Interval-Based Memory Reclamation

Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott
Computer Science Department
University of Rochester
Rochester, NY, USA
{hwen5,jhi1,wcai6,hbeadle,scott}@cs.rochester.edu

Abstract

In this paper we present *interval-based reclamation* (IBR), a new approach to safe reclamation of disconnected memory blocks in nonblocking concurrent data structures. Safe reclamation is a difficult problem: a thread, before freeing a block, must ensure that no other threads are accessing that block; the required synchronization tends to be expensive. In contrast with epoch-based reclamation, in which threads reserve all blocks created after a certain time, or pointer-based reclamation (e.g., hazard pointers), in which threads reserve individual blocks, IBR allows a thread to reserve all blocks known to have existed in a bounded *interval* of time. By comparing a thread's reserved interval with the lifetime of a detached but not yet reclaimed block, the system can determine if the block is safe to free. Like hazard pointers, IBR avoids the possibility that a single stalled thread may reserve an unbounded number of blocks; unlike hazard pointers, it avoids a memory fence on most pointer-following operations. It also avoids the need to explicitly "unreserve" a no-longer-needed pointer.

We describe three specific IBR schemes (one with several variants) that trade off performance, applicability, and space requirements. IBR requires no special hardware or OS support. In experiments with data structure microbenchmarks, it also compares favorably (in both time and space) to other state-of-the-art approaches, making it an attractive alternative for libraries of concurrent data structures.

CCS Concepts • **Theory of computation** → *Shared memory algorithms*; • **Software and its engineering** → *Garbage collection*;

Keywords shared memory, garbage collection, parallel algorithms

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPoPP '18, February 24–28, 2018, Vienna, Austria

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-4982-6/18/02…$15.00
https://doi.org/10.1145/3178487.3178488

1 Introduction

Pointer-rich nonblocking concurrent data structures pose a difficult safety challenge for memory managers. In a lock-based structure, a block (node) that is detached from the structure can generally be reclaimed (freed) safely and immediately, because the locking discipline ensures that no other thread is using it. In contrast, in a nonblocking structure, the detaching thread must generally wait until it knows that no other thread retains a local pointer to the block before reclaiming it. Otherwise, the local pointer may become a dangling reference, leading to incorrect behavior.

Some languages and systems address the problem by relying on fully automatic garbage collection, typically based on tracing. While this tends to eliminate large classes of bugs, it also tends to result in both higher and more variable time and space overheads, and precise collection requires a type-safe language.

In this paper we focus on manual storage management as found in languages like C and C++. Here the programmer must explicitly *retire* no-longer-needed blocks. For nonblocking algorithms, manual storage management introduces the problem that retirement (and reclamation) of a block may race with concurrent accesses to the block. Past researchers have introduced a variety of mechanisms to resolve this race, all of which can be seen, broadly, as allowing a thread to *reserve* the set of blocks it is using, preventing concurrent reclamation. A thread that detaches a block from the structure calls into the memory manager to retire the block, but the memory manager refrains from actually reclaiming the block until it is no longer reserved by any thread.

Epoch-based memory management [12, 15] is usually the fastest manual approach, and is relatively easy to implement. In this approach, a global *epoch counter* is incremented periodically. A thread performing an operation on the data structure records the epoch in which it begins the operation (typically in a global array), implicitly reserving all blocks that were not retired before the start of that epoch. The problem with this approach is that if some thread stalls in the middle of a data structure operation, all other threads may be prevented from reclaiming any blocks. We use the term *robust* [3, 8] to describe a memory management scheme that does not suffer from this problem.

Fine-grained approaches to memory reclamation, such as hazard pointers [17, 20], allow a thread to reserve only

those blocks it is actually using, avoiding unbounded implicit reservation. Such approaches are robust, but tend to have high run-time overhead (in particular, they tend to incur a write-read memory fence each time a pointer is dereferenced). Fine-grained approaches also generally require the programmer to manage reservations explicitly, and in particular to call an "unreserve" routine each time a pointer is no longer needed. (While reserving may sometimes be hidden within the pointer-dereference operation [e.g., using smart pointers], only the programmer knows when unreserving should occur.) Recent proposals [3, 25] have tried to reduce the run-time overhead of fine-grained approaches, but retain the need to manage reservations explicitly on a block-by-block basis.

In this paper, we introduce *interval based reclamation* (IBR), a new approach to memory management inspired by the epoch-based reclamation of RCU [15, 19]. IBR has low run-time overhead, small constant space per thread (regardless of data structure size or operation complexity), and an API that avoids the need for unreserve. IBR is robust and nonblocking, and requires neither special hardware nor assistance from the operating system. With minor modifications to data structure methods (Sec. 4.3.1), IBR can also bound the memory that may be reserved by a starving (but not stalled) thread.

2 Background

Interval-based reclamation schemes fall into a category of memory managers that might be described as "lightweight"—that is, they require only the standard support from the underlying system (no kernel modifications or signal or page fault handlers), impose minimal burden on the programmer, and, in general, have no restrictions on the class of data structures they can manage. In short, we are interested in reclamation schemes that can be data structure specific and need not be integrated into a wider garbage collection system. Prior art that fits these requirements is described in this section, together with the programming model these systems (and ours) expect; Section 6 takes a closer look at more specialized or heavyweight systems.

2.1 Programming Model and Definitions

In this paper, we focus on languages (e.g. C/C++) in which the programmer allocates and reclaims memory explicitly. To be correct, a program must ensure that no memory block is ever accessed after it is reclaimed, and no block is reclaimed more than once. The challenge for nonblocking data structures is then to determine when no other thread is still accessing a block that has been detached from the data structure. This determination is the purview of the reclamation system, which provides programmers with a retire operation and takes responsibility for detached blocks, reclaiming them only once outstanding thread-local references are known to have disappeared.

As is conventional, we assume a programming model in which a block has the following life course. It begins by being *allocated*. After the allocating thread has initialized the block, the block is *published* to the shared space (using a CAS or write) so that other threads can find it. Once the block has served its use, it is *detached* from all other blocks using another CAS or write; no thread that has not yet dereferenced the block can subsequently access it (but some threads may still hold a reference from the time during which it was shared). The block, being detached, can then be *retired* and handed over to the reclamation system. The reclamation system monitors potential outstanding references to the retired block and, once all such references are lost, the block is automatically *reclaimed*. Blocks that have been retired but not yet reclaimed are stored in a thread-local or shared *retired list*. Correct applications will retire any given block no more than once, and only after the block has been detached. Following standard convention, we assume that no thread holds a reference to a block across data structure operations.

In general, threads *reserve* references to blocks that they might dereference by updating some sort of metadata accessible to the reclamation system; blocks that are reserved cannot, in general, be reclaimed. Memory management schemes differ widely in the form that these *reservations* take. As mentioned in the introduction, it is desirable for a reclamation scheme to be *robust*[1] to stalled (e.g., preempted) threads; that is, the reservation of a stalled thread should prevent only a bounded number of blocks from being reclaimed.

