Turning Manual Concurrent Memory Reclamation into Automatic Reference Counting

Daniel Anderson*
Carnegie Mellon University
Pittsburgh, PA, USA
dlanders@cs.cmu.edu

Guy E. Blelloch*
Carnegie Mellon University
Pittsburgh, PA, USA
guyb@cs.cmu.edu

Yuanhao Wei*
Carnegie Mellon University
Pittsburgh, PA, USA
yuanhao1@cs.cmu.edu

Abstract

Safe memory reclamation (SMR) schemes are an essential tool for lock-free data structures and concurrent programming. However, manual SMR schemes are notoriously difficult to apply correctly, and automatic schemes, such as reference counting, have been argued for over a decade to be too slow for practical purposes. A recent wave of work has disproved this long-held notion and shown that reference counting can be as scalable as hazard pointers, one of the most common manual techniques. Despite these tremendous improvements, there remains a gap of up to 2x or more in performance between these schemes and faster manual techniques such as epoch-based reclamation (EBR).

In this work, we first advance these ideas and show that in many cases, automatic reference counting can in fact be as fast as the fastest manual SMR techniques. We generalize our previous algorithm called Concurrent Deferred Reference Counting (CDRC) to obtain a method for converting any standard manual SMR technique into an automatic reference counting technique with a similar performance profile. Our second contribution is extending this framework to support weak pointers, which are reference-counted pointers that automatically break pointer cycles by not contributing to the reference count, thus addressing a common weakness in reference-counted garbage collection.

Our experiments with a C++-library implementation show that our automatic techniques perform in line with their manual counterparts, and that our weak pointer implementation outperforms the best known atomic weak pointer library by up to an order of magnitude on high thread counts. All together, we show that the ease of use of automatic memory management can be achieved without significant cost to practical performance or general applicability.

*Authors are listed in alphabetical order.

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523730

CCS Concepts: • **Computing methodologies → Concurrent algorithms**.

Keywords: automatic memory reclamation, concurrency, smart pointers, lock-free

ACM Reference Format:
Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. 2022. Turning Manual Concurrent Memory Reclamation into Automatic Reference Counting. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3519939.3523730

1 Introduction

Manually managing memory for concurrent programs is notoriously difficult and prone to errors. One solution is to only work in fully garbage-collected languages but this is not always possible, and comes with its own performance problems, often yielding no control to the user. On the other hand, manually managing memory can be challenging even for sequential programs, but the concurrent setting makes it significantly more difficult. In particular, concurrent programs can suffer from read-reclaim races [12] with potentially disastrous results. For example, one thread could overwrite a location containing a pointer to an object, and then reclaim the memory for that object since it is not being referred to anymore. Another thread executing concurrently could read the location just before it is overwritten. It could then access the contents of the object, which by now might have been reclaimed and perhaps even been reallocated for another use, or returned to the operating system.

Over the past two decades, researchers have developed a broad set of techniques to avoid such read-reclaim races. The goal of these techniques is to delay the destruction and reclamation on an object until it can be ensured that no thread can still access the object. These techniques are generally referred to as *safe memory reclamation* (SMR), and include approaches such as read-copy-update (RCU) [10], epoch-based-reclamation (EBR) [8], hazard-pointers [19], pass-the-buck [13], pass-the-pointer [4], interval-based reclamation (IBR) [30], Hyaline [25], and others [2, 27]. All these approaches replace the destruction of an object with a retire operation, which delays the actual destruction until it is safe.

The SMR approaches differ in how they ensure the reclamation and destruction is safe. The approaches can be partitioned broadly into two classes. *Protected-region techniques*, such as RCU, EBR, IBR, and hyaline, protect regions of code, while *protected-pointer techniques*, such as hazard-pointers, pass-the-buck, and pass-the-pointer, are based on protecting individual pointers. The protected-region techniques tend to be faster since they only need a memory barrier on every critical region instead of every read, but require more space due to longer delays between a retire and reclamation. Both classes of manual techniques, however, are difficult to use and can lead to subtle and hard to reproduce bugs. As evidence, Anderson *et al.* [1] noted several instances where manual techniques of both kinds were applied incorrectly.

An alternative approach for memory management in languages without built-in garbage collection (or even with) is to use reference counting. Reference counting requires very few modifications for programmers to integrate into their code, and provides memory safety and leak freedom automatically as long as the programmer does not create reference cycles. Reference counting is still vulnerable to read-reclaim races , but the race can be managed by the reference counting library itself instead of by the user.

Reference cycles can be broken with so-called weak pointers [16], allowing the cycles to be collected. Weak pointers do not prevent the collection of the objects they point to, but are different from raw pointers in that they provide a way of checking if the object they point to is still alive (i.e. not yet reclaimed), and if so, they can be upgraded to strong pointers. Weak pointers can be used to store back/parent pointers in many data structures or for other pointers that would have otherwise caused a troublesome cycle. As long as a node is not part of any strong reference cycle by the time it becomes unreachable, it will be automatically freed.

Owing to the ease of use of automatic reference counting, there has been increasing interest in concurrent (atomic) reference-counted pointers (both strong and weak), as evidenced by their inclusion in the most recent C++ standard (C++20), and many recent papers on the topic [1, 4, 29]. Early approaches [6, 13] suffered severe performance issues due to contention on the reference counts, but more recent approaches, such as FRC [29], OrcGC [4], and CDRC [1], have been able to avoid this problem by temporarily protecting pointers without incrementing the reference counts. However, even these recent efficient concurrent reference counting approaches can have significant performance degradation relative to manual reclamation. The CDRC paper reports up to a factor of two in performance degradation relative to manual collection via EBR. The main issue is the use of protected-pointer techniques which require extra memory barriers on every read (even if the count is not incremented).

