We present a family of safe memory reclamation schemes, Hyaline, which are fast, scalable, and transparent to the underlying lock-free data structures. Hyaline is based on reference counting – considered impractical for memory reclamation in the past due to high overheads. Hyaline uses reference counters only during reclamation, but not while accessing individual objects, which reduces overheads for object accesses. Since with reference counters, an arbitrary thread ends up freeing memory, Hyaline’s reclamation workload is (almost) balanced across all threads, unlike most prior reclamation schemes such as epoch-based reclamation (EBR) or hazard pointers (HP). Hyaline often yields (excellent) EBR-grade performance with (good) HP-grade memory efficiency, which is a challenging trade-off with all existing schemes.

Hyaline schemes offer: (i) high performance; (ii) good memory efficiency; (iii) robustness: bounding memory usage even in the presence of stalled threads, a well-known problem with EBR; (iv) transparency: supporting virtually unbounded number of threads (or concurrent entities) that can be created and deleted dynamically, and effortlessly join existent workload; (v) autonomy: avoiding special OS mechanisms and being non-intrusive to runtime or compiler environments; (vi) simplicity: enabling easy integration into unmanaged C/C++ code; and (vii) generality: supporting many data structures. All existing schemes lack one or more properties.

We have implemented and tested Hyaline on x86(-64), ARM32/64, PowerPC, and MIPS. The general approach requires LL/SC or double-width CAS, while a specialized version also works with single-width CAS. Our evaluation reveals that Hyaline’s throughput is very high – it steadily outperforms EBR by 10% in one test and yields 2x gains in oversubscribed scenarios. Hyaline’s superior memory efficiency is especially evident in read-dominated workloads.

Abstract

We present a family of safe memory reclamation schemes, Hyaline, which are fast, scalable, and transparent to the underlying lock-free data structures. Hyaline is based on reference counting – considered impractical for memory reclamation in the past due to high overheads. Hyaline uses reference counters only during reclamation, but not while accessing individual objects, which reduces overheads for object accesses. Since with reference counters, an arbitrary thread ends up freeing memory, Hyaline’s reclamation workload is (almost) balanced across all threads, unlike most prior reclamation schemes such as epoch-based reclamation (EBR) or hazard pointers (HP). Hyaline often yields (excellent) EBR-grade performance with (good) HP-grade memory efficiency, which is a challenging trade-off with all existing schemes.

Hyaline schemes offer: (i) high performance; (ii) good memory efficiency; (iii) robustness: bounding memory usage even in the presence of stalled threads, a well-known problem with EBR; (iv) transparency: supporting virtually unbounded number of threads (or concurrent entities) that can be created and deleted dynamically, and effortlessly join existent workload; (v) autonomy: avoiding special OS mechanisms and being non-intrusive to runtime or compiler environments; (vi) simplicity: enabling easy integration into unmanaged C/C++ code; and (vii) generality: supporting many data structures. All existing schemes lack one or more properties.

We have implemented and tested Hyaline on x86(-64), ARM32/64, PowerPC, and MIPS. The general approach requires LL/SC or double-width CAS, while a specialized version also works with single-width CAS. Our evaluation reveals that Hyaline’s throughput is very high – it steadily outperforms EBR by 10% in one test and yields 2x gains in oversubscribed scenarios. Hyaline’s superior memory efficiency is especially evident in read-dominated workloads.

1 Introduction

Modern computer systems increasingly rely on parallelism. Programming paradigms are also changing accordingly: the use of scalable non-blocking data structures is preferred to more traditional lock-based approaches.

Aside from general memory allocation and reclamation problems, non-blocking data structures also present a number of unique challenges that do not manifest in lock-based programming. One of the most fundamental problems for lock-free data structures that use dynamic memory allocation is that memory objects need to be safely deallocated. The problem arises when one thread wants to deallocate a memory object, but concurrent threads still have stale pointers and are unaware of ongoing memory deallocation. Garbage collectors avoid this problem by deferring the deallocation until no thread has pointers to the deallocated memory object. However, fully lock-free garbage collectors are challenging to design, especially with consistent and limited overheads.

Moreover, it is often impractical to use garbage collectors in languages that are designed for unmanaged code such as C and C++. To support concurrent data structures in unmanaged languages, a number of techniques have been developed for safe memory reclamation (or SMR). Many existing approaches for SMR originate from, or improve upon, epoch-based reclamation (EBR) [24, 26] and hazard pointers (HP) [32]. Unlike garbage collectors, these schemes do not automatically determine when memory becomes free. Instead, such schemes are predicated on user-specified retire statements, which are roughly analogous to free, with the only difference being that retire does not necessarily deallocate memory right away.

EBR uses a simple API and achieves good performance, but lacks protection against stalled threads. This can prevent timely reclamation, resulting in blocking behavior due to memory exhaustion. HP does not suffer from this problem, but is harder to use and slower in practice. Some of the other SMR algorithms [11–13, 15, 42] rely on special operating system (OS) abstractions, which make them difficult to use.
in certain settings such as within OS kernels or platform-independent code. In general, all SMR schemes have different trade-offs in terms of API simplicity, throughput, average memory efficiency, and protection against stalled threads.

Although a number of existing SMR schemes [24, 39, 44] achieve excellent throughput, their memory efficiency is limited. An implicit assumption of these algorithms is that all threads get more or less even shares of memory objects to reclaim. In most existing SMR schemes, a thread that detaches an object from a data structure must eventually reclaim it. This can cause an unbalanced reclamation workload, especially in read-dominated scenarios, where most threads are reading and only a fraction of them modifies data (see examples in Section 6). When most threads are reading and are therefore not deallocating, the reclamation parallelism is reduced, which degrades memory efficiency. To make matters worse, threads also need to periodically peruse their local lists of not-yet-reclaimed objects to check if an object can be safely reclaimed (as in HP) or check the status of all threads to advance an epoch (as in EBR). The reclamation workload that is skewed toward the writer threads and consequent delayed reclamation can eventually degrade performance (see Section 6). Such performance degradation becomes even more evident in oversubscribed scenarios where there are more threads than cores available. Note that oversubscription is not that uncommon in practice (e.g., consider Go, Erlang, and proposed C++23 concurrency contracts).

Lock-free reference counting (LFRC) [33, 43], another SMR discipline, enables better parallelism in theory: a thread with the last reference frees an object, which often means that an arbitrary thread ends up freeing memory. Unfortunately, LFRC typically performs poorly since every object access, even just for reading, requires memory writes and barriers.

In this paper, we revisit reference counting – considered impractical for concurrent algorithms in the past – and design an SMR scheme called Hyaline (Sections 3 and 4). The key idea of Hyaline is to actively use reference counters only during reclamation, but not while accessing individual objects. This reduces overheads for object accesses, while ensuring that the reclamation workload is balanced across all threads, yielding excellent performance as well as excellent memory efficiency. We establish Hyaline’s core properties including reclamation safety, lock-freedom, reclamation cost bounds, and robustness (Section 5).

