Universal Wait-Free Memory Reclamation

Ruslan Nikolaev, Binoy Ravindran
rnikola@vt.edu,binoy@vt.edu
Virginia Tech
Bradley Department of Electrical and Computer Engineering
Blacksburg, VA, USA

Abstract

In this paper, we present a universal memory reclamation scheme, *Wait-Free Eras* (WFE), for deleted memory blocks in wait-free concurrent data structures. WFE's key innovation is that it is completely wait-free. Although some prior techniques provide similar guarantees for certain data structures, they lack support for *arbitrary* wait-free data structures. Consequently, developers are typically forced to marry their wait-free data structures with lock-free Hazard Pointers or (potentially blocking) epoch-based memory reclamation. Since both these schemes provide weaker progress guarantees, they essentially forfeit the strong progress guarantee of wait-free data structures. Though making the original Hazard Pointers scheme or epoch-based reclamation completely wait-free seems infeasible, we achieved this goal with a more recent, (lock-free) Hazard Eras scheme, which we extend to guarantee wait-freedom. As this extension is non-trivial, we discuss all challenges pertaining to the construction of universal wait-free memory reclamation.

WFE is implementable on ubiquitous x86_64 and AArch64 (ARM) architectures. Its API is mostly compatible with Hazard Pointers, which allows easy transitioning of existing data structures into WFE. Our experimental evaluations show that WFE's performance is close to epoch-based reclamation and almost matches the original Hazard Eras scheme, while providing the stronger wait-free progress guarantee.

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

Keywords wait-free, non-blocking, memory reclamation, hazard pointers, hazard eras

PPoPP '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374540

1 Introduction

Most modern general purpose systems use the shared-memory architecture, which stipulates solving the synchronization problem when accessing shared data. The easiest way to solve this problem is to use locks, but due to scalability issues, non-blocking data structures have been studied over the years. A downside of non-blocking data structures is that they require a special *memory reclamation* scheme. Whereas mutual exclusion locks can guarantee that no other thread is using a memory block (node) that is in the process of deletion, this is generally not true for non-blocking designs.

Wait-freedom, the strongest of non-blocking progress guarantees, is critically important in many latency-sensitive applications where execution time of all operations must be bounded [22]. In wait-free algorithms, all threads must eventually complete any operation after a bounded number of steps. Nonetheless, such algorithms have not had significant practical traction due to a number of reasons. Traditionally, wait-free algorithms were difficult to design and were much slower than their lock-free counterparts. Kogan and Petrank's *fast-path-slow-path* methodology [24] largely solved the problem of creating efficient wait-free algorithms. The design of wait-free algorithms, however, is still challenging, as the Kogan-Petrank methodology implicitly assumes wait-free memory reclamation. A case in point: Yang and Mellor-Crummey's wait-free queue [40] uses the Kogan-Petrank methodology. But, as pointed out by Ramalhete and Correia [32], [40]'s design is flawed in its memory reclamation approach which, strictly speaking, forfeits wait-freedom.

Although a number of memory reclamation techniques [4–10, 16, 19–21, 27, 28, 30, 33, 38, 39] have been proposed, only a fraction of them can be used for arbitrary data structures and are truly non-blocking [9, 20, 21, 27, 28, 30, 33, 38, 39]. At present, no *universal* memory reclamation technique exists that guarantees wait-freedom for *arbitrary* wait-free data structures. Typically, prior efforts on wait-free data structures have ignored the memory reclamation problem entirely or have harnessed lock-free memory reclamation schemes such as Hazard Pointers [27], essentially forfeiting strict wait-freedom guarantees. The recently proposed Ramalhete and Correia's wait-free queue [35] can be implemented using Hazard Pointers, but the approach is too specific to the queue's design and cannot be applied for other data structures. [36] presents wait-free memory reclamation for software transactional memory (STM), but the work does

not consider wait-free memory reclamation for handcrafted data structures. Although STM is an important general synchronization technique, handcrafted data structures usually perform better and still need wait-free memory reclamation.

We present a universal scheme, *Wait-Free Eras* (WFE), that solves the wait-free memory reclamation problem for *arbitrary* data structures. For the first time, wait-free data structures such as the Kogan-Petrank queue [23] can be implemented fully wait-free using WFE. Additionally, WFE can help simplify memory reclamation for existing wait-free data structures which use Hazard Pointers in a special way to guarantee wait-freedom.

WFE is based on the recent Hazard Eras [33] memory reclamation approach, which is lock-free. We demonstrate how Hazard Eras can be extended to guarantee wait-freedom. In our evaluation, we observe that WFE's performance is close to epoch-based reclamation and almost matches the original Hazard Eras scheme.

2 Background

For greater clarity and completeness, we discuss relevant memory reclamation schemes and the challenges in designing them with wait-free progress guarantees.

2.1 Progress Guarantees

Non-blocking data structures can provide different *progress guarantees*. In *obstruction-free* algorithms, a thread performs an operation in a finite number of steps if executed in isolation from other threads. In *lock-free* algorithms, at least one thread always makes progress in a finite number of steps. Finally, *wait-freedom* – the strongest progress guarantee – implies that *all* threads make progress in a finite number of steps. Wait-free data structures are particularly useful for latency-sensitive applications which usually have quality of service constraints.

Memory reclamation adds extra requirements to progress guarantees. Unless memory usage is bounded, threads will be unable to allocate memory at some point. This effectively blocks threads from making further progress. Thus, memory reclamation schemes must guarantee that stalled or preempted threads will not prevent timely memory reclamation.

The epoch-based reclamation (EBR) scheme [16, 19] can have unbounded memory usage, preventing its use in wait-free algorithms. In contrast, reclamation schemes such as Hazard Pointers [27] and Hazard Eras [33] provide strict memory bounds as long as programs properly use these schemes. However, both schemes lack *wait-free* progress guarantees.

2.2 Atomic Operations

Typically, lock-free and wait-free data structures are implemented using compare-and-swap (CAS) operations. CAS is more general than other atomic operations such as fetch-and-add (F&A), which can be emulated by CAS. Nonetheless, x86_64 and AArch64 (ARM; as of version 8.1) implement F&A natively due to its better efficiency in hardware. Moreover, hardware's F&A execution time is bounded, which makes it appealing for wait-free algorithms.

In this paper, we also use wide CAS (WCAS), which updates two *adjacent* memory words. WCAS is not to be confused with double-CAS, which updates two *arbitrary* words but is rarely supported in commodity hardware. WCAS, however, is available in x86_64 and AArch64 architectures. Furthermore, WCAS is required by commodity OSes such as Windows 8.1 or higher [2].

2.3 Hazard Eras

The Hazard Eras [33] memory reclamation scheme merges EBR with Hazard Pointers. Each allocated object retains two fields used by the reclamation scheme: *"alloc_era"* and *"retire_era"*. When allocating a new object, its *alloc_era* is initialized with the global era clock value, which is periodically incremented. When the object is retired, its *retire_era* is also initialized with the global era value. The lifespan of the object is determined by these two eras. When a thread accesses a hazardous reference, it publishes the current era. Thus, if the lifespan of an object falls within *any* of the published eras, it will not be reclaimed.

Hazard Eras' API is mostly compatible with that of Hazard Pointers and consists of the following operations:

- *get_protected()*: safely retrieve a pointer to the protected object by creating a reservation; each object needs an *index* that identifies the reservation.
- *retire()*: mark an object for deletion; the retired object must be deleted from the data structure first, i.e., only in-flight threads can still access it.
- *clear()*: reset all prior reservations made by the current thread in *get_protected()*.
- *alloc_block()*: a special operation unique to Hazard Eras; it allocates a memory block and initializes its *alloc_era* to the global era clock value.

Figure 1 presents the Hazard Eras algorithm. In the algorithm, we assume that the maximum number of threads is *max_threads*. The maximum number of reservations per each thread is *max_hes*. All retired nodes are appended to the thread-local *retire_list*. The algorithm periodically scans this list to check if old nodes can be safely de-allocated by calling *cleanup()*. To guarantee that the memory usage is bounded, the algorithm periodically increments the global era clock in *alloc_block()* and *retire()*. Arguments on correctness and bounded memory usage can be found in [34].