In this paper, we show that reference counting can be nearly as fast as *any* manual technique while using a similar amount of memory (in most cases), thus showing that the ease-of-use of automatic approaches comes at no significant cost to practical performance. Our method is based on CDRC, which combines reference counting and hazard pointers in a novel way. Unlike traditional methods which use hazard pointers to protect a block of memory from being freed, the key insight in CDRC is that hazard pointers can be used to protect *the reference count itself* from being decremented. This simple insight leads to two crucial patterns. First, *deferred decrements* allow increments to proceed without fear of racing with a decrement that might set the counter to zero, thus solving the read-reclaim race. Second, and critically for performance, being able to temporarily protect the reference count from decrements enables readers to safely read the managed object without fear of its destruction and without the performance cost of incrementing the reference count.

One of the contributions of this paper is generalizing the CDRC technique so that the hazard pointer scheme can be replaced with just about any standard SMR scheme to yield an automatic version of that scheme with a similar performance profile. We apply this to three (very different) state-of-the-art manual techniques, EBR, IBR and Hyaline, to yield automatic versions of all three. To the best of our knowledge, this is the first time reference counting has been combined with any manual technique outside of variations of hazard-pointers. The resulting algorithms are all lock-free, assuming that the SMR scheme being automated is lock-free.

As a second contribution, we show how this framework can be extended even further to support lock-free atomic weak pointers that also allow safe reads without incrementing the reference count. We use them to implement a concurrent doubly-linked-list based queue [26], and show that our implementation is several times faster than the only other lock-free atomic weak pointer that we are aware of [32].

A key challenge with weak pointers is supporting the upgrade to strong pointers efficiently. This requires being able to atomically increment the reference count only if it is not already zero. This operation is typically implemented using a CAS-loop [17] which takes up to $O(P)$ amortized time per process if P processes perform this upgrade at the same time. Instead, we show how to implement a so-called sticky counter primitive that supports an *increment-if-not-zero* operation so that reading and incrementing/decrementing take only $O(1)$ time in the worst case.

Contributions.

- We show that a wide range of manual SMR techniques can be made automatic using reference counting.
- We show experimentally that our automatic techniques have similar throughput and memory usage to their manual counterparts. (This represents a 2x-3x throughput improvement over existing concurrent reference counting implementations.)
- We show how to extend our reference counting techniques to efficiently support atomic weak pointers.

- To do so, we implement a theoretically and practically efficient sticky counter primitive.
- We show that our weak pointers significantly outperform existing weak pointers in practice.

Outline. In Section 2, we introduce some important background information and we defer a broader discussion of related works to Section 6. Section 3 describes a general technique for making manual memory reclamation automatic. In Section 4, we show how to extend our algorithms with support for weakly reference-counted pointers to handle reference cycles. An experimental evaluation of the techniques described in this paper is presented in Section 5. Finally, we conclude in Section 7.

2 Preliminaries

Model. We work in the concurrent shared memory model with P asynchronous processes assuming sequential consistency [15]. Memory barriers or memory ordering instructions are needed for weaker memory models, and are included in our C++ implementations. We use the standard definitions of wait-free, lock-free and linearizability [15]. Essentially, lock-freedom guarantees that some process makes progress, while wait-freedom guarantees that every process makes progress. Roughly speaking, linearizability means that each operation appears to take effect atomically at some point during its execution interval. Beyond reads and writes, we assume the existence of three atomic read-modify-write primitives: **compare_and_swap** (CAS), **fetch_and_store** (FAS), and **fetch_and_add** (FAA). All three instructions are supported by modern processors.

Manual SMR. Most manual SMR schemes have similar interfaces built around a common set of operations. These operations include:

- **retire**(x): Indicate that an allocated object x is no longer reachable by the program, i.e., that it is safe to delete after all readers currently reading it are finished.
- **eject**(): Returns a previously retired object that is now safe to delete. The caller should then free this object.

The retire operation is the critical one; it is what replaces completely manual memory management (explicit freeing). A retire operation is essentially a "delayed free". Rather than being freed immediately, the object is freed once any lingering readers have finished with it. The eject operation is optional and is often performed implicitly by retire, but separating the two can allow the programmer greater control over exactly when or how memory is freed.

The difference between *protected-pointer* and *protected-region* techniques is in how they determine when the lingering readers have finished with a retired object, making it safe to free. Protected-region techniques implement the following pair of operations:

- **begin_critical_section**(): Indicate the beginning of a read critical section.
- **end_critical_section**(): Indicate the end of the current read critical section.

For correctness, all reads of objects that are protected by the SMR scheme must be performed while inside a read critical section. A retire operation is able to deduce that a retired object x is safe to eject once all critical sections that were active at the time of its retirement have ended. Protected-pointer techniques use the following operations instead:

- **acquire**(m): Indicate the intention to read the contents of a shared pointer located at the memory location m, and return the current value of the shared pointer.
- **release**(p): Indicate that the pointer obtained from a shared location by **acquire** is no longer being read.

All reads of objects that are protected by the SMR scheme must be done so via an acquire operation, and ended by a corresponding release operation. A retire operation is then able to safely deduce that a retired object x is safe to eject once all active acquires of it at the time of its retirement have been released. Note that in many protected pointer schemes such as hazard-pointer and pass-the-buck, the acquire operation can fail, forcing the program to retry or take a data structure specific fallback plan.

The difference between protected-pointer and protected-region techniques is that protected-region techniques prevent *all* objects from being ejected during their read critical sections, while protected-pointer techniques are more granular and only protect the objects actually being read. Protected-region techniques are therefore usually faster since they require less bookkeeping, but accumulate more garbage because they overprotect objects from being ejected.

CDRC. The key idea behind CDRC is to combine manual SMR and reference counting by using hazard pointers (a protected-pointer technique) to defer reference-count decrements until they no longer race with increments. Essentially, instead of protecting an object from being freed, an acquire operation protects an object's reference count from being decremented until a corresponding release is issued, and a retire operation issues a delayed decrement, which is performed by an eject at a later time when it is not protected by an active acquire.

