Brief Announcement: SCOT: Fix non-blocking data structures, not memory reclamation

Md Amit Hasan Arovi
The Pennsylvania State University
University Park, PA, USA
arovi@psu.edu

Ruslan Nikolaev
The Pennsylvania State University
University Park, PA, USA
rnikola@psu.edu

Abstract

We present Safe Concurrent Optimistic Traversals (SCOT), to address a well-known problem related to optimistic traversals with both classical and more recent memory reclamation schemes, such as Hazard Pointers (HP), Hazard Eras (HE), Interval-Based Reclamation (IBR), and Hyaline. For these schemes, unlike for Epoch-Based Reclamation (EBR), existing data structure implementations are either buggy (e.g., Natarajan-Mittal tree) or come with performance trade-offs (e.g., Harris-Michael modified list).

Unlike past approaches, our method keeps the memory reclamation scheme intact but requires data structure adaptations. SCOT enables the first correct implementations of Harris' list and Natarajan-Mittal tree with optimistic traversals for HP, HE, IBR, and Hyaline. Our evaluation shows that our technique enables high throughput, comparable to EBR, especially when used with IBR and Hyaline.

CCS Concepts

• **Theory of computation → Concurrent algorithms**.

Keywords

hazard pointers, non-blocking, Harris list, Natarajan-Mittal tree

ACM Reference Format:
Md Amit Hasan Arovi and Ruslan Nikolaev. 2025. Brief Announcement: SCOT: Fix non-blocking data structures, not memory reclamation. In *37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25), July 28–August 1, 2025, Portland, OR, USA*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3694906.3743348

1 Introduction

When using manual memory management in non-blocking data structures, memory has to be safely reclaimed by freeing memory only when *all* ongoing operations with stale pointers complete. One fast and easy-to-use *safe memory reclamation* (SMR) scheme is Epoch-Based Reclamation (EBR) [3]. Unfortunately, EBR has a serious drawback that any stalled thread results in unbounded memory usage, i.e., it lacks *robustness*.

A number of researchers proposed various robust SMR schemes in C, C++, or Rust. However, it was proven [14] that they always make compromises, e.g., in terms of ease of integration or compatibility with data structures (*wide applicability*). Among well-known robust schemes is Hazard Pointers (HP) [7]. Prior to this work, data structures such as Harris' linked list [4] with optimistic traversals and Natarajan-Mittal binary search tree [9] were *not known* to work[1] with HP [1, 5]. Likewise, hash tables and skip lists [3] had similar issues. A number of recent schemes, including Interval-Based Reclamation (IBR) [17], Hyaline-1S [11], and Hazard Eras (HE) [13] are often faster but have issues similar to those of HP.

HP++ [5], a recent SMR scheme, is slower than HP and also makes some compromises with respect to wide applicability. However, HP++ still demonstrates support for the above two data structures. We are inspired by HP++'s success with these data structures, which is indicative that **many existing data structures can still be implemented, with an additional effort**, despite prior beliefs regarding limited applicability of the SMR schemes in this category.

This paper departs from the typical strategy of designing a "silver-bullet" SMR approach. Instead, we focus on fixing problematic data structures using existing SMR techniques. In Section 4, we evaluate data structures with optimistic traversals using HP, HE, IBR, and Hyaline-1S schemes. We also compare results to EBR and show that performance benefits can be preserved in the same manner.

2 Background

Harris' List. In the algorithm, a thread that removes a node first marks it as "logically" deleted by updating its next pointer. Subsequently, the node is "physically" unlinked from the list. Harris' list supports *optimistic traversals*, i.e., the search operation bypasses logically deleted nodes without their immediate physical removal.

This approach presents fundamental challenges for HP, as we show in Figure 1. Suppose Thread 3 (not shown) previously marked N2 for deletion but has not yet unlinked it. Then, Thread 2 marks N3 for deletion and unlinks the entire chain of nodes between N1 and N4 (assuming all *consecutive* nodes in the chain are logically deleted). Meanwhile, Thread 1 is traversing the list; it reaches N2 before Thread 2 physically removes the chain. Although N2 is now *retired* by Thread 2, it still remains in the HP limbo list. This is because Thread 1 has made a reservation *prior* to N2's physical deletion. N3 is also retired by Thread 2, but because Thread 1 never reserves N3, it is not guaranteed to stay in the HP limbo list, i.e., N3 is returned to the OS. Thus, Thread 1 fails when accessing N3.