Hyaline also has a number of other important properties. Unlike most SMR algorithms, which typically require globally visible, private, per-thread state (either in static arrays or in dynamically managed lists), Hyaline supports virtually unbounded number of threads using a relatively small (fixed) number of shared slots, entities that can be shared by multiple threads. Since Hyaline’s reclamation is asynchronous (i.e., any thread can free memory allocated by any thread), threads can immediately be recycled without worrying about the fate of its previously deleted but not-yet-freed objects. These two properties ensure that Hyaline is less intrusive to applications, enabling its greater transparency. Hyaline is also 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 (Section 6). Our experimental results reveal that, in a number of cases, Hyaline demonstrates both excellent throughput as well as excellent memory efficiency, which is difficult for many past SMR schemes to achieve together. Hyaline’s substantial throughput gains are also evident: in the Bonsai Tree benchmark, Hyaline’s steady gains over EBR, one of the fastest schemes, are ≈10%. In oversubscribed scenarios, Hyaline particularly shines: up to 2x throughput gains for high-throughput data structures.

We also present an extension of Hyaline, called Hyaline-S, to deal with stalled threads (Section 4). Similar to EBR, basic Hyaline’s memory usage can become unbounded if some threads are stalled. We partially adopt the birth eras idea, inspired by similar usage in interval-based reclamation (IBR) [44] and hazard eras (HE) [39], and demonstrate how this idea helps to deal with stalled threads in Hyaline-S.

The paper’s research contribution is the Hyaline algorithm and its variants, which are the first SMR schemes that achieve excellent performance, memory efficiency, and other aforementioned properties over a broad range of workloads including read-dominated and oversubscribed scenarios.

2 Background

For greater clarity and completeness, we discuss properties and challenges of the existent memory reclamation schemes.

Read-Modify-Write. Lock-free algorithms typically use read-modify-write (RMW) operations, which atomically read a memory variable, perform some operation on it, and write back the result. Modern CPUs implement RMWs via compare-and-swap (CAS) or a pair of load-link (LL)/store-conditional (SC) instructions. For better scalability, some CPUs support specialized fetch-and-add (FAA) and swap operations. Also, x86-64 and ARM64 support operations on two adjacent CPU words (double-width RMW) via the cmpxchg16b [4] and 1daxp/st1xp [2] instructions, respectively.

Hyaline requires either double-width CAS, or ordinary LL/SC (see Appendix A regarding LL/SC and PowerPC) since a reference counter needs to be coupled with a pointer in some places. In rare cases, such as in SPARC [7], where neither is supported, reference counters can be squeezed with pointers. (SPARC uses 54-bit virtual addresses; 48-bit cache-line aligned pointers where lower 6 bits are 0s can be 1

1Specifically, this is useful for global data structures within OS kernels that support kernel-mode preemption, e.g., Linux. We also have preliminary results with experimental OS designs [37].
squeezed with 16-bit counters.) We also present a specialized Hyaline-1 version for single-width CAS.

**API Model.** We focus on the SMR problem in unmanaged code environments such as C/C++. Hyaline’s and Hyaline-1’s basic programming model is similar to that of EBR [24] and is defined as follows. Memory objects are allocated using standard OS-defined means. They additionally incorporate SMR-related headers and are initialized appropriately. Once memory objects appear in a lock-free data structure, they must be reclaimed using a two-step procedure. After deleting a pointer from the data structure, a memory object needs to be retired. A memory object is returned to the OS only after the object becomes unreachable by any other concurrent thread. All data structure operations must be encapsulated between enter and leave calls that trigger the use of SMR.

**Robustness.** One of the biggest downsides of the simplistic API model described above is that memory usage becomes unbounded in the presence of stalled threads. We call an algorithm robust if it bounds memory usage for stalled threads.²

HP [32] was among the first to recognize this problem and propose a safer API model, which wraps every pointer access. HP more precisely tracks retired objects using pointers and assigns special indices to accessed objects. HE [39] adopted the same API model but proposed to internally record eras (epochs) rather than pointers. More recently, IBR [44] simplified HE’s API by abolishing indices when wrapping pointers, bringing it closer to the original EBR model.

We added additional robustness guarantees to Hyaline and Hyaline-1 by extending the API with a pointer wrapper method, deref, as in IBR. Our robust schemes, Hyaline-S and Hyaline-1S, adopt the birth eras approach from HE and IBR to guard against stalled threads. The main idea is to mark each allocated object with the global counter, so that stalled threads will only hold older objects. Note that whereas HE and IBR also use retire eras to identify reclamation intervals, Hyaline relies on reference counting.

**Reclamation Cost.** Many reclamation schemes have a non-constant reclamation cost. For example, HP, HE, and IBR need to periodically peruse thread local lists of not-yet-reclaimed objects to check if an object can be safely reclaimed. Hyaline does not need to periodically check thread local lists. We discuss and prove reclamation costs in Section 5.

**Transparency.** Most existing SMR schemes maintain special entries throughout thread lifecycles – e.g., static arrays indexed by thread IDs. In practice, threads can be created and deleted dynamically, and practical implementations [3] maintain lists rather than arrays with per-thread entries.

²Sometimes, this property is also called “lock-freedom” (memory-wise). Since memory is finite, stalled threads prevent other threads from allocating memory when memory is exhausted. We use the terminology from [13, 21, 44] to capture this property more precisely.

³Hyaline-S adaptively changes the number of slots to guarantee robustness.

However, this puts an extra burden on programmers who have to explicitly register and unregister threads. This also breaks seamless integration, as concurrent data structures cannot be accessed outside thread contexts – e.g., signal handlers or OS interrupt contexts. Moreover, unregistration is blocking, as a thread needs to complete deallocation, which is impossible until all other threads promise not to access its locally retired objects. In Hyaline, threads can completely forget about previously retired objects after calling leave, as they are already (or will be) taken care of by the remaining threads. (In Section 3.2, retire uses local batches, but they can be immediately finalized by allocating a finite number of dummy nodes.) We call a scheme transparent if it avoids the problems mentioned above.

**Snapshot-Freedom.** To make certain schemes (HP, HE, IBR, etc) more efficient, when examining what objects can be deleted safely, a snapshot of global state (i.e., all hazard pointers or eras) is taken. The per-thread snapshot sidesteps expensive cache misses since it can be consulted repeatedly for all not-yet-freed objects. Per-thread snapshots are typically preallocated, resulting in extra $O(n^2)$ memory usage, which is substantial as the number of threads, $n$, grows. Furthermore, pre-allocated memory needs to be expanded if the number of threads grows dynamically, which presents additional challenges for transparency. In contrast, EBR consults the global state only once per each examination and does not take snapshots. All Hyaline schemes are also snapshot-free.

**Semantics.** Most robust schemes provide different semantics in handling memory objects that have never been dereferenced. Whereas non-robust schemes, such as EBR, can work with the original lock-free linked list [25], robust schemes (HP, HE, and IBR) require a modification [32] that timely retires deleted list nodes. Our non-robust and robust Hyaline schemes have a similar distinction. FreeAccess [17] – a recent scheme – specifically tackles the semantics problem. The scheme is still robust, but falls short on transparently handling the swap operation (can be used for better scalability), and needs compiler modifications. It also uses a garbage collector which is undesirable when fully transparent memory management is needed, such as in OS kernels.