To achieve this, Anderson et al. [1] introduce an interface called acquire-retire, which exposes the same four operations as protected-pointer schemes: `acquire`, `release`, `retire`, and `eject`, but generalizes hazard pointers by allowing a pointer to be retired multiple times, which is not allowed by traditional hazard pointers. This additional feature is important because each retire corresponds to a delayed decrement, and there could be multiple of those on the same pointer. This interface allows them to implement reference-counted pointers as follows:

- To copy a shared reference-counted pointer, it is first *acquired* to protect the reference count from being decremented below one (which would destroy the object and create a race). The reference count is then incremented, the protection *released*, and a new copy of the pointer is safely returned.
- To overwrite a shared reference-counted pointer with a new value, the reference count of the desired value is incremented, and the previous value is replaced via an atomic exchange (i.e. `fetch_and_store`). The previous value is then *retired*, which issues a deferred decrement to its reference count, which will be applied by a later *eject* once it is no longer protected by a corresponding acquire.

Using this idea, CDRC implements a C++ library containing three reference-counted pointer types: `atomic_shared_ptr`, `shared_ptr`, and `snapshot_ptr`. Applying this library involves replacing raw pointers with one of these three smart pointer types. The interface they support is closely modeled after `atomic<shared_ptr>` and `shared_ptr` from the C++ standard. `atomic_shared_ptr` supports arbitrary concurrent accesses, but is the most expensive to use. `shared_ptr` supports everything except read-write races. `snapshot_ptr` avoids incrementing reference counts in common case and is the most efficient, but cannot be shared between threads.

Figure 1 shows a code snippet from Natarajan-Mittal's BST [21] using both manual SMR and CDRC. It shows that manually calling retire sometimes adds non-trivial code. For example, all the code between lines 12 and 20 can be avoided with CDRC. This loop is responsible for retiring all the nodes removed by the pointer swing on line 10. This loop is easy to forget because in the common (sequential) case, each pointer swing only removes one internal node, and this bug has appeared in the artifacts of several papers [4, 5, 9, 23, 30]. Therefore, reference counting techniques like CDRC are often easier to use and less error-prone.

3 Making Manual SMR Automatic

In this section, we describe how to make manual SMR automatic by combining it with reference counting. Our approach extends CDRC [1], our previous concurrent reference counting algorithm, which uses a hazard-pointer-like technique called acquire-retire to delay reference count decrements until they no longer race with increments. Our insight is that this approach would work for virtually any manual SMR technique, not just hazard-pointers. Note that the process of converting from manual to automatic SMR is not automatic, but we present an easy-to-apply framework and show examples of how to use it.

To generalize CDRC, we first generalize its acquire-retire interface and then show that this generalized interface can be implemented from a wide range of manual techniques. Then we show how to implement concurrent reference counting using this generalized interface.

```
1  class Node { K key; atomic<Node*> left, right; };
2  class SeekRecord { Node *ancestor, *successor, *parent, *leaf;};
3  thread_local SeekRecord seekRecord;

5  void cleanup() { // helper function called by remove()
6    Node* ancestor = seekRecord.ancestor;
7    Node* successor = seekRecord.successor;
8    ...
9    /* Update the left child of ancestor to point to sibling */
10   if(ancestor->left.compare_and_swap(successor, sibling)) {
11     /* retire nodes on path from successor to sibling */
12     for(Node* n = successor; n != subling;) {
13       Node* tmp = n;
14       if(getFlag(n->left)) {
15         retire(n->left);
16         n = n->right;
17       } else {
18         retire(n->right);
19         n = n->left; }
20       retire(tmp); }
21     return true;
22   } else return false; }
```

(a) Manual SMR

```
23 class Node { K key; atomic_shared_ptr<Node> left, right; };
24 class SeekRecord {
25   snapshot_ptr<Node> ancestor, successor, parent, leaf; };
26 thread_local SeekRecord seekRecord;

28 void cleanup() { // helper function called by remove()
29   snapshot_ptr<Node>& ancestor = seekRecord.ancestor;
30   snapshot_ptr<Node>& successor = seekRecord.successor;
31   ...
32   /* Update the left child of ancestor to point to sibling */
33   return ancestor->left.compare_and_swap(successor, sibling));}
```

(b) Reference Counting

Figure 1. Code snippet from Natarajan-Mittal's BST [21] using (a) manual SMR and (b) reference counting (C++-like pseudocode).

3.1 Generalized Acquire-Retire Interface

Our generalized acquire-retire interface shown in Figure 2 has several advantages over the original. The original interface is well-suited for capturing protected-pointer SMR techniques (because `acquire` protects a specific pointer), but not for capturing other types of SMR techniques. We added three new methods to make the interface more general: `alloc`, and `begin_` and `end_critical_section`. Adding `alloc` to the interface is important for techniques like IBR and HE, which tag each object with a birth timestamp on allocation.

Beyond generality, another benefit of the interface in Figure 2 is that it gives us a clean way of implementing snapshot pointers. In CDRC, supporting snapshot pointers requires reaching into the internals of their acquire-retire implementation. So unlike the rest of their reference counting algorithm, their algorithm of snapshot pointers only works for their specific implementation of acquire-retire. We fix this problem by breaking their `acquire` into two operations, an `acquire` and a `try_acquire`. Both operations return a pointer as well as a guard variable that protects the pointer. The pointer can be unprotected at any point by passing the

```
class AcquireRetire<T> {
// Allocate object of type T
Function alloc(): T*

// Delays destructing ptr
Function retire(T* ptr): void

// Returns a previously retired pointer
// that is no longer protected.
Function eject(): optional<T*>

Function begin_critical_section(): void
Function end_critical_section(): void

// Reads a pointer from shared memory and protects it.
// Can only protect one pointer at a time.
Function acquire(T** ptraddr): pair<T*, Guard>

// Reads a pointer from shared memory and tries to protect it
// Can fail and return ⊥.
Function try_acquire(T** ptraddr): optional<pair<T*, Guard>>

// Releases protection
Function release(Guard guard): T* };
```