In Figure 2, we present an example of Treiber's stack [37] implementation using Hazard Eras. The stack is a linked-list of nodes which store pointers to inserted objects. Each node also encapsulates a memory reclamation header block. When

```
1  // Constants
2  const int era_freq, cleanup_freq;
3
4  // Global variables
5  thread-local int retire_counter = 0;
6  thread-local int alloc_counter = 0;
7  thread-local list retire_list = EMPTY;
8  int global_era = 0;
9  int reservations[max_threads][max_hes] = { ∞ };
10
11 // Read a block pointer
12 block* get_protected(block** ptr, int index) {
13    int prevEra = reservations[tid][index];
14    while (true) {
15       block* ret = *ptr;
16       int newEra = global_era;
17       if (prevEra == newEra) return ret;
18       reservations[tid][index] = newEra;
19       prevEra = newEra;
20    }
21 }
22
23 // Retire a memory block
24 void retire(block* ptr) {
25    ptr->retire_era = global_era;
26    retire_list.append(ptr);
27    if (retire_counter++ % cleanup_freq == 0) {
28       if (ptr->retire_era == global_era)
29          F&A(&global_era, 1);
30       cleanup();
31    }
32 }
```

```
33 // Clear all hazard eras for the current thread
34 void clear() {
35    for (int j = 0; j < max_hes; j++)
36       reservations[tid][j] = ∞;
37 }
38
39 // Allocate a memory block
40 block* alloc_block(int size) {
41    if (alloc_counter++ % era_freq == 0)
42       F&A(&global_era, 1);
43    block* ptr = new block(size);
44    ptr->alloc_era = global_era;
45    return ptr;
46 }
47
48 // Internal functions
49 void cleanup() {
50    foreach blk in retire_list
51       if (can_delete(blk, 0, max_hes))
52          free(blk);
53 }
54
55 bool can_delete(block* ptr, int js, int je) {
56    for (int i = 0; i < max_threads; i++) {
57       for (int j = js; j < je; j++) {
58          int era = reservations[i][j];
59          if (era ≠ ∞ && ptr->alloc_era ≤ era
60                && ptr->retire_era ≥ era)
61             return false;
62    }  }
63    return true;
64 }
```

Figure 1. The Hazard Eras memory reclamation scheme.

enqueuing an object, we allocate a node using *alloc_block()*, store a pointer to the object, and update the stack pointer. When dequeuing, we dereference the top of the stack using *get_protected()*. We only use *index 0* since we dereference just one pointer at a time. The top of the stack is then updated to refer to the next node. We retrieve an object pointer and retire the dequeued node. Finally, *clear()* resets all reservations (i.e., *index 0*) for the current thread.

2.4 Challenges in Wait-Free Memory Reclamation

Only a few memory reclamation schemes are truly non-blocking, i.e., do not use any OS mechanisms and also guarantee bounded memory usage. Although certain OS mechanisms, such as signals, can transparently handle reclamation, it is difficult to guarantee non-blocking behavior in general, as locks are often used inside OS kernels when dispatching signals. We also focus on manual memory reclamation techniques, i.e., garbage collectors are beyond the scope of this paper. Though EBR is almost wait-free for most operations, it is blocking due to its potentially unbounded memory usage. Since truly wait-free algorithms must also be non-blocking, we only consider those algorithms. We narrow our scope to Hazard Pointers [27], Hazard Eras [33], and Interval-Based Reclamation (IBR) [39].

In both Hazard Pointers and Hazard Eras, most operations are already wait-free. However, access to hazardous references through *get_protected()* is a noticeable exception. Both schemes need unbounded loops: Hazard Eras check that the published global era value has not changed while reading the reference; Hazard Pointers publish the reference itself but still need to validate that the reference has not changed since it was published. Despite this similarity, the two schemes differ drastically. To make Hazard Pointers wait-free, we must ensure that the references do not change, which seems generally impossible to do in a wait-free manner. In Hazard Eras, we only need to make sure that the *global era* value is unchanged. Although the original Hazard Eras approach could not solve this problem, we demonstrate a viable solution.

We also considered IBR, especially because of its simpler API. However, we preferred to extend the Hazard Eras scheme instead due to its strict memory usage guarantees, even in the presence of starving or slow threads. (Wait-free memory reclamation can still be used for ordinary, lock-free data structures.) IBR requires additional changes to data structures to guard against this condition [39]. Certain tagged versions of IBR also require more work to make them wait-free. Nonetheless, our approach is applicable to the 2GEIBR version where only hazardous reference accesses need to be made wait-free.

3 Wait-Free Memory Reclamation

The proposed WFE algorithm employs the traditional *fast-path-slow-path* idea but avoids the Kogan-Petrank methodology [24] due to the complexity of memory reclamation

```
1  struct node_s {     // Stack element:
2      block header;   // 1. hazard eras header
3      node_s* next;   // 2. next stack element
4      void* obj;      // 3. stored object
5  };
6  node_s* stack = nullptr;   // Top of the stack
7
8  // Remove an object from the stack
9  void* dequeue() {
10     void* obj = nullptr;
11     while (true) {
12         node_s* node = get_protected(&stack, 0);
13         if (node == nullptr) break;
14         if (CAS(&stack, node, node->next)) {
15             obj = node->obj;
16             retire(node);
17             break;
18         }
19     }
20     clear();
21     return obj;
22 }
23
24 // Insert an object onto the stack
25 void enqueue(void* obj) {
26     node_s* node = alloc_block(sizeof(node_s));
27     node->obj = obj;
28     do {
29         node->next = stack;
30     } while (!CAS(&stack, node->next, node));
31 }
```

Figure 2. Usage example: Treiber's lock-free stack.

and uniqueness of its challenges. The high-level idea is to execute an almost unchanged Hazard Eras scheme on the fast path. When *get_protected()* fails to complete after a specified number of steps, the slow path is taken where threads collaborate to guarantee wait-freedom.

3.1 Assumptions

We assume that the hardware supports WCAS and wait-free F&A. We also assume a 64-bit CPU since we need to atomically manipulate eras (sometimes using WCAS) which, for safety, are typically at least 64 bits wide in Hazard Eras. Our assumptions are true for common-place x86_64 and more recent versions of AArch64 (ARM) processors.

Although not all other microarchitectures currently satisfy these requirements, they can easily fall back to the original Hazard Eras algorithm to retain API compatibility at the cost of losing wait-freedom.

To simplify our pseudo-code, we will further assume that the memory model is sequentially consistent [25]. Our actual implementation relaxes this requirement to benefit architectures with weaker memory models such as AArch64.

Finally, we assume that the number of threads is bounded, which is a reasonable assumption made by most reclamation schemes.

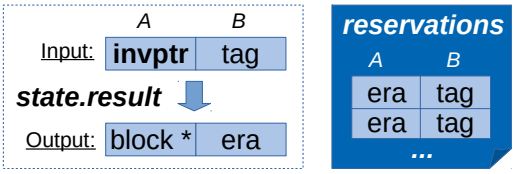


Figure 3. WFE state and reservations.

3.2 Data Fields and Formats

We inherit most existing data fields from Hazard Eras. However, we modify the *reservations* array to record *pairs* instead of eras. Each pair consists of an *era* as well as a *tag* for the current reservation. The tag is only accessed on the slow path and identifies the slow-path cycle. The tag is monotonically increased after each such cycle and protects against spurious (delayed) data changes happening afterwards.

We also reserve a special pointer value, **invptr**, which should never be used by data structures. Since **nullptr** is still occasionally used, we reserve the maximum integer value instead; it should never be stored in valid pointers (e.g., mmap(2) returns the same value for MAP_FAILED).

For the slow path, each reservation keeps *state* (in a separate array). We discuss all fields of *state* below. One of its fields, *result*, is used for both input and output. On the input, it records the *tag* of the current slow-path cycle. On the output, it contains a dereferenced pointer as well as the era that needs to be set in *reservations* for this pointer. To distinguish these two cases, we place **invptr** in the place of the dereferenced pointer and *tag* in the place of *era*. The pointer value distinguishes whether the result is already produced.

We summarize new data fields in Figure 3.

3.3 Bird's-Eye View

Assuming the presence of wait-free F&A, as discussed in [34], Hazard Eras is already wait-free for all operations except *get_protected()*, which is used to dereference pointers. The *get_protected()* operation contains a potentially unbounded loop and, consequently, is only lock-free. The only reason why this loop may never converge is due to the changing value of the global era clock. The global era value changes periodically when allocating new objects or retiring old objects. Thus, to make the Hazard Eras algorithm wait-free, threads that call *alloc_block()* and *retire()* have to collaborate with threads that call *get_protected()*.

