Brief Announcement: Hyaline: Fast and Transparent Lock-Free Memory Reclamation

Ruslan Nikolaev
Virginia Tech, USA
rnikola@vt.edu

Binoy Ravindran
Virginia Tech, USA
binoy@vt.edu

ABSTRACT

We present a new lock-free safe memory reclamation algorithm, Hyaline, which is fast, scalable, and transparent to the underlying data structures. Hyaline easily handles virtually unbounded number of threads that can be created and deleted dynamically, while retaining $O(1)$ reclamation cost. We also extend Hyaline to avoid situations where stalled threads prevent others from reclaiming newly allocated objects, a common problem with epoch-based reclamation. Our evaluation reveals that Hyaline's throughput is high – it steadily outperformed other reclamation schemes by > 10% in one test and yielded even higher gains in oversubscribed scenarios.

CCS CONCEPTS

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

KEYWORDS

lock-free; memory reclamation; concurrent data structures

ACM Reference Format:
Ruslan Nikolaev and Binoy Ravindran. 2019. Brief Announcement: Hyaline: Fast and Transparent Lock-Free Memory Reclamation. In *2019 ACM Symposium on Principles of Distributed Computing (PODC '19), July 29–August 2, 2019, Toronto, ON, Canada.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3293611.3331575

1 INTRODUCTION

One of the most fundamental problems for lock-free data structures that use dynamic memory allocation is that memory blocks need to be safely deallocated. Safe memory reclamation (SMR) techniques are typically needed for unmanaged (C/C++) code. Many existing approaches for SMR originate from or improve upon epoch-based reclamation (EBR) [4, 5] and hazard pointers (HP) [6]. EBR provides a simple API but lacks protection against stalled threads that can prevent timely reclamation and lead to memory exhaustion. HP does not suffer from this problem, but is harder to use and slower in practice. (SMR schemes that defend against stalled threads are called *robust* [1, 10].) Furthermore, some algorithms [1, 2] rely on special OS abstractions, making it difficult to use them in certain cases, such as within OS kernels or platform-independent code.

We present Hyaline, a new algorithm that is not based on EBR or HP directly, has very low overhead, and scales well. Unlike most SMR algorithms, which typically reserve per-thread entries, Hyaline supports virtually unbounded number of threads using a small number of *slots*. Moreover, Hyaline bounds the cost of reclamation to $\approx O(1)$ per operation, irrespective of the total number of threads.

In oversubscribed scenarios, Hyaline particularly shines due to its unique asynchronous block tracking mechanism (we saw more than 30% gain over other algorithms). Threads do not need to periodically check if block(s) can be safely freed. Instead, tracking is reminiscent of reference counting, but Hyaline avoids the prohibitive cost of classical reference counting [9].

Hyaline is well suited for preemptive environments where the number of threads substantially exceeds the number of cores and can change dynamically such as in OS kernels and server applications. Unlike many other techniques, in Hyaline, threads that delete blocks (nodes) are not necessarily those that end up freeing them. This results in better *transparency*, as threads are "off the hook" as soon as they complete data structure operations. Unlike in EBR or HP, Hyaline's threads can immediately be recycled or destroyed without worrying about the fate of their previously deleted blocks.

2 OVERALL DESIGN

In Hyaline, programs explicitly *retire* objects and ensure that retired objects are not reachable from subsequent operations on the data structures. Each operation on the data structures must be enclosed between *enter* and *leave* calls. Hyaline's use of reference counters is triggered only when handling retired nodes.

We first present Hyaline's simpler version that manipulates just a single *retirement list*. Hyaline's key idea is that all threads participate in the tracking of retired nodes between *enter* and *leave* in this global list even if they are not actively retiring any nodes themselves. A special `Head` tuple is associated with this list. The `HPtr` field of this tuple is a pointer to the beginning of the list, and the `HRef` field counts the number of active threads. When each thread *enters*, it atomically increments the `HRef` field to indicate that a new thread has arrived. At the same time, the thread records a snapshot value of `HPtr` into a special per-thread `Handle` variable.

Each node for a data structure incorporates a special header which contains `Next` and `NRef` fields. `Next` is a pointer to the next node in the list, and `NRef` of every non-`Head` node counts threads that can still access this node. For the very first node, `HRef` itself serves this purpose. When retiring a new node, its `NRef` is set to 0. After appending the node, the current thread atomically adds the snapshot value of `HRef` to the `NRef` field of the predecessor node (`Head` no longer points to the predecessor node).