Figure 2. Generalized acquire-retire interface.

```
class AcquireRetireEBR<T> {
  using Guard = void; // empty type, never used
  using Epoch = int;
  Epoch ann[P]; // initialized to INT_MAX
  Epoch curEpoch = 0;
  thread_local List<pair<T*, Epoch>> retired;

  T* alloc() { return new T(); }
  void begin_critical_section() { ann[pid] = curEpoch; }
  void end_critical_section() { ann[pid] = INT_MAX; }
  void release(Guard guard) {}

  pair<T*, Guard> acquire(T** ptraddr) {
    return [*ptraddr, void]; }

  optional<pair<T*, Guard>> try_acquire(T** ptraddr) {
    return [*ptraddr, void>]; }

  // retire + eject implemented as in Figure 2 of [30] };
```

Figure 3. Generalized acquire-retire implemented with epoch-based-reclamation. We assume each process knows its process id *pid*.

guard variable to `release`. In HP and HE, this guard variable would be a pointer to the announcement slot that protects the pointer. `acquire` can only protect one pointer at a time, so the user must alternate between calling `acquire` and `release`. `try_acquire` on the other hand can protect multiple pointers with different guards. However `try_acquire` may fail and return ⊥ if it runs out of guards (e.g. running out of hazard-pointers). We use `try_acquire` to implement `snapshot_ptrs` in a black box manner in Section 3.4.

Lastly, just like in the original acquire-retire interface, the `retire` operation in Figure 2 takes as input a pointer which will be returned by a future `eject` operation when it is no longer protected.

3.2 Implementing Generalized Acquire-Retire

This new acquire-retire interface can be implemented from almost any manual SMR technique. Figures 3 and 4 show implementations from EBR and IBR, respectively. In this

```
class AcquireRetireIBR<T> {
  using Guard = void; // empty type, never used
  using Epoch = int;
  Epoch emptyann = INT_MAX;
  Epoch beginAnn[P], endAnn[P]; // initialized to emptyann
  Epoch curEpoch = 0;
  thread_local Epoch prev_epoch = emptyann;
  thread_local int counter = 0;

  void begin_critical_section() {
    beginAnn[pid] = endAnn[pid] = prev_epoch = curEpoch; }
  void end_critical_section() {
    beginAnn[pid] = endAnn[pid] = emptyAnn; }
  void release(Guard guard) {}
  class Tagged<T> { Epoch birthEpoch; T t; };

  T* alloc() {
    Tagged<T>* taggedObj = new Tagged<T>();
    taggedObj->birthEpoch = curEpoch;
    if(counter++ % epoch_freq == 0) curEpoch.fetch_add(1);
    return addressof(taggedObj->t); }

  pair<T*, Guard> acquire(T** ptraddr) {
    while(true) {
      T* ptr = *ptraddr;
      Epoch cur_epoch = curEpoch;
      if(prev_epoch == cur_epoch) return [ptr, void];
      else endAnn[pid] = prev_epoch = cur_epoch; } }

  optional<pair<T*, Guard>> try_acquire(T** ptraddr) {
    return acquire(ptraddr); }

  // retire + eject implemented as in [30] };
```

Figure 4. Generalized acquire-retire implemented with interval-based-reclamation (specifically, 2GEIBR).

section, we will discuss some general patterns in these implementations. Most manual SMR algorithms combine the functionality of `retire` and `eject` into a single `retire` operation, but this is easy to split into two operations. A more important difference is that manual SMR is typically used to delay freeing objects. So instead of returning retired pointers to the user, their `retire` function calls `free` on pointers that are no longer protected. We require pointers to be returned to the user because our `retire` can be used to delay arbitrary operations on the pointer, for example decrementing the pointer's reference count. In our implementation of weak pointers in Section 4, we use three instances of `AcquireRetire`, each delaying a different type of operation.

Another reason for having `eject` return a pointer instead of directly applying the delayed operation is to prevent `eject` from recursively calling itself. For example, if the delayed operation is a reference count decrement, then this might trigger recursive reference count decrements, which might lead to recursive calls to `eject`. The `eject` operation is not guaranteed to behave correctly if called recursively, so we disallow this possibility by not applying the delayed operation inside the `eject`. The final difference between our `retire` and the one supported by existing SMR techniques is that we allow a pointer to be retired any number of times before it is ejected a single time. Luckily, most SMR algorithms work properly in this kind of situation even though

they were not designed with it in mind. Protected-pointer approaches sometimes need to be modified to keep track of the number of times a pointer is `retired` and `acquired`. `eject` also has to be modified so that it returns only the pointers that have been `retired` more times than `acquired`. No such modifications are needed for protected region approaches.

Next, we focus on how to implement `acquire`, `release`, and `try_acquire`. For protected-region SMR techniques like EBR, and Hyaline, these operations are trivial to implement because the critical section on its own is enough to protect all the pointers returned by `acquire`. So `acquire` and `try_acquire` simply load the pointer and `release` is a no-op. For protected-pointer approaches like HP and PTB, `try_acquire` has to look for an empty announcement slot to act as the guard. If all announcement slots are in use, then `try_acquire` fails, returning $\perp$. For `acquire`, we reserve a special guard / announcement slot that cannot be used by `try_acquire`. This ensures that `acquire` always succeeds but it means that only one pointer can be protected by `acquire` at a time.

Finally, the operations for beginning and ending a critical section are implemented in the exact same way as in the corresponding manual SMR technique. So for EBR, they would just announce and unannounce an epoch, and for protected-pointer approaches, they would be no-ops.