To address this problem, Michael's approach [6] ensures that whenever a thread encounters a node marked as logically deleted

SPAA '25, Portland, OR, USA

ACM ISBN 979-8-4007-1258-6/25/07
https://doi.org/10.1145/3694906.3743348

[1] Despite presenting Natarajan-Mittal tree results in [11, 17] for HP and other SMR schemes, it was later argued [1] that such implementations are buggy and leak memory. Moreover, [1] pointed out two bugs, and only one bug was resolved in [1]. Prior to our work, the second bug related to optimistic traversals remained unresolved.

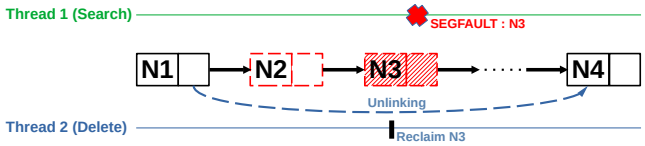

Figure 1: Unsafe HP traversals: accessing 1st logically deleted node (N2) is safe but subsequent nodes may cause SEGFAULT.

during traversal, it *immediately* attempts to physically remove the node from the list. Unfortunately, Michael's approach increases the number of CAS operations and may lead to a higher contention among threads. It also makes read-only traversals impossible.

Why does Michael's approach work? Recall that accessing N2 is still safe because its HP reservation is made prior to N2's physical deletion. (To reserve *successfully*, HP confirms that the pointer from N1 to N2 remains intact.) Once N2 is physically deleted, it is retired and ends up in the HP limbo list. If we now attempt to reserve N3, HP falsely succeeds as *the pointer from N2 to N3 remains intact unlike the pointer from N1 to N2*. Thus, Michael's approach ensures that pointers always change during physical removal, which happens automatically when we remove one node at a time.[2]

Natarajan-Mittal Tree. In the tree, every node keeps `left` and `right` pointers. The actual keys and values are stored in leaf nodes, while internal nodes are simply guiding in which direction (left or right) the underlying search operation has to descend to the next level. A concept similar to logical deletion also exists in this tree. There are two types of marking: "flagging" when marking the leaf node for its logical deletion followed by "tagging" of its sibling node.

When performing the underlying search operation, we keep the `successor` node, which is the last untagged node, as well as its preceding node – `ancestor`. We also keep the immediate `parent` of the `leaf` node. The algorithm makes a very crucial observation: a chain of consecutively tagged edges can be eliminated with one CAS operation by updating `ancestor`'s link from `successor` to the remaining `leaf` node. This idea is very similar to that of removing a chain of logically deleted nodes in Harris' linked list.

3 Algorithm Description

The crux of the problem with the HP implementation is its inability to properly track physically deleted nodes. In Figure 1, N2 is still *safe* to access because it is not physically deleted, i.e., N1 still points to N2. Moreover, according to Harris' list, we always make sure that the node to the left is not logically deleted and thus **never unlink nodes in the middle of the chain**, i.e., we can delete just N2, or both N2-N3, but we never delete N3 while keeping N2 in the list. In other words, we either remove the entire chain of logically deleted nodes or a subchain with the truncated tail.

3.1 Solution

Combining the observations above, we make one crucial insight: it is still safe to access N3 and the following nodes as long as at every step, i.e., after making an HP reservation for the next logically deleted node, we verify that N1 still points to N2. Once we reach N2, we declare it to be a "dangerous zone". Until we reach the end of the chain of consecutive logically deleted nodes, we are going to stay in the dangerous zone and perform additional checks. More specifically, after retrieving N3's pointer and creating its HP reservation, we check that N1 still points to N2. We will continue to perform these checks at every iteration until we reach the end of the chain. If the check fails, we cannot proceed further and need to restart the search operation from the very beginning. (See Figure 3.)