**Memory Overhead (Header Size).** SMR schemes also differ in extra memory required per each node. For example, EBR, HP, and PEKR store a (thread-local) list pointer per node. HE and IBR additionally require two 64-bit eras. All Hyaline variants require 3 words which is equivalent to HE and IBR for 64-bit CPUs and more efficient for 32-bit CPUs.

Although EBR/HP/PEBR’s overhead can be fully eliminated by allocating an intermediate container object when retiring, this causes undesirable circular allocator dependency. In the same vein, HE, IBR, and Hyaline schemes can reduce the overhead to just 1 word.
Table 1. Comparison of Hyaline with existing SMR approaches.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Based on</th>
<th>Performance</th>
<th>Robust</th>
<th>Transparent</th>
<th>Header Size</th>
<th>Usage/API</th>
</tr>
</thead>
<tbody>
<tr>
<td>LFRC [33, 43]</td>
<td>-</td>
<td>Very Slow</td>
<td>Yes</td>
<td>Partially</td>
<td>N/A, but 1 extra word per pointer</td>
<td>Harder, intrusive</td>
</tr>
<tr>
<td>HP [32]</td>
<td>-</td>
<td>Slow</td>
<td>Yes</td>
<td>No (retire)</td>
<td>1 word</td>
<td>Harder</td>
</tr>
<tr>
<td>EBR [24, 26]</td>
<td>RCU [31]</td>
<td>Fast</td>
<td>No</td>
<td>No (retire)</td>
<td>1 word</td>
<td>Very easy</td>
</tr>
<tr>
<td>DEBRA+ [15]</td>
<td>EBR</td>
<td>Fast</td>
<td>Partially</td>
<td>No (OS)</td>
<td>1 word + desc</td>
<td>Harder</td>
</tr>
<tr>
<td>PEBR [30]</td>
<td>EBR, HP</td>
<td>Medium</td>
<td>Yes</td>
<td>No (retire, OS)</td>
<td>1 word</td>
<td>Medium</td>
</tr>
<tr>
<td>HE [39]</td>
<td>EBR, HP</td>
<td>Medium</td>
<td>Yes</td>
<td>No (retire)</td>
<td>3 words (64 bit)</td>
<td>Harder</td>
</tr>
<tr>
<td>IBR (2GE) [39]</td>
<td>EBR, HP</td>
<td>Fast</td>
<td>Yes</td>
<td>No (retire)</td>
<td>3 words (64 bit)</td>
<td>Medium</td>
</tr>
<tr>
<td>Hyaline</td>
<td>-</td>
<td>Fast</td>
<td>No</td>
<td>Yes</td>
<td>3 words</td>
<td>Very easy</td>
</tr>
<tr>
<td>Hyaline-1</td>
<td>-</td>
<td>Fast</td>
<td>No</td>
<td>Partially</td>
<td>3 words</td>
<td>Very easy</td>
</tr>
<tr>
<td>Hyaline-S</td>
<td>Hyaline, part. HE/IBR</td>
<td>Fast</td>
<td>Yes</td>
<td>Yes (^3)</td>
<td>3 words</td>
<td>Medium</td>
</tr>
<tr>
<td>Hyaline-1S</td>
<td>Hyaline-1, part. HE/IBR</td>
<td>Fast</td>
<td>Yes</td>
<td>Partially</td>
<td>3 words</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Summary. Existing approaches are discussed in detail in Section 7. Table 1 presents a qualitative and quantitative comparison of Hyaline with other schemes on metrics including performance, robustness, and transparency. We also categorize API as hard, medium, etc., similar to the discussion in [44]. Although this categorization is somewhat subjective, we note that the medium difficulty in robust Hyaline-S and Hyaline-1S implies that deref on pointers can be fully hidden using standard language idioms, such as smart pointers in C++, and no extra programming language or OS support is needed. This is not true for schemes that rely on OS mechanisms. Furthermore, schemes that use HP’s API require assigning indices to reserved objects and annotating where a pointer is used for the last time. These cannot be hidden in smart pointers easily and need to be handled explicitly by a programmer.

3 Hyaline

Hyaline is a member of the family of memory reclamation techniques where programs explicitly retire objects and ensure that retired objects are not reachable from subsequent operations on the data structures. In each operation on the data structures must be enclosed between enter and leave calls as presented in Figure 1. Hyaline keeps track of all active threads using special reference counters. Those counters are not typical and do not represent the number of references to objects directly. Unlike traditional per-object counting, the use of reference counters is triggered only when handling retired objects (nodes). Thus, insertions and read-only traversals avoid expensive (and inconvenient) per-access counting.

```
handle_t Handle = enter();
// deref is for Hyaline-S,
// not needed in Hyaline
List = deref(&LinkedList);
Node = deref(&List->Next);
retire(Node);
// Do something else...
leave(Handle);
// Transparency: the thread
// need not check any of the
// retired nodes after this
```

Figure 1. Hyaline’s transparent API.

3.1 Simplified Version

We first describe a simpler version of Hyaline that manipulates only a single retirement list. This version is more prone to CAS contention, a problem addressed by a scalable version that we present in Section 3.2.

Hyaline’s key idea is that all threads participate in the tracking of retired nodes in the global list even if they are not actively retiring any nodes themselves. A special Head tuple is associated with the retirement list. The tuple consists of HPtr and HRef fields. HPtr is a pointer to the beginning of the list, and HRef counts the number of active threads. Initially, when the list is empty, HPtr = Null and HRef = 0.

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 at the moment it entered. The thread stores this snapshot value in a special per-thread Handle variable. Since updates on the [HRef, HPtr] tuple have to be atomic, we use double-width RMW to update Head.
As nodes are retired, threads append them to the list (Figure 2). Since nodes need to be connected in the list, each node incorporates a special header in addition to any other fields used for representing the encapsulating data structure. The header 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. (We will describe how NRef is initialized later.)

When a thread completes a data structure operation (leaves), it decrements HRef to indicate that one thread has just left. Simultaneously, it retrieves the HPtr pointer and then traverses a sublist of nodes from HPtr to Head that were retired since it initiated the operation (enter). While traversing the sublist, the thread decrements NRef counters for every non-Head node. The first node’s counter (HRef) is already incremented. A node is freed when its counter becomes 0.

Using HRef for the first node prevents the ABA problem as a thread has a reference to every node through its Handle inclusively – i.e., no other thread can recycle these nodes.

We now describe how NRef values get propagated across the list. When retiring a node, its NRef is set to 0, as the actual counter for the very first node in the list is inferred from the HRef field of Head. As threads insert retired nodes, they initialize Next and atomically update Head (shown with dashed lines in Figure 3, part (a)). NRef of the predecessor is initially 0. However, since a node was just added, the predecessor is no longer the first node, and any concurrent thread may decrement its NRef, converting it to a negative value, but will not deallocate this predecessor (NRef must become 0 for the node to be deallocated). Finally, the current thread atomically adds the snapshot value of HRef (obtained while appending the new node) to the NRef field of the predecessor, and its new adjusted value becomes ≥ 1 (Figure 3, part (b)). Retiring is now complete (Figure 3, part (c)).