In other words, *alloc_block()* and *retire()* should not be incrementing the global era clock unless they can guarantee that all other running threads succeed in their *get_protected()* call. However, global era increments cannot be simply postponed since they are crucial to guarantee that the memory usage is bounded. Likewise, *alloc_block()* or *retire()* cannot block on *get_protected()* either. The idea employed in WFE is that before incrementing the global era, *alloc_block()* and *retire()* will first check if *any* thread needs helping.

```
1  int counter_start = 0;
2  int counter_end = 0;
3  int_pair  // Each pair is { .A,.B }
4    reservations[max_threads][max_hes+2] = { ∞,0 };
5
6  struct state_s {
7     int_pair result; // Initially: { nullptr,∞ }
8     int era; // Initially: ∞
9     void* pointer; // Initially: nullptr
10 };
11 state_s state[max_threads][max_hes];
12
13 // Read a block pointer
14 block* get_protected(block** ptr, int index,
15                      block* parent) {
16    int prevEra = reservations[tid][index].A;
17    int attempts = max_attempts;
18    while (--attempts ≠ 0) { // Fast path
19       block* ret = *ptr;
20       int newEra = global_era;
21       if (prevEra == newEra) return ret;
22       reservations[tid][index].A = newEra;
23       prevEra = newEra;
24    }
25    // Fetch parent's era for [ptr] protection
26    if (parent == nullptr) alloc_era = ∞;
27    else alloc_era = parent->alloc_era;
28    // Slow path, request helping
29    F&A(&counter_start, 1);
30    state[tid][index].pointer = ptr;
31    state[tid][index].era = alloc_era;
32    int tag = reservations[tid][index].B;
33    state[tid][index].result = {invptr,tag};
34    do {   // Bounded by # of in-flight threads
35       block* ret = *ptr;
36       int newEra = global_era;
37       if (prevEra == newEra &&
38           WCAS(&state[tid][index].result,
39                {invptr,tag}, {nullptr,∞}) {
40          reservations[tid][index].B = tag + 1;
41          F&A(counter_end, 1);
42          return ret;
43       }
44       // Ignore failures, the loop will exit
45       WCAS(&reservations[tid][index],
46            {prevEra,tag}, {newEra,tag});
47       prevEra = newEra;
48       res_ptr = state[tid][index].result.A;
49    } while (res_ptr == invptr);
50    int res_era = state[tid][index].result.B;
51    reservations[tid][index].A = res_era;
52    reservations[tid][index].B = tag + 1;
53    F&A(&counter_end, 1);
54    return res_ptr;
55 }
56
57 void cleanup() { // A new cleanup() procedure
58    foreach blk in retire_list {
59       int ce = counter_end;
60       if (!can_delete(blk, 0, max_hes) ||
61           !can_delete(blk, max_hes, max_hes+1))
62          continue;
63       if (ce == counter_start ||
64           (can_delete(blk, max_hes+1, max_hes+2)
65             && can_delete(blk, 0, max_hes))
66          free(blk);
67 }  }
```

```
68 // Allocate a memory block
69 block* alloc_block(int size) {
70    if (alloc_counter++ % era_freq == 0)
71       increment_era(tid);
72    block* ptr = new block(size);
73    ptr->alloc_era = global_era;
74    return ptr;
75 }
76
77 // Retire a memory block
78 void retire(block* ptr) {
79    ptr->retire_era = global_era;
80    retire_list.append(ptr);
81    if (retire_counter++ % cleanup_freq == 0) {
82       if (ptr->retire_era == global_era)
83          increment_era(tid);
84       cleanup();
85    }
86 }
87
88 // Help others before incrementing an era
89 void increment_era(int tid) {
90    int ce = counter_end;
91    int cs = counter_start;
92    if (cs - ce ≠ 0) {
93       for (int i = 0; i < max_threads; i++) {
94          for (int j = 0; j < max_hes; j++) {
95             void* ptr = state[i][j].result.A;
96             if (ptr == invptr)
97                help_thread(i, j, tid);
98    }  }  }
99    F&A(&global_era, 1);
100 }
101
102 // An internal function to help other threads
103 void help_thread(int i, int j, int tid) {
104    int_pair res = state[i][j].result;
105    if (res.A ≠ invptr) return;
106    int era = state[i][j].era;
107    reservations[tid][max_hes].A = era;
108    block** ptr = state[i][j].pointer;
109    int tag = reservations[i][j].B;
110    if (tag ≠ res.B) goto exit;
111    // All state data were read consistently
112    int prevEra = global_era;
113    do {   // Bounded by # of in-flight threads
114       reservations[tid][max_hes+1] = prevEra;
115       block* ret_ptr = *ptr;
116       int newEra = global_era;
117       if (prevEra == newEra) {
118          if (WCAS(&state[i][j].result,
119                res, {ret_ptr,newEra}) {
120             do { // At most 2 iterations
121                old = reservations[i][j];
122                if (old.B ≠ tag) break;
123                ok = WCAS(&reservations[i][j],
124                          old, {newEra,tag+1});
125             } while (!ok);
126          }
127          break;
128       }
129       prevEra = newEra;
130    } while (state[i][j].result == res);
131    reservations[tid][max_hes+1] = ∞;
132 exit:
133    reservations[tid][max_hes].A = ∞;
134 }
```

Figure 4. The Wait-Free Eras (WFE) memory reclamation scheme.

3.4 Wait-Free Eras (WFE)

Figure 4 presents WFE's pseudo-code for functions that diverge from Hazard Eras. All other operations that use the *reservations* array need to be modified accordingly to access the *A* component of the pair which retains an era. The *reservations* array is also extended by two additional reservations per each thread, i.e., its size is now *max_hes+2*. These two new reservations should never be used by the application; they are internal to *help_thread()*.

WFE slightly alters the API for *get_protected()*. Its extra argument, *parent*, needs to provide the area (block) where the hazardous reference is located. Typical data structures keep all references inside their blocks (e.g., a linked-list node keeps the reference to the next node) and the parent parameter simply refers to the previously retrieved hazardous reference. Since the topmost references do not have any parent, we also allow to pass **nullptr** for them.

Lines 16-24 of *get_protected()* represent the fast path and are identical to the original implementation. Lines 26-27 retrieve the *alloc_era* from the parent block (if any), so that the block can be saved from being reclaimed while retrieving the hazardous reference from it by a helper method.

To facilitate fast detection if any thread needs helping, we maintain global variables *counter_start* and *counter_end*, which are atomically incremented using F&A. Their purpose is twofold. First, the difference between these two variables indicate the number of threads that need helping. Second, if the value of *counter_start* changes, it indicates that some new thread entered the slow path region.

On the slow path, a thread initializes *state* with a hazardous reference pointer and the *alloc_era* from the parent block where it is located. As the final step, the thread atomically flips the *result* pair from some valid (previously used) pointer and era to **invptr** and the current slow-path cycle tag obtained from the *reservations* array. Right after that instant, concurrent threads in *increment_era()* (Line 96), called from *alloc_block()* and *retire()*, can detect that this thread needs helping.

While waiting for help, the thread resumes the loop where it attempts to retrieve the hazardous reference. If the thread succeeds, it simply cancels the request by flipping its *state* to some valid pointer and era (**nullptr** and ∞ in the pseudo-code). At that point, only the tag from the *reservations* array needs to be incremented to prepare for the next slow-path cycle (Line 40). The corresponding WCAS (Line 38) call can also fail, indicating that some helping thread already produced the output which must be used instead. Either way, the thread attempts to update its current reservation by using WCAS (Line 45) which can only fail if a helping thread already produced an output and modified the corresponding entry in *reservations*. The loop exits when the pointer field is no longer **invptr**. Finally, the thread modifies the entry in *reservations* from the output value in the *result* pair. (Note that the entry can already be set by the helping thread, in which case the same value will be just written again.)

We will now discuss the *help_thread()* procedure. First, this procedure must set a reservation for the parent block, so that it can be safely accessed (Line 107). While doing so, the original *get_protected()* may already complete. Thus, we check the tag (Line 110) before proceeding. Once the reservation is set, the new *cleanup()* routine will make sure that the parent block is not reclaimed (see Section 4). When *help_thread()*'s loop converges and the output is produced (Line 118), WCAS in Line 123 will attempt to update the *reservations* array on behalf of the *get_protected()* thread. It only succeeds if the tag still refers to the previous slow-path cycle.