3.3 Defining Correctness

Just like with the original acquire-retire interface, the tricky part of defining correctness for the generalized version is handling the case where a pointer gets retired multiple times before any copy gets ejected. Fortunately, we can use the original correctness definition with just some small modifications. The idea behind the original definition is to map acquires to retires and ejects to retires such that if an acquire and an eject get mapped to the same retire, then the acquire must be inactive by the time the eject is executed. This formalizes the intuition that a pointer can only be returned by eject if it is not protected by any active acquire. We begin by defining what it means for an acquire to be *active*.

Definition 3.1 (active vs. inactive acquires)**.** *We say that an* `acquire` *or a successful* `try_acquire` *is* active *between when it was invoked and when the guard it returns is passed to* `release`*. After its guard is released, we say it is* inactive*.*

Our acquire-retire interface imposes some restrictions on how it can be used. These restrictions are captured in the following definition of *proper executions*.

Definition 3.2 (proper execution)**.** *We say that a concurrent execution involving acquire-retire operations is* proper *if (1) each active acquire is contained in a critical section, (2) each guard returned by* `acquire` *or* `try_acquire` *is passed to* `release` *at most once, and (3) a process cannot call* `acquire` *while its previous* `acquire` *is still active.*

```
1  class snapshot_ptr<T> { T* ptr; optional<Guard> guard; };

3  AcquireRetire<T> ar;

5  snapshot_ptr<T> atomic_shared_ptr<T>::get_snapshot() {
6    auto ptr, guard = ar.try_acquire(addressof(this->ptr));
7    if(guard != ⊥) return snapshot_ptr<T>(ptr, guard);
8    ptr, guard = ar.acquire(addressof(this->ptr));
9    increment(ptr); // increment reference count
10   ar.release(guard);
11   return snapshot_ptr<T>(ptr, ⊥); }

13 void snapshot_ptr<T>::release() {
14   if(this->guard != ⊥) ar.release(this->guard);
15   else decrement(this->ptr); }

17 void begin_critical_section() { ar.begin_critical_section(); }
18 void end_critical_section() { ar.end_critical_section(); }
```

Figure 5. Implementing snapshot pointers using the generalized acquire-retire interface from Figure 2.

The first property in Definition 3.2 is easy to ensure by beginning a critical section before any calls to acquire and making sure all acquires are inactive before ending the critical section. The third property just says that acquire can only be used to protect a single pointer at a time. Now we are ready to formally define the sequential specifications of acquire-retire.

Definition 3.3 (acquire-retire)**.** *Any proper, concurrent execution can be linearized to a sequential history with the following guarantees:*

- *Successful* `try_acquire(pptr)` *and* `acquire(pptr)` *operations return the pointer currently stored in* $*pptr$*.*
- *Let f be a function that maps each* `acquire` *returning p and each successful* `try_acquire` *returning p to either a later* `retire(p)` *or* $\perp$*. Let g be an injective (one-to-one) function that maps each* `eject` *returning p to an earlier* `retire(p)`*. For all f, there is a g such that whenever $f(A) = g(E)$, the* `acquire` *or* `try_acquire` *A is inactive by the time* `eject` *E is executed.*

3.4 Concurrent Reference Counting

Using the generalized acquire-retire interface, we can implement concurrent reference counting in much the same way as CDRC. The main difference is in our implementation of `snapshot_ptr`s shown in Figure 5. The code for the other two reference-counted pointer types, `atomic_shared_ptr` and `shared_ptr`, remains the same except for some minor updates to use the new acquire-retire interface.

We support snapshot pointers by implementing an operation called `get_snapshot` which loads an atomic shared pointer and creates a `snapshot_ptr`, and by implementing a `release` operation which destructs a `snapshot_ptr`. `get_snapshot` first tries to take the fast path which consists of protecting the pointer with just a `try_acquire`. If this `try_acquire` fails, then it reverts to the slow path which consists of protecting the pointer using an `acquire`, then incrementing the reference count of the pointer, and then

releasing the previous acquire since the pointer is now protected by the incremented reference count. In the slow path, `get_snapshot` then constructs and returns a `snapshot_ptr` with its `guard` field set to $\perp$ to indicate that the slow path was taken. A `snapshot_ptr`'s destructor calls `ar.release()` if it was constructed via the fast path and `decrement` otherwise. As long as a process does not hold on to too many `snapshot_ptrs`, `get_snapshot` will always take the fast path and not perform any reference count updates. This is why `snapshot_ptr` can be cheaper than `shared_ptrs`.

This is different from CDRC's `get_snapshot` implementation which only works for a specific acquire-retire implementation based on hazard-pointers. In their algorithm, `get_snapshot` first looks for an empty announcement location and if all of them are taken, it evicts one of the announcement hazard pointers and increments the reference count of the evicted pointer to ensure that it stays protected. Then `get_snapshot` uses the newly emptied announcement location to protect the pointer it reads.

Another difference from CDRC's implementation is that we require all racy[1] reads and writes on atomic shared pointers as well as all snapshot pointer lifetimes to be contained in a critical section. When applying our reference counting algorithm to a concurrent data structure, this requirement can be satisfied by wrapping each data structure operation in a critical section and only holding on to snapshot pointers during the operation.

4 Weak Pointers

The second classical drawback of reference counting is its inability to clean up garbage that contains cyclic references. A common approach to mitigate this issue at the library level is to include a "weak pointer" type. Weak pointers complement shared pointers (or "strong pointers") by holding a reference to a shared object without contributing to the reference count. If the reference count of the managed object reaches zero, it is destroyed, despite any weak pointers that may have a reference to it.

The advantage of weak pointers over raw pointers is that, unlike raw pointers, which are unsafe to dereference if they might point to an already freed object, weak pointers can tell whether they point to a managed object that has already been destroyed. This is usually achieved by storing a second reference count that counts the number of weak pointers to the managed object. When the (strong) reference count reaches zero, the managed object is destroyed, but the control data containing the reference counts is kept intact until both the strong and weak reference counts reach zero. This allows weak pointers to safely check that the managed object is alive by checking that the strong reference count is non-zero.