Essentially, NRef is the difference between two logical variables: the number of times the node is acquired and the number of times it is released. Due to concurrency interleaving, NRef is relaxed and can be negative.

Figure 4 shows an example with 3 threads. Initially, HRef is 0 and HPtr is Null. (a) Thread 1 calls enter to atomically increment HRef and retrieve its handle. (b) Thread 1 retires node N1; as the list is empty, there is no predecessor to adjust. (c) Thread 2 enters, but (d) it stalls while retiring N2. Meanwhile, (e) Thread 3 enters. (f) Thread 1 leaves and dereferences all nodes in the list through its handle N1. Since N2 is the first node, HRef is decremented, but N2’s NRef field remains intact. N1 stays as its NRef is now negative. (g) Thread 2 resumes and completes its adjustment for N1. (h) Thread 2 then leaves, dereferences all nodes, and deallocates N1. (i) Finally, Thread 3 leaves and deallocates N2.

Although this version is not yet optimized for performance, we make one important observation regarding the algorithm’s asynchronous tracking mentioned in introduction: threads traverse lists just once when dereferencing nodes in leave. This is unlike EBR, where all threads are periodically checked if they are past the retired node(s) epoch. Section 6 reveals this to be Hyaline’s advantage for oversubscribed tests.

**Alternative Designs.** Hyaline carefully avoids the ABA problem which is possible with other designs. A straightforward alternative is to use NRef in the first node as usual, i.e., to indicate its reference counter. HRef can then indicate
the reference counter of the node to be retired in the future. However, this design triggers the ABA problem, as the node pointed to by Handle (end-of-list marker) can be recycled and may reappear in the retirement sublist when leave is called. The sublist will get truncated, and the remaining nodes will never be dereferenced. An extra ABA tag could prevent this problem but Head already needs double-width operations.

NRef could also store the reference counter of the next rather than the current node. HRef would indicate the reference counter of the first node as in Hyaline. While this design is ABA-free, it has a major drawback: nodes must be dereferenced in the reversed order, as they are dependent on each other. It complicates the implementation, as backward links also need to be stored. Moreover, Head cannot be updated until the retirement sublist is fully traversed, and by that time, other nodes may already be retired. If deallocation is slow, one unlucky thread can easily get stuck in a state where it has to constantly deallocate nodes retired by other threads. Other threads will simply continue dereferencing their counters and keep retiring more and more nodes. Hyaline avoids this problem by immediately decrementing HRef, as all nodes are independent from each other there.

3.2 Scalable Multiple-List Version

If deletions are frequent, retire calls may create contention on Head. To alleviate the contention, threads create local lists of nodes to be retired. Threads retire entire batches of nodes and keep a single reference counter per batch rather than individual nodes. Batches do not have direct analogues in epoch-based approaches, where all retired nodes are always in thread-local lists. However, batch size (the number of nodes in a batch) impacts the cost of retirement in a way that is similar to the frequency of epoch counter increments.

Frequent enter and leave calls also create contention on Head, which is undesirable for high-throughput data structures. To address this problem, we introduce the concept of slots, which a thread chooses randomly or based on its ID. Slots do not need to be statically assigned: they can change from one operation to another. Each slot has its own Head, and thus, we end up with multiple retirement lists. When a batch is retired, it needs to be added to each slot that has its HRef ≠ 0 (i.e., slots with active threads).

Since batches are added atomically only to one slot at a time, slots may end up with non-identical order of batches. To support this, we need individual list pointers. We take advantage of the fact that we retire entire batches rather than nodes. To that end, 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 retirement list, and one additional node will keep the per-batch NRef counter instead. Additionally, 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.

In Section 3.1, we described reference adjustments using signed integer arithmetic. The same argument applies to unsigned numbers, in which case negative numbers represent very large integers. We generalize this idea to accommodate Hyaline’s multiple-list version. When adjusting a predecessor in slot i, we add Adjs + HRef rather than just HRef, 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.

For example, if k = 1 (simple version), Adjs cancels out right away. When k = 8, assuming 64-bit integers, Adjs = 2^{41}. It is easy to see that more generally, Adjs cancels out after k additions: k × Adjs = 0 due to unsigned integer overflow.

When retiring a batch, a predecessor batch has to accumulate Adjs for all k lists for the adjustment to complete. Since some slots have no active threads, we accumulate Adjs for them when inserting the batch, and atomically add the net value to NRef of the current batch as the final step.

In Figure 5, we present an example with k = 8 slots. For the purpose of this example, we enumerated nodes to reflect their relative slot positions (skipping empty slots) and corresponding batch numbers. For convenience, Empty denotes adjustments for five empty slots (5 × Adjs) that need not be handled. Batches are added one slot at a time, and two concurrent threads insert them in an interleaved fashion. When Batch 0 is inserted, it ends up in the first position for slot 0 and the second position in all other active slots. NRef for Batch 0 is stored in the node R0 and contains Empty for empty slots, 0 for slot 0 (not yet adjusted), 2 × Adjs for slots 2 and 4 (adjusted when retiring Batch 1), and the actual counter component Δ0. Δ0 contains the snapshot values of
HRef\_2 and HRef\_4 when Batch 1 is inserted. A similar breakdown is shown for Batch 1. For Batch \( m \), all adjustments are already cancelled out, and its NRef node contains just \( \Delta_m \).

**Hyaline-1 for Single-width CAS.** In a special case, if every thread allocates its own unique slot, we can squeeze HRef into one bit and merge it with HPtr. This simplifies enter and leave, since they can use ordinary write and swap in lieu of CAS, making these operations wait-free. Adjustments can be also simplified: instead of adjusting predecessors and empty slots, we count the number of slots a batch is added to. (\( k \) does not have to be a power of 2.) After adding the batch to the last slot, NRef of the batch is adjusted by this counter.

**Contention.** Assuming that slots are cache-line aligned, CAS on Head is almost uncontended, and MESIF/MOESI protocols used by modern Intel/AMD CPUs incur no substantial performance penalty [41] (contrary to popular belief). The cost of enter and leave is therefore relatively small. Section 6 further shows that there is very little difference in Hyaline’s or Hyaline-1’s overall performance even for high-throughput data structures, which confirms that CAS in enter or leave is not a source of any measurable performance penalty.

**Costs.** Hyaline-1’s and EBR’s enter/leave costs are similar at the very least for x86 (write+barrier is replaced with swap, i.e., xchg, by recent gcc/clang compilers; AMD explicitly recommends xchg for sequentially-consistent writes [9]). Due to low CAS contention (used in lieu of swap), more general Hyaline also exhibits very similar performance. Though leave is longer in Hyaline and Hyaline-1 due to list traversals, this cost is simply incurred elsewhere in EBR, e.g., in retire.

### 4 Algorithm Descriptions

The main idea of Hyaline is that it always implicitly keeps track of concurrent threads. The number of concurrent threads gets reflected in the counter for each retired batch. Each of the concurrent threads has to decrement this counter explicitly. When the counter gets to 0, the batch is reclaimed.

