Hazard Eras - Non-Blocking Memory Reclamation

Pedro Ramalhete[1] **and Andreia Correia**[2]

1 **Cisco Systems**
pramalhe@gmail.com
2 **Concurrency Freaks**
andreiacraveiroramalhete@gmail.com

Abstract

Dynamic memory reclamation is perceived to be the biggest open problem in concurrent data structure design. For non-blocking data-structures, only 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 a multitude of systems.

1998 ACM Subject Classification D.3.4 - Processors - Memory Management (garbage collection)

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

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Concurrent data structures are often measured on two vectors: the throughput they provide and the progress they guarantee. All software engineers aim for high throughput and they generally desire a data structure that provides lock-free progress or, in scenarios where the tail latency is important, strive for wait-free progress.

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). *Automatic* memory management greatly simplifies the design of concurrent data structures [9, 17]. However, while the existence of lock-free garbage collection has been demonstrated [15], and there has been much work on automatic garbage collectors that obtain some partial guarantee for progress [15, 16, 25, 24, 2], there is no garbage collection that supports wait-free execution in the current literature [23]. Known schemes for *manual* concurrent reclamation of unused objects can be difficult to use, or impose a significant overhead on the execution, or both. 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 [10], Harris [12], or Userspace RCU [13] 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 [8].

Epoch-based and Userspace RCU (URCU) memory reclamation work on the principle that reclaimers wait for ongoing readers to complete before deleting an object. Such techniques are inherently blocking, even if used with *deferral* or *delegation*.

When using *deferral*, an object that may still be in use by a reader is put in a list, thus deferring the deletion to a later time. This is blocking for reclaimers because a non-progressing reader will cause all objects meant to be reclaimed to be put on the deferred list, which will grow *unbounded*, eventually exhausting all memory available to the application.

When using *delegation*, an object that may still be in use by a reader is put in a list associated with reader that may be using it. Typically, the last reader to end its critical section will reclaim all associated objects. Again, memory exhaustion can occur due to a non-progressing reader.

When using neither deferral nor delegation, a reclaimer blocks waiting for ongoing readers to complete before safely deleting an object. In this case, at any given time there can be $O(N_{threads})$ objects waiting to be deleted.

For a data structure or memory reclamation technique to be lock-free (or wait-free) all of its methods must be lock-free (or wait-free), further implying that it is possible to make progress even in the presence of faults. Although URCU and Epoch-based reclamation are typically wait-free for readers, they are blocking for reclaimers, even when using deferral or delegation. In other words, a single blocked or sleepy reader is enough to prevent any further memory from being reclaimed.

Reference counting techniques [28, 11] require expensive synchronization on the readers side [13], can have contention among the readers, at best are wait-free only on architectures with a native instruction for `fetch_add()`, and they have several limitations as described in chapter 9.1 of [18]. The solution by Valois [28] can not be used for memory reclamation, allowing only the re-usage of objects while preventing the ABA problem [21].

In reference counting, each reader does one atomic to increment the counter for each object that is to be used, and another atomic decrement to the counter for each object that is no longer in use. This incurs a high cost in synchronization, causing low throughput for readers and low scalability. On [28] a `fetch_add()` can be used for the atomic increment and decrement, which on x86 results in a single atomic instruction, implying that the readers are wait-free population oblivious (wfpo). Other approaches [11] are lock-free regardless of the architecture. In reference counting it is up to the last reader using an object the responsibility of deleting it, an operation that is not always lightweight, but it guarantees that at any given time there is at most a finite bound on the number of objects to be deleted per thread.

Pointer-based techniques, such as Hazard Pointers [20], Pass The Buck [14], or Drop The Anchor [3], explicitly mark live objects (objects accessible by other threads) which can not be de-allocated. These techniques can be wait-free for reclaimation and it is possible to use them in a wait-free way for readers in some situations [26], but they are typically deployed lock-free. Pointer-based schemes suffer from two limitations: they must be customized to the data structure at hand, which makes them difficult to deploy; they publish each pointer that is used in a shared memory location, which is expensive in terms of synchronization.

	Readers progress	Reclaimers progress	Bound on memory usage	Average per-node synchronization
Reference Count	lock-free/wfpo	lock-free/wfb	$O(N_{threads})$	2 `fetch_add()`
Epoch-based	wfpo	blocking	unbounded	minor
Userspace RCU	wfpo	blocking	$O(N_{threads})$	minor
Hazard Pointers	lock-free/wfb	wfb	$O(N^2_{threads})$	2 `load()` + 1 `store()`
Drop the Anchor	lock-free	lock-free	$O(_{AnchorInterval} \times N^2_{threads})$	2 `load()`
Hazard Eras	lock-free/wfb	wfb	finite (equation 1)	2 `load()`

Table 1 Progress conditions and synchronization cost of different memory reclamation techniques.

Hazard Pointers (HP) and other pointer-based techniques will typically publish the pointer to each object they use, and then check that the pointer has not changed in the meantime. Such approach guarantees that an object which has been deleted will not be later de-referenced, at the cost of each reader doing synchronization on a per-object basis. In the C/C++ Memory Model, this implies doing two sequentially consistent (seq-cst) loads for every object that is traversed and one sequentially consistent store, using the default memory order of `memory_order_seq_cst`. Variants like Drop The Anchor, trade off complexity of deployment for throughput, and need to do it just for a subset of the objects/nodes, the anchor nodes, but the concept is the same. Because of this, Drop The Anchor is capable of a throughput that can go up to 5x the throughput of HP, however, it is not as generic as HP [3] and can be substantially harder to deploy correctly.