[1]Two operations are said to *race* if they both access the same atomic shared pointer and one of them is a write.

The C++ standard library includes support for weak pointers, and, as of C++20, support for atomic weak pointers. However, currently the only standard library implementation of atomic weak pointers is Microsoft's STL [20], and it is lock-based. We know of one commercial implementation in the just::thread library [32]. We describe how our approach can be extended to efficiently support weak pointers.

4.1 Library Interface

We add the following types to the reference-counted pointer library. Figure 6 depicts the relationship between them.

- **atomic_weak_ptr**: Analogous to `atomic_shared_ptr`, an `atomic_weak_ptr` facilitates atomically loading, storing, and CASing a `weak_ptr` into a shared mutable location. In addition to `load`, it also supports a `get_snapshot` method, which grants safe local access to the managed object without modifying the reference count.
- **weak_ptr**: A `weak_ptr` is modeled after C++'s standard weak pointer. Unlike `shared_ptr`, a `weak_ptr` cannot be directly dereferenced. To access the managed object, the `weak_ptr` must be upgraded to a `shared_ptr`. If the managed object has *expired*, the obtained `shared_ptr` will be null to indicate this.
- **weak_snapshot_ptr**: A `weak_snapshot_ptr` allows safe access to the object managed by the `atomic_weak_ptr` as of the time it was created, even if the reference count of the managed object reaches zero during its lifetime. Creating and reading a `weak_snapshot_ptr` does not incur a modification to the reference count. A `weak_snapshot_ptr` will be null if the managed object has expired at the time of its creation.

The subtle difference between a `weak_snapshot_ptr` and a `snapshot_ptr` is that a `snapshot_ptr` guarantees that the managed object doesn't expire (has reference count at least one) throughout its lifetime, while a `weak_snapshot_ptr` only guarantees that the managed object is safely readable, though it may expire (reach reference count zero) during the lifetime of the snapshot.

We first describe the main primitives needed to implement deferred reference counting with weak pointers. We then describe how to support the main operations on the various weak pointer types in our library.

4.2 Managing the Managed Object

First, to implement weak pointers, each managed object is augmented with a second reference count. We distinguish between the original (strong) reference count and the new (weak) reference count. When the strong reference count reaches zero, the managed object is ready to be destroyed. However, the control data attached to the managed object (the reference counts plus any extra scheme-specific metadata) cannot be destroyed and freed yet, because there might still exist weak pointers that attempt to access those fields.

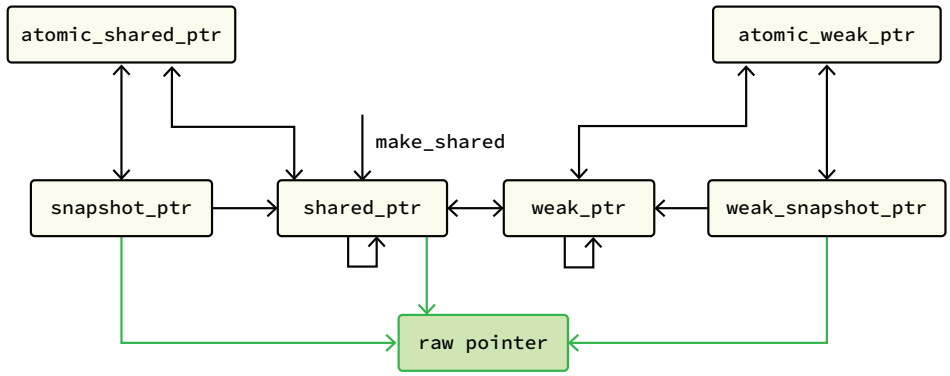


Figure 6. The managed pointer types in our library. Arrows between types denote that it is possible to store/load one type in/from the other, or that it is possible to convert from one type to the other. The three types `snapshot_ptr`, `shared_ptr`, and `weak_snapshot_ptr` can be safely dereferenced/converted into raw pointers.

Only once both the strong and weak reference counters hit zero can the entire control block (the managed object plus the control data) be freed. To correctly detect when both counters hit zero in the presence of concurrent updates, we use the standard trick [17, 20] of storing

$$\text{weak\_cnt} = \#\text{weak refs} + \begin{cases} 1 & \text{if } \#\text{strong refs} > 0 \\ 0 & \text{otherwise.} \end{cases}$$

When the strong count hits zero, it can destroy the managed object and decrement one from the weak count. To be precise, this destruction and corresponding decrement must be delayed in the presence of weak pointers. We will discuss this in Section 4.4. When the weak count hits zero, the entire control block is ready to be freed immediately.

In the strong-only setting, the reference count will only ever be incremented when there already exists at least one reference, and hence the increment can always be performed with a fetch-and-add operation. In the weak setting, however, it is possible that a weak pointer points to a managed object whose strong reference count could be decremented to zero at any moment. Attempting to increment the strong reference count with a fetch-and-add could therefore result in incrementing the counter from zero, thus resurrecting a dead object. Our algorithms therefore require an *increment-if-not-zero* operation, which can return false if the reference count is zero, and hence should not be incremented.

The increment-if-not-zero operation is traditionally implemented as a simple CAS loop, which continuously attempts to add one to reference count as long as it is not zero, or returns false otherwise. This results in the increment having lock-free but not wait-free progress. In the next section, we describe a simple, but to the best of our knowledge, novel implementation of a constant-time wait-free counter that supports the increment-if-not-zero operation. This data structure in general is sometimes referred to as a sticky counter. Specifically, our data structure implements an atomic counter that supports increment-if-not-zero, decrement, and load, all in constant time using single-word atomic instructions.