Figure 6 presents the layout of nodes, batch structure, and global state. Batches are first accumulated locally. All retired nodes are linked together using BatchNext. The very last node (with a reference counter) is denoted as NRefNode, its BatchNext points to the very first node (i.e., in a cyclic list manner). Every node has a pointer to NRefNode.

#### 4.1 Basic Hyaline

In Figure 7, we present pseudocode for the enter, leave, and retire operations. enter atomically increments the HRef variable while fetching the current pointer in a given slot. retire inserts a batch to all slots. For empty slots, it counts Empty adjustments and adds Empty to NRef of the batch in the very end. For each slot, a predecessor is adjusted by the corresponding HRef snapshot value and Adjs. leave decrements HRef, but also reads Next from the node Head is pointing to. Since a thread always has a reference to the head of the list, reading the first node is safe. The last thread replaces the first node with Null treating it as a predecessor in retire. Finally, succeeding nodes (if any) are dereferenced in the traverse helper method.

Hyaline-1 in Figure 8 replaces enter and leave with simpler equivalents. Since one thread is the sole owner of all nodes, leave can detach the first node immediately and read the node that follows after that. This simpler scheme also does not adjust predecessor nodes.

#### 4.2 Hyaline-S

To deal with stalled threads in Hyaline, we extend Hyaline by partially adopting the idea from HE and IBR to record birth eras when allocating memory. The high-level idea is to mark each allocated object with the global counter, so that stalled threads will only hold older objects. Objects allocated after those threads stall will be counted only towards active threads. Birth eras simply facilitate detection of stalled threads in Hyaline. (Compare it to HE and IBR, where birth and retire eras define actual reclamation intervals.) Unlike HE/IBR, birth eras share space with other variables, e.g., Next, as they are not required to survive retire.

Hyaline-S, unlike Hyaline-1S, supports multiple threads per each slot, so we have to record eras such that they can be shared across multiple threads. That presents extra challenges when dealing with stalled threads since they may interleave with non-stalled threads.

In Figure 9, we present Hyaline-S. Our API model is reminiscent of 2GE-IBR [44] which only requires to additionally wrap all pointer reads in a special deref call. 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. When dereferencing pointers, threads call deref to update a per-slot access era value. Since Hyaline-S allows arbitrary number of threads per slot, threads must share per-slot eras, and the maximum era needs to...
forall head_t Head ∈ Heads[k] do  // Initialization
  Head.HRef = 0; Head.HPtr = Null;
handle_t t enter(int slot)
  Last = FAA(Heads[slot], { HRef=1, HPtr=0 });
  return Last.HPtr;  // Returns a handle
void leave(int slot, handle_t handle)
  do
    Head = Heads[slot];
    Curr = Head.HPtr;
    if ( Curr # handle )
      New.HPtr = Curr;
      New.HRef = Head.HRef - 1;
      while not CAS(Heads[slot], Head, New);
      if ( Head.HRef = 1 and Curr )
        // Treat Curr as if
        adjust(Curr, Adj);
      // it were a predecessor
    else if ( Curr # handle )
      // Non-empty list
      traverse(Next, handle);
  void adjust(node_t *node, int val)
    do
      Ref = node->NRefNode;
      // free_batch() frees all nodes by iterating
      traverse(Next, handle);
      // first node in the batch.
      if ( FAA(&Ref->NRef, val) = -val ) free_batch(Ref->BatchNext);
  exit
  /\ Initialization
  AllocEra = 0;
thread int AllocCounter = 0;
forall int Access ∈ Accesses[k] do Access = 0;
forall signed int Ack ∈ AckS[k] do Ack = 0;
node_t *deref(int slot, node_t *ptr_node)
  Access = Accesses[slot];
  while True do
    node_t *Node = (*ptr_node);
    Alloc = AllocEra;
    if ( Access = Alloc ) return Node;

    doAdj = False, Empty = 0, Inserts = 0;
    do Adjacency
      doAdj = True, Empty += Adjs;
      // Returns a handle
    void leave(int slot, handle_t handle)
      do
        Head = Heads[slot];
        if ( Head.HRef = 0 )
          // 0
            doAdj = True, Empty += Adj;
        // continue with the next slot;
        New.HPtr = CurrNode;
        New.HRef = Head.HRef;
        New.HPtr->Next = Head.HPtr;
        while not CAS(Heads[slot], Head, New);
        CurrNode = CurrNode->BatchNext;
        adjust(HHead.Ptr, Adj + Head.HRef);
      // 0
      if ( doAdj ) adjust(batch->FirstNode, Empty);
      // 0
      void traverse(node_t *next, handle_t handle)
        do
          Curr = next;
          if ( Curr = Null ) break;
          next = Curr->Next;
          Ref = Curr->NRefNode;
          if ( FAA(&Ref->NRef, -1) = -1 )
            // BatchNext of Ref points to the
            free_batch(Ref->BatchNext);  // first node in the batch.
      traverse(Next, handle);
      if ( FAA(&Ref->NRef, val) = -val ) free_batch(Ref->BatchNext);

Figure 7. Hyaline for double-width CAS.

Figure 8. Hyaline-1 for single-width CAS (substitutes for functions).

to be set using the touch helper function. (Hyaline-1S can just write the new era, as there is a 1:1 thread-to-slot mapping.) Since all active threads update eras when calling deref in their slots, retire simply 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, it is crucial to stay away from slots occupied by stalled threads when entering. Each slot keeps a special Ack value incremented by retire. Ack accumulates the total number of active threads across all retired batches in the slot. Since all retired batches inevitably appear in the retirement sublists of all active threads, each thread acknowledges that it no longer references batches by decrementing Ack in traverse. Ack can be negative temporarily if traverse takes place before FAA in retire. (Nonetheless, Ack only increases after finite number of retirements when at least one thread is stalled, i.e., it does not call traverse.) Ack may also be positive, but after some threshold (e.g., 8192), enter can assume that the corresponding slot is occupied by stalled threads. Ack's do not incur any measurable penalty as evidenced by Section 6 where Hyaline-S and Hyaline-1S have roughly similar performance.

All slots may end up being occupied by stalled threads in Hyaline-S. To guarantee robustness, we can adaptively increase the number of slots by using an extra array which stores pointers to arrays of slots (Section 4.3).

4.3 Adaptive Resizing for Hyaline-S

Unlike Hyaline-1S which allocates a dedicated slot for each thread and is fully robust, Hyaline-S caps the total number of slots. This limits robustness guarantees for Hyaline-S in rare situations when all slots fill up with stalled threads and they begin to interfere with active threads.

We now describe an approach which makes Hyaline-S fully robust by adaptively increasing the number of available slots, \( k \), as a larger number of threads are stalled. We denote the initial \( k \) value (a constant), \( K\text{min} \). The current \( k \) value is stored in a global atomic variable.

When a batch is finalized and retired, we read the current \( k \) value. (There is no problem if concurrent threads increase the \( k \) value right after we read it, as new slots will be used by new enter calls which need not account for already retired nodes. A larger than necessary \( k \) is also not a problem since the batch will simply be added to extra slots.) We calculate the \( Adjs \) value based on the current \( k \) value and store it in each batch. Each node in a batch contains a pointer to \( NRefNode \), but \( NRefNode \) itself does not need to keep this pointer. Instead, we use this variable to store the current \( Adjs \) value for the batch.