When a thread *leaves*, it decrements `HRef`. At the same time, it retrieves the `HPtr` pointer and traverses a sublist of nodes from `HPtr` to `Handle` that were retired since it *entered*. While traversing, the thread decrements `NRef` counters for every non-Head node.

PODC '19, July 29–August 2, 2019, Toronto, ON, Canada

ACM ISBN 978-1-4503-6217-7/19/07.
https://doi.org/10.1145/3293611.3331575

```
1  forall head_t Head ∈ Heads[k] do                                  // Initialization
2      Head.HRef = 0, Head.HPtr = Null;
3  handle_t enter(int slot)
4      Last = FAA(&Heads[slot], { .HRef=1, .HPtr=0 });
5      return Last.HPtr;                                             // Returns a handle
6  void leave(int slot, handle_t handle)
7      do                                                            // Decrement HRef and fetch Next
8          Head = Heads[slot];
9          Curr = Head.HPtr;
10         if ( Curr ≠ handle )
11             Next = Curr->Next;
12         New.HPtr = Curr;
13         if ( Head.HRef = 1 ) New.HPtr = Null;
14         New.HRef = Head.HRef - 1;
15     while not CAS(&Heads[slot], Head, New);
16     if ( Head.HRef = 1 and Curr ≠ Null )                          // Treat Curr as
17         adjust(Curr, Adjs);                                       // if it were a predecessor
18     if ( Curr ≠ handle )                                          // Non-empty list
19         traverse(Next, handle);
20 handle_t trim(int slot, handle_t handle)
21     Head = Heads[slot];                                           // Do not alter head
22     Curr = Head.HPtr;
23     if ( Curr ≠ handle )                                          // Non-empty list
24         traverse(Curr->Next, handle);
25     return Curr;                                                  // Returns a new handle
26 void retire(batch_t *batch)
27     doAdj = False, Empty = 0, Inserts = 0;
28     batch->NRefNode()->NRef = 0;
29     forall int slot ∈ 0..k-1 do
30         do                                                        // Add the batch to this slot
31             Head = Heads[slot];
32             if ( Head.HRef = 0 )                                  // REF #1#
33                 doAdj = True, Empty += Adjs;
34                 continue with the next slot;
35             New.HPtr = batch->NextNode();
36             New.HRef = Head.HRef;
37             New.HPtr->Next = Head.HPtr;
38         while not CAS(&Heads[slot], Head, New);
39         adjust(Head.HPtr, Adjs + Head.HRef);                      // REF #2#
40     if ( doAdj ) adjust(batch->FirstNode(), Empty);               // REF #3#
41 void adjust(node_t *node, int val)
42     Ref = node->NRefNode;
43     if ( FAA(&Ref->NRef, val) = -val ) free_batch(Ref->First);
44 void traverse(node_t *next, handle_t handle)
45     do                                                            // Traverse the retirement sublist
46         Curr = next;
47         if ( Curr = Null ) break;
48         next = Curr->Next;
49         Ref = Curr->NRefNode;
50         if ( FAA(&Ref->NRef, -1) = 1 ) free_batch(Ref->First);
51     while Curr ≠ handle;
```

Figure 1: Hyaline for double-width CAS.

```
1  handle_t enter(int slot)
2      Heads[slot] = { .HRef=1, .HPtr=Null };
3      return Null;                                                  // Returns a handle
   // Replace #2# in retire() with: Inserts++
4  void leave(int slot, handle_t handle)
5      Head = SWAP(&Heads[slot], { .HRef=0, .HPtr=Null });
6      if ( Head.HPtr ≠ Null ) traverse(Head.HPtr, handle);
   // Replace #3#: adjust(batch->FirstNode(), Inserts)
```