Because it requires validation of the pointer that will be accessed next, Hazard Pointers are lock-free for readers, although in some situations they can be made wait-free for readers. HP is wait-free bounded for reclamation, with the bound being proportional to the number of threads times the number of hazard pointers, because each reclaimer has to scan all the hazard pointers of all the other threads before deleting a node. In HP the retired nodes are placed in a retired list which is scanned once its size reaches an `R` threshold. In terms of memory usage, when the `R` factor is set to the lowest setting of 1, each reclaimer can have at most a list of retired nodes with a size equal to the number of threads minus 1, times the number of hazard pointers. If each thread has one such list of nodes pending to be deleted, at any given point in time there are at most $O(N^2_{threads})$ nodes to be deleted.

Table 1 shows a comparison of some of the characteristics of well known memory reclamation techniques that need solely the C++ Memory Model, without requiring OS support, or signals, or special hardware instructions, or transactional memory. The second and third rows are quiescent based techniques and the three bottom rows are pointer based techniques. The first and second column represent the progress condition of the call to a pointer protection and pointer reclamation procedures respectively. The third column gives a bound on memory usage, where epoch-based is assumed to be using either unbounded deferral or delegation and therefore be unbounded, and URCU is assumed to use neither or use bounded deferral. The fourth column shows the amortized number of atomic operations needed by a reader to protect a node.

Recent work [5] employed hardware transactional memory to speed up common components of known memory reclamation schemes. Other techniques exist that use OS-specific functionality, like RCU's Linux kernel implementation, or uses POSIX signals [4, 1], or they use special CPU instruction or capabilities [8], or require custom allocators [6]. Our goal is to achieve a generic algorithm whose implementation relies solely on an atomics API and a memory model like the one in C11/C++11, so as to make it OS and CPU agnostic, and therefore, we do not consider techniques that require functionality outside the C++ Memory Model.

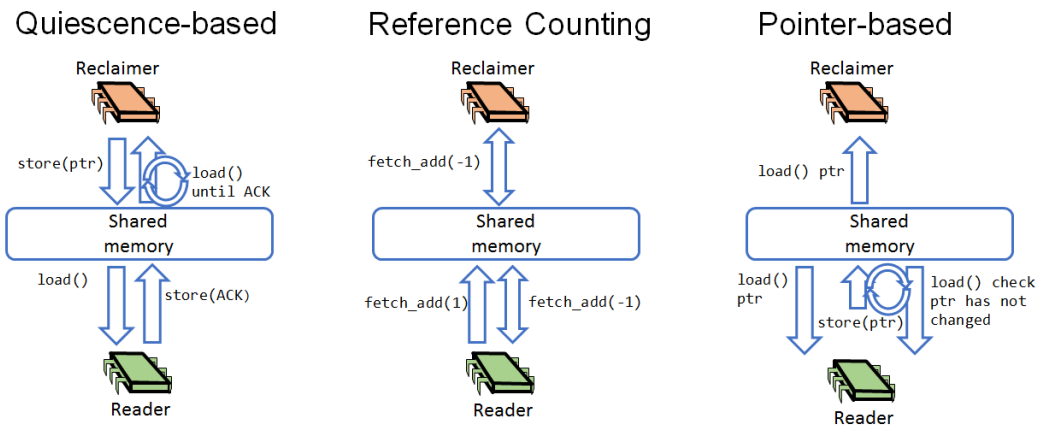

Figure 1 The three different types of memory reclamation.

At its essence, memory reclamation is a problem that involves (at least) two threads, the *reader* and the *reclaimer*, where the reclaimer needs to know when the reader will no longer access a certain resource. In order to do this, some kind of *communication* must occur between the two threads, which in the Memory Model is done with atomic stores and atomic loads. Figure 1 shows a schematic of the three different families of memory reclamation.

On the left side of figure 1 is a typical *quiescent based* technique. The reclaimer needs to advertise the pointers that are safe (or unsafe) to be used by the reader, and then wait for an acknowledgment from the reader. Due to this wait, this technique is implicitly blocking for reclaimers. In practice, few or no algorithms actually publish the pointers themselves, publishing instead a *proxy* of the pointer(s), such as an epoch of when the object was made inaccessible, which the reader then publishes to shared memory to let the reclaimer know the change has been seen and pointers previous to the change in epoch are no longer in use.

In the middle of figure 1 is a simplistic *reference counting* technique. The reader atomically increments a counter in the object before using it and atomically decrements it when it will no longer de-reference the pointer. The reclaimer atomically decrements the counter. Upon atomically decrementing, the thread that sees 0 is responsible for deleting the object. As mentioned before, this technique is slow for readers and has several limitations [21].

On the right side of figure 1 is a *pointer based* technique. The reader publishes each pointer before using it, then checks the pointer has not changed. The reclaimer scans all pointers in use by the reader and can not delete an object if there is still a reader using it.

2 Overall Design

The idea behind Hazard Eras (HE) is to combine the low synchronization overhead of epoch-based techniques with the non-blocking properties of Hazard Pointers for both readers and reclaimers, while providing the same API as Hazard Pointers, which is currently being proposed to the C++ standard library [22]. In HE we track the object's lifetime 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.