When calling adjust, we use the corresponding batch’s \( Adjs \) value. In Figure 7, we have three adjust calls: Line 17 uses \( Adjs \) for the curr’s batch, Line 38 uses \( Adjs \) for HPtr’s batch, and Line 39 uses \( Adjs \) for the current batch.

When stalled threads occupy all slots (Figure 9, Line 26), we adaptively increase the number of slots. Since we cannot resize the initial array of slots easily, we maintain a directory of slots, an array of pointers to arrays of slots, as shown in Figure 10. This array is fixed-size and small, e.g., for 64-bit CPUs, it never exceeds 64 entries. Initially, only index 0 points to the array of slots with \( K\text{min} \) entries. As enter runs out of slots, we allocate an additional array of \( (2 \times K\text{min} – K\text{min}) \) slots such that the total number of slots doubles. We atomically change index 1 to point to the new array (we also offset this pointer by \( K\text{min} \) to simplify the slot position calculation). If a concurrent thread also changes index 1, the thread for which the corresponding CAS fails will discard the allocated buffer. The aforementioned procedure applies to all arrays which use slots: Heads, Accesses, and Ack's.

To access a slot, we use the formula from Figure 10, which calculates a directory array index. The \( log_2 \) operation, including a special case of \( log_2(0) = -1 \), is efficiently implemented by the leading zero count instruction, available on modern CPUs, by using \( log_2(x) = N – lzcnt(x) – 1 \), where \( N \) is bit-length. Since we always double the number of slots, \( k \), and the initial \( K\text{min} \) value is a power-of-two number, our assumption that \( k \) is a power-of-two number is still valid.

We always increase the number of slots as we detect more stalled threads and run out of slots. However, the number of slots is bounded by the total number of stalled threads (rounded to the next power-of-two number). Since the number of threads is finite, memory occupied by slots is bounded, i.e., our algorithm is still robust. Existing robust SMR schemes similarly require dedicated slots per each thread.

4.4 Usage Preference

It should be feasible to always use Hyaline-1 in lieu of EBR, and Hyaline-1S in lieu of HE, HP, or IBR given Hyaline’s performance benefits (Section 6). One exception is when users deliberately want to avoid reclamation by read-only threads due to some extremely rigid latency requirements. This scenario seems uncommon for general-purpose systems, and it would not improve the overall throughput anyhow.

Hyaline-1 and Hyaline-1S are very portable and expose a relatively simple API. Hyaline and Hyaline-S provide full transparency but additionally require LL/SC or double-width CAS, which degrades portability. All Hyaline schemes simplify integration, e.g., it is much easier to register/unregister threads dynamically than with the aforementioned schemes. Garbage collectors have different trade-offs, and Hyaline’s applicability in the corresponding applications is similar to that of EBR, HE, HP, and IBR.

5 Correctness

We now prove correctness, lock-freedom, and robustness.

Theorem 1. All Hyaline variants are reclamation-safe.

Proof. In a correct program, a retired batch cannot be accessed by subsequent operations. Only concurrent operations may still access it. Each of those concurrent operations starts by calling Hyaline’s enter. Any batch retired during this concurrent execution will have its NRef ≠ 1 (Lines 26 and 39). If another thread executes leave after Line 36 and before the last adjustment in Line 39, then it will start by decrementing the retired object’s reference count such that it will be a
very large number (Line 46). Only objects with a new reference count of zero are reclaimed (Lines 22 or 46). Thus, those retired objects with very large reference counts are safe from being reclaimed. After executing Line 39 across all slots where the batch is placed, the retired object’s reference count will reflect the correct number of concurrent threads that have not executed leave yet. Hence, the object will not be reclaimed until all these threads execute leave.

Hyaline-(1)S, regardless of HRef values, skips slots with eras that are smaller than min_birth from a retired batch. min_birth signifies the oldest node in the batch. deref always updates per-slot eras to keep them in sync with the global era clock. Thus, the retired batch must have been covered by per-slot eras unless none of its nodes is ever dereferenced.

**Theorem 2.** All Hyaline variants are lock-free. (With respect to CPU progress only, see Theorem 5 for robustness.)

**Proof.** Hyaline has two unbounded loops (Lines 7-15 and 28-36). If the CAS operation fails in the first loop (Lines 7-15) causing it to repeat, it means that Head is changed by another thread executing enter, leave, or retire in the same slot. Thus, that other thread is making progress – i.e., successfully executing enter, leave, or retire in the same slot, and finishing modification of the same Head. The same argument applies to the second loop (Lines 28-36). The loop in traverse is bounded by the number of batches retired between executing enter and leave, and this number is finite.

Hyaline-S (Figure 9) has two additional loops (Lines 7-11 and 20-23). If CAS fails in touch (Lines 20-23) causing it to repeat, it means that another thread calling touch succeeds. The other loop (Lines 7-11) converges unless the global era clock is incremented. In the latter case, another thread is making progress, i.e., initializes a new node in Line 17.

**Theorem 3.** Hyaline and Hyaline-S have \(O(n/k)\) reclamation cost.

**Proof.** The reclamation cost in Hyaline consists of two parts: 1) the direct cost of retire and 2) the cost of retire incurred later, during list traversal in leave.

Retiring is a simple \(O(1)\) linked-list (batch) insertion operation. Upon reaching the maximum batch size, \(s\), retire inserts the batch into slots with active threads. Since the number of slots is \(k\), \(s \geq k + 1\). Each batch is inserted into at most \(k\) slots after \(s\) per-node retire calls, making the average cost of retire \(O(1)\).

The list traversal takes places for all batches retired between enter and leave. A batch is retired after \(s \geq k + 1\) retire calls (for individual nodes). Each batch maintains a single reference counter per \(\geq (k + 1)\) nodes. The batch’s reference counter needs to be decremented by all active threads (at most, \(n\) threads). Thus, the average cost to update a reference counter per node (i.e., the indirect cost of a single retire incurred in leave) is \(O(n \times \frac{1}{k}) = O(n/k)\). □

**Theorem 4.** Hyaline-1 and Hyaline-1S have \(O(1)\) reclamation cost.

**Proof.** Hyaline-1 and -1S are special cases of Hyaline and Hyaline-S, where \(k = n\) (i.e., the number of threads equals to the number of slots). Thus, the reclamation cost is \(O(1)\). □

**Theorem 5.** Hyaline-S and Hyaline-1S are robust.4

**Proof.** Since slots with stalled threads are detected after a finite number of retire calls and avoided by active threads in their following operations, we assume, without loss of generality, that slots do not reference any active threads. (Although batches are potentially added to every slot, one stalled thread can only make unusable the slot which was used by the last enter operation of the stalled thread. Only this slot references this thread. Newly allocated nodes will skip this slot due to its stale era and consequently will not reference the stalled thread when these nodes are retired.)