4.3 Wait-Free Increment-if-Not-Zero

Our algorithm can implement a b-bit wait-free counter using $b + 2$ bits, that is, we use two bits for bookkeeping purposes. The main idea is simple, we use the highest bit of the reference counter to indicate whether the reference count is zero. Any bit pattern in which the highest bit is set is interpreted as zero, and otherwise is not. Note importantly, that this means that the stored value being zero is not interpreted as the reference count being zero! The implementation is described below and depicted in Figure 7. This technique of using the high bits to store a flag above a counter is similar to that of Correia and Ramalhete [3] who implement reader-writer locks that store a count of the number of shared readers. Our technique generalizes theirs by allowing constant-time linearizable reads of the counter.

Increment. Since the presence of the high bit indicates whether the counter is zero, the increment operation can just perform a fetch-and-add operation, and check whether the result has the high bit set. If so, it returns false.

Decrement. The decrement operation should decrement the reference count and return true if the reference count was brought to zero, or false otherwise. To decrement the counter, the algorithm uses a fetch-and-add and checks whether the counter hits zero. If it does, it must attempt to set the high bit to indicate this. This is done with a CAS. Note that if the CAS fails, it must be the case that an increment occurred that brought the counter back up from zero. In this case, the decrement can simply act as if the increment occurred before it, and hence report that it did not bring the counter to zero. A decrement that races with a load must handle one additional case described in the next paragraph.

Load. At first glace, the algorithm could try to just load the stored value, and return zero if the high bit is set. This however, is not necessarily correct if the stored value is zero. If the stored value is zero, the high bit might be about to be set, but an increment might race with it and bring the counter above zero. Reporting zero would therefore be incorrect. In order to achieve wait-freedom, the load operation therefore

```
unsigned int zero = 1 << (b - 1);
unsigned int help = 1 << (b - 2);
unsigned int x;

bool increment_if_not_zero() {
  auto val = x.fetch_add(1);
  return (val & zero) == 0; }

bool decrement() {
  if (x.fetch_sub(1) == 1) {
    unsigned int e = 0;
    if (x.compare_exchange(e, zero)) return true;
    else if ((e & help) && (x.exchange(zero) & help)) return true;
  } return false; }

unsigned int load() {
  auto e = x.load();
  if (e == 0 && x.compare_exchange(e, zero | help)) return 0;
  return (e & zero) ? 0 : e; }
```

Figure 7. An implementation of a wait-free reference counter with constant time increment-if-not-zero, decrement, and load. Note that the `compare_exchange` operation, if unsuccessful, atomically loads the value of `x` into `e`.

attempts to help set the high bit. If it successfully sets the high bit, it can return zero. If it fails, the unsuccessful CAS will return the current value of the counter.

If the load operation successfully helps to store the high bit, one of the decrements still needs to take responsibility for being the one who brought the counter to zero. To achieve this, the helping operation additionally writes the second-highest bit, to indicate to the decrement operation that it was helped. If a decrement operation fails to CAS the high bit but detects the helper bit, it can then perform a fetch-and-store (exchange in C++) to remove the helper bit. If it removes the helper bit, it takes credit for bringing the counter to zero.

4.4 Primitives for Weak Reference Counting

The addition of a weak reference count requires us to make changes to the use of the acquire-retire interface used behind our reference counting scheme. In the strong-only setting, a retired pointer always corresponds to a delayed decrement of the reference count. In the weak setting, our algorithm also needs to be able to delay decrements of the weak count.

Additionally, in the strong-only setting, obtaining a snapshot pointer to a managed object meant that the strong reference count was at least one, and since the pointer through which it was obtained is protected, it is guaranteed to remain at least one. However, this property cannot be guaranteed for a *weak snapshot*, because a thread might be about to decrement the last remaining strong reference right as we acquire it. Therefore, to make weak snapshots safe, an additional round of deferral is required to defer the destruction of the managed object after its reference count hits zero. This guarantees that after an acquire, if the strong reference count is at least one, the object will not be destroyed until after the protection of the snapshot is released. We refer to the destruction of the managed object as a *dispose* operation.

```
1  AcquireRetire<T> strongAR, weakAR, disposeAR;

3  void delayed_decrement(T* p) {
4    strongAR.retire(p);
5    auto x = strongAR.eject();
6    decrement(x); }

8  void delayed_weak_decrement(T* p) {
9    weakAR.retire(p);
10   auto x = weakAR.eject();
11   weak_decrement(x); }

13 void delayed_dispose(T* p) {
14   disposeAR.retire(p);
15   auto x = disposeAR.eject();
16   dispose(x); }

18 T* load_and_increment(T** p) {
19   auto ptr, guard = strongAR.acquire(p);
20   if (ptr) increment(ptr);
21   strongAR.release(guard);
22   return ptr; }

24 T* weak_load_and_increment(T** p) {
25   auto ptr, guard = weakAR.acquire(p);
26   if (ptr) weak_increment(ptr);
27   weakAR.release(guard);
28   return ptr; }

30 bool increment(T* p) {
31   return p->ref_cnt.increment_if_not_zero(); }

33 void weak_increment(T* p) {
34   p->weak_cnt.increment_if_not_zero(); }

36 void decrement(T* p) {
37   if (p->ref_cnt.decrement(1)) {
38     delayed_dispose(p); } }

40 void dispose(T* p) {
41   destroy(p->object);
42   weak_decrement(p); }

44 void weak_decrement(T* p) {
45   if (p->weak_cnt.decrement(1)) {
46     delete p; } }

48 bool expired(T* p) {
49   return p->ref_cnt.load() == 0; }
```

Figure 8. Primitives for implementing deferred reference counting with support for weak pointers.

To facilitate these additional needs, instead of using a single instance of acquire-retire, our enhanced algorithm makes use of three instances—one for strong reference count decrements, one for weak decrements, and one for disposals.