4 Correctness

We assume that programs are *well-behaved*, i.e., they call provided API functions appropriately. We focus on the arguments related to wait-freedom and reclamation safety. General non-blocking and memory usage arguments remain the same as in [34] since WFE is based on Hazard Eras. We will denote the number of threads by *n*.

Lemma 1. *The loop in Lines 34-49 is bounded by at most n iterations.*

Proof. A thread initiates the slow path in Line 33. The loop can only become unbounded due to changing *global_era*. At most *n* in-flight threads can already be executing *increment_era()*, from *alloc_block()* or *retire()*, prior to Line 99, which increments *global_era*, but after Line 96, which detects threads that need helping. Each of these in-flight threads will execute Line 99, potentially causing the loop in *get_protected()* to fail and repeat. Subsequent *increment_era()* calls detect threads that need helping and only increment *global_era* after *help_thread()* (Line 97) is complete. □

Lemma 2. *The loop in Lines 113-130 is bounded by at most n iterations.*

Proof. The proof is similar to that of Lemma 1. The only difference is that we need to also consider a case when the loop potentially never terminates because the same thread keeps requesting the slow path over and over again. However, Line 130 also checks the *tag* which guarantees that *help_thread* handles just one slow-path cycle. □

Lemma 3. *The loop in Lines 120-125 is bounded by at most 2 iterations.*

Proof. For this loop to continue, the tag must remain intact (Line 122), i.e., the corresponding *get_protected()* call is still pending. The loop is only executed when WCAS in Line 118 succeeds. Consequently, the output is produced, which prompts Line 49 to terminate the slow-path loop in *get_protected()*. Prior to that loop termination, WCAS can still

change the *reservations* array one more time (Line 45). Thus, the loop in Lines 120-125 may repeat one more time. □

Lemma 4. *A parent object is not reclaimed while running help_thread() as long as the object reclamation procedure first checks normal reservations [0..max_hes-1] and then the first special reservation.*

Proof. For an object to be removed, no reservation should overlap with it. By the API convention, the parent object already has a corresponding reservation for it (except when it is **nullptr**). However, this reservation is not guaranteed to last after *get_protected()* is complete. If the tag in Line 110 matches the corresponding field in *result*, the reservation was set while *get_protected()* was still active. For other cases, we simply exit from *help_thread()*. Since the order of making reservations coincides with the order of checks, the parent object will be covered by at least one reservation. □

Lemma 5. *An object referred to by a hazardous entry is not reclaimed while handing over a reservation from help_thread() to get_protected() as long as the object reclamation procedure first checks the second special reservation and then normal reservations [0..max_hes-1].*

Proof. Similar to a parent object, a hazardous entry object obtained in *help_thread()* is protected by a special and normal reservations. Compared to Lemma 4, the order of reservations is different. Using similar arguments as in Lemma 4, we conclude that the order of checks must also happen in the opposite direction. □

Theorem 1. *get_protected() is wait-free bounded.*

Proof. The number of iterations on the fast path is bounded. For the slow path, Lemma 1 guarantees that the output must be produced at most after n iterations, or the corresponding loop must converge due to the global era value staying intact. When the output is produced, the slow-path loop terminates (Line 49). □

Theorem 2. *alloc_block() is wait-free bounded.*

Proof. *alloc_block()* periodically calls *increment_era()*. Loops inside *increment_era()* are already bounded. On each iteration, *help_thread()* is called. The *help_thread()* function is bounded due to Lemmas 2 and 3. □

Theorem 3. *retire() is wait-free bounded.*

Proof. The proof is similar to that of Theorem 2. □

Theorem 4. *WFE's cleanup() is safe for memory reclamation.*

Proof. The reservation scanning discipline in *cleanup()* satisfies both Lemmas 4 and 5. It also satisfies Hazard Eras' original discipline for all other blocks. □

5 Performance Results

We performed all tests on an x86_64 machine with 256GB of RAM and four Intel Xeon E7-8890 v4 (2.20GHz) processors, each with 24 cores. Processors have separate L1/L2 caches per each core and the L3 cache is shared across each processor. We pinned the first 24 threads to one processor, next 24 threads to another processor, and so on. We also disabled SMT (simultaneous multithreading), i.e., splitting one physical core into several virtual cores, as it is typically recommended to disable SMT for more predictable measurements.

We ran the benchmark in [39] that already implements existing reclamation approaches and additionally implemented our approach. The schemes include:

WFE: our wait-free eras scheme presented in this paper.
HE: the hazard eras scheme [33], which WFE extends.
HP: the classical hazard pointers scheme [27].
EBR: the epoch-based reclamation scheme.
2GEIBR: the interval-based reclamation approach of [39]; we used the 2GEIBR version which does not tag pointers.
Leak Memory: a baseline which just leaks memory, i.e., provides no memory reclamation.

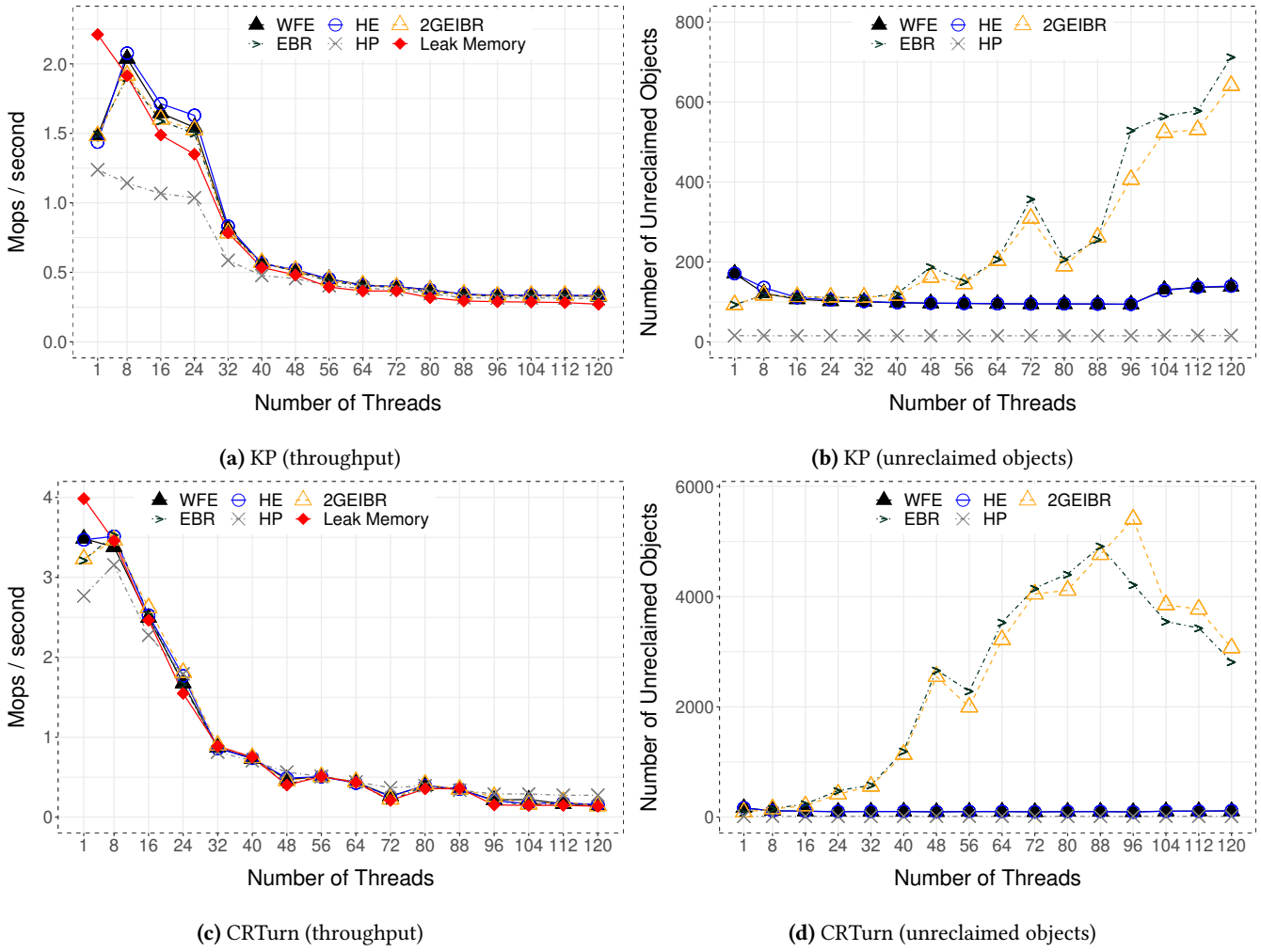
We have fixed a potential race condition in HE's *retire()*, which probably went unnoticed in the original benchmark due to subtle differences in *retire()* between HE and IBR.