Our interval-based schemes (and most of the comparison schemes) share an API, the methods of which are shown in Figure 1 along with their default implementations for schemes that do not overload the method.

2.2 Epoch-Based Reclamation

Epoch-based reclamation (EBR) uses *global epochs* as thread reservations. When a thread begins a data structure operation, it reads the global epoch and posts the value as its reservation. When a thread wishes to reclaim a retired block, it verifies that the block was retired (and therefore detached) before the earliest epoch reserved by any active thread. If this condition is satisfied, the block cannot be referenced by any active thread, since all such threads started their operations after the block was disconnected; the block can therefore be reclaimed safely. Notably, epoch-based reclamation schemes are not robust: a stalled thread can prevent all future blocks (even those not yet allocated) from being reclaimed.

Epoch-based reclamation schemes differ in the way that they increment the global counter. In Fraser's original proposal [12], a thread will increment the counter once all threads have made a reservation in the current epoch. In

[1]Our use of the term "robust" is consistent with that of Dice et al. and Balmau et al. [3, 8]. Other authors (e.g., Hart et al. [15]) have called this property "nonblocking," a term we avoid due to overloaded meaning.

```
// Allocate memory for a block on demand:
Function alloc(int size) : block*

// Indicate that a block is detached and will not be
   used by any future application operation:
Function retire(block* ptr) : void

// Indicate start of an application operation:
Function start_op() : void

// Indicate end of an application operation:
Function end_op() : void

// Read a block pointer from shared memory:
Function read(block** ptraddr) : block*
    return *ptraddr
// Update a shared pointer:
Function write(block** ptraddr, block* ptr) : void
    *ptraddr = ptr
// Conditionally update a shared pointer:
Function CAS(block** ptraddr, block* old, block* new)
  : bool
    return ptraddr → compare_and_swap(old, new)
// Optional, not needed for IBR:
Function unreserve(block* ptr) : void
```

Figure 1. Memory Management API.

quiescent-state–based reclamation (QSBR) [15], a thread will increment the epoch once it has verified that all threads have moved through a *quiescent state*, in which they hold no pointers (e.g., they have exited a data structure operation). As the epoch updates only periodically, we can generally assume that a 64-bit integer will never overflow in practice.

Code for a simple epoch-based scheme appears in Figure 2. A thread increments the epoch counter after every x retirements (in the experiments of Section 5, we used a value that leads to roughly one update every $100\mu s$). A thread posts its reservation (a copy of the global epoch) upon entering a data structure operation, and sets the reservation to a maximum value upon leaving. Retired blocks are stored in a thread-local list. Periodically, each thread scans all reservations of threads currently active in the data structure, taking note of the minimum. It then traverses its retire list and reclaims any block retired before the earliest reserved epoch.

Figure 3 shows a simple example of the APIs in Figure 2 in a non-blocking concurrent linked list. Please note that the set-ups happen before any thread is invoked, and functions of threads A and B can be invoked concurrently.

2.3 Pointer-Based Reclamation

In contrast to EBR, which reserves whole sets of blocks associated with a global epoch number, pointer-based approaches to reclamation reserve individual blocks. The canonical example is Michael's *hazard pointer* (HP) scheme [20], developed concurrently with the similar pass-the-buck [17] scheme.

```
   // globals
1  int epoch
2  int reservations[thread_cnt]
3  thread_local int counter
4  thread_local list retired
5  epoch_freq // freq. of increasing epoch
6  empty_freq // freq. of reclaiming retired
   // private function
7  Function empty() : void
8      max_safe_epoch = reservations.min()
9      for block ∈ retired do
           /* all blocks retired IN or AFTER
              max_save_epoch will be protected        */
10         if block.retire_epoch < max_safe_epoch then
11             free(block)
   // public interface
12 Function retire(block* ptr) : void
13     retired.append(ptr)
14     ptr→retire_epoch = epoch
15     counter++
16     if counter % epoch_freq == 0 then
17         fetch_and_increment(epoch)
18     if retired.cnt % empty_freq == 0 then
19         empty()
20 Function start_op() : void
21     reservations[tid] = epoch
22 Function end_op() : void
23     reservations[tid] = MAX
```

Figure 2. EBR memory management.

With hazard pointers, a thread posts a pointer to a shared block immediately before accessing that block. For nontrivial structures, a thread may need to reserve multiple blocks concurrently, for which it needs multiple hazard pointers. The details are somewhat subtle: a thread must read the pointer, write it to a hazard pointer, issue a write-read memory fence, and re-read the pointer to make sure it hasn't changed; only then can it safely dereference it. Periodically, each thread scans all hazard pointers of all threads. It can then safely reclaim all blocks that were on the retired list at the beginning of the scan, and for which no hazard pointer was found.

Unfortunately, where EBR requires the programmer to call only start_op, retire, and end_op, pointer-based reclamation schemes require the programmer to indicate every point in the code at which a pointer is read for the first time and every point at which it is used for the last time. Initial use annotations can, with some overhead, be rolled into the pointer read

```
// set-ups
1  memory_manager<node> mm; // 3 nodes with
   values 0, 2 and 4
2  node n0(0), n1(2), n2(4)
3  n0.next = &n1
4  n1.next = &n2
5  n2.next = NULL
6  node* head = &n
   // 2 worker threads
7  Function tA() : int
       // Thread A tries to read n1
8      mm.start_op()
9      node* target = &n1
10     node* p1 = mm.read(&target)
11     if p1 ≠ NULL // n1 still exists
12     then
13         int ret = p1->val; mm.end_op()
14         return ret;
15     else
16         mm.end_op()
17         return NULL;
18 Function tB() : void
       // Thread B tries to update n1
19     while true do
20         mm.start_op()
21         node* new_n1 = mm.alloc(sizeof(node))
22         new_n1->val = 3
23         new_n1->next = &n2
24         node* target = &n0
25         node* p0 = mm.read(&target)
26         if p0 ≠ NULL then
27             if mm.CAS(&p0->next, &n1, new_n1) then
28                 mm.retire(&n1)
29                 mm.end_op()
30                 return
```

Figure 3. EBR usage example.

operator (e.g., using the smart pointer idiom in C++), but final use annotations must generally be explicit—hence the unreserve method in Figure 1. To minimize instrumentation overhead, straightforward implementations of pointer-based schemes also tend to provide a fixed number of hazard pointers for each thread; this limitation can be problematic for data structures like balanced search trees, in which a potentially rotation-inducing update may want to reserve every node on a path of arbitrary length.

A new *Hazard Era* (HE) scheme, developed by Ramalhete and Correia [25], merges hazard pointers and epochs in a way that defies easy categorization. As in EBR, threads periodically increment a global epoch number. As in HP, each thread makes a reservation before accessing a block, and clears the reservation when the block leaves its working set. Instead of the address of the block, however, the reservation indicates the epoch in which the block was accessed. Moreover, each block is tagged at allocation time with the epoch in which it was created. By looking at the set of all thread reservations, a reclaimer can determine the set of "active epochs"— the epochs in which a pointer to a block may have been read. Any block that was retired before any active epoch or that was created after any active epoch (and subsequently retired) can safely be reclaimed. Ramalhete and Correia make a key observation in this work: block lifetimes can be used to determine reachability for reclamation. We leverage this idea further in our work.

