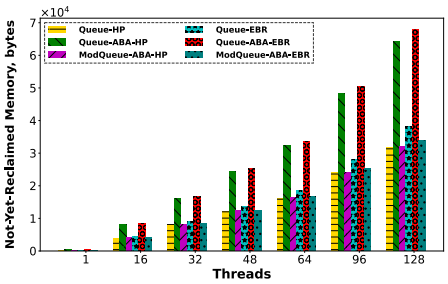


(c) No Payload, 50% recycle

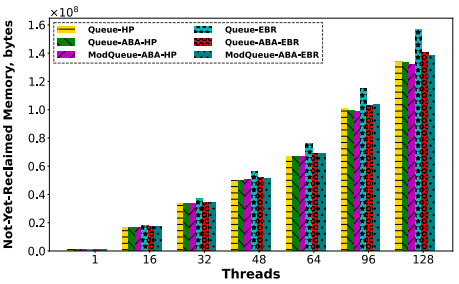


(d) 64KiB Payload, 50% recycle

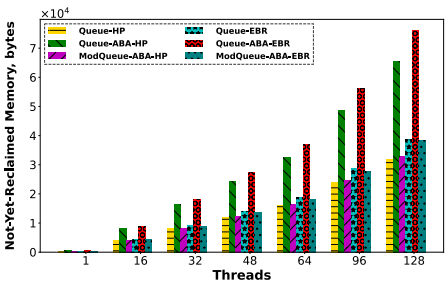
Fig. 9. Queue [Pairwise Operations], Average Memory Consumption (waste): *lower is better*.



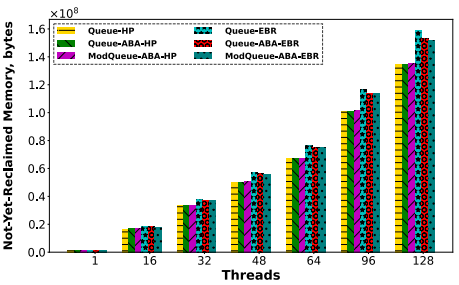
(a) No Payload, 90% recycle



(b) 64KiB Payload, 90% recycle



(c) No Payload, 50% recycle



(d) 64KiB Payload, 50% recycle

Fig. 10. Queue [Random Operations], Average Memory Consumption (waste): *lower is better*.

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 tags and pointers. This measurement ensures that any additional overhead brought in by our tagging scheme is fully included in the results. Although each node's size increases slightly due to tags and pointers, our method decreases the number of nodes overall held in the limbo state, thus lowering overall memory consumption.

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

For queues, we run the benchmark with 128 pre-filled elements, conducting 5 runs each lasting 20 seconds. Figure 7 and Figure 8 illustrate that our queue algorithm outperforms the original M&S (ABA-safe) algorithm in terms of throughput while operating under both 50% and 90% recycling scenarios across most thread counts for both Pairwise and Random operations. Figure 9 and Figure 10 show that even when the payload is non-existent, the new algorithm is twice as more memory efficient as M&S queue due to smaller nodes. Moreover, the new queue achieves the same memory efficiency as ABA-unsafe M&S queue while getting the performance benefits of ABA-safe M&S queue. Therefore, our modified version of the M&S queue with both HP and EBR consistently outperforms the original M&S queue.

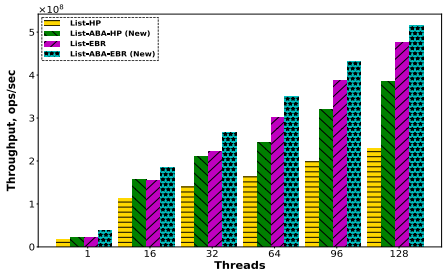
For some thread counts, our new queue has a higher throughput than the original M&S ABA-safe queue due to the use of single-width rather double-width CAS. On the other hand, under some scenarios, the throughput might also slightly drop due to the longer CAS loop which calculates a new tag in the last 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 [24], which do not use tags for the `next` pointers. Moreover, our proposed queue is substantially more memory efficient since tags are only needed for the head and tail pointers.

For linked lists, we perform 5 runs with 1024 pre-filled elements, each running for 60 seconds. Therefore, 1024 is the maximum key for the list. As illustrated in Figure 11, our ABA-safe linked list solution demonstrates highly competitive throughput compared to existing Harris' solutions for both 128B and 64KiB payloads, under both 50% and 90% recycle scenarios. Our solution shows a substantial performance boost (up to 4x) for 64KiB payloads. For 128-byte payloads, the gain is more modest. However, Figure 12 shows significant memory savings (up to 6x) for 128-byte payloads, which are due to more memory being recycled directly and not held in the limbo state. Albeit smaller, memory savings also exist for 64KiB due to less memory being held in the limbo state.

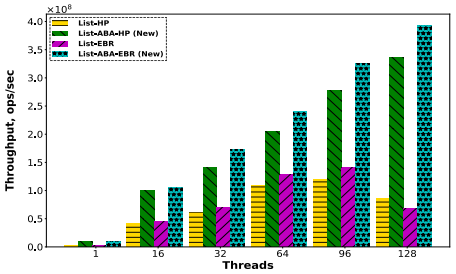
6 Related Work

Memory management of concurrent data structures, both in general and specifically lock/wait-free ones, has been studied for over two decades. In this section, we will focus on some of the related work that represents key ideas in the field.