We compiled the benchmark, which is written in C++, using g++ 8.3.0 (-O3 optimizations). Similar to [39], we used jemalloc [15] due to its better performance.

The goal of our evaluation is twofold. First, we wish to understand how universal our scheme is and its performance when used with common lock-free data structures. Second, we wish to understand the effectiveness of using our reclamation scheme with wait-free data structures while providing wait-free progress guarantees. Since wait-free data structures are typically much harder to implement, we focused on a select few.

For lock-free data structures, we used the existing tests from the benchmark: a sorted *Linked List* [18] (includes a modification from [27]), *Natarajan BST* (binary search tree) [29], and *Hash Map* [27]. For wait-free data structures, we extended the benchmark to implement Kogan-Petrank (*KP*) [23] and *CRTurn* [35] wait-free queues for all reclamation schemes; we based our implementation on the existent code for Hazard Pointers [1]. The original KP queue uses a garbage collector, for which no known wait-free implementation exists; thus, we are the first to evaluate the KP queue with wait-free reclamation.

In the evaluation, we used both write-dominated tests, where one half of all operations are insertions and the other half are deletions, as well as read-mostly tests, where 90% of all operations are *get()* and the remaining 10% are *put()*. Each data structure implements an abstract key-value interface with the corresponding operations, i.e., *insert()*, *delete()*, *get()*,



(a) KP (throughput)

(b) KP (unreclaimed objects)

(c) CRTurn (throughput)

(d) CRTurn (unreclaimed objects)

Figure 5. Wait-free queues (*50% insert()* and *50% delete()*).

and *put()*. Following the methodology from [39], each test measured a single data point by pre-filling the data structure with 50K elements and then ran 10 seconds (repeated 5 times). The key for each operation is randomly chosen from the range (0, 100000).

The benchmark allows tuning of certain parameters for each test. Specifically, the epoch counter for WFE, HE, EBR, and 2GEIBR is incremented after $n \times \nu$, where n is the number of all active threads and ν is the per-thread frequency of epoch or era increments. Similar to [39], we used $\nu = 150$, which is large enough to avoid performance bottlenecks for the epoch counter increments in our setup. Similar to [39], the per-thread scanning frequency of retired lists is at least 30 but depends on the algorithm due to differences in *retire()*. Finally, for WFE, we set the number of attempts on the fast-path to 16. Even if the number of attempts is that small, the slow path is taken rarely. (We also tested our algorithm by forcing the slow path to be taken all the time to validate that our implementation still works correctly under stress conditions.)

Figure 5 shows the throughput for wait-free queues. For queues, typically, only *insert()* and *delete()* operations make sense. Thus, we only present results for the write-dominated workload. Generally speaking, both KP and CRTurn queues show similar throughput (Figures 5a and 5c) for all schemes except HP, which is sometimes slower. Queues generally do not scale very well. With respect to the average number of unreclaimed objects per operation, a metric which measures the memory reclamation speed, we found that HE and WFE are slightly less efficient than HP, but better than EBR and 2GEIBR (Figures 5b and 5d).

For Linked-List (Figures 6 and 9), we found that EBR is marginally better than all other schemes except HP, which exhibits the worst performance. WFE is marginally worse than HE. Our investigation has shown that an average linked-list traversal operation is long (due to sequential search) and dereferences many pointers. Consequently, (inlined)

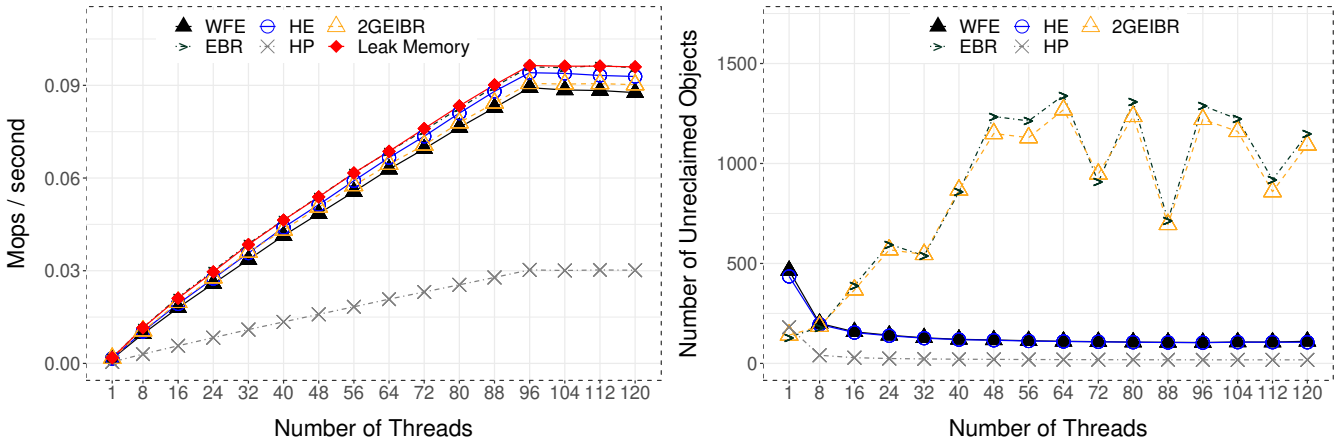


Figure 6. Linked List (*50% insert()* and *50% delete()*).

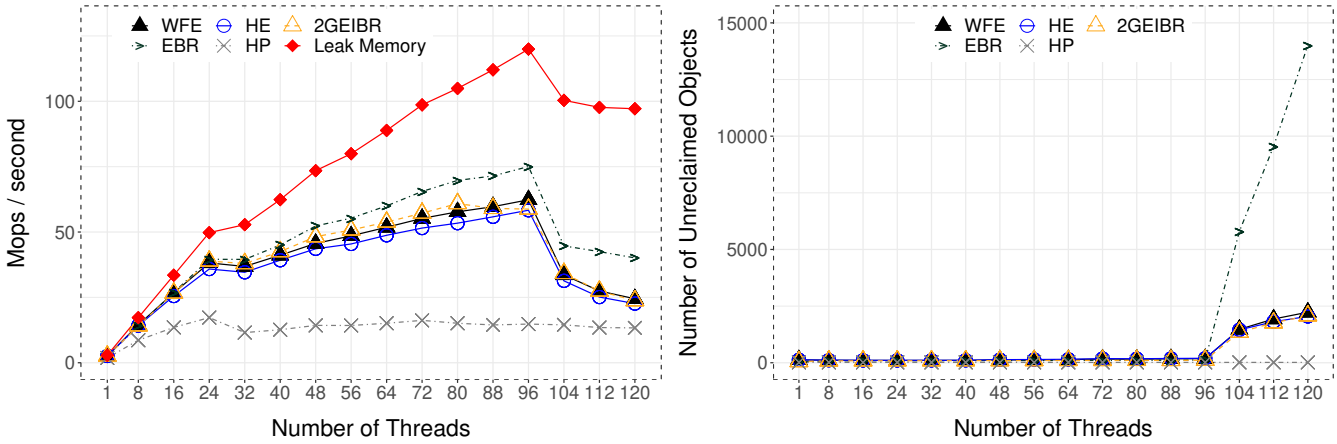


Figure 7. Hash Map (*50% insert()* and *50% delete()*).

