Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Andreia Correia
Concurrency Freaks
andreiacraveiroramalhete@gmail.com

ABSTRACT

For non-blocking data-structures, only memory reclamation with pointer-based techniques can maintain non-blocking progress, but there can be high overhead associated to these techniques, with the most notable example being Hazard Pointers.

We present a new algorithm we named Hazard Eras, which allows for efficient lock-free or wait-free memory reclamation in concurrent data structures and can be used as drop-in replacement to Hazard Pointers. Results from our microbenchmark show that when applied to a lock-free linked list, Hazard Eras will match the throughput of Hazard Pointers in the worst-case, and can outperform Hazard Pointers by a factor of 5x. Hazard Eras provides the same progress conditions as Hazard Pointers and can equally be implemented with the C11/C++11 memory model and atomics, making it portable across multiple systems.

CCS CONCEPTS

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

KEYWORDS

memory reclamation; hazard pointers; lock-free; wait-free; non-blocking; concurrent data structures

1 INTRODUCTION

Concurrent data structures are often measured on two vectors: the throughput they provide and the progress they guarantee. Data structures with lock-free progress are not that common, and with wait-free characteristics even less. To make things worse, the progress conditions of the memory reclamation technique can further reduce the progress of either *readers* (threads calling methods that de-reference pointers in the data structure) or *reclaimers* (threads calling methods that attempt to reclaim and delete an object/node in the data structure). Existing techniques for manual concurrent memory reclamation fall into one of three groups: *quiescence-based*, *reference counting*, and *pointer based*.

Quiescence-based techniques, like the Epoch-based by Fraser [3], Harris [5], or Userspace RCU [6], reclaim memory whenever readers pass though a *quiescent* state in which no reader holds a reference to a shared object. These techniques have light synchronization and can be wait-free for readers, but their throughput is significantly impacted by delays on readers, which can block the reaching of quiescent states, thus preventing any other thread from reclaiming memory [2]. Moreover, they can have an unbounded amount of unreclaimed memory, which can fatally exhaust all memory available to the application when there is a slow reader, or just because there is high oversubscription (more threads than cores to run them on).

Reference counting techniques [4] require expensive synchronization on the readers side [6], cause contention among readers and have several limitations as described in chapter 9.1 of [8].

Pointer-based techniques, such as Hazard Pointers [9], Pass The Buck [7], or Drop The Anchor [1], explicitly mark live objects (objects accessible by other threads) which can not be de-allocated. These techniques are wait-free for reclamation and it is possible to use them in a wait-free way for readers for some algorithms [11], but they are typically deployed lock-free.

2 OVERALL DESIGN

Hazard Eras (HE) combines the low synchronization overhead of epoch-based techniques with the non-blocking properties of Hazard Pointers (HP) for both readers and reclaimers, while providing the same API as Hazard Pointers, specifically, the API that is currently being proposed to the C++ standard library [10].

In HE, the object's lifetime is tracked with a global monotonic clock, the `eraClock`. When an object is created, the current value of `eraClock` is stored in `object.newEra`, and when the object is retired, the current value of `eraClock` is stored in `object.delEra`, and subsequently the `eraClock` is atomically incremented. Every time an object's lifetime arrives at an end, the current era ends and a new era begins.

Unlike HP, where readers publish the pointer that they are using, in HE the reader publishes the *era* that was in `eraClock` at the point in time when the hazardous reference was read. One era must be published for each different pointer in use, just like on HP.

A reader that publishes an era with a value of x is guaranteeing that (after re-validation of `eraClock`) no object with a lifetime that encompasses x will be deleted. By definition, all objects currently in a *live* state are now protected from deletion, however, objects created after this era may be subsequently deleted, which is not possible in Epoch-based reclamation. In Epoch-based memory reclamation, all objects retired *after* a reader has started, will not be deleted unless the reader completes, because there is still an ongoing reader and it may be accessing those objects.

In HE, all objects created with a `newEra` greater than the highest of the published eras by all readers, can be retired and deleted. The algorithm guarantees that readers which published a precedent era can not have access to the objects of a higher era, and to be able to access those objects they will observe the era has changed and