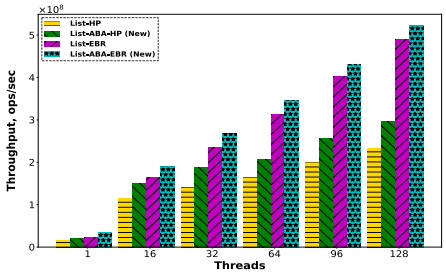
There is a cornucopia of both manual [5, 7, 18, 20, 22, 23, 26] and automatic schemes [3, 4]. In this work, without loss of generality, we considered manual reclamation schemes, which are more suitable for languages such as C and C++. As discussed in [12], the ABA problem is orthogonal to the underlying memory reclamation method and can still happen with garbage collectors [8] 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 memory reclamation and ABA problems.



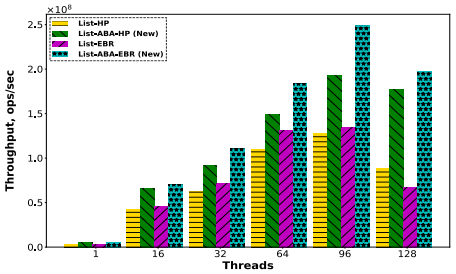
(a) 128B Payload, 90% recycle



(b) 64KiB Payload, 90% recycle

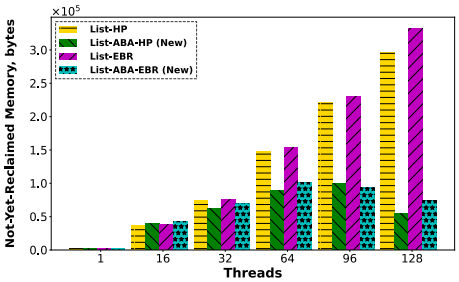


(c) 128B Payload, 50% recycle

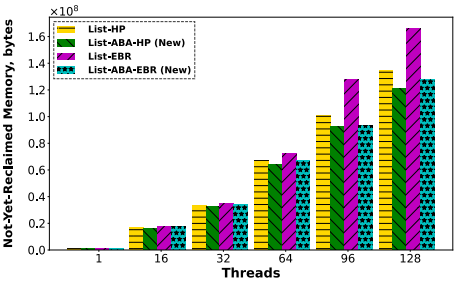


(d) 64KiB Payload, 50% recycle

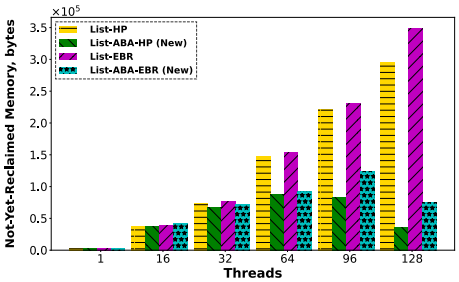
Fig. 11. Linked List Throughput: *higher is better.*



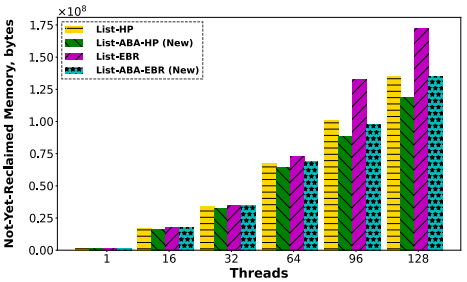
(a) 128B payload, 90% recycle



(b) 64KiB payload, 90% recycle



(c) 128B payload, 50% recycle



(d) 64KiB payload, 50% recycle

Fig. 12. Linked List Average Memory Consumption (waste): *lower is better.*

Most manual schemes can be categorized as epoch- or pointer-based reclamation schemes. In *epoch-based* reclamation (EBR) [5, 7], 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, 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) [12], 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 [24] 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 [13] 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 [6] 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 [12] 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 skip list is a probabilistic data structure known for its efficient search, insertion, and deletion capabilities. It normally operates with logarithmic complexity on average. A lock-free implementation of skip lists [5, 8], 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 [11]. Thus, the RRR-safe solution solves the corresponding problem for hash tables.

7 Conclusion

We introduced RRR (memory-recyclable) versions of the lock-free queue and linked list. We also briefly discussed an RRR Natarajan 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 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 rather than just one node) was implemented with Natarajan-Mittal Tree, where the same practical benefits are observed. Due to space limits and complexity of Natarajan-Mittal Tree, we omitted the full discussion and evaluation of RRR Natarajan-Mittal Tree. These will be included into an extended technical report after the paper is accepted.

Our full implementation will be released as open source after the paper is accepted.

References

[1] ARM. 2022. ARM Architecture Reference Manual. http://developer.arm.com/.

[2] 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. https://doi.org/10.1145/3503221.3508433

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

[4] 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.

[5] 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

[6] 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.

[7] 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. https://doi.org/10.1016/j.jpdc.2007.04.010

[8] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

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

[10] Intel. 2022. Intel 64 and IA-32 Architectures Developer's Manual. http://www.intel.com/.

[11] 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. https://doi.org/10.1145/564870.564881

[12] 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.

[13] 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. https://doi.org/10.1145/248052.248106

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

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

[16] MIPS. 2022. MIPS32/MIPS64 Rev. 6.06. http://www.mips.com/products/architectures/.

[17] 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. https://doi.org/10.1145/2555243.2555256

[18] 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. https://doi.org/10.1145/3453483.3454090

[19] 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. https:

//doi.org/10.1145/3381052.3381316

[20] 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. https://doi.org/10.1145/3087556.3087588

[21] RISC-V Foundation. 2022. RISC-V Books. http://riscv.org/risc-v-books/.

[22] 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. https://doi.org/10.1145/3409964.3461817

[23] 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. https://doi.org/10.1145/3437801.3441625

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

[25] 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. https://doi.org/10.1145/3178487.3178488

[26] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. 2018. Interval-based memory reclamation. *SIGPLAN Not.* 53, 1 (feb 2018), 1–13. https://doi.org/10.1145/3200691.3178488