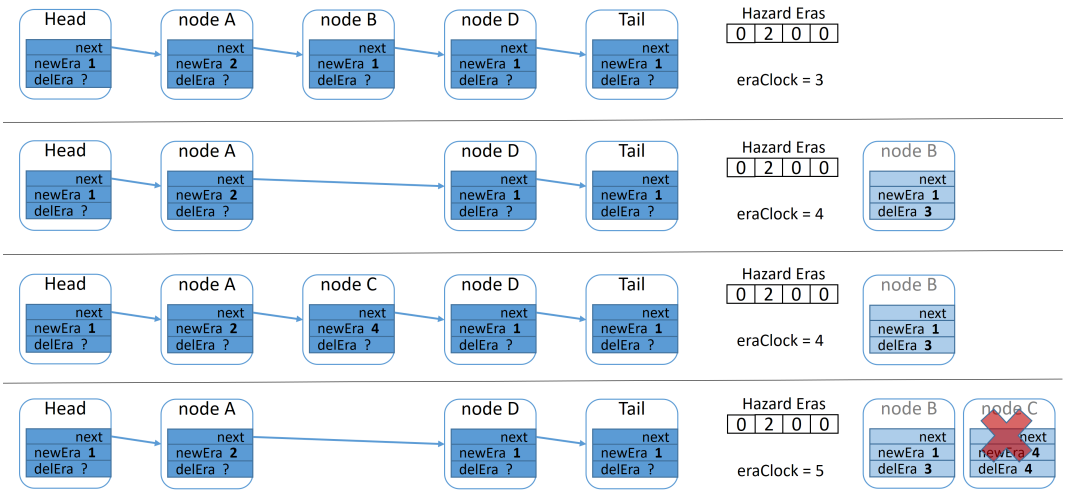

Figure 2 Four schematics depicting the removal of nodes B and C. Time flows top to bottom.

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. For example, on the Maged-Harris list [19], three hazard pointers are required to track traversals on the list and therefore, three hazard eras will be required as well.

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. In other words, no object with a `newEra` equal or lower than x **and** a `delEra` equal or higher than x can 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 will be forced to re-read the atomic variable corresponding to the pointer and then re-publish the most recent era. In other words, when a thread reads the global era and then reads an hazardous reference and publishes the latest read value of the global era (after it has confirmed that the global era has not changed), we are certain that the object to which the hazardous reference is referencing can not be deleted. Furthermore, an *era* value that has been published and validated in the array of hazard eras, protects *all* objects whose lifetime encompass it, i.e. protects all objects for which $newEra <= era <= delEra$.

Similarly to all other pointer-based techniques, an object that is no longer accessible from shared memory must have all its pointers to other objects in shared memory invalidated before being retired. In the case of the Maged-Harris list, this is done when the bit is set on the next pointer of the node, but other data structures will often use self-linking of the nodes. In other words, before being retired, an object can no longer be accessible from any other object in the data structure, nor can it provide access to another object in the data structure. This is a necessary condition for both HP and HE techniques, as well as all other

pointer-based techniques we're aware of.

Figure 2 shows schematics for an example of what happens when nodes are removed from a list. The topmost schematic shows that the list starts with three nodes A, B, and D, with the `eraClock` at 3 and there is a reader that published the era 2 in its entry in the array of hazard eras. A thread removes node B from the list, saves the current era on the `delEra` of node B, and retires it, causing the `eraClock` to advance to 4. Then, on the third schematic, a new node C is inserted in the list, with a `newEra` equal to the current `eraClock` of 4. Notice that node B can not yet be deleted because a reader thread has published an era of 2 and it might be accessing node B. On the bottom schematic node C is removed and its `delEra` will be 4, the `eraClock` goes to 5. Node C can be immediately deleted because there is no reader that can access it.

3 Hazard Eras Algorithm

The C++14 code shown throughout this text assumes the usage of a language with a sequentially consistent memory model or equivalent [7], like C11 or C++1x.

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` (of type `std::vector`), 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 to 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`. In this implementation, the type `T` must have the members `newEra` and `delEra`, both of type `uint64_t`. Neither of these variables needs to be atomic because they are only read after being placed in a retired list, by the thread that put them there.

Similarly to hazard pointers [22], in hazard eras we have three main APIs, plus one extra:

- `get_protected()`: Publishes the current `eraClock` value and returns a pointer to a protected object. Any instance `T` whose lifetime encompassed that era, can not be deleted. It runs in a lock-free loop which completes once a publishing/validation is achieved. See Algorithm 2.
- `clear()`: Clears all hazard eras in use by the calling thread. It is meant to be called when no pointer is being used. See Algorithm 2.
- `retire()`: Retires an object of type `T`. This object must no longer be accessible from a shared memory location by the time this method is called. If there are no readers active with an era published within the visibility window of the object defined by `[newEra;delEra]`, the object will be deleted immediately, otherwise, it is placed on a list of retired objects until a later time when it is safe to delete the object. See Algorithm 3.
- `getEra()`: Returns the current value of the global era `eraClock`. Its returned value is meant to be stored in the member `newEra` of a newly created object of type `T`.

The methods `get_protected()`, `clear()` and `getEra()` are called during a read operation, done by *readers*, and the method `retire()` is called during a reclamation operation, done by *reclaimers*. The first three methods have the same signature and semantics as the equivalent

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   }
```

methods in HP. The method `getEra()` is unique to Hazard Eras and its return value is meant to be stored in a newly created object before the object is made visible to other threads, i.e. before it is inserted in the data structure. Although our implementation has extra arguments (`index` and `tid`) relative to the one currently being proposed to the C++ standard [22], it can be adapted to use the exact same API.

The method `get_protected()` in Algorithm 2 shows a lock-free loop for protecting an hazardous reference using Hazard Eras. The first argument is the atomic variable that is to be read. The second argument is the `index` to identify which of the hazard eras we're publishing. The third argument is the id of the thread, used to identify its entry in the array of hazard eras, where a thread-local variable can be used instead. There is a kind of *fast-path-slow-path* approach. When the `eraClock` has not changed from the previous published value (line 33) then there is no need to publish again the same era value and pay the synchronization cost of doing a seq-cst store (line 34). By following this fast-path, we do only 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 the seq-cst load relative to seq-cst store is much lower, thus providing higher throughput also on non TSO architectures like PowerPC and ARM. Notice we don't count the load for `prevEra` because it is relaxed and can even be replaced with a stack variable.

The `retire()` method will save the current era in `object.delEra` and put the object in the `retiredList` of the current thread. Then, to guarantee progress, it will advance the `eraClock` if another thread has not done so in the meantime (line 50), 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 58) the object can not be deleted yet.

Algorithm 2 Reader's API