SPAA '17, July 24-26, 2017, Washington DC, USA

ACM ISBN 978-1-4503-4593-4/17/07.
https://doi.org/10.1145/3087556.3087588

Algorithm 1 Hazard Eras class

```
1  template<typename T> class HazardEras {
2
3  private:
4    static const uint64_t NONE = 0;
5    const int maxHEs;
6    const int maxThreads;
7    std::atomic<uint64_t> eraClock = { 1 };
8    std::atomic<uint64_t> he[MAX_THREADS][MAX_HES];
9    std::vector<T*> retiredList[MAX_THREADS];
10
11 public:
12   HazardEras(int maxHEs, int maxThreads) :
13               maxHEs{maxHEs}, maxThreads{maxThreads} {
14     for (int ith = 0; ith < MAX_THREADS; ith++) {
15       for (int ihe = 0; ihe < MAX_HES; ihe++) {
16         he[ith][ihe].store(NONE, std::memory_order_relaxed);
17       }
18     }
19   }
20
21   ~HazardEras() {
22     for (int ith = 0; ith < MAX_THREADS; ith++) {
23       for (auto i = 0; i < retiredList[ith].size(); i++) {
24         delete retiredList[ith][i];
25       }
26     }
27   }
```

will be forced to re-read the atomic variable corresponding to the pointer and then re-publish the most recent era.

3 HAZARD ERAS ALGORITHM

In our implementation, the HazardEras class is composed of three objects, a global clock named `eraClock`, a bidimensional array of *eras* named he, and an array of lists named `retiredList`, as shown in Algorithm 1. The era timestamps are 64 bit integers so that they are large enough not to cause ABA issues and still be used atomically on recent CPUs. An instance of type `T` has two eras associated to it: the era of its birth `newEra`, which indicates the moment the instance was made visible to other threads, and the era of its death `delEra`, where `delEra+1` indicates the moment in time after which it is no longer visible for new accesses to objects on the data structure. The birth era of each object must be set on `newEra` before the object is made visible to other threads, i.e. inserted in the data structure, which can be easily done in the constructor of `T` or one of its base classes. The death era follows a reversed procedure, where the object is first removed from the data structure and only then is the `eraClock` read and its value stored in `delEra`.

Similarly to Hazard Pointers [10], in Hazard Eras we have three main APIs (`get_protected()`, `clear()`, `retire()`) plus one extra (`getEra()`). The method `retire()` is called during a reclamation

Algorithm 2 Reader's API

```
28 T* get_protected(std::atomic<T*>& atom, int index, int tid) {
29   auto prevEra = he[tid][index].load(std::memory_order_relaxed);
30   while (true) {
31     T* ptr = atom.load();
32     auto era = eraClock.load(std::memory_order_acquire);
33     if (era == prevEra) return ptr;
34     he[tid][index].store(era);
35     prevEra = era;
36   }
37 }
38
39 void clear(const int tid) {
40   for (int ihe = 0; ihe < maxHEs; ihe++) {
41     he[tid][ihe].store(NONE, std::memory_order_release);
42   }
43 }
44
45 int64_t getEra() { return eraClock.load(); }
```

operation, done by *reclaimers*, while the other three methods are called during a read operation, done by *readers*.

The method `get_protected()` in Algorithm 2 shows a lock-free loop for protecting an hazardous reference. There is a kind of *fast-path-slow-path* approach. When the `eraClock` has not changed from the previous published value (line 33), there is no need to publish again the same era value and pay the synchronization cost of doing a sequentially-consistent (seq-cst) store (line 34). By following this fast-path, the reader does two seq-cst loads instead of the two seq-cst loads and one seq-cst store that are needed for HP. This may seem a minor gain, however, seq-cst loads have an almost free cost on x86 architectures, having very little overhead, and therefore, providing high throughput for the readers. Even on non-x86 architectures, the price of doing a seq-cst load relative to a seq-cst store is much lower, thus providing higher throughput also on architectures like PowerPC and ARM. In the object's constructor or before inserting it in the data structure, the return value of `getEra()` must be placed in `object.newEra`.

The `retire()` method saves the current era in `object.delEra` and puts the object in the thread's `retiredList`. Then, to guarantee progress, it will advance the `eraClock` if another thread has not done so in the meantime (line 51), and scan the `retiredList` for objects that can be safely deleted. If there is at least one thread with a published era in the range `[newEra;delEra]` (line 68) the object can not be deleted yet.

A rarely mentioned advantage of HP is its low bound on memory usage. Quiescent-based techniques with *delegation* or *deferral* typically have no bound on memory usage. When using HP with an R factor of 1, there is the guarantee that there are at most $\texttt{MAX\_THREADS} \times \texttt{MAX\_HPS}$ objects in the retired list of each reclaimer, and therefore, there may be at most $\texttt{MAX\_THREADS}^2 \times \texttt{MAX\_HPS}$ retired objects waiting to be deleted.