3 Interval-Based Reclamation

Like EBR, *interval-based reclamation* (IBR) maintains a global epoch counter. Like HE, it also tracks the *birth epoch* and *retire epoch* (if any) of each block. By identifying a finite range of epochs, a thread can reserve all blocks with intersecting birth–retire *lifetimes*. Since threads reserve only a finite number of epochs, IBR is robust. At the same time, the fact that a reserved epoch can "cover" many blocks eliminates the need for unreserve, making IBR as easy to use as EBR.

Interval-based reclamation is in general agnostic with regard to the technique used to increment the global epoch counter. In our prototype implementations, threads increment the counter periodically as they allocate blocks, much as in Figure 2. This convention bounds the number of blocks allocated in any given epoch, making it particularly easy to demonstrate robustness. Blocks in IBR are tagged with a birth epoch upon allocation; this 64 bit field is generally stored in the block header managed by the allocator (and hidden from the application). Retire epochs are added upon retirement. Like all other presented schemes, our IBR schemes use thread-local *retire lists* to store blocks that have been retired; each thread periodically traverses its list to find blocks that are safe to reclaim.

3.1 Persistent Object IBR

Our simplest scheme is primarily applicable to data structures that are *persistent*, in the sense that they preserve their own history [10]. Persistent data structures follow one critical rule: all pointers other than the root are immutable; modifications always entail linking new node(s) to unchanged parts of the existing structure. Simple examples of persistent concurrent data structures are the Treiber lock-free stack [26] and anything built using Herlihy's lock-free universal construction [16]. Data structures of this sort are widely used in functional programming languages, where

the ability to share space among multiple versions provides an efficient alternative to mutating a single version [24].

Pseudocode for persistent-object IBR (POIBR) appears in Figure 4. Its operation resembles that of EBR, except that instead of reserving all blocks not retired before a given epoch, a reservation reserves only blocks whose lifetimes intersect the reserved epoch. A thread executing an application method reserves the epoch in which it first reads the root of the data structure. In a manner similar to the setting of a hazard pointer, the thread reads the global epoch and the root pointer, posts the epoch to a globally visible location to reserve it, executes a write-read fence, and then double-checks the global epoch to make sure it hasn't changed. This "snapshot" technique ensures that the root's contents were indeed active during the reserved epoch (see the read method, line 25 in Figure 4). Due to the "immutable pointers" rule of persistent data structures, every block reachable from this root is also known to have been active during the reserved epoch, so the thread's reservation intersects the lifetimes of all blocks it could possibly read during the remainder of its application operation.

3.2 Tagged Pointer IBR

Many data structures, of course, are not persistent. Our second IBR scheme, *tagged pointer IBR* (TagIBR) is applicable to arbitrary nonblocking concurrent structures. It allows each thread to reserve a finite *range* of epochs. It also maintains, as an extra field in each pointer, an epoch number guaranteed to be greater than or equal to the *birth epoch* of the block to which the pointer refers. Pseudocode for a portable variant of TagIBR is shown in Figure 5.

When starting an application operation, a thread sets both the upper and lower endpoints of its reservation to match the current epoch. Each time it reads a pointer (but before dereferencing that pointer), it checks whether the epoch in the pointer is less than or equal to the upper endpoint of the reservation range; if not, it updates the endpoint to cover it (see the read method at line 46 of Fig. 5). Since we know that each encountered block was reachable after the lower endpoint of the thread's epoch range, and since the upper endpoint is kept greater than or equal to the epoch in which the block we are starting to read was created, the thread's reserved interval is guaranteed to intersect with the lifetime of the block.

Whenever we perform a write or CAS of a shared pointer, both the born_before and address (p) fields of the pointer may need to be updated. In the absence of hardware support for multi-word atomic update (more on this below), we will need to modify the two fields separately. To maintain correctness, we adopt the convention of never reducing the value in the born_before field. During a write or CAS operation, we first update the born_before field if it currently seems to be too small, using compare_and_swap to make sure it increases

```
   // global structures same as in EBR
   // private function
 1 Function empty() : void
 2     for block ∈ retired do
 3         bool conflict = false; /* block is protected if
               some reserved epoch is in its interval    */
 4         for res ∈ reservations do
 5             if block.birth_epoch ≤ res ≤
                 block.retire_epoch then
 6                 conflict = true
 7         if !conflict then
 8             free(block)
   // public interface
 9 Function alloc(int size) : block*
10     counter++
11     if counter % epoch_freq == 0 then
12         fetch_and_increment(epoch)
13     block b = new block(size)
14     b⟶birth_epoch = epoch
15     return b
16 Function retire(block* ptr) : void
17     retired.append(ptr)
18     ptr⟶retire_epoch = epoch
19     if retired.cnt % empty_freq == 0 then
20         empty()
21 Function start_op() : void
22     reservations[tid] = epoch
23 Function end_op() : void
24     reservations[tid] = MAX
   // used only when reading data structure root
25 Function read(block** rootaddr) : block*
26     while true do
27         reservations[tid] = epoch
28         ret = *rootaddr
29         if reservations[tid] == epoch then
30             return ret
```

Figure 4. Persistent Object IBR.

monotonically. We then update the address field, with a store or compare_and_swap, as appropriate.

When updating a pointer to refer to an older block, or when racing with another thread, this two-step technique may lead to arbitrary amounts of "slack" in the born_before field. Fortunately, this slack is very close to harmless in practice. Since a thread sets the lower endpoint of its epoch to

```
1  Class Reservation
2      int lower, upper
3  Class TPointer
4      int born_before // monotonically increasing
5      block* p
6      Function protected_CAS(block* ori, block* new) : bool
7          repeat
8              ori_bb = born_before
9          until new→birth_epoch ≤ ori_bb or
             compare_and_swap(&born_before, ori_bb,
             new→birth_epoch)
10         return compare_and_swap(&p, ori, new)
11     Function protected_write(block* ptr) : void
12         repeat
13             ori_bb = born_before
14         until ptr→birth_epoch ≤ ori_bb or
             compare_and_swap(&born_before, ori_bb,
             ptr→birth_epoch)
15         p = ptr
16 int epoch
17 Reservation reservations[thread_cnt]
18 thread_local int counter
19 thread_local list retired
20 int epoch_freq // freq. of increasing epoch
21 int empty_freq // freq. of reclaiming retired
22 Function empty() : void
23     for block ∈ retired do
24         bool conflict = false
25         for res ∈ reservations do
               /* block protected if some epoch reserved by some
                  thread is in its interval                    */
26             if block.birth_epoch ≤ res.upper and
                 block.retire_epoch ≥ res.lower then
27                 conflict = true
28         if !conflict then
29             free(block)
   // public interface
30 Function alloc(int size) : block*
31     counter++
32     if counter % epoch_freq == 0 then
33         fetch_and_increment(epoch)
34     block b = new block(size)
35     b→birth_epoch = epoch
36     return b
37 Function retire(block* ptr) : void
38     retired.append(ptr)
39     ptr→retire_epoch = epoch
40     if counter % empty_freq == 0 then
41         empty()
42 Function start_op() : void
43     reservations[tid].lower = reservations[tid].upper =
         epoch
44 Function end_op() : void
45     reservations[tid].lower = reservations[tid].upper = MAX
46 Function read(TPointer* ptraddr) : block*
47     while true do
48         ret = ptraddr→p
49         reservations[tid].upper =
             max(reservations[tid].upper,
             ptraddr→born_before)
50         if reservations[tid].upper ≥ ptraddr→born_before
             then
51             return ret
52 Function write(TPointer* target_ptraddr, block* ptr) : void
53     return target_ptraddr→protected_write(ptr)
54 Function CAS(TPointer* target_ptraddr, block* ori, block*
     new) : bool
55     return target_ptraddr→protected_CAS(ori, new)
```