```
28 T* get_protected(std::atomic<T*>& atom, int index, int tid) {
29   auto prevEra = he[tid][index].load(memory_order_relaxed);
30   while (true) {
31     T* ptr = atom.load();
32     auto era = eraClock.load(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, memory_order_release);
42   }
43 }
44
45 int64_t getEra() { return eraClock.load(); }
```

Algorithm 3 Reclaimers API - retire()

```
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   rlist.erase(
53     std::remove_if(
54       rlist.begin(),
55       rlist.end(),
56       [this] (T* obj) {
57         for (int tid = 0; tid < maxThreads; tid++) {
58           for (int ihe = 0; ihe < maxHEs; ihe++) {
59             const auto era = he[tid][ihe].load();
60             if (era == NONE || era < obj->newEra || era > obj->delEra) continue;
61             return false;
62           }
63         }
64         delete obj;
65         return true;
66       }),
67     rlist.end());
68   }
69 }
70 };
```

3.1 Bound on memory usage

One of the rarely mentioned advantages 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 `MAX_THREADS` × `MAX_HPS` objects in the retired list of each reclaimer, and therefore, there may be at most `MAX_THREADS` $^2\times$ `MAX_HPS` retired objects waiting to be deleted. Drop The Anchor [3] has an higher bound proportional to the maximum number of objects between two anchors.

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:

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

where HEs(t) is the set of all clock eras published in the hazard eras array 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`. 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.

If we consider a constant flow of live objects being added to the data structure at `eraClock` i, during which no object is removed. All these objects are alive and there is currently no unreclaimed object. As soon as one object is removed from the data structure and retired, the `eraClock` will be advanced to $i + 1$, thus limiting the number of unreclaimed objects at era i. From this moment on, the constant flow of added abjects to the data structure can continue, but these objects will be created already in the `eraClock` $i + 1$. It is the existence of a memory reclamation event (a call to `retire()`) that gives a bound to the number of objects that can remain unreclaimed. 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, unless it is the lastest `eraClock` and at that era it still hasn't occurred a memory reclamation event so there isn't any unreclaimed objects for the latest `eraClock`.

Another way to validate that there is a bound to the number of unreclaimed objects, is to consider the scenario where all threads (minus one) die after publishing an era in the HE array. This will prevent the reclamation of objects whose lifetime includes any of the published eras. However, all new objects will have a lifetime which will not contain any of the published eras, therefore, they will not be prevented from being reclaimed. This implies that the amount of objects which can potentially remain unreclaimed is known, and can even be calculated if an inspection is done to the data structure, counting the number of objects whose lifetime includes any of the published eras.

3.2 Progress Conditions

A method is said to be *wait-free bounded* if there is a finite and known bound on the number of steps required to complete it. The `getEra()` method has no loops, and therefore has a constant number of steps, meaning it is wait-free population oblivious. The `clear()` method has a `for()` loop bounded by the number of threads, thus being wait-free bounded. The method `get_protected()` contains a loop that could go on indefinitely if the `clockEra` keeps changing at each iteration, but when `clockEra` changes, it means that some other thread has made progress, and therefore, this method is lock-free. This implies that *readers* in HE using `get_protected()` are *lock-free*, like on HP. Similarly to HP, it is possible to use HE in a wait-free algorithm [26], maintaining its wait-free progress.

`retire()` has no retries and will always complete in a finite number of steps, meaning it is wait-free. Reclaimers calling `retire()` may have to traverse a large list containing all the reclaimed objects that can not yet be deleted. This list is finite in size and there is no constant bound to it. However, we know that the maximum number of retired objects which can not be deleted is bounded by (1) and therefore, the progress for `retire()` is *wait-free bounded*.

3.3 Correctness

The correctness of a safe memory reclamation algorithm is attained if the following condition holds: An object's memory can only be deleted if and only if no reader will be allowed to access the object's contents. The object is considered to be unique, with a visible lifetime that starts at `object.newEra` and ends at `object.delEra`. Even if its memory location may be re-used by another object, the new object's visible lifetime will not overlap the visible lifetime of any other object at the same memory location.

Invariant 1. *A reader willing to access the contents of `object` will have to publish the current `eraClock`, which is comprised between `object.newEra` and `object.delEra`*