In Figure 2, we present our approach for Harris' list. Compared to Michael's approach, we need an additional hazard pointer index (Hp3) which protects the first *unsafe*, i.e., logically deleted, node. This extra hazard pointer prevents the ABA problem, i.e., a case when the unsafe node gets recycled while we are traversing the list. By holding this extra hazard pointer, we can solely rely on pointer comparison. In other words, Hp3 protects N2 from our example above irrespective of where we currently are in the chain of logically deleted nodes. Like in Michael's approach, we need Hp2, which protects the `prev` area. We note that in Harris' list, it will be the last *safe*, i.e., not logically deleted, node.

The HP reclamation provides a special `protect` function to safely retrieve an object and make the corresponding reservation. Additionally, we implement `dup` to duplicate the existing hazard pointer when moving to the next iteration. It is crucial to duplicate hazard pointers such that the old HP index has a lower numerical value than the new HP index (e.g., Hp0 to Hp1). This allows to avoid a small race window when iterating in the same (ascending) order of indices in `retire`. (An alternative to `dup` would be index renaming via an indirection array. From our empirical observations, we found duplication to be generally cheaper.)

One tricky part is to avoid a premature duplication into Hp3, which would induce a memory barrier not present in Harris-Michael approach, potentially making the algorithm more costly. We achieve that by unrolling the loop in the original Harris' algorithm and splitting it into two phases: (1) *iteration through the safe zone* and (2) *iteration through the dangerous zone*. Phase 1 only duplicates (shifts) `prev` and `curr` hazard pointers from `curr` and `next`, correspondingly. This is analogous to Harris-Michael algorithm. Upon leaving Phase 1, `curr` is duplicated (Hp3) in L52. After going to Phase 2, we no longer duplicate `curr` into `prev` (Hp2), which gives us an extra benefit compared to Harris-Michael approach as the number of memory barriers through the dangerous zone is reduced.

Once we get to the dangerous zone of logically deleted nodes, Figure 2 runs a check in L58 which will ensure that the reservation made in L57 is correct. Once the chain (or its subchain) gets unlinked, L58 fails and we restart the operation from the very beginning. We highlight in the green color all major changes related to SCOT in HP. We use pink to highlight the dangerous zone traversal.

Recovery Optimization. For simplicity, we have stated that when validation fails, we go back to the beginning of the list. While this is acceptable, this validation check is not always critical: the last safe node may simply point to another node now (e.g., when a new node was inserted, or the chain of logically deleted nodes was already eliminated by a concurrent thread). In such cases, we can simply escape from the dangerous zone and continue traversal from the

[2] Another way to think about Michael's approach is that if we remove nodes one by one, N3 takes place of N2, i.e., right after N1. N3 will be the first logically deleted node in the chain when making the HP reservation, which is still safe to access.

```
1  struct {
2    node_t * Next;        // Next node
3    key_t Key;            // Any key type
4  } node_t;
5  node_t * Head;    // Head of the list
6
7  void Init()
8    Head = malloc(sizeof(node_t));
9    Head->Next = nullptr;
10   Head->Key = ∞;
11 bool Insert(key_t key)
12   node_t **prev, *curr, *next, *new;
13   new = malloc(sizeof(node_t));
14   new->Key = key;
15   while true do
16     if ( Do_Find(key, &prev, &curr,
         &next, false) )
17       free(new);
18       return false;
19     new->Next = curr;
20     if ( CAS(prev, curr, new) )
         return true;
21 bool Delete(key_t key)
22   node_t **prev, *curr, *next;
23   while true do
24     if ( !Do_Find(key, &prev, &curr,
         &next, false) ) return false ;
25     if ( !CAS(&curr->Next, next,
         getMarked(next)) ) continue;
26     if ( CAS(prev, curr, next) )
         smr_retire(curr);
27     return true;
28 bool Search(key_t key)
29   node_t **prev, *curr, *next;
30   return Do_Find(key, &prev, &curr,
       &next, true);
31
   // Hazard Pointer Indices
32 const int Hp0 = 0;    // Protects next
33 const int Hp1 = 1;    // Protects curr
34 const int Hp2 = 2; // Safe node (prev)
35 const int Hp3 = 3; // 1st unsafe node
```

```
36 bool Do_Find(key_t key,
     node_t ***p_prev, node_t **p_curr,
     node_t **p_next, bool is_search)
37   node_t **prev = &Head, *curr, *next;
38   node_t *prev_next = nullptr;
39   curr = hp.protect(Head, Hp1);
40   next = hp.protect(curr->Next, Hp0);
41   while true do
42     do
43       if ( curr && curr->Key ≥ key )
           goto 60 ;
44       prev_next = nullptr;
45       prev = &curr->Next;
46       hp.dup(Hp1,Hp2); // Hp2=Hp1:curr
47       curr = getUnmarked(next);
48       if ( curr == nullptr ) goto 68 ;
49       hp.dup(Hp0,Hp1); // Hp1=Hp0:next
50       next = hp.protect(curr->Next, Hp0);
51     while !isMarked(next);
     // Hp3=Hp1:curr, return a hazard ptr
52   prev_next = hp.dup(Hp1, Hp3);
     // Check last safe node (N1) still
        points to the first unsafe (N2) node
53   do
54     curr = getUnmarked(next);
55     if ( curr == nullptr ) goto 60 ;
56     hp.dup(Hp0,Hp1); // Hp1=Hp0:next
57     next = hp.protect(curr->Next, Hp0);
58     if ( *prev != prev_next ) goto 37;
59   while isMarked(next);
60 if ( !is_search && prev_next != nullptr
       && prev_next != curr )
61   if ( !CAS(&prev, prev_next, curr )
62     goto 37;
63   do  // Retire unlinked chain of nodes
64     node_t *node =
         getUnmarked(prev_next->Next);
65     smr_retire(prev_next);
66     prev_next = node;
67   while (prev_next != curr);
68 *p_curr = curr;
69 *p_prev = prev; *p_next = next;
70 return curr && (curr->Key == key);
```