Figure 2: Hyaline-1 for single-width CAS.

```
1  int AllocEra = 0;                                                 // Initialization
2  thread int AllocCounter = 0;
3  forall int Access ∈ Accesses[k] do Access = 0;
4  forall signed int Ack ∈ Acks[k] do Ack = 0;
5  node_t *deref(int slot, node_t **ptr_node)
6      Access = Accesses[slot];
7      while True do
8          node_t * Node = (*ptr_node);
9          Alloc = AllocEra;
10         if ( Access = Alloc ) return Node;
11         Access = touch(slot, Alloc);
12 void retire(batch_t *batch)
       // Replace #1# in retire() with:
13     Access = Accesses[slot];
14     Min = batch->MinBirth();
15     if ( Head.HRef = 0 or Access < Min ) ...;
16     FAA(&Acks[slot], Head.HRef);                                  // Skip in Hyaline-1S
17 void init_node(node_t *node)
18     if ( AllocCounter++ mod Freq = 0 ) FAA(&AllocEra, 1);
19     node->BirthEra = AllocEra;                                    // Shares space with Next
20 int touch(int slot, int era)
21     do                                                            // Hyaline-1S: touch is ordinary memory write
22         Access = Accesses[slot];
23         if ( Access ≥ era ) return Access;
24     while not CAS(&Accesses[slot], Access, era);
25     return era
   // Changes below are for Hyaline-S, not Hyaline-1S
26 handle_t enter(int *slot)
27     while Acks[*slot] ≥ Threshold do
28         *slot = (*slot + 1) mod k;                                // Try out all k slots
29 void traverse(int slot, node_t *next, handle_t handle)
30     Counter = 0;
31     do Counter++ ... while ...;
32     FAA(&Acks[slot], −Counter);
```

Figure 3: Hyaline-S: dealing with stalled threads.

To alleviate contention due to *retire*, threads retire nodes in batches and keep a single reference counter per batch rather than individual node. To alleviate contention due to *enter* and *leave*, we introduce the concept of *slots*, which a thread chooses randomly. Each slot has its own `Head`, and thus, we end up with multiple retirement lists. When a batch is retired, it is added to every slot that has its `HRef` $\neq 0$ (i.e., slots with active threads). Since slots may end up with non-identical order of batches, we require the number of nodes in batches to be strictly greater than the number of slots. Each node in a batch keeps the `Next` pointer for the corresponding slot's list, and a dedicated node keeps the per-batch `NRef` counter instead. All nodes in the batch are linked together, and each node has an extra pointer to the node with `NRef`. Thus, each node keeps three variables irrespective of batch sizes and total number of slots.

We generalize the idea of `NRef` adjustments to accommodate Hyaline's multiple-list version. When adjusting a predecessor in slot i, we add $Adjs + HRef_i$ rather than just $HRef_i$, where $Adjs$ is a special constant which prevents the adjustment for the predecessor to complete until all slots are handled. Assuming that the number of slots, k, is a power of 2, and the maximum representable unsigned integer value is $2^N - 1$, we calculate: $Adjs = \left\lfloor \frac{2^N - 1}{k} \right\rfloor + 1$.

Hyaline-1 for Single-width CAS. If every thread allocates its own unique slot, we can squeeze `HRef` into a single bit and merge it with `HPtr`. This approach simplifies *enter* and *leave*, and

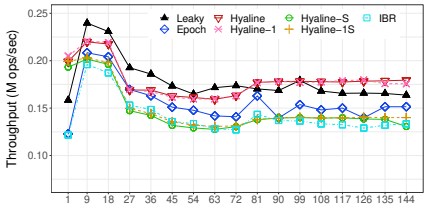
adjustments. Instead of adjusting predecessors and empty slots, we count the number of slots a batch is added to. After adding the batch to the last slot, `NRef` of the batch is adjusted by this counter.

3 ALGORITHM DESCRIPTIONS

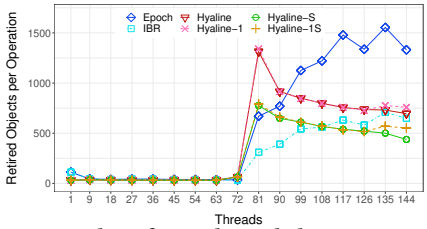
In Figure 1, we present pseudocode. (Hyaline-1 in Figure 2 replaces *enter* and *leave* with simpler equivalents.) Hyaline requires double-width CAS but can also be implemented with LL/SC; see [7] for details. Correctness arguments are given in [7]. *enter* atomically increments `HRef` while fetching the current pointer in a given slot. *retire* inserts a batch to all slots. *leave* decrements `HRef` and dereferences preceding nodes in the *traverse* helper method. An optional *trim* operation (we do not use it in Section 4) is equivalent to *leave* followed by *enter*, but avoids the unwanted alteration of `Head`.