By design, before the access of `object`'s contents, a reader will call `get_protected()`, thus publishing the current `eraClock`. In order to guarantee that the published era is within the `object`'s visible lifetime, between `object.newEra` and `object.delEra`, when calling `get_protected()`, the load of the reference to `object` from shared memory has to be done before validating `eraClock` has not changed. Let us consider that the `eraClock` on re-validation is x. The load of `object` before the load of `eraClock` gives the guarantee that the `object` was accessible on era x, because `eraClock` has not changed. If the `object` is still accessible it means its `object.delEra` is not yet filled and will always be greater or equal than x. Also, `object.newEra` will have to be equal or less than x because it is always filled before the `object` is placed in the data structure and made accessible to other threads.

Invariant 2. *A reader with a published era that is lower than `object.newEra` can not have access to the `object`'s contents*
The value stored in `object.newEra` is the era value taken from the global `eraClock`. It is only after the assignment to `object.newEra` that the `object` will be placed in the data structure and made accessible to other threads. This implies that once the `object` is visible to other threads, the `eraClock` will have to be equal or higher than `object.newEra`. Also, every access to the contents of an object on the data structure is protected by publishing the current `eraClock`, Invariant 1, thus forcing any thread attempting to access the contents of `object` to publish an era equal or higher than `object.newEra`. If the reader published an era prior to `object.newEra` then it is impossible for it to have access to `object`, unless it re-publishes an era within the `object`'s lifetime.

Invariant 3. *A reader with a published era that is higher than `object.delEra` will never access `object`*
For any thread wishing to retire an `object`, it will first make sure the `object` is no longer accessible through the data structure, and afterward, it will store in `object.delEra` the era value taken from the global `eraClock`. After the assignment to `object.delEra`, the thread that wishes to delete the `object` will make sure `eraClock` is higher than the value published in `object.delEra`, by incrementing it or by verifying that other threads have already done so. This implies that all readers with a published era higher than `object.delEra` can not have access to `object`, because `object` was no longer visible from `eraClock` at `object.delEra`+1.

Invariant 4. *A reclaimer will only be allowed to free the memory allocated to `object` if and only if no reader will be allowed to access the contents of `object`.*
A thread wishing to free an object's memory has to guarantee no reader is allowed to access that object. From Invariant 2, any reader publishing an era lower than `object.newEra`, can not access the `object`'s contents. Even if it retries to publish, the current `eraClock` is already

higher than `object.delEra` because, before checking the published eras, the reclaimer thread has made the object inaccessible and made sure `eraClock` has advanced to a value higher than `object.delEra`. Also, from Invariant 3, any reader that has published an era higher than `object.delEra` is guaranteed to never access `object`. Therefore, any thread willing to free an object's memory will only do so after verifying all reader threads have a publish era outside of the `object`'s visible lifetime, making sure from that moment on, no thread will access the contents of `object`.

3.4 Further Improvements

We now explain some of the improvements that can be done on the Hazard Eras technique.

The object does not need a `delEra` member. The `delEra` requirement can be foregone when using a *map* associating the pointer with the era of when the object was deleted. This typically involves memory allocation on the `retire()` method, therefore we decided in our implementation to have it as part of the object.

Similarly to `delEra`, `newEra` does not need to be a member of the object and can instead be placed in a concurrent map which maps pointers to eras. The memory usage is however still there, and the need of a concurrent map that maintains the desired non-blocking progress conditions makes this a non-trivial problem, therefore, we decided in our implementation to make `newEra` part of the object.

One possible optimization to HE is to advance the `eraClock` less often. Instead of advancing for every call to `retire()`, a thread-local counter can be kept that only increments the `eraClock` if it has not advanced after a certain k number of calls to `retire()`. Having the global era change less often will cause the readers to have to do a seq-cst store with the updated era fewer times, thus improving their throughput. The disadvantage of this approach is that it will have more objects in memory waiting to be removed because if the era does not advance for k times, then there will be k times more objects waiting to be reclaimed, with a proportionally higher bound on memory usage.