Figure 2: SCOT for Harris' list using HP-style reclamation.

new node. We must, however, ensure that the last safe node is still not logically deleted. When it is deleted, the last safe node (`prev`) is in a dangerous zone itself (though still safe to access due to prior reservation). For different SMR schemes, we have different choices.

In HP and HE, reservations are precise: `protect` uses indices and cancels previous reservations. As keeping a predecessor of `prev` would induce extra hazard pointers and duplication (hence memory barriers), it makes sense to simply go back to the beginning of the list when `prev` gets logically deleted.

For IBR and Hyaline-1S, `protect` has a cumulative effect and does not cancel previous reservations. For these schemes, we implement a small on-stack ring buffer of `prev` pointers, which allows us to fall further back in the list when `prev` gets logically deleted. Empirically, we found that a ring buffer of size 8 is more or less optimal. Only when exhausting all 8 predecessors, do we go back to the beginning of the list. (See Figure 4.)

3.2 SCOT for Natarajan-Mittal Tree

For the tree, we allocate 5 hazard pointers to protect nodes in the underlying search procedure: `current`, `leaf`, `parent`, `successor`,

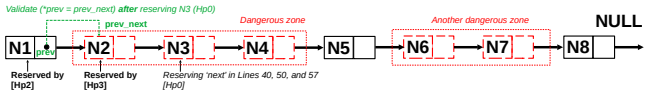

Figure 3: SCOT for Harris' list: validating *prev = prev_next at every iteration while traversing the dangerous zone.

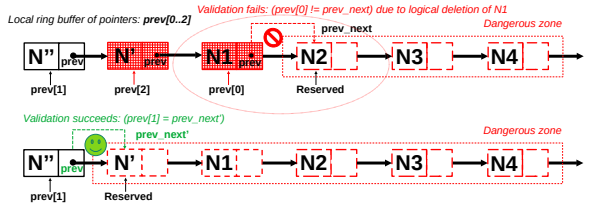

Figure 4: IBR and Hyaline-1S can go back from N1 to N''.

and `ancestor`. The `current` node points to the lowest node that is currently considered.

To properly support optimistic traversals, after each HP reservation of the `current` node, if the corresponding node is flagged or tagged, we need verify that ancestor still points to successor. If ancestor points to some other node or successor becomes tagged, we need to restart from the very beginning.

This idea is largely similar to the above-mentioned Harris' list approach, except that no additional hazard pointer is required, as we already need to protect both ancestor and successor nodes. Additionally, we need to check both *tagged* and *flagged* conditions for the current node, as (flagged) leaf nodes' hazard pointer reservations need to be also verified.