Figure 5. Tagged pointer interval-based memory management.

the (monotonically increasing) global epoch whenever it starts an application operation, and only updates its upper endpoint, a "too large" born_before field has an impact only when an application operation that began execution long ago actively reads a new pointer. If we assume that application operations are all of modest length, this scenario is likely to arise only between the resumption of a previously preempted thread and the end of its current application operation.

3.2.1 TagIBR Variants

Our default TagIBR scheme requires no special hardware support beyond compare_and_swap, but it doubles the number of such instructions in each application operation. We can reduce this overhead on machines with appropriate instructions.

Using Fetch_and_add Like compare_and_swap (CAS), the atomic fetch_and_add (FAA) operation is widely available. Unlike CAS, however, FAA never fails. If n threads attempt a

CAS simultaneously, only one will succeed; if the remaining always retry, $O(n^2)$ time will be required for all to complete. By contrast, n FAA operations will complete in $O(n)$ time.

When updating the born_before field in a pointer, we can reduce overhead under contention by using FAA to add the difference between the current value and the desired value, instead of CAS-ing the new value in. This technique has the added advantage of making write and CAS operations wait free instead of merely lock free (read remains lock free). The downside is the potential for extra "slack" in a born_before field when competing threads update it concurrently.

Using Wide or Double CAS On a machine that can update two adjacent (wide CAS) or arbitrary (double CAS) memory words atomically, we can obviously dispense with the monotonically increasing convention for born_before fields in pointers, allowing us both to minimize the number of expensive instructions and to maintain the precise birth epoch of blocks. Like the FAA variant above, WCAS- or DCAS-based TagIBR makes write and CAS operations wait free.

Using a Type Preserving Allocator Type-preserving allocators provide the guarantee that a chunk of memory, once used for some type of data, will continue to be used for the same type. If we use a type-preserving allocator, we can remove the born_before field in the pointer and store this field in the block itself. If thread t reads a pointer from shared memory, and the block to which the pointer refers is retired and reclaimed before t has a chance to reserve it, the pointer may be dangling by the time t reads the born_before field. With type-preserving allocation, however, this read is still certain to return a valid epoch number, which does no harm if reserved by t. (Of course, if the block is reused, the fact that it was reclaimed implies that the global epoch counter will have changed, and t's double-check of the birth_epoch field in the block will fail, forcing a retry of the loop in read.)

By leveraging type preservation, we eliminate both the space overhead of born_before fields in pointers and the time overhead of a extra CAS instruction on every write or CAS operation. Finally, like both of our previous variants, this technique makes write and CAS operations wait free.

3.3 Two Global Epochs IBR

We can obtain all the advantages of the final TagIBR variant above—normal-sized pointers and no extra CASes—in a portable fashion if we are willing to tolerate somewhat less precise reservations. As in TagIBR, threads in two-global-epochs IBR (2GEIBR) reserve a range of epochs, and set the lower endpoint at the beginning of each application operation. When reading a pointer, however, a 2GEIBR thread updates its endpoint to the value of the global epoch counter, rather than the born_before field in the pointer. As in our previous schemes, it employs a "snapshot" idiom: first the pointer and global epoch are read, then the epoch is reserved if it is larger than the current upper epoch, and finally the global epoch is verified to be unchanged. Since the target of the pointer is active during the current global epoch, it was necessarily "born before" it. Thus, 2GEIBR inherits its correctness from TagIBR. Figure 6 shows the parts of 2GEIBR that differ from TagIBR.

```
/* global structures, alloc, retire, empty, start_op,
   and end_op same as in TagIBR; write and CAS same as
   in default (no instrumentation)                   */

1 Function read(block** ptraddr) : block*
2   while true do
3     ret = *ptraddr
4     reservations[tid].upper =
        max(reservations[tid].upper, epoch)
5     if reservations[tid].upper == global_epoch
       then
6       return ret
```

Figure 6. 2GE Interval-based memory management.

Key properties of existing and IBR reclamation techniques are summarized in Figure 7.

4 Correctness

This section provides arguments for the correctness of our IBR algorithms. Our arguments reason over *abstract histories*, composed of calls to and returns from the memory management API routines, and *concrete histories*, which include the instructions comprising the implementations of those routines. Note that start_op, alloc, read, write, CAS, retire, and end_op operations appear in both abstract and concrete histories; reclaim operations appear only in concrete histories.

In Section 4.2, we demonstrate the key safety property: no block is ever accessed after it is reclaimed. In Section 4.3, we show that our IBR schemes are both robust—a stalled thread can prevent the reclamation of only a bounded number of retired blocks—and nonblocking—in a bounded number of steps, some thread always makes progress.

4.1 Assumptions

We assume that any application using the memory management system is *well-behaved*. That is, it meets the following assumptions. First, the memory management operations performed by any one thread (the abstract *thread subhistory*) have the form (start_op (alloc, read, write, CAS, retire)* end_op)*, where each start_op…end_op sequence comprises an *application operation*—the execution of some method of the data structure. Second, the memory management operations performed on any one block (the abstract *block subhistory*) have the form alloc (read, write, CAS)* retire (read, write, CAS)*, with the added proviso that any access to the block that occurs after the retire must be within an

	Per-thread reservation	Pros and cons
EBR	Start epoch; covers everything not retired before then.	Unbounded reservation for stalled thread.
HP	Copy of every active pointer; covers precisely those blocks.	Non-constant reservation size; needs explicit unreserve.
HE	Epoch number for every active pointer; covers all blocks accessed in specified epochs.	Less precise than HP, but fewer memory fences.
POIBR	Start epoch; covers everything reachable from root at start time.	All pointers but the root must be immutable.
TagIBR	Start epoch + latest born-before value seen so far; covers all blocks whose [birth, retirement] intersects thread's epoch range.	Default implementation doubles size of pointers; imprecision due to slack from 2-step update. TagIBR-FAA has less contention, but more slack. TagIBR-WCAS has no slack, but requires WCAS/DCAS. TagIBR-TPA has no contention and no extra space per pointer, but requires type-preserving allocator.
2GEIBR	Start epoch + latest epoch in which a pointer was read; same coverage as TagIBR.	Less precision than TagIBR.