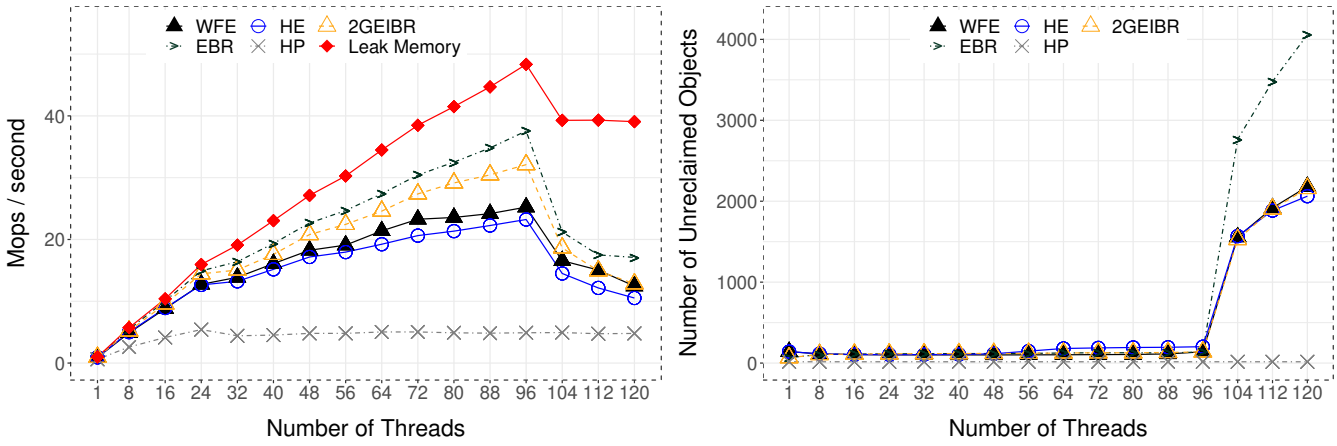


Figure 8. Natarajan BST (*50% insert()* and *50% delete()*).

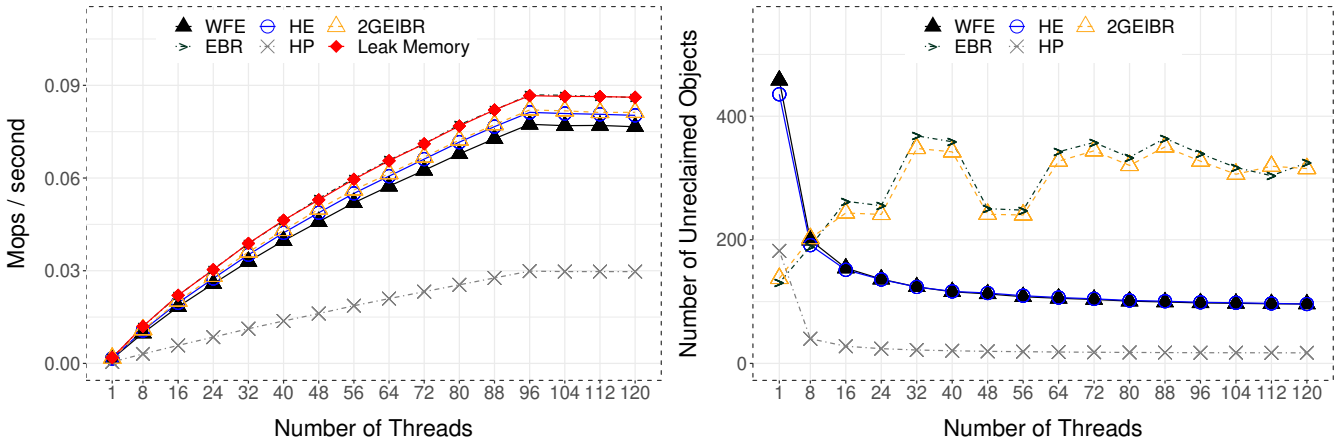


Figure 9. Linked List (*90% get()* and *10% put()*).

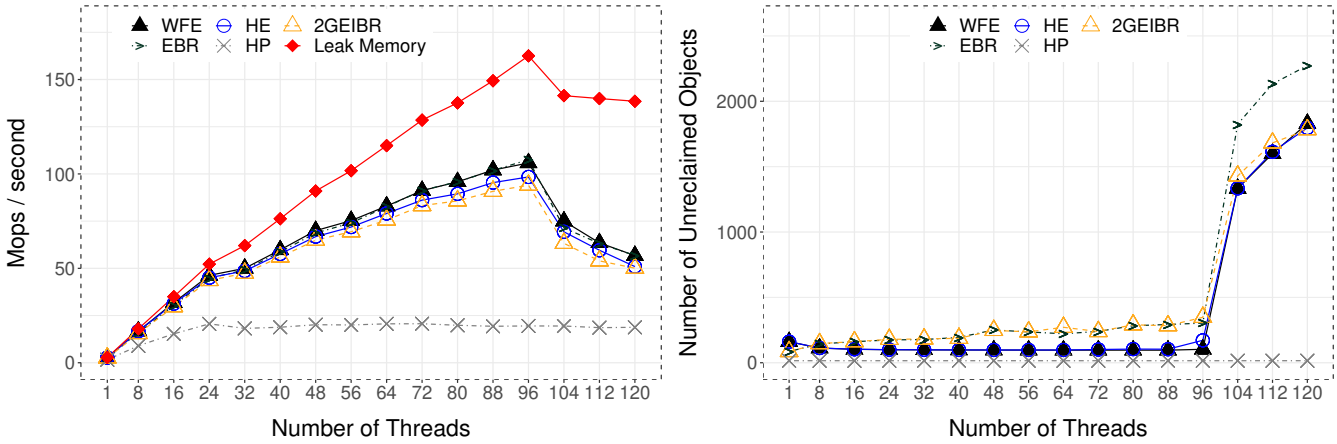


Figure 10. Hash Map (*90% get()* and *10% put()*).

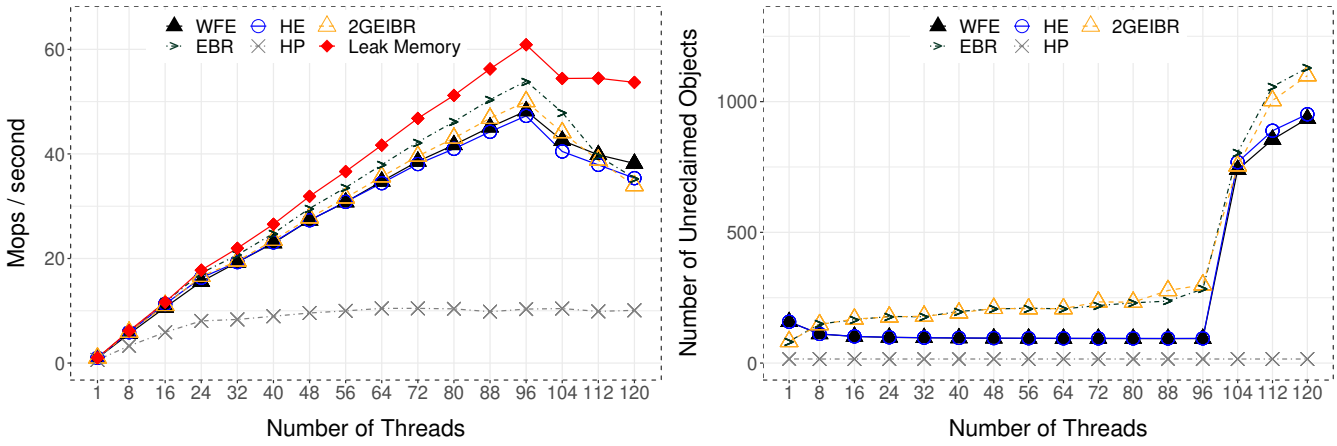


Figure 11. Natarajan BST (*90% get()* and *10% put()*).

get_protected() calls have to be very efficient. Since WFE also needs to call the slow-path procedure, higher register pressure forces the compiler to generate less efficient code. However, this overhead is still quite insignificant. If desired, the overhead can be eliminated by customizing a calling convention for the slow-path procedure: the customized call can reduce the register pressure on the fast path by ensuring that no registers need to be saved by callees. With respect to the average number of unreclaimed objects per operation, except for smaller concurrency, HE and WFE are less efficient than HP, but better than EBR and 2GEIBR.