To deal with stalled threads in Hyaline, we create Hyaline-S (Figure 3) by partially adopting the idea from hazard eras (HE) [8] and interval-based reclamation (IBR) [10] to record *birth eras* when allocating memory. However, we do not use retire eras and also support multiple threads per each slot. Our API is similar to 2GE-IBR [10]: all pointer dereferences must be wrapped in *deref*. The eras are 64-bit numbers which are assumed to never overflow in practice. When nodes are allocated, *init_node* initializes their birth eras with the era clock value. Threads must share per-slot eras, and the maximum era needs to be set (when dereferencing) using the *touch* helper function. *retire* uses the minimum birth era across all nodes in a batch, and skips slots with stale eras.

Since threads share per-slot eras in Hyaline-S, *enter* avoids slots occupied by stalled threads by using `Ack` values. To retain robustness, we adaptively change the number of slots; see [7].



(a) Throughput



(b) Average number of unreclaimed objects per operation

Figure 4: Bonsai Tree (50% *insert*, 50% *delete*).

4 EVALUATION

We extended the test framework of [10] to support Hyaline. We present results for Bonsai Tree [3] under write-intensive workload (50% *insert*, 50% *delete*). For more results, see [7]. We ran our tests for up to 144 threads on a 72-core machine consisting of four Intel Xeon E7-8880 v3 2.30 GHz (45MB L3 cache) CPUs with hyper-threading disabled and 128GB of RAM. Threads are pinned in order, socket by socket. We use jemalloc and clang 6.0 with `-O3`. For each data point, the experiment starts by prefilling the data structure with 50,000 elements and runs 10 seconds (5 times). Each thread then randomly performs the aforementioned operations. The key used in each operation is randomly chosen from the range of 0 to 100,000 with equal probability.

We compare all Hyaline variants against a variant [10] of EBR and 2GE-IBR. The benchmark parameters [10] for these algorithms already appear to be optimized for high throughput. We also present **Leaky** which does not perform any memory reclamation. For Hyaline(-S), we cap the number of slots, k, at 128 (rounded number of cores). We use batches of at least 64 nodes and at most $k + 1$.

Throughput (Figure 4a) decreases for all schemes as we approach 18 per-CPU cores. Hyaline has the best performance and steadily outperforms Epoch by ≈10%. IBR, Hyaline-S, and Hyaline-1S have similar performance; it is worse than their non-robust counterparts due to a substantial number of pointer dereferences. The number of unreclaimed objects in Figure 4b for Hyaline and Hyaline-S is generally smaller than that of Epoch and IBR, respectively.

5 CONCLUSION

We presented Hyaline, a new algorithm for memory reclamation which has a number of advantages: great performance and scalability, easy integration with underlying data structures, and handling of stalled threads (in Hyaline-(1)S). All Hyaline schemes are transparent and suitable for environments where threads are created, recycled, and deleted dynamically: threads are "off-the-hook" as soon as they *leave* and need not check retirement lists afterwards.

We thank the anonymous reviewers for their valuable feedback. We also thank Mohamed Mohamedin for helping with experiments for an early version of the algorithm. This work is supported in part by AFOSR under grants FA9550-15-1-0098 and FA9550-16-1-0371.

Complete details of the algorithms, analysis, and experimental results are available in [7]. We provide code for all Hyaline variants at https://github.com/rusnikola/lfsmr.

REFERENCES

[1] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. 2016. Fast and Robust Memory Reclamation for Concurrent Data Structures. In *The 28th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA '16)*. 349–359.
[2] Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC '15)*. 261–270.
[3] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2012. Scalable Address Spaces Using RCU Balanced Trees. In *The 17th Inter. Confer. on Architectural Support for Programming Languages and OS (ASPLOS XVII)*. 199–210.
[4] Keir Fraser. 2004. *Practical Lock-freedom (Ph.D. dissert.)*. University of Cambridge.
[5] 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.
[6] Maged M. Michael. 2004. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Trans. on Parallel and Distributed Systems* 15, 6 (June 2004), 491–504.
[7] Ruslan Nikolaev and Binoy Ravindran. 2019. Hyaline: Fast and Transparent Lock-Free Memory Reclamation (full paper, arXiv). http://arxiv.org/abs/1905.07903.
[8] 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 (SPAA '17)*. 367–369.
[9] John D. Valois. 1995. Lock-free Linked Lists Using Compare-and-swap. In *The 14th ACM Symposium on Principles of Distributed Computing (PODC '95)*. 214–222.
[10] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. 2018. Interval-based Memory Reclamation. In *Proceedings of the 23rd ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. 1–13.