Since threads update their per-slot eras in monotonically increasing order when calling deref, each slot \(i\) ends up with some era \(A_i\) when it contains only stalled threads. Let \(E_{\max} = \max(A_i)\) across all slots \(i\) with stalled threads. We use \(E_i\) to denote a global era clock value when the earliest stalled thread from slot \(i\) entered. All previously retired

---

4As in IBR, we consider only stalled threads – i.e., threads that are stopped indefinitely as opposed to threads that are simply paused briefly. Also, as in IBR, “starved” threads that are running but unable to make any progress can still potentially reserve an unbounded number of objects; this is prevented by bounding the number of CAS failures in data structure operations and restarting from the very beginning (implemented by Section 6’s benchmark).
nodes must have been retired before (or at) \( E_i \). Let \( \delta \text{Era} = \text{Era}_{\text{max}} - \min(E_i) \) across all \( i \) slots with stalled threads. All potentially unreclaimable batches will have their \( \min \text{ birth} \leq \text{Era}_{\text{max}} \) (Line 14 of Figure 9). As each thread periodically increments the era value, a number of unreclaimable batches is bounded by \( \delta \text{Era} \times \text{Freq} \times n \), where \( n \) is the number of threads and \( \text{Freq} \) is the frequency used in the algorithm.

Batch sizes can be capped by \( k + 1 \), where \( k \) is the number of slots. Thus, the number of unreclaimable nodes is bounded by \( \delta \text{Era} \times \text{Freq} \times n(k + 1), k \leq n \).

6 Evaluation

We used and extended the test framework of [44] to support Hyaline. The framework consists of four benchmarks representing different data structures: the sorted linked-list [25, 32], lock-free hash map [32], a variant of the Bonsai Tree [16], which is a self-balancing lock-free binary tree, and Natarajan and Mittal’s binary tree [34].

We run 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. We chose Clang 11.0.1 with the -O3 optimization flag due to its better support of double-width RMW as used by Hyaline. We saw no visible difference between GCC and Clang for existing algorithms. We used jemalloc [23] to alleviate the standard library malloc’s poor performance [1].

Since a number of different techniques exist, we focus on well-established or state-of-the-art algorithmic schemes that have similar properties or programming models as Hyaline. We do not evaluate classical reference counting because it uses an intrusive model and is already known to be slower than other evaluated schemes. We skip OS-based approaches since they are inevitably blocking. We skip PEBR [30] due to significant API differences. We note that PEBR authors only compare against EBR, and PEBR’s performance appears to be 85–90% of EBR’s, worse than that of Hyaline. Since Hyaline aims to achieve excellent throughput while also retaining good memory efficiency, we are comparing against schemes with excellent throughput, such as epoch-based reclamation, and excellent memory efficiency, such as hazard pointers.

We compare all four Hyaline variants against:

- HP – hazard pointers [32].
- HE – hazard eras [39].
- IBR – the interval-based technique 2GE-IBR [44].
- Epoch – a variant [44] of the epoch-based approach.\(^5\)
- No MM – running the test without any memory reclamation, which serves as a general baseline.

In the results, it is more fair to compare Hyaline and Hyaline-1 against (non-robust) Epoch, and Hyaline-S and Hyaline-1S against (robust) HP, IBR, and HE.

The original benchmark code we used [44] implemented snapshots only for IBR. HP and HE were suboptimal due to excessive cache misses when scanning lists of retired nodes. We modified these implementations accordingly. EBR and Hyaline are snapshot-free and do not need this optimization.

Note that the actual throughput can exceed No MM as it can be faster to recycle old objects. As memory deallocation slows down due to a number of factors, including number of freed objects, any memory reclamation scheme can also become objectively slower than No MM.

We use both a write-intensive workload (50% insert, 50% delete), which stresses reclamation techniques through a large number of insertions and deletions, as well as read-dominated workload (90% get, 10% put), which represents a more reclamation-unbalanced and yet common scenario.

For each data point, the experiment starts by prefilling the data structure with 50,000 elements and runs 10 seconds. 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 run the experiment 5 times and report the average.

Reclamation algorithms need to be adjusted to gain good performance. Although this process is tricky, we found more or less reasonable parameters for a fair comparison such that existing algorithms achieve the highest possible throughput while retaining as much of memory efficiency as possible. For our machine, benchmark parameters \( \text{epochf} = 150 \) and \( \text{emptyf} = 120 \) appear to be optimal for existing schemes in this regard. \( \text{epochf} \) amortizes the frequency of epoch counter increments for Epoch, IBR, and HE. \( \text{emptyf} \) reduces other overheads for all algorithms, e.g., amortizing the frequency of list traversals. For Hyaline and Hyaline-S, we cap the number of slots, \( k \), at 128 (the next power of 2 of the number of cores). All variants use batches of at least 64 and at most \( k + 1 \) nodes (as required by the Hyaline algorithms).

Figure 11a shows the throughput of (sorted) Linked-list, which is a good example of an unbalanced workload since operations are slow and dominated by the long traversal required to find an element (even in the write-intensive scenario). Figure 12a, which shows the average number of retired but not-yet-reclaimed objects per operation (allows us to estimate how fast memory is reclaimed), demonstrates that Hyaline has excellent memory efficiency, which is much better than that of Epoch, HE, or IBR. This validates our claim that Hyaline’s efficiency is better in unbalanced settings. All Hyaline variants also have marginally higher throughput than the other schemes. Although HP is also efficient, its throughput is visibly worse due to so many memory barriers incurred while traversing the list. Similar trends are also observed for the read-dominated case (Figures 11d and 12d).

Figure 11b shows hash map’s throughput using the write-intensive workload. Hash map operations are very short and significantly stress memory reclamation systems. Because operations are short, HP’s performance does not degrade...
as much as in the prior test. The gap between No MM and memory reclamation techniques substantially increases as the number of threads begins to exceed the number of cores. However, all Hyaline variants still perform well after 72 threads. (The gap between Hyaline and Epoch gets as large as 2x for 81 threads.) For a smaller number of threads, retirement in Hyaline can be slightly more expensive than in Epoch and IBR. The average number of unreclaimed objects (Figure 12b) for all Hyaline variants is comparable to HP
and smaller than that of IBR, HE, or Epoch before the over-
subscribed scenario (not visible due to a smaller scale). Al-
though it temporarily increases afterwards, the correspond-
ing throughput is also substantially higher than that of other
schemes. Hence, one possible explanation for this increase
is that Hyaline simply allocates and reclaim more objects
(compared to other schemes) in the first place. Since this
workload is already very balanced, Hyaline also does not
get any extra benefit due to reclamation balancing. Hash
map’s results are somewhat more interesting for the read-
dominated case (Figures 11e and 12e), where Hyaline is more
memory efficient than IBR, HE, or Epoch. Hyaline’s through-
put remains very high, even in oversubscribed scenarios.
Natarajan & Mittal tree (Figures 11c, 12c, 11f, and 12f)
shows similar trends to that of hash map. HP is slower due
to longer operations. Throughput gains of Hyaline are more
visible here. With respect to memory efficiency, we see the
same benefit in the read-dominated workload as in hash map.
Before oversubscription, Hyaline’s efficiency is close to HP’s.