Unlike Harris' list, empirically, we found that *recovery* does not help improve performance for various key ranges, primarily because of the hierarchical structure of the tree (e.g., when there is a mismatch, likely, the tree has diverged substantially anyway). If ancestor no longer points to the original successor, we simply restart the search from the very beginning.

Due to complexity of the algorithm and very similar checks for optimistic traversals, we omit pseudocode in this paper. In Section 4, we evaluate Natarajan-Mittal Tree with SCOT.

3.3 SCOT for Hash Tables and Skip Lists

SCOT for these data structures is largely similar to Section 3.1. Hash tables are based on linked lists directly. Fraser's skip list [3] uses the same idea of logically deleted nodes as Harris' list.

3.4 Wait-Free Optimistic Traversals

With our approach, optimistic traversals may need to occasionally restart the search operation from the beginning if, due to overlapping modifications, the state of the data structure diverges significantly and cannot be recovered locally. This guarantees lock-free but not wait-free progress. This limitation is very similar to that of HP++'s existing solution which also does not support wait-free optimistic traversals due to potential restarts.

However, this restart is inherently unavoidable with data structures that use optimistic traversals: it is still needed for `Insert`

and `Delete`, regardless of the search operation or the reclamation scheme used. We note that we can still implement wait-free search traversals by using Timnat-Petrank's method [15, 16] which allows restarts. This method can be applied to various data structures, including linked lists. Timnat-Petrank's original wait-free linked list, implemented in HP, is similar to Harris-Michael list. With SCOT, it can now use Harris' optimistic traversals for faster performance.

4 Evaluation

Our benchmark reuses and substantially extends the test harness from [13]. We evaluate Harris-Michael List (HMList), Harris' List (HList), and Natarajan-Mittal Tree (NMTree) with EBR, HP, HE, IBR, and Hyaline-1S (HLN) by carefully considering all well-known optimizations. We also implemented extra optimizations for HP, HE, and IBR, which allow to greatly reduce overheads due to cache misses by capturing local snapshots of shared data in *retire* [11]. We did not implement HP++ [5] due to its substantial API differences (i.e., lack of an easy integration). HP++ is already known to perform mostly worse than HP for the *same* data structure [5], i.e., HP++ shows some performance benefits only when comparing different data structures (Harris-Michael HP list vs. Harris' HP++ list). We also did not evaluate WFE [10] or Crystalline [12], but their trends should be largely similar to those of HE and Hyaline-1S, respectively. Finally, we present the No Reclamation (NR) baseline, which simply leaks memory and demonstrates the "upper bound" for performance. For NMTree, everyone uses SCOT except NR/EBR. These algorithms were previously infeasible for HP/IBR/HE/Hyaline-1S or contained serious bugs in the available code, as discussed in [1, 5].

We calibrated all memory reclamation schemes to provide the highest possible throughput while also minimizing the number of not-yet-reclaimed objects. We found that amortizing *retire* list scans with frequency of 128 works well for EBR, HP, HE, IBR, and Hyaline-1S. Additionally, for EBR, IBR, HE, and Hyaline-1S we select the epoch increment frequency which corresponds to the number of threads multiplied by 12.

We run our experiments on Ubuntu 22.04.5 LTS using an AMD EPYC 9754 system with 128 physical cores (256 hardware threads w/ hyperthreading enabled), 384 GiB of RAM, and a maximum clock speed of 3.10 GHz. The benchmark is written in C++ and compiled using Clang++ 14.0.0 with -O3 optimizations because its compiled code tends to perform slightly better than the code generated by Ubuntu's (default) GCC 11.4.0. For memory allocation, we use **Microsoft's `mimalloc`** [8] since it scales much better in multi-threaded code compared to glibc's stock malloc.

We present complete results, including memory usage and recovery optimization impact (which improves performance by up to 25%), in [2] for various key ranges and workloads. In this section, we focus on throughput for mixed reads and writes (50% read – 50% write) using *optimized* SCOT. Each benchmark begins with prefilling the data structure with unique keys using 50% of the specified key range. We conduct 5 runs, each lasting 10 seconds, and report the median. We evaluate up to 256 threads and also show oversubscription (384 threads) when results differ from 256 threads.