When deploying Hazard Pointers in some complex concurrent data structures it can sometimes happen that the number of hazard pointers needed may be large, which can reduce throughput considerably. One such example is when doing traversals on binary trees that may require protecting all the nodes from the root to the leaf. Hazard Eras can be modified to publish only the highest and lowest of the eras which further improves throughput over HP. An internal per-thread list of hazardous eras must still be kept on the reader side so as to publish the lowest and highest of the eras, and only when unchanged. In line 60 of `retire()` we would use the condition shown below, preventing the object from being deleted when true, where $minEra_i$ and $maxEra_i$ are the minimum and maximum for thread i:

$$(minEra_i \leq newEra \leq maxEra_i) \;||$$
$$(minEra_i \leq delEra \leq maxEra_i) \;||$$
$$(newEra \leq minEra_i \;\&\&\; delEra \geq maxEra_i)$$

4 Performance Evaluation

We compared the throughput of Hazard Eras versus two other common memory reclamation techniques, namely Hazard Pointers, and Userspace-RCU. For Hazard Pointers we made our own implementation, sharing as much code as possible with the Hazard Eras implementation, using also a two-dimensional array to store the hazard pointers, and thread-local lists of type

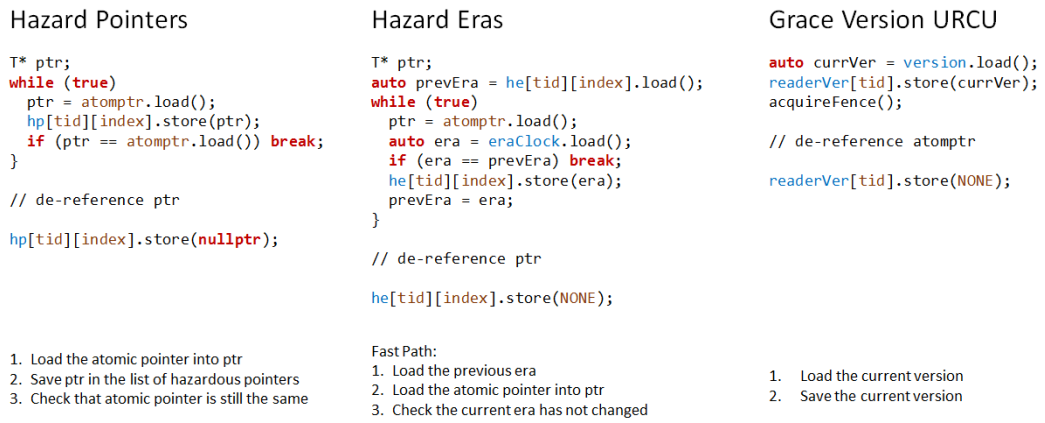

Figure 3 Reader-side with three different memory reclamation techniques.

`std::vector` to store the retired nodes. For Userspace RCU, we chose the currently fastest simple URCU based on the C++ memory model, named Grace Version URCU [27], which provides high scalability for the readers, and reclaimers due to sharing of grace periods. The goal of comparing against URCU is to show an upper limit on throughput of the Maged-Harris list when doing mostly reads. In figure 3 we display side by side the implementations of the three memory reclamation techniques under consideration in our microbenchmarks, for the reader-side operations. The HP and HE correspond to `get_protected()` and `clear()`, and for URCU it corresponds to `rcu_read_lock()` and `rcu_read_unlock()`.

We applied these memory reclamation techniques to the Maged-Harris list and made three C++ implementations whose throughput we show in figure 4, where all values are normalized to Maged-Harris list with HP. The `remove()` method in the implementation using URCU is blocking due to it calling `synchronize_rcu()`, while all other methods for all three implementations are non-blocking.

We executed the microbenchmark on an AMD Opteron 6272 server with a total of 32 cores, running Ubuntu with GCC 6.2.0, using the following procedure: A list is filled with 100 items; we randomly select doing either a lookup or an update, whose probability depends on the percentage of updates for this particular workload; for a *lookup*, we randomly select one item of the 100 and call `contains(item)`; for an *update*, we randomly select one item of the 100 and call `remove(item)`, and if the removal is successful, we re-insert the same item with a call to `add(item)`, thus maintaining the size of the list at 100, minus any ongoing removals. Notice that although we remove and re-insert the same item, internally, the lock-free list will have to retire the old node and create a new node. We repeated the procedure for lists with 1000 and 10000 items (figure 4).

Looking at the read-only plots (leftmost three plots) in figure 4 we see that URCU is the clear winner, with a an advantage in throughput that goes up to 8x the throughput of HP, as is to be expected due to it having no synchronization per-node during a traversal of the list. For the plots more to the right, the number of updates increases and the advantage of URCU reduces, becoming worse than HP and HE with oversubscription. This happens because a preempted reader may block one or multiple reclaimers for long periods of time.

In our microbenchmarks, HE is faster than HP for most scenarios, with a throughput ratio of 0.8x-4.4x for a list with 100 items, 0.8x-5.3x for 1000 items, and 1.8x-5.1x for 10000 items. It's only with small lists and a high number of updates and threads, that the throughput of HE drops to values near or slightly below HP, making HE the dominant strategy, at least in

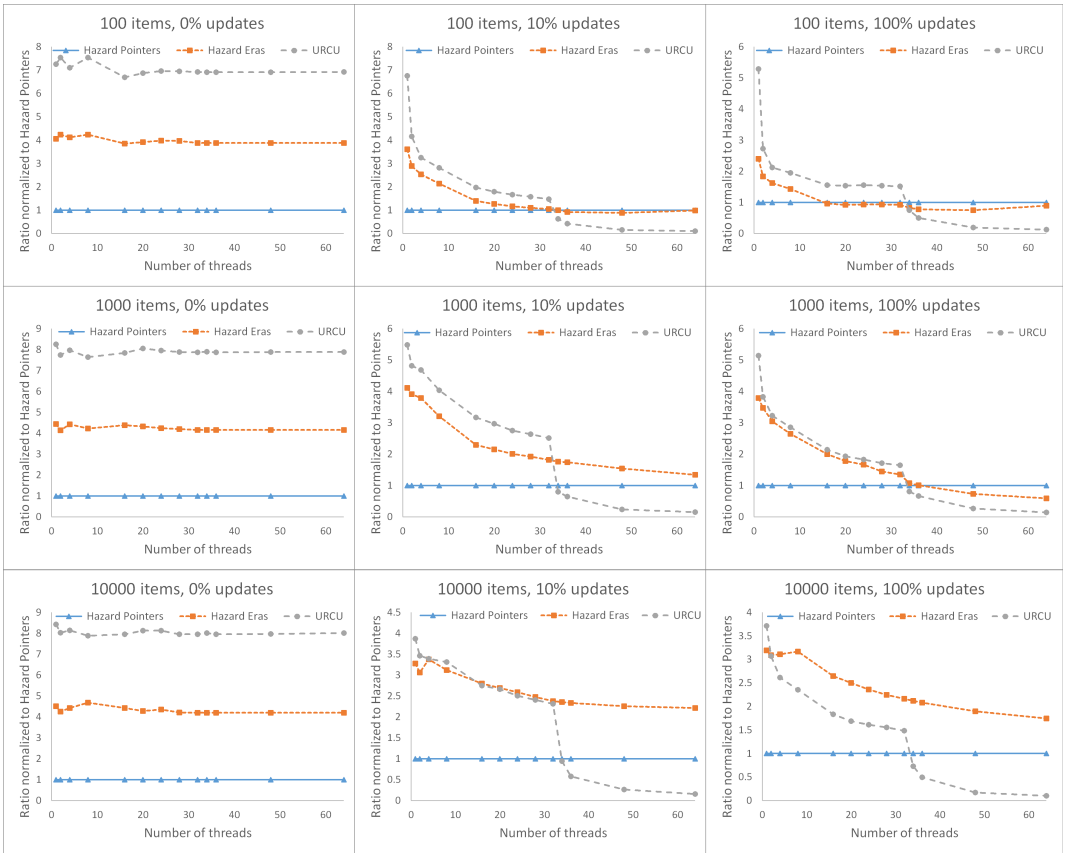

Figure 4 Microbenchmarks comparing Hazard Pointers, Hazard Eras and URCU, using Maged-Harris lists with 100, 1000, or 10000 items (top to bottom), under different ratios of updates of 0%, 10%, 100% (left to right). The vertical axis is the ratio of total number of operations, normalized to the value for Hazard Pointers. Higher is better.

our microbenchmarks. This drop in throughput happens because high update ratios cause the global era to change frequently, thus triggering a seq-cst store for almost every node that is traversed. As long as both the ratio of updates and contention are not extremely high, HE's throughput will be multiple times higher than HP.

5 Conclusion