For Hash-Map (Figures 7 and 10), we found that EBR is at the same level as or marginally outperforms other schemes (except HP which has the worst performance). WFE is at the same level as HE or even marginally outperforms it. Natarajan BST (Figures 8 and 11) shows similar trends, except that the gap between EBR and HE (or WFE) is larger. For the write-dominated tests, we found that EBR is significantly less memory efficient than all other schemes when threads are preempted.

Overall, the results show that WFE's performance is comparable to that of other high-performant non-blocking algorithms such as HE and 2GEIBR. At the same time, WFE provides the stronger wait-free progress guarantee.

6 Related Work

The literature presents a number of memory reclamation techniques for concurrent data structures. We classify them into different categories.

The *first category* includes schemes that use epochs such as EBR [16], which originates from RCU [26]. In these approaches, a thread records the global epoch value to make a reservation at the beginning of an operation. Then, at the end of the operation, it resets the reservation. A related approach, quiescent-state reclamation [19], increments the counter after all threads transition through a state where they hold no pointers. Stamp-it [31] extends EBR to bound reclamation cost to $O(1)$.

Due to potentially unbounded memory usage, all these techniques can be blocking when threads are preempted or stalled. Hazard Eras [33] and IBR [39] implement a non-blocking epoch-based approach. Our Wait-Free Eras scheme extends Hazard Eras, but the same idea can also be straightforwardly applied to certain versions of IBR, e.g., 2GEIBR.

The *second category* includes reclamation techniques that deal with pointers. Hazard Pointers [27] record all pointers that are currently in use. The technique has a relatively high overhead due to its extensive use of memory barriers for each pointer dereference. The original paper presents Hazard Pointers as a "wait-free" scheme. However, the difficult part comes during traversals, when an advertised pointer changes and needs to be read again. (Granted, [27] sidesteps an explicit *get_protected()* operation, which we discuss in this paper.) Pass-the-buck [20, 21] uses a similar model. Another technique, drop-the-anchor [7], is designed specifically for linked-lists, and outperforms hazard pointers. This approach, however, does not seem to be directly applicable to other data structures. Optimistic Access [11] is more general, and leverages a "dirty" flag instead of publishing hazard pointers, but requires data structures to be written in a "normalized form." Automatic Optimistic Access [10] relies on a data structure-specific garbage collector to make reclamation more automatic, but still requires data structures to be written in a normalized form. FreeAccess [9] forgoes this requirement by extending the LLVM compiler to make the process fully automatic.

We considered making Hazard Pointers wait-free. Just like Hazard Eras, the Hazard Pointers scheme is also mostly wait-free except the *get_protected()* operation. However, Hazard Pointers use pointers instead of epochs, and there does not seem to be a straightforward way to adopt our approach for Hazard Pointers or any other technique that tracks pointers in the same manner.

The *third category* is reference counting [12, 17, 28, 38]. A memory object is reclaimed when the reference counter, associated with the object, reaches zero. General-purpose reference counting is typically lock-free and very intrusive: a pointer and a reference counter are adjacent and need to be both atomically updated using WCAS. Reference counting typically performs poorly on read-dominated workloads, as read operations must update reference counters, which requires additional memory barriers. Hyaline [30], an approach that implements distributed reference counting, forgoes this requirement and achieves excellent performance. However, Hyaline is still only lock-free.

Reclamation schemes of the *fourth category* use special OS mechanisms. For these reclamation schemes, it is generally hard to guarantee non-blocking behavior since an OS can use locks internally. DEBRA+ [8] uses signals to add fault tolerance to EBR, i.e., a stalled thread which does not advance its epoch is interrupted by an OS signal. This signal triggers a restart operation, for which special recovery code needs to be written. ThreadScan [5] implements a mechanism which uses a shared *delete buffer*. A thread triggers reclamation by sending signals to all other active threads, which scan their stacks and registers to mark deleted nodes in the shared buffer if they are still used. ForkScan [4] also uses signals as well as copy-on-write OS optimizations for fork(2). A reclaimer thread creates a child process which contains a "frozen" memory snapshot; this process can scan deleted nodes in parallel. QSense [6] uses quiescent-state reclamation in its fast path, but hazard pointers back it up when threads do not respond. It also relies on the specific behavior of an OS scheduler, which needs to context switch threads periodically. Yet another approach [13] uses an OS page fault mechanism to alleviate the costs related to memory barriers in hazard pointers.

Software transactional memory (STM) can simplify concurrent programming and memory reclamation. OneFile [36] is a recent wait-free STM implementation with its own, STM-specific memory reclamation. While OneFile's framework enables the implementation of a wide range of wait-free algorithms by directly converting sequential data structures, customized wait-free data structures can often better utilize parallelism and achieve higher overall performance. Our work, which presents a universal memory reclamation scheme for such *arbitrary* data structures, closes this gap.

Hardware transactional memory (HTM) mechanisms are also widely used in concurrent programming. For example, reference counting can be accelerated using HTM [14]. Another approach, StackTrack [3], encapsulates read operations in HTM transactions; a concurrent thread will abort an HTM transaction when an object is no longer live but still in use. Since typical HTMs lack wait-free progress guarantees, it is not clear how to use HTMs for wait-free memory reclamation.

7 Conclusion

We presented the first practical wait-free memory reclamation scheme called *Wait-Free Eras* (WFE). Like Hazard Eras, it achieves great performance while providing compatibility with Hazard Pointers. Unlike Hazard Eras or any other existing universal technique, WFE guarantees that *all* memory reclamation operations are wait-free.

Since WFE eliminates a serious obstacle in wait-free programming, i.e., wait-free memory reclamation, we hope that it spurs further research in this area and the practical adoption of wait-free algorithms.

Availability

WFE's code is available at https://github.com/rusnikola/wfe.

Acknowledgments

We thank the anonymous reviewers for their insightful comments, which helped improve the paper. This work is supported in part by ONR under grants N00014-16-1-2711 and N00014-18-1-2022 and AFOSR under grant FA9550-16-1-0371.

References

[1] 2016. KPQueue and CRTurnQueue for Hazard Pointers. https://github.com/pramalhe/ConcurrencyFreaks.

[2] 2017. Windows 8.1 System Requirements. https://support.microsoft.com/en-us/help/12660/windows-8-system-requirements/.

[3] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. 2014. StackTrack: An Automated Transactional Approach to Concurrent Memory Reclamation. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 25, 14 pages. https://doi.org/10.1145/2592798.2592808

[4] Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. 2017. Forkscan: Conservative Memory Reclamation for Modern Operating Systems. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 483–498. https://doi.org/10.1145/3064176.3064214

[5] Dan Alistarh, William M. Leiserson, Alexander Matveev, and Nir Shavit. 2015. ThreadScan: Automatic and Scalable Memory Reclamation. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '15)*. ACM, New York, NY, USA, 123–132. https://doi.org/10.1145/2755573.2755600

[6] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. 2016. Fast and Robust Memory Reclamation for Concurrent Data Structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '16)*. ACM, New York, NY, USA, 349–359. https://doi.org/10.1145/2935764.2935790

[7] 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 (SPAA '13)*. ACM, New York, NY, USA, 33–42. https://doi.org/10.1145/2486159.2486184

[8] Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC '15)*. ACM, New York, NY, USA, 261–270. https://doi.org/10.1145/2767386.2767436

[9] Nachshon Cohen. 2018. Every Data Structure Deserves Lock-free Memory Reclamation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 143 (Oct. 2018), 24 pages. https://doi.org/10.1145/3276513

[10] Nachshon Cohen and Erez Petrank. 2015. Automatic Memory Reclamation for Lock-free Data Structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 260–279. https://doi.org/10.1145/2814270.2814298

[11] Nachshon Cohen and Erez Petrank. 2015. Efficient Memory Management for Lock-Free Data Structures with Optimistic Access. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '15)*. ACM, New York, NY, USA, 254–263. https://doi.org/10.1145/2755573.2755579

[12] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. 2002. Lock-free reference counting. *Distributed Computing* 15, 4 (01 Dec 2002), 255–271. https://doi.org/10.1007/s00446-002-0079-z

[13] 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 (ISMM 2016)*. ACM, New York, NY, USA, 36–45. https://doi.org/10.1145/2926697.2926699

[14] Aleksandar Dragojević, Maurice Herlihy, Yossi Lev, and Mark Moir. 2011. On the Power of Hardware Transactional Memory to Simplify Memory Management. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC '11)*. ACM, New York, NY, USA, 99–108. https://doi.org/10.1145/1993806.1993821

[15] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the BSDCan Conference, Ottawa, Canada*.