In Figure 5, HList consistently outperforms HMList. Generally, IBR and Hyaline-1S exhibit excellent performance on HList with SCOT traversal, matching or approaching that of EBR. HE and HP

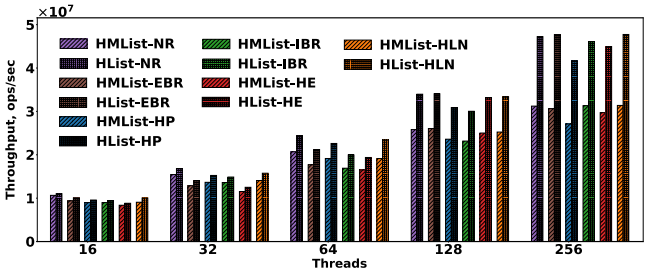

Figure 5: Linked List Throughput (Key Range = 512).

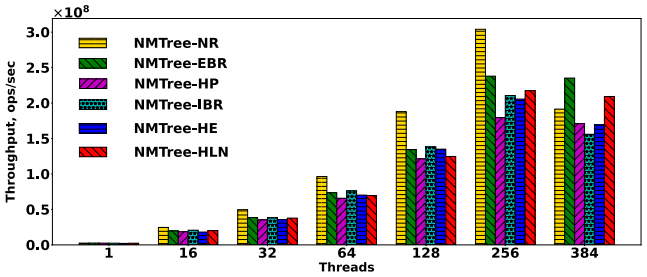

Figure 6: NM Tree Throughput (Key Range = 100,000).

also show the relative benefits of SCOT traversal on HList, though their overall performance is somewhat lower compared to EBR, IBR, and Hyaline-1S, as expected. When comparing with previously reported HP++'s results, we observe that our approach with HP consistently outperforms HP++.

Figure 6 highlights the scalability of the tree at higher key ranges, where the NMTree achieves up to **240 million operations per second** at 256 threads using the EBR scheme. Hyaline-1S also performs competitively, reaching **210 million operations per second**, making it the closest in performance to EBR in this configuration. When comparing our results with HP++'s reported results, we observe that our implementation is roughly 4x faster. Moreover, for the key range of 100,000, the reported HP++ vs. EBR gap was around 30%, whereas our HP vs. EBR gap is less than 30% and our IBR/HE/HLN vs. EBR gap is less than 10%.

5 Conclusion

We introduced safe concurrent optimistic traversals (SCOT), a technique which enables support for data structures that are incompatible with HP, IBR, HE, Hyaline-1S, and similar SMRs that lack the wide applicability support. Despite prior beliefs [5] of incompatibility of Natarajan-Mittal Tree and Harris' List with HP, we have demonstrated that not only they are feasible with HP, IBR, HE, and Hyaline-1S, but they also can largely retain performance benefits when comparing to equivalent data structures without optimistic traversals. We hope that our work will help to spur further research and reevaluate the existing challenges related to SMR because lack of the wide applicability property can be addressed by other means.

We thank the anonymous reviewers for their invaluable feedback. Complete details of the algorithm, analysis, and evaluation are available in [2]. SCOT's code and benchmark for all evaluated data structures are available at https://github.com/rusnikola/scot.

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

[3] 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

[4] 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. doi:10.1007/3-540-45414-4_21

[5] 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

[6] 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

[7] 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

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

[9] 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

[10] 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

[11] 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

[12] Ruslan Nikolaev and Binoy Ravindran. 2024. A Family of Fast and Memory Efficient Lock- and Wait-Free Reclamation. In *Proceedings of the 2024 ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Copenhagen, Denmark) *(PLDI '24)*. Association for Computing Machinery, New York, NY, USA, 235:1–235:25. doi:10.1145/3658851

[13] 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

[14] Gali Sheffi and Erez Petrank. 2023. The ERA Theorem for Safe Memory Reclamation. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing* (Orlando, FL, USA) *(PODC '23)*. ACM, New York, NY, USA, 102–112. doi:10.1145/3583668.3594564

[15] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. 2012. Wait-Free Linked-Lists. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 330–344. doi:10.1007/978-3-642-35476-2_23

[16] Shahar Timnat and Erez Petrank. 2014. A Practical Wait-Free Simulation for Lock-Free Data Structures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. ACM, New York, NY, USA, 357–368. https://doi.org/10.1145/2555243.2555261

[17] 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. doi:10.1145/3200691.3178488