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 \(\rightarrow\) Concurrent algorithms.

KEYWORDS
lock-free; memory reclamation; concurrent data structures

ACM Reference Format:

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 \(\text{HPtr}\) field of this tuple is a pointer to the beginning of the list, and the \(\text{HRef}\) field counts the number of active threads. When each thread enters, it atomically increments the \(\text{HRef}\) field to indicate that a new thread has arrived. At the same time, the thread records a snapshot value of \(\text{HPtr}\) into a special per-thread Handle variable.

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

When a thread leaves, it decrements \(\text{HRef}\). At the same time, it retrieves the \(\text{HPtr}\) pointer and traverses a sublist of nodes from \(\text{HPtr}\) to \(\text{Handle}\) that were retired since it entered. While traversing, the thread decrements \(\text{NRef}\) counters for every non-Head node.
To alleviate contention due to retire, threads retire nodes in batches and keep a single reference counter per batch rather than individual nodes. 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. 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 Adj + HRef to rather than just HRef, where Adj 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 representative unsigned integer value is $2^N - 1$, we calculate: Adj = $\left\lceil\frac{2^N - 1}{k}\right\rceil + 1$.

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

Figure 1: Hyaline for double-width CAS.

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

Figure 3: Hyaline-S: dealing with stalled threads.
adjusted. 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, NRRef 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].

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.

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/lfsr.

REFERENCES