To the untrained eye, it may look as though there is little difference between an Epoch-based memory reclamation and Hazard Eras, however, there are a few details of vital importance. In Epoch-based reclamation, each thread does a 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.

For scenarios where reclaimers need not be lock-free nor wait-free, quiescence-based techniques like Epoch or URCU seem preferable due to their simplicity of usage and their low synchronization cost for readers, as long as there is no oversubscription of threads to cores. However, if non-blocking progress is a requirement for readers and reclaimers, then one of the pointer-based techniques must be used instead. In that respect, HE are at an advantage over the other pointer-based techniques because they have the same deployment complexity as Hazard Pointers, with a lower synchronization cost for the readers, which allows HE to provide an increased throughput, up to **5**x the throughput of HP. Moreover, non-blocking algorithms that need a large number of simultaneous hazard pointers, previously deemed too slow to be practical, are now practical thanks to HE's high throughput.

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. We believe these qualities make Hazard Eras the best of its class for most scenarios, capable of surpassing Hazard Pointers in throughput, which can be implemented using solely the C11/C++11 memory model with its atomics API, being efficient and simple enough to be deployed in production systems.

References

1 Alistarh, D., Leiserson, W. M., Matveev, A., and Shavit, N. Threadscan: Automatic and scalable memory reclamation. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures* (2015), ACM, pp. 123–132.

2 Auerbach, J., Bacon, D. F., Cheng, P., Grove, D., Biron, B., Gracie, C., McCloskey, B., Micic, A., and Sciampacone, R. Tax-and-spend: democratic scheduling for real-time garbage collection. In *Proceedings of the 8th ACM international conference on Embedded software* (2008), ACM, pp. 245–254.

3 Braginsky, A., Kogan, A., and Petrank, E. 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* (2013), ACM, pp. 33–42.

4 Brown, T. A. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing* (2015), ACM, pp. 261–270.

5 Carpen-Amarie, M., Dice, D., Thomas, G., and Felber, P. Transactional pointers: experiences with htm-based reference counting in c++. In *International Conference on Networked Systems* (2016), Springer, pp. 102–116.

6 Cohen, N., and Petrank, E. Efficient memory management for lock-free data structures with optimistic access. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures* (2015), ACM, pp. 254–263.

7 CPP-ISO-committee. C++ Memory Order. `http://en.cppreference.com/w/c/atomic/memory_order`, 2013.

8 Dice, D., Herlihy, M., and Kogan, A. Fast non-intrusive memory reclamation for highly-concurrent data structures. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management* (2016), ACM, pp. 36–45.

9 Ellen, F., Fatourou, P., Ruppert, E., and van Breugel, F. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing* (2010), ACM, pp. 131–140.

10 Fraser, K. *Practical lock-freedom.* PhD thesis, University of Cambridge, 2004.

11 Gidenstam, A., Papatriantafilou, M., Sundell, H., and Tsigas, P. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems 20*, 8 (2009), 1173–1187.

12 Harris, T. L. A pragmatic implementation of non-blocking linked-lists. In *Distributed Computing*. Springer, 2001, pp. 300–314.

13 Hart, T. E., McKenney, P. E., Brown, A. D., and Walpole, J. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing 67*, 12 (2007), 1270–1285.

14 Herlihy, M., Luchangco, V., and Moir, M. The repeat offender problem: A mechanism for supporting dynamic-sized lock-free data structures.

15 Herlihy, M. P., and Moss, J. E. B. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 3*, 3 (1992), 304–311.

16 Hudson, R. L., and Moss, J. E. B. Sapphire: Copying gc without stopping the world. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande* (2001), ACM, pp. 48–57.

17 Kogan, A., and Petrank, E. Wait-free queues with multiple enqueuers and dequeuers. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 223–234.

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

19 Michael, M. M. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures* (2002), ACM, pp. 73–82.

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

21 Michael, M. M., and Scott, M. L. Correction of a memory management method for lock-free data structures. Tech. rep., DTIC Document, 1995.

22 Michael, M. M., Wong, M., McKenney, P., and O'Dwyer, A. Hazard Pointers - Safe Resource Reclamation for Optimistic Concurrency. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0233r2.pdf`, 2016.

23 Petrank, E. Can parallel data structures rely on automatic memory managers? In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (2012), ACM, pp. 1–1.

24 Pizlo, F., Petrank, E., and Steensgaard, B. A study of concurrent real-time garbage collectors. In *ACM SIGPLAN Notices* (2008), vol. 43, ACM, pp. 33–44.

25 Pizlo, F., Ziarek, L., Maj, P., Hosking, A. L., Blanton, E., and Vitek, J. Schism: fragmentation-tolerant real-time garbage collection. In *ACM Sigplan Notices* (2010), vol. 45, ACM, pp. 146–159.