Figure 7. Summary of tradeoffs among memory reclamation techniques.

application operation in which a read of a pointer to the block occurred before the retire (i.e. only threads with a local reference to a block can access it after it is retired).

This final proviso is more subtle than it might at first appear. It outlaws code in which, for example, thread t_1 reads a pointer to block $\mathcal{B}$, thread t_2 retires block C, and then thread t_1 finds and uses a pointer to C within block $\mathcal{B}$. A sufficient (though not necessary) means of avoiding this scenario is to overwrite all shared pointers to a block before retiring it. This proviso is shared with both hazard pointers and hazard eras.

4.2 Safety

We argue that our IBR schemes display what we call *reclamation safety*: they guarantee that no block access can happen after the block has been reclaimed (or, equivalently, they delay reclamation until after the final access to the block).

Theorem 1 (Reclamation Safety)**.** *All presented IBR algorithms (POIBR, TagIBR, and 2GEIBR) are reclamation safe.*

Proof. For any given block $\mathcal{B}$ in concrete history H, consider the point in H at which the memory manager invokes free on $\mathcal{B}$. We wish to argue that after this point, no application operation will access $\mathcal{B}$ again. By well-behavedness, we can disregard all but active application operations that have already read a pointer to $\mathcal{B}$. We now consider these active application operations with a reference to block $\mathcal{B}$ for each IBR scheme.

POIBR: Since the root is the newest block in the data structure, and since all other pointers are immutable, the epoch in which the root was dereferenced by an active application operation will intersect the lifetimes of all blocks reachable from the root. Thus, any application operation that can read block $\mathcal{B}$ must have reserved an epoch that intersects with $\mathcal{B}$'s lifetime, and, consequently, the retire method will delay reclamation of $\mathcal{B}$ until all such application operations have completed.

TagIBR: Any application operation that has obtained a reference to block $\mathcal{B}$ will have reserved, through the read method, an interval that includes the lifetime of $\mathcal{B}$: the block was alive after the application operation began and the application operation's reserved range extends at least until the read of a pointer to the block. The retire method will therefore delay reclamation of $\mathcal{B}$ until all such application operations have completed.

2GEIBR: Any application operation that has obtained a reference to $\mathcal{B}$ will have reserved, through the read method, an interval that includes the lifetime of $\mathcal{B}$, since the block was still attached during the epoch of the dereference. The retire method will therefore delay reclamation of $\mathcal{B}$ until all such application operations have completed. □

4.3 Liveness

We now investigate two aspects of liveness for memory management schemes: robustness, which bounds the number of blocks that a stalled thread can reserve, and progress, which ensures that some thread always completes a memory management operation in a bounded number of steps.

4.3.1 Robustness

Given a history H, a set of "stalled" threads T, a history H' extending H without steps by threads in T, and a block $\mathcal{B}$ retired in H', we say $\mathcal{B}$ is *unreclaimable* with respect to H and T ($\mathcal{B} \in unrec(H,T)$) if $\forall H''$ extending H' without steps by threads in T, $\mathcal{B}$ is unreclaimed. Additionally, we say a memory management algorithm is *robust* if $\forall H, T\ \exists k$ s.t. $|unrec(H,T)| < k$.

We prove robustness by showing that, at any time, there is a bound on the number of unreclaimable blocks.

Theorem 2 (Robustness)**.** *All presented IBR algorithms—POIBR, TagIBR, and 2GEIBR—are robust.*

Proof. The same proof works for all IBR methods. According to the alloc method, there can be at most epoch_freq $\times$ num_threads blocks born in any given epoch e. Suppose

we are given a finite concrete history H and a finite set of threads T that are currently stalled. Clearly the stalled threads are collectively reserving a finite set of epochs $\mathcal{E}$. If the maximum epoch in $\mathcal{E}$ is E_{max}, then all blocks $\mathcal{B}$ that can possibly be made unreclaimable by reserving $\mathcal{E}$ have $\mathcal{B}$.birth_epoch $\leq E_{max}$. This implies that the number of unreclaimable blocks is bounded by $E_{max} \times$ epoch_freq $\times$ num_threads. □

Note that if some thread is "starved" rather than "stalled"—that is, it executes an application operation that takes an unbounded number of steps without completing—then the thread may still reserve an unbounded number of blocks. The solution to this problem—adopted in our code—is to modify each application operation so that it restarts from the beginning (with a new start epoch) after some fixed number of failed CAS operations.

4.3.2 Progress

We argue that our techniques are nonblocking; specifically:

Theorem 3 (Progress)**.** *All presented IBR algorithms (POIBR, TagIBR, and 2GEIBR) are lock free.*

Proof. **POIBR and 2GEIBR:** There is one unbounded loop in these algorithms, in read. If this loop retries, then either the global epoch counter has changed (due to a thread making progress on allocation), or the pointer value has changed due to a thread making progress in the application. Either way, some thread has made progress, so these algorithms are lock free.

TagIBR: There are two unbounded loops in the algorithm: one in protected_CAS and the other in read. For protected_CAS, the only reason to repeat the loop is that another thread swapped in a new value of born_before; a failed compare_and_swap is always caused by a successful one. For read, if the loop continues then the dereferenced pointer was modified—a write operation in another thread succeeded. Either way, a continuation in those loops means other threads are making progress, and, consequently, the algorithm is lock free. □

5 Performance Results

Our experimental evaluation was performed on an Intel machine with two Xeon E5-2699 v3 processors, each with 18 cores and 2 hyperthreads per core (72 hardware threads total). Each core has private L1 and L2 caches and an L3 cache shared across the processor. For all experiments, the first 18 threads are pinned to the 18 cores of one processor; the next 18 threads are pinned to the same cores' hyperthreads. We fill then fill the second processor in the same manner. Our code is written in C++ and compiled with g++ 6.3.1 and −O3 optimization. To eliminate contention on the global malloc, we used the jemalloc library [11].

Each test is a fixed-time microbenchmark in which threads randomly call operations on a shared key-value data structure using a random key, chosen from the range (0, 65536). Our tests are write-dominated. We first fill the key-value store with three quarters of the keys, then each thread randomly inserts or removes a random key. As representative data structures we picked the ordered list of Harris [14] and Michael [20], Michael's lock-free hash map [20], the binary tree of Natarajan and Mittal [23], and a lock-free variant of the Bonsai Tree [6], a persistent and balanced binary tree.

For memory managers that use global epoch counters, each thread increments the counter once every $n \times k$ allocations, where n is the number of active threads and k is tuned for high average performance across benchmarks and reclamation schemes (150 in our experiments). The dependence on n ensures that the rate of epoch number increase is roughly the same in all experiments, regardless of thread count.

We measured both throughput and space overhead (average number of retired but unreclaimed blocks) of:

No MM, a baseline that never reclaims memory.
Epoch, the EBR scheme of Section 2.2.
HP, the traditional hazard pointer scheme of Michael, introduced in Section 2.3 [20].
HE, the hazard era scheme of Ramalhete and Correia, introduced in Section 2.3 [25].
POIBR, our persistent object IBR scheme of Section 3.1.
TagIBR, our default tagged pointer IBR scheme (Fig. 5).
TagIBR-FAA, our tagged pointer scheme using FAA.
TagIBR-WCAS, our tagged pointer scheme using WCAS.
2GEIBR, our two-global-epochs IBR scheme of Section 3.3.