[16] Keir Fraser. 2004. *Practical lock-freedom*. Technical Report UCAM-CL-TR-579. University of Cambridge, Computer Laboratory. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf

[17] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. 2009. Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting. *IEEE Transactions on Parallel and Distributed Systems* 20, 8 (Aug 2009), 1173–1187. https://doi.org/10.1109/TPDS.2008.167

[18] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-lists. In *Distributed Computing*, Jennifer Welch (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 300–314.

[19] 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. https://doi.org/10.1016/j.jpdc.2007.04.010

[20] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. 2005. Nonblocking Memory Management Support for Dynamic-sized Data Structures. *ACM Trans. Comput. Syst.* 23, 2 (May 2005), 146–196. https://doi.org/10.1145/1062247.1062249

[21] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2002. The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures. In *Proceedings of the 16th International Conference on Distributed Computing (DISC '02)*. Springer-Verlag, Berlin, Heidelberg, 339–353. http://dl.acm.org/citation.cfm?id=645959.676129

[22] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[23] Alex Kogan and Erez Petrank. 2011. Wait-free Queues with Multiple Enqueuers and Dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. ACM, New York, NY, USA, 223–234. https://doi.org/10.1145/1941553.1941585

[24] Alex Kogan and Erez Petrank. 2012. A Methodology for Creating Fast Wait-free Data Structures. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 141–150. https://doi.org/10.1145/2145816.2145835

[25] L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (Sept. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[26] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, O. Krieger, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. 2001. Read-Copy Update. In *In Ottawa Linux Symposium*. 338–367.

[27] Maged M. Michael. 2004. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (June 2004), 491–504. https://doi.org/10.1109/TPDS.2004.8

[28] Maged M. Michael and Michael L. Scott. 1995. *Correction of a Memory Management Method for Lock-Free Data Structures.* Technical Report. University of Rochester, Computer Science.

[29] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, New York, NY, USA, 317–328. https://doi.org/10.1145/2555243.2555256

[30] Ruslan Nikolaev and Binoy Ravindran. 2019. Brief Announcement: Hyaline: Fast and Transparent Lock-Free Memory Reclamation. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC '19)*. ACM, New York, NY, USA, 419–421. https://doi.org/10.1145/3293611.3331575

[31] Manuel Pöter and Jesper Larsson Träff. 2018. Brief Announcement: Stamp-it, a More Thread-efficient, Concurrent Memory Reclamation Scheme in the C++ Memory Model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA '18)*. ACM, New York, NY, USA, 355–358. https://doi.org/10.1145/3210377.3210661

[32] Pedro Ramalhete and Andreia Correia. 2016. A Wait-Free Queue with Wait-Free Memory Reclamation (Full Paper). https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/crturnqueue-2016.pdf

[33] Pedro Ramalhete and Andreia Correia. 2017. Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '17)*. ACM, New York, NY, USA, 367–369. https://doi.org/10.1145/3087556.3087588

[34] Pedro Ramalhete and Andreia Correia. 2017. Hazard Eras - Non-Blocking Memory Reclamation (Full Paper). https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/hazarderas-2017.pdf

[35] 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 (PPoPP '17)*. ACM, New York, NY, USA, 453–454. https://doi.org/10.1145/3018743.3019022

[36] P. Ramalhete, A. Correia, P. Felber, and N. Cohen. 2019. OneFile: A Wait-Free Persistent Transactional Memory. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 151–163. https://doi.org/10.1109/DSN.2019.00028

[37] R. K. Treiber. 1986. *Systems Programming: Coping with Parallelism.* Technical Report RJ 5118. IBM Almaden Research Center.

[38] John D. Valois. 1995. Lock-free Linked Lists Using Compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*. ACM, New York, NY, USA, 214–222. https://doi.org/10.1145/224964.224988

[39] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. 2018. Interval-based Memory Reclamation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/3178487.3178488

[40] Chaoran Yang and John Mellor-Crummey. 2016. A Wait-free Queue As Fast As Fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 16, 13 pages. https://doi.org/10.1145/2851141.2851168

A Artifact appendix

A.1 Abstract

The artifact includes our extended version of the benchmark [39]. It includes our new WFE scheme as well as existing schemes: HE, HP, EBR, and 2GEIBR. The test set of lock-free data structures is extended with KP and CRTurn wait-free queues. The benchmark requires Linux with libjemalloc and libhwloc (on an x86_64 machine that supports cmpxchg16b). The gcc compiler must support C++11 as well as extended inline assembly features for cmpxchg16b (we tested gcc 8.3.0).

A.2 Artifact check-list (meta-information)

- **Algorithm:** New algorithm, Wait-Free Eras (WFE).
- **Program:** The benchmark with WFE implementation.
- **Compilation:** gcc with GCC_ASM_FLAG_OUTPUT and C++11 support.
- **Binary:** Linux ELF (x86_64) executables.
- **Run-time environment:** Ubuntu 18.04.3 LTS.
- **Hardware:** Any multi-core x86_64 with cmpxchg16b support; we tested it on 4x24 Intel Xeon E7-8890 v4 (2.20GHz).
- **Execution:** The execution time is passed through a program parameter.
- **Output:** The output is produced in the CSV format. PDF plots can be generated from CSV files.
- **Experiments:** A single python script runs all presented test cases.
- **Workflow frameworks used?:** No.
- **Publicly available?:** Yes.
- **Artifacts publicly available?:** Yes.
- **Artifacts functional?:** Yes.
- **Artifacts reusable?:** Yes.
- **Results validated?:** Yes.

A.3 Description

A.3.1 How delivered

The artifact is available through the public repository: https://github.com/rusnikola/wfe.

A.3.2 Hardware dependencies

Any multi-core x86_64 with cmpxchg16b support; we tested the benchmark on 4x24 Intel Xeon E7-8890 v4 (2.20GHz). For the Leak Memory experiment, 256GB of RAM is recommended, albeit RAM can still be much smaller if occasional outliers are acceptable.

A.3.3 Software dependencies

Linux with gcc (C++11 and GCC_ASM_FLAG_OUTPUT support), python, libjemalloc, and libhwloc. In our setup with Ubuntu 18.04.3, the following packages were installed: g++-8, gcc-8, libhwloc-dev, libjemalloc-dev, python. (The default gcc version was set to 8.) To draw charts, R is required: littler, r-cran-plyr, r-cran-ggplot2.

A.4 Installation

```
make [Release Version]
make debug [Debug Version]
```

A.5 Experiment workflow

- Compile the benchmark.
- Run tests. We provide testscript_wfe.py which runs all tests presented in the paper.
- For individual tests, you can also invoke tests directly. For example, for WFE's hash map test (10 seconds):

  ```
  ./bin/main -i 10 -m 3 -v -r 1 -o hashmap.csv
             -t 4 -d tracker=WFE
  ```

  For HE's hash map:

  ```
  ./bin/main -i 10 -m 3 -v -r 1 -o hashmap.csv
             -t 4 -d tracker=HE
  ```

  (You can see all options by running ./bin/main -h.)
- Plot the results. See below.

A.6 Evaluation and expected result

Throughput and the number of unreclaimed objects of WFE should roughly correspond to HE. Other algorithms such as 2GEIBR and EBR should have relatively similar performance to WFE and HE, i.e., the gap is typically not very large. HP, on the other hand, should typically have significantly worse throughput but a smaller number of unreclaimed objects.

Running all tests:

```
cd ./ext/parharness/scripts
mkdir -p data/final
nohup ./testscript_wfe.py &
```

The results will be in data/final/*.csv. Note that this script takes long time to complete. For rough results, the number of iterations in testscript_wfe.py can be reduced.

Drawing PDF plots:

```
mv ./ext/parharness/scripts/data/final
       ./data/final
cd ./data/scripts
./genplots.sh
```

PDF plots will be placed in ./data/final/*.pdf. Note that genplots.sh runs *.R scripts from the same directory. These scripts are adjusted for the parameters in ./testscript_wfe.py. When you change ./testscript_wfe.py, these scripts may need to be changed accordingly.

A.7 Experiment customization

The original benchmark we used [39] is highly-customizable. New reclamation schemes can be added to src/trackers (we added WFE), whereas new data structure tests can be added to src/rideables (we added KP and CRTurn queues).