26 Ramalhete, P., and Correia, A. A Wait-Free Queue with Wait-Free Memory Reclamation. `https://github.com/pramalhe/ConcurrencyFreaks/papers/crturnqueue-2016.pdf`, 2016.

27 Ramalhete, P., and Correia, A. Grace Sharing Userspace-RCU. `https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/gracesharingurcu-2016.pdf`, 2016.

28 Valois, J. D. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing* (1995), ACM, pp. 214–222.

A Epoch vs Hazard Eras

At first sight, it may seem that Hazard Eras is equivalent to an epoch based memory reclamation where the epoch is updated regularly, as opposed to just once at the beginning of the access to objects.

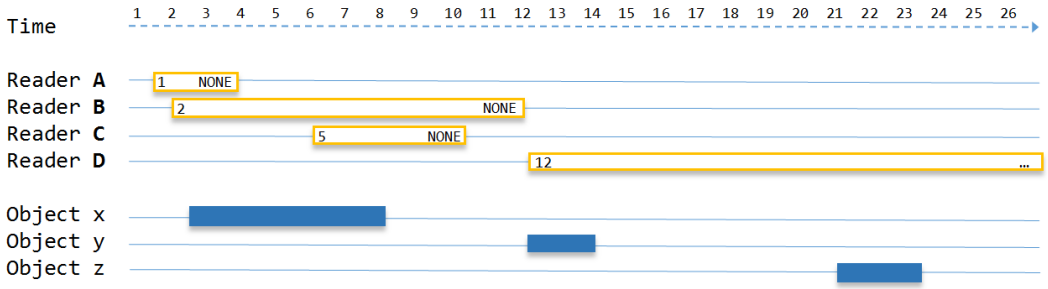

Figure 5 Schematic of 4 readers using Epoch-based memory reclamation. Time flows from left to right. Node x can not be deleted until readers B completes. Nodes y and z can not be deleted until reader D completes, possibly, never.

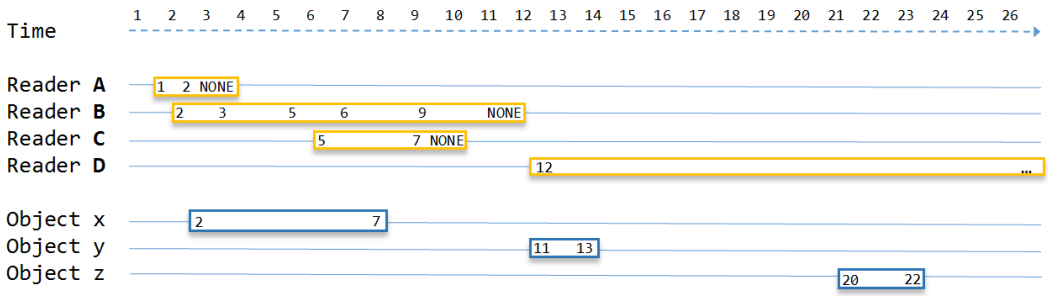

Figure 6 Schematic of 4 readers using Hazard Eras memory reclamation. Time flows from left to right. Node x can not be deleted until reader C completes. Node y can not be deleted until reader D completes, possibly, never. Node z can be deleted immediately.

Looking at the schematics in figures 5 and 6, we see four different readers that have started on a read-side critical path at different instants in time, and three different objects with different visibility windows (the time during which these objects are visible to other threads). The `retire()` method will be called immediately after the end of the objects visibility window, at times 7, 13, and 22, for objects x, y and z respectively. In figure 6, the numbers at the edges of the visibility windows represent the values of `newEra` and `delEra` for x, y and z, respectively. Of particular importance is reader D, which has not completed and therefore, when using an epoch-based algorithm, prevents all further memory reclamation after epoch 12 in figure 5. When using Hazard Eras (figure 6), reader D prevents the deletion of node y, but all nodes whose `newEra` is a value higher than 12 can be safely deleted, namely, node z can be immediately deleted after retired.

This example shows that although some nodes may remain undeletable until a sleepy reader wakes up, newly inserted nodes can be safely deleted, thus providing non-blocking progress and a bound in memory usage for HE.

B On the overflow of `eraClock`

What happens in the event of an overflow of the `eraClock` when a reader is still pending on an ancient era?

Our HE implementation is incapable of handling such a situation, but even on a 3 GHz machine with a CAS instruction which takes a single clock cycle, it would take at least 195 years of continuous increments to overflow a 64 bit register. In HE we increment the `eraClock` only when there is a call to `retire()` and even this can be reduced by incrementing `eraClock` only every k calls to `retire()`. The increments are *shared* by multiple threads, therefore, adding more threads calling `retire()` will not cause the eraClock to increment any faster. It is unlikely that a thread/process with a stuck reader will survive such a large amount of years in a pending state. Alternatively on x86, a Double-Word-CAS can be used on a 128 bit `eraClock`, with `newEra` and `delEra` being 128 bits as well, thus increasing the minimum span of 195 years to a value higher than the age of the known universe before any such issue could theoretically occur.

C Progress Definitions

There are multiple levels of progress guarantees for non-blocking methods in data structures. A method acting on a concurrent object is:

- *obstruction-free* if a thread can perform an arbitrary operation on the object in a finite number of steps when it executes in isolation;
- *lock-free* if it guarantees that at least one thread running one of the concurrent methods of the object finishes in a finite number of steps;
- *wait-free* if it guarantees that every call finishes its execution in a finite number of steps;

Wait-free progress can be further classified as:

- *wait-free unbounded* if it is wait-free and there is no known bound on the number of steps;
- *wait-free bounded (wfb)* if there is a known bound and it depends on the number of threads/processes.
- *wait-free populations oblivious (wfpo)* if there is a known bound and it does not depend on the number of threads/processes; This implies the bound is a constant;

A concurrent object is said to be lock-free if all of its methods are lock-free. A concurrent object is said to be wait-free if all of its methods are wait-free, and therefore, for a memory reclamation algorithm to be wait-free, it must have all of its methods wait-free.

Designers of concurrent data structures are not always aware that memory management with support for certain progress guarantees is not available. For example, there is little benefit in designing a wait-free queue and then use a quiescence-based memory reclamation for the unused nodes, knowing that such a technique is blocking for reclaimers, i.e. for dequeuing operations. Even more, many designers do not apply a memory reclamation technique to their algorithms, leaving what is arguably the hardest task to the library implementer.