Algorithm 3 Reclaimer's API

```
46 void retire(T* ptr, const int mytid) {
47   auto currEra = eraClock.load();
48   ptr->delEra = currEra;
49   auto& rlist = retiredList[mytid*CLPAD];
50   rlist.push_back(ptr);
51   if (eraClock == currEra) eraClock.fetch_add(1);
52   for (unsigned iret = 0; iret < rlist.size();) {
53     auto obj = rlist[iret];
54     if (canDelete(obj, mytid)) {
55       rlist.erase(rlist.begin() + iret);
56       delete obj;
57       continue;
58     }
59     iret++;
60   }
61 }
62
63 private:
64 bool canDelete(T* obj, const int mytid) {
65   for (int tid = 0; tid < maxThreads; tid++) {
66     for (int ihe = 0; ihe < maxHEs; ihe++) {
67       const auto era = he[tid][ihe].load();
68       if (era < obj->newEra || era > obj->delEra ||
69           era == NONE) continue;
70       return false;
71     }
72   }
73   return true;
74 }
75 };
```

Hazard Eras' upper bound is limited to the number of objects that were in the data structure at a given clock era published in the hazard eras array, for all reader threads. At a given time t, the bound on the maximum number of unreclaimed objects is given by:

$$\#\left\{ \bigcup_{\substack{x \in X(t) \\ era \in \text{HEs(t)}}} x : x.\text{newEra} \leq era \leq x.\text{delEra} \right\} \quad (1)$$

where HEs(t) is the set of all clock eras published by readers at time t, X(t) is the set of objects created until `clockEra` at time t. By definition, the `delEra` of a live object is the highest possible value of `eraClock`.

The number of objects that can remain unreclaimed on each `era` becomes limited as soon as a memory reclamation event occurs (a call to `retire()`). Only the latest `era` can have an unbounded amount of live objects, and any unreclaimed objects are always on previous eras. As such, it is not possible for the program to allocate an unbounded number of objects in a single `era`, except on the latest `era` which only includes live objects. Depending on the number of such objects and the number of threads, the bound for HE may be higher, or lower than the bound for HP.

4 CONCLUSION

To the untrained eye, it may look as though there is little difference between an Epoch-based memory reclamation and Hazard Eras. However, in Epoch-based reclamation, each thread does one single publishing of the global epoch it saw per method call, causing it to have a small synchronization cost, but *unbounded* memory usage: a single sleeping or blocked reader is enough to prevent *any further memory reclamation*, even in variants of Epoch-based reclamation where the epoch is updated regularly by the readers. In Hazard Eras, each reader thread publishes the global era it saw, for each new pointer that is accessed, if and only if the era has changed, which incurs a small synchronization cost for each pointer. Furthermore, HE have *bounded* memory usage: a sleeping or blocked reader may prevent all currently allocated objects from being reclaimed, but newly allocated objects can be subsequently reclaimed.

Compared with Hazard Pointers, Hazard Eras have the same deployment complexity, with a lower synchronization cost for the readers, which gives HE up to **5**x the throughput of HP. This increase in throughput comes with a price tag: higher memory usage. Unlike HP, HE requires each tracked object to have a `newEra` once created, a `delEra` once deleted, and the number of objects in memory which have been retired but not yet deleted, although finite, can be higher for HE than for HP.

Hazard Eras fall between Epoch-based and Hazard Pointers, providing the best characteristics of each: high throughput due to its low synchronization; non-blocking progress for readers and reclaimers; and a bound on memory usage.

REFERENCES

[1] Anastasia Braginsky, Alex Kogan, and Erez Petrank. 2013. Drop the anchor: lightweight memory management for non-blocking data structures. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 33–42.

[2] Dave Dice, Maurice Herlihy, and Alex Kogan. 2016. Fast non-intrusive memory reclamation for highly-concurrent data structures. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*. ACM, 36–45.

[3] Keir Fraser. 2004. *Practical lock-freedom*. Ph.D. Dissertation. University of Cambridge.

[4] Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. 2009. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems* 20, 8 (2009), 1173–1187.

[5] Timothy L Harris. 2001. A pragmatic implementation of non-blocking linked-lists. In *Distributed Computing*. Springer, 300–314.

[6] 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.

[7] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2002. The repeat offender problem: A mechanism for supporting dynamic-sized lock-free data structures. (2002).

[8] Paul E McKenney. 2011. Is parallel programming hard, and, if so, what can you do about it? *Linux Technology Center, IBM Beaverton* (2011).

[9] Maged M Michael. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on* 15, 6 (2004), 491–504.

[10] Maged M. Michael, Michael Wong, Paul McKenney, Arthur O'Dwyer, and David Hollman. 2017. Hazard Pointers - Safe Resource Reclamation for Optimistic Concurrency. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0233r3.pdf. (2017).

[11] Pedro Ramalhete and Andreia Correia. 2017. POSTER: A Wait-Free Queue with Wait-Free Memory Reclamation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 453–454.