Figures 13a and 13b show Bonsai tree’s throughput. HP
and HE are not implemented due to the complexity of the tree
rotation operations [44], for which the number of local point-
ers cannot be determined in advance. Throughput drops for
all schemes as we approach 18 per-socket cores, most likely
due to over-socket contention [44]. Hyaline and Hyaline-1
achieve the best performance and steadily outperform Epoch
by ≈10%. All robust schemes presented for this benchmark
(IBR, Hyaline-S, and Hyaline-1S) have similar performance; it
is worse than their non-robust counterparts due to increased
number of pointer dereferences. The number of unreclaimed
objects (Figures 13c) for Hyaline and Hyaline-S is mostly
smaller than that of Epoch and IBR, respectively.

**Snapshots.** Snapshots also impact memory utilization. For
144 threads and 16 local (concurrently reserved) pointers in
HP and HE, snapshots **additionally require 2.5 MB.** Al-
though this size depends on the number of threads, and the
number of local pointers is typically smaller, we present
this figure to give some perspective. For example, even if
the number of unreclaimed objects is as high as 4000, and
each object is 128 bytes, we still use less than 0.5 MB. To
retain the same evaluation methodology as in prior works,
our results above disregard snapshot overheads. However,
for snapshot-based schemes (IBR, HP, and HE), snapshots
alone can create more memory inefficiency than the scheme
itself, a fact rarely acknowledged in prior works.

7 **Related Work**

A number of approaches for safe memory reclamation (SMR)
were proposed over the last two decades.

Most SMR approaches are either pointer- or epoch- based.
Pointer-based techniques such as hazard pointers (HP) [32]
are typically fine-grained and track every accessed object.
Unfortunately, this approach degrades performance as pointer
dereferencing incurs additional overheads, such as mem-
ory writes and barriers. Pass-the-buck [27, 28] has a simi-
lar model. Drop the anchor [14] is designed specifically for
linked-lists and outperforms hazard pointers, but the ap-
proach is not directly applicable to other data structures.
Optimistic Access [19] and Automatic Optimistic Access
(AOA) [18] are more universal techniques, but they require
data structures to be written in a “normalized form.” FreeAc-
cess [17] drops this requirement and implements a garbage
collector. FreeAccess, however, needs to divide a program
into read-only and write-only periods, which makes it im-
possible to directly use certain operations such as swap.
Or-
eGC [20], another fully lock-free garbage collector, achieves
good performance but is still slower in some tests than HP.

In epoch-based reclamation (EBR) [24, 26], which is based
on the read-copy-update (RCU) [31] paradigm, objects are
marked with the current epoch value at the time they are
retired. A memory object is deallocated only when all thread
reservations are ahead of the object’s retire epoch and no
thread can reach it. Stamp-it [38] extends EBR to guarantee
O(1) reclamation cost but is not robust and requires per-
thread control blocks. It extends EBR by using a doubly-
linked list, and requires ABA tags [29]. Stamp-it squeezes
17-bit tags directly into control block pointers, but for ABA
safety, it is better to use larger tags and double-width CAS.
The hazard eras (HE) approach [39] attempts to reconcile EBR with HP: HE is robust, but uses "eras" (i.e., epochs) instead of pointer addresses to accelerate the algorithm. When allocating memory objects, they are tagged with the birth era, and when objects are retired, they are tagged with the retire era. Lifecycles of objects are controlled by these eras. Similarly to HP’s API model, indices must be assigned to all accessed objects in HE. A subsequent work [36] makes HE wait-free. Interval-based reclamation (IBR) [44] employs the idea of birth and retire eras but forgoes the need to explicitly assign indices making its API model, especially in its 2GE-IBR variant, reminiscent of EBR and easier to use.

Some approaches exploit OS support. PEBR [30] relies on OS tricks [5] to avoid extra memory barriers, which makes it intrusive to execution environments. DEBRA+ [15], NBR [42], and QSense [13] improve EBR to make it robust, but they rely on OS signals or scheduler support. They are robust but not in a fully lock-free manner as typical OSs such as Linux inevitably use locks. ThreadScan [12] and Forkscan [11] are other examples of schemes that rely on signals.

Another approach that is simple to implement but has a high overhead is lock-free reference counting (LFRC) [33, 43]. In this approach, each object is associated with a reference count. An object can be safely reclaimed when the reference count reaches zero. The reference count is updated with every access, which converts read-accesses into write-accesses with a memory barrier. This significantly impacts performance. Hyaline uses a completely different approach, wherein objects are accessed without modifying reference counters. Since active threads are tracked only in the list of retired objects, Hyaline’s overhead is significantly smaller.

Some approaches rely on hardware transactional memory (HTM) to speed up reference counting [22] using HTM transactions. Another approach [10] executes any read operation on the data structure as an HTM transaction. When a conflict occurs in a concurrent thread that reclaims memory, the transaction is aborted. Some other approaches [21] optimize performance by using page protection mechanisms which issue a page fault that forces a global memory barrier.

8 Conclusion

We presented Hyaline, a new lock-free algorithm for safe memory reclamation. Hyaline uses LL/SC or double-width CAS, which are available on most modern architectures. A specialized Hyaline-1 algorithm uses single-width CAS and can be implemented on all architectures. We also presented Hyaline-S and Hyaline-1S extensions, which bound memory usage even in the presence of stalled threads. Compared to other common approaches, the Hyaline schemes balance the reclamation workload due to their underlying asynchronous nature of reclamation. This often manifests in improved memory efficiency without sacrificing performance.

All Hyaline schemes are suitable for environments where threads are created and deleted dynamically: threads are “off-the-hook” as soon as they leave and do not need to check retirement lists afterwards. Hyaline and Hyaline-S are fully transparent as they need not explicitly register or unregister threads; they can allocate a fixed number of slots roughly corresponding to the number of cores and still support any number of threads. Hyaline-1 and Hyaline-1S are less transparent in this sense but can be implemented everywhere.

Hyaline schemes do not take snapshots, which can help reduce memory footprints as the number of threads grow.

We tested all Hyaline versions on x86(-64), ARM32/64, PowerPC, and MIPS architectures. For these architectures, all Hyaline variants exhibit very high throughput on various data structures, and ensure that the number of retired, but not-yet-reclaimed objects is small. We presented results for x86-64, a ubiquitous architecture. Hyaline’s benefits are especially visible in certain read-dominated workloads. Moreover, in oversubscribed scenarios, Hyaline obtains up to 2x throughput gain over other algorithms, including EBR.

Availability

We provide code for the modified benchmark and all Hyaline variants at https://github.com/rusnikola/lfsmr.


Acknowledgments

A preliminary version of the algorithm previously appeared as a brief announcement at PODC ’19 [35].

We would like to thank the anonymous reviewers and our shepherd Tony Hosking for their insightful comments and suggestions, which helped greatly improve the paper. 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 and ONR under grants N00014-18-1-2022 and N00014-19-1-2493.

References


We keep ARM32 MPCore+ and ARM64 [2] implement double-width A Hyaline(-S) for Single-width LL/SC modify as false negatives are impossible unless concurrent threads looping SC; single-width atomicity for failures is acceptable HRef than atomicity is guaranteed only when SC succeeds. Our algo-...