The tuning of memory managers is a trade-off between throughput and space utilization: trying to empty retire lists more frequently will minimize wasted space, but may waste time scanning reservations. For our microbenchmarks, if we scan on every kth retirement, space use climbs roughly linearly with k, while throughput remains essentially constant for $1 \leq k \leq 50$, suggesting that reclamation (which occurs largely in parallel, and off the critical path) is not the bottleneck. In a full application, however, overly aggressive reclamation might reduce the amount of time available for other useful work. For our experiments, we set $k = 30$ somewhat arbitrarily.

Figure 8 shows throughput results on our four data structures. In general, interval based reclamation schemes outperform hazard pointers (HP) by a large margin, and have similar performance to (fast but non-robust) epoch-based reclamation. For the linked list, in which each operation traverses a large number of shared pointers, TagIBR and its variants do particularly well. We didn't include precise approaches (HP and HE) for the Bonsai Tree because tree rotations require a statically unknown number of reservations. These can be accommodated, but only at the cost of

more expensive instrumentation and scans, which we opted not to impose on other data structures.

In contrast to the other three data structures, performance for the Bonsai Tree (Fig. 8d) peaks at just two threads, presumably due to the root-pointer bottleneck. There is also a dramatic drop in performance at around 35 threads. We believe this to be due to L1/L2 cache contention. Our thread pinning strategy, as described earlier, puts one thread on each core of the first socket, then populates the hyperthreads of that socket, before doing the same for the second socket. At 35 or 36 threads, only zero or one has a core to itself, and we lack the two fast threads required to saturate the bottleneck at the root. At 38 threads we again have two (on the second socket) that are able to run at full speed, and throughput recovers to almost what it was at 34 threads.

Figure 9 shows the average number of locally-retired but not yet reclaimed blocks at the start of each data structure operation. This metric gives a good estimate for how much space overhead the particular reclamation scheme incurs. The portion of each curve beyond a thread count of 72 captures the case in which threads are certain to be stalled. On the Bonsai Tree, Michael's hash map, and the Natarajan-Mittal tree, the IBR mechanisms consume up to 35% more memory than Hazard Eras and 10% to 40% less than Epoch-based reclamation; this fits our expectations well. On the linked list, which has relatively little memory churn relative to the size of operations, IBR doesn't deviate much from EBR, in either time or space.

As a final experiment, we repeated all the tests on a read-dominated workload (90% reads and 10% inserts and removes). Results were very similar to the write-dominated case. The one exception, shown in Figure 10, is the retired but not reclaimed block count on the Natarajan-Mittal tree, where 2GEIBR was the only IBR that outperformed EBR. Here again the explanation appears to be the more frequent scanning performed by performance-tuned EBR.

6 Related Work

Memory management for nonblocking data structures has been a research topic for over twenty years. However, the research community has yet to settle on any single scheme, since every presented alternative embodies major trade-offs.

As mentioned in Section 2, IBR is most closely comparable to epoch-based reclamation [12, 19], quiescent-state–based reclamation [15], hazard pointers [17, 20], and hazard eras [25]. Other, less self-contained approaches leverage OS support or specific hardware implementations. Several authors have used *signals*, issued by a reclaiming thread, to interrupt the execution of otherwise "invisible" reader threads. In DEBRA+ [5], the signal forces readers to abandon their current application operation and restart, thereby eliminating any stale pointers. Similarly, Threadscan [2] uses signals on reclamation to query active threads, who then respond to the reclaimer with a summary of their active state.

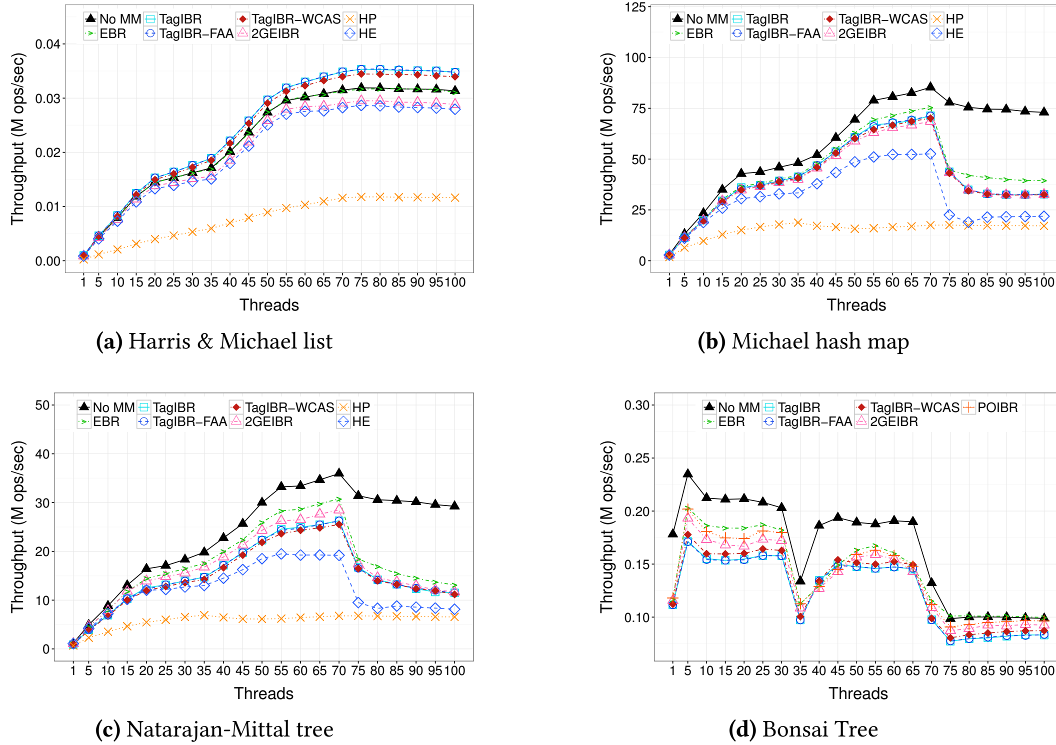
Other authors have leveraged *hardware transactional memory* to simplify and accelerate memory reclamation. Dragojević et al. [9] propose two schemes, one, using HTM to maintain reference counts, the other using HTM to protect a detaching CAS and the companion call to free. Stacktrack [1] improves on this scheme by using HTM to protect a reader's entire application operation, so that reclamation of a still-in-use block will trigger a transactional abort.

Dice et al. propose using page permissions to force page faults when a reclaimer wishes to reclaim blocks. These page faults ensure the global visibility of otherwise thread-local hazard pointer updates, while allowing the usual write-read fence to be elided on the default read path [8]. In a similar vein, Morrison et al. propose a memory consistency model called temporally bounded total store ordering (TBTSO), which guarantees that unsynchronized writes will eventually be propagated, again eliding fences on the default read path[22]. This scheme can be emulated on current OSs, but requires hardware modifications to be fully safe. Finally, in QSense [3] QSBR is used optimistically with hazard pointers as a back-up. To maintain robustness, threads are periodically swapped out by timed context switches, forcing hazard pointer modifications to become globally visible.

Other schemes for memory reclamation require less system overhead and are more portable, but are incomparable to IBR for other reasons. "Drop-the-anchor" [4] is a scheme inspired by hazard pointers for linked lists, where thread reservations cover a section of a list instead of a single block. Unfortunately, the scheme does not seem to generalize to other data structures. The more recent optimistic access strategy of Cohen and Petrank [7] is more general, but requires that data structures be written in a "normalized form" and that reads of reclaimed memory never trigger an exception—a difficult property to ensure. Finally, several nonblocking reference counting systems have been published [13, 18, 21, 27]. These systems associate a reference count with every block, update it on every pointer access, and reclaim blocks when counts reach zero. As noted by Hart et al. [15], such systems tend to be quite slow: each access requires a write-read fence.

7 Conclusion

In this paper, we have introduced an interval-based approach to memory reclamation for nonblocking concurrent data structures. IBR is fast, nonblocking, and robust: like hazard pointers, but unlike EBR, it bounds the space that may be tied up by a stalled (preempted) thread; like EBR, but unlike hazard pointers, it does so without substantial run-time overhead, without the awkwardness of manual reservation management, and without the need for any a priori limit on the number of reservations in a given application operation. Given these properties, we recommend IBR for any



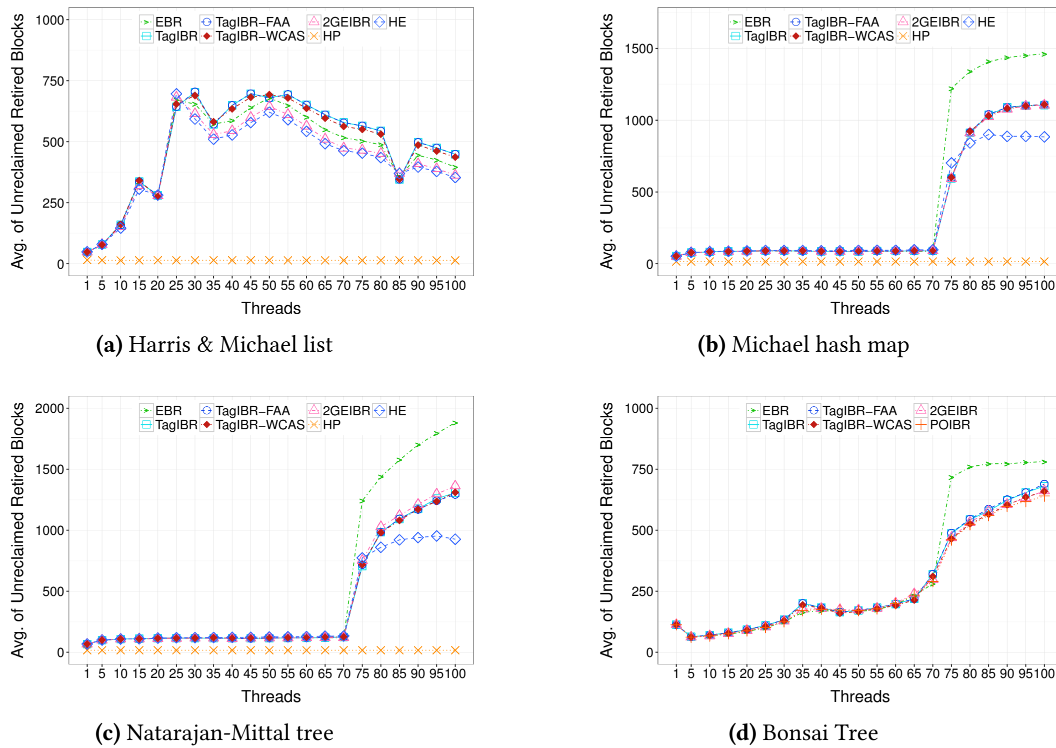
(a) Harris & Michael list

(b) Michael hash map

(c) Natarajan-Mittal tree

(d) Bonsai Tree

Figure 8. Throughput (operations per second).



(a) Harris & Michael list

(b) Michael hash map

(c) Natarajan-Mittal tree

(d) Bonsai Tree

Figure 9. Average unreclaimed but retired blocks per operation.

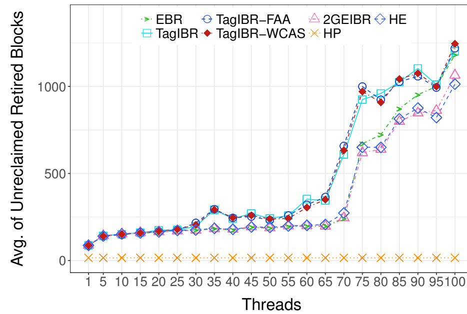


Figure 10. Natarajan-Mittal tree, read-dominated workload.

nonblocking concurrent data structure that needs to reclaim memory, particularly in the presence of multiprogramming or large numbers of application threads, when there is a significant chance that threads will be preempted.

Acknowledgments

This work was supported in part by NSF grants CNS-1319417, CCF-1337224, CCF-1422649, and CCF-1717712, and by a Google Faculty Research award.

References

[1] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. 2014. StackTrack: An Automated Transactional Approach to Concurrent Memory Reclamation. In *Proc. of the 9th European Conf. on Computer Systems (EuroSys '14)*. Amsterdam, The Netherlands, 25:1–25:14.

[2] Dan Alistarh, William M. Leiserson, Alexander Matveev, and Nir Shavit. 2015. ThreadScan: Automatic and Scalable Memory Reclamation. In *Proc. of the 27th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA '15)*. Portland, OR, USA, 123–132.

[3] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. 2016. Fast and Robust Memory Reclamation for Concurrent Data Structures. In *Proc. of the 28th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA '16)*. Pacific Grove, CA, USA, 349–359.

[4] Anastasia Braginsky, Alex Kogan, and Erez Petrank. 2013. Drop the Anchor: Lightweight Memory Management for Non-blocking Data Structures. In *Proc. of the 25th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA '13)*. Montréal, Québec, Canada, 33–42.

[5] Trevor Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proc. of the 2015 ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing (PODC '15)*. Donostia-San Sebastián, Spain, 261–270.

[6] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2012. Scalable Address Spaces Using RCU Balanced Trees. In *Proc. of the 17th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*. London, England, UK, 199–210.

[7] Nachshon Cohen and Erez Petrank. 2015. Efficient Memory Management for Lock-Free Data Structures with Optimistic Access. In *Proc. of the 27th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA '15)*. Portland, Oregon, USA, 254–263.

[8] Dave Dice, Maurice Herlihy, and Alex Kogan. 2016. Fast Non-intrusive Memory Reclamation for Highly-concurrent Data Structures. In *Proc. of the 2016 ACM SIGPLAN Intl. Symp. on Memory Management (ISMM '16)*. Santa Barbara, CA, USA, 36–45.

[9] Aleksandar Dragojević, Maurice Herlihy, Yossi Lev, and Mark Moir. 2011. On the Power of Hardware Transactional Memory to Simplify Memory Management. In *Proc. of the 2011 ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing (PODC '11)*. San Jose, CA, USA, 99–108.

[10] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. 1986. Making Data Structures Persistent. In *Proc. of the 18th Ann. ACM Symp. on Theory of Computing (STOC '86)*. Berkeley, CA, USA, 109–121.

[11] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the 2006 BSDCan Conf.*

[12] Keir Fraser. 2004. *Practical lock-freedom*. Ph.D. Dissertation. Computer Laboratory, University of Cambridge. No. UCAM-CL-TR-579.

[13] Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. 2009. Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting. *IEEE Trans. on Parallel and Distributed Systems* 20, 8 (Aug 2009), 1173–1187.

[14] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proc. of the 15th Intl. Conf. on Distributed Computing (DISC '01)*. Springer-Verlag, Lisbon, Portugal, 300–314.

[15] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of Memory Reclamation for Lockless Synchronization. *Journal of Parallel and Distributed Computing* 67, 12 (Dec. 2007), 1270–1285.

[16] Maurice Herlihy. 1993. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Trans. on Programming Languages and Systems* 15, 5 (Nov. 1993), 745–770.

[17] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. 2005. Nonblocking Memory Management Support for Dynamic-sized Data Structures. *ACM Trans. on Computer Systems* 23, 2 (May 2005), 146–196.

[18] Håkan Sundell. 2005. Wait-Free Reference Counting and Memory Management. In *19th IEEE Intl. Parallel and Distributed Processing Symp.* Denver, CO, USA, 24b–24b.

[19] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. 2002. Read Copy Update. In *2002 Ottawa Linux Symp.*

[20] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. on Parallel and Distributed Systems* 15, 8 (Aug. 2004), 491–504.

[21] Maged M. Michael and Michael L. Scott. 1995. *Correction of a Memory Management Method for Lock-Free Data Structures.* Technical Report TR 599. Dept. of Computer Science, Univ. of Rochester.

[22] Adam Morrison and Yehuda Afek. 2015. Temporally Bounding TSO for Fence-Free Asymmetric Synchronization. In *Proc. of the 20th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. Istanbul, Turkey, 45–58.

[23] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-free Binary Search Trees. In *Proc. of the 19th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '14)*. Orlando, FL, USA, 317–328.

[24] Chris Okasaki. 1999. *Purely functional data structures.* Cambridge University Press.

[25] Pedro Ramalhete and Andreia Correia. 2017. Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation. In *Proc. of the 29th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA '17)*. Washington, DC, USA, 367–369.

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

[27] John D. Valois. 1995. Lock-free Linked Lists Using Compare-and-swap. In *Proc. of the 1995 ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing (PODC '95)*. Ottowa, Ontario, Canada, 214–222.

A Artifact appendix

A.1 Abstract

This artifact contains the program we used for all experiments in the paper, including implementations of our 5 tested memory managers (POIBR, TagIBR, TagIBR-FAA, TagIBR-WCAS and 2GEIBR) and 3 existing algorithms (Hazard Pointers, Hazard Eras and EBR). The program requires a POSIX compliant operating system (like LINUX) on a multi-core x86-64 machine, with libjemalloc, libhwloc, and gcc supporting C++11.

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

- **Algorithm:** We're presenting new algorithms.
- **Program:** Microbenchmarks are included in the program.
- **Compilation:** We require gcc with C++11 support.
- **Run-time environment:** It's tested on Linux.
- **Hardware:** The program works on any x86 (multi-core) processor. However, we recommend at least 20 hardware threads (hyperthreads) in total to make the experiment interesting.
- **Run-time state:** The program is sensitive to cache/memory contention. Please make sure no other cache/memory consuming programs are running concurrently.
- **Execution:** The program uses thread pinning on hyperthreads at run time. The execution time can be customized.
- **Output:** The program itself outputs a table in csv format, which contains information like total/per-thread operation counts and unreclaimed block counts. An example R script for plotting the figures in the paper is provided.
- **Experiments:** We use Python scripts to operate a test framework, which can reproduce any result as desired.
- **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 at:
https://github.com/roghnin/Interval-Based-Reclamation
The whole program will be about 300MB after compilation.

A.3.2 Hardware dependencies

A multi-core x86-64 processor with more than 20 hyperthreads is recommended. In order to run the multi-threaded tests with the "no memory manager (memory leaking)" option for tens of seconds, about 10GB of RAM is necessary.

A.3.3 Software dependencies

The program assumes Linux and depends on gcc with C++11 support, libjemalloc, and libhwloc. Python and R are also recommended, in order to operate the parharness execution script and to post-process with our plotting script.

A.3.4 Data sets

We don't use any input data set.

A.4 Installation

To compile for release:
$ make
To compile for debugging:
$ make debug

A.5 Experiment workflow

Our workflow can be described as:

- Implement memory managers (MM)
- Choose and implement data structures (rideables) to test on
- Implement desired tests
- Compile source code
- Customize tests with parharness: combinations of MM, rideables, threadcount range and interval, running time, etc.
- Run tests, get results
- Plot results with the R script; analyze

To run the test manually, use
$ bin/main -h
to get usage information. As an example,
$ bin/main -r3 -m12 -t32 -i10 -dtracker=HE -v -ooutput.csv
stands for "run the 12th test mode on the 3rd rideable with 32 threads and Hazard Eras as memory manager for 10 seconds; send results to output.csv." To run tests in batches, edit and run
$ ext/parharness/scripts/testscript.py
in which "mata arguments" are supported. For example, argument
–meta t:5:10:15 –meta i:5:10
will result in 6 different test configurations of thread counts and intervals, as the Cartesian product of the two sets indicates. To plot out data saved in data/final use
$ cd data/script
$ Rscript genfigs.R

A.6 Evaluation and expected result

With rideables and memory managers already implemented in the repository, running
$ ext/parharness/scripts/testscript.py
and plotting with
$ cd data/script
$ Rscript genfigs.R
will reproduce the experiments reported in the paper.
Please note that results may vary depending on the architecture of the test machine. For example, the "gap" in the throughput performance of the Bonsai Tree may not appear on a single-socket machine. But on any given machine, (1) throughputs of IBRs should lie between EBR and HP-like algorithms and be lower than EBR by about 5%; (2) with threads oversubscribed (or stalled), space usage (retired but not reclaimed blocks) of IBRs should be about 30% above that of HP-like algorithms and below that of EBR.

A.7 Experiment customization

Our experiments are done by customizing tests—we change the thread counts and types of memory managers to get time and space curves for different MMs as a function of thread count.
New candidates of rideables and MMs can be added into src/rideables and src/trackers respectively, and new modes of tests can be added into src/CustomTests.hpp. Newly implemented rideables, trackers, and tests should be registered into src/main.cpp and int-main.cpp to appear in the menu.