Integrating these ideas, we extend the set of primitives for deferred reference counting with weak pointers as follows. Pseudocode is given in Figure 8. The **delayed_decrement**, **delayed_weak_decrement**, and **delayed_dispose** operations make use of three different instances of acquire-retire to delay a decrement to the strong or weak reference count, or the destruction of the managed object, until it is no longer protected by a corresponding acquire.

load_and_increment and **weak_load_and_increment** atomically load the value of the pointer stored at the given location and perform a safe increment of the strong or weak reference count respectively. Note that `load_and_increment`

does not check whether the increment was successful, because these functions are only ever called on a pointer location that is storing a strong or weak reference respectively, and hence the reference count is already guaranteed to not be zero. It is a precondition violation to call this function on a pointer location that stores an object whose strong reference count is already zero.

increment and **weak_increment** attempt to increment the reference count or weak reference count respectively. The first returns true if successful. Note that `weak_increment` does not need to check for success because objects with a zero weak reference count are instantly destroyed, and hence it would be unsafe to attempt to increment the counter anyway. **decrement** decrements the strong reference count, and if it reaches zero, queues up a delayed *dispose*. A **dispose** destroys[2] the managed object and decrements the weak reference count. Similarly, **weak_decrement** decrements the weak reference count, and if it hits zero, immediately frees the managed object and its control data. Lastly, **expired** checks whether the managed object is still considered alive by checking that the reference count is not zero.

4.5 Algorithms for Atomic Weak Pointers

Using the primitives from Figure 8, the algorithms for storing and loading to/from and CASing into an atomic weak pointer are very similar to those in CDRC [1]. The main difference is that we must be careful to use the correct instance of acquire-retire for protection, and the correct kinds of increments/decrements. The algorithm that is most different from its strong counterpart is `get_snapshot`. Pseudocode is given in Figure 9 and described below.

Storing a weak_ptr in an atomic_weak_ptr. This works the same as storing a `shared_ptr` in an `atomic_shared_ptr`. The algorithm first increments the weak reference count of `desired`, then uses a fetch-and-store (exchange in C++) to swap the managed object with the given one, and finally performs a delayed decrement of the weak reference count of the previously stored object.

Loading a weak_ptr from an atomic_weak_ptr. This is essentially the same as loading from an `atomic_shared_ptr`. The managed object is atomically loaded and has its weak reference count safely incremented, returning a `weak_ptr` to the managed object.

CASing into an atomic_weak_ptr. Compare and swap begins by protecting the pointer owned by `desired`. If the CAS is successful, it increments the weak reference count of desired and performs a delayed decrement of the weak reference count of `expected`. Note that the guard must be acquired before performing the CAS because otherwise, the CAS might succeed while another process clobbers `desired`, destroying it before the reference count increment happens.

```
1  void atomic_weak_ptr<T>::store(const weak_ptr<T>& desired) {
2    if (desired.ptr) weak_increment(desired.ptr);
3    auto old_ptr = this->ptr.exchange(desired.ptr);
4    if (old_ptr) delayed_weak_decrement(old_ptr); }

6  weak_ptr<T> atomic_weak_ptr<T>::load() {
7    auto ptr = weak_load_and_increment(addressof(this->ptr));
8    return weak_ptr(ptr); }

10 bool atomic_weak_ptr<T>::compare_and_swap(
11   const weak_ptr<T>& expected, const weak_ptr<T>& desired) {
12   auto ptr, guard = weakAR.acquire(addressof(desired.ptr));
13   if (compare_and_swap(this->ptr, expected.ptr, ptr)) {
14     if (ptr) weak_increment(ptr);
15     if (expected.ptr) delayed_weak_decrement(expected.ptr);
16     weakAR.release(guard);
17     return true; }
18   else {
19     weakAR.release(guard);
20     return false; } }

22 weak_snapshot_ptr<T> atomic_weak_ptr<T>::get_snapshot() {
23   while (true) {
24     auto ptr, weak_guard = weakAR.acquire(addressof(this->ptr));
25     auto _, dispose_guard=disposeAR.try_acquire(addressof(ptr));
26     if (dispose_guard == ⊥ && ptr) increment(ptr);
27     if (ptr && !expired(ptr)) {
28       weakAR.release(weak_guard);
29       return weak_snapshot_ptr(ptr, dispose_guard); }
30     else {
31       disposeAR.release(dispose_guard);
32       weakAR.release(weak_guard);
33       if (ptr == null || this->ptr == ptr)
34         return weak_snapshot_ptr(null); } }

36 void weak_snapshot_ptr<T>::release() {
37   if (this->guard != ⊥) disposeAR.release(this->guard);
38   else decrement(this->ptr); }
```

Figure 9. C++-like pseudo-code for atomic weak pointers.

Creating a snapshot from an atomic_weak_ptr. Creating a snapshot from an `atomic_weak_ptr` is slightly more complicated than taking one from an `atomic_shared_ptr`. The main idea is to try to acquire a protected pointer to the managed object that prevents the object from being disposed, and, if the managed object has not expired (the strong reference count is at least one), return a snapshot containing the protected pointer. If the try_acquire fails, the backup plan is to attempt to increment the reference count[3]. In case the managed object has already been disposed before protecting the pointer, the algorithm first acquires protection against a possible weak decrement, since, otherwise, the control data could be deleted mid-operation.

If the strong reference count is zero, the obvious algorithm would just return a snapshot containing a null pointer. However, this strategy would result in the operation not being linearizable, because the reference count could be in the process of being decremented right as the pointer is acquired. This would allow for situations where the `atomic_weak_ptr`

[2]We use destroy in the object-oriented sense to mean to recursively destroy all of its fields. If any of its fields are themselves reference-counted pointers, this would trigger their reference count decrements.

[3]This only happens with the hazard pointer implementation if too many snapshots are held at once such that the announcement array runs out of slots. EBR, IBR and Hyaline never fail.

```
1  class doubly_linked_queue<V> {
2    struct Node {
3      V value;
4      atomic_shared_ptr<Node> next;
5      atomic_weak_ptr<Node> prev;
6      Node(V v) { value = v; next = null; prev = null; } };
8    atomic_shared_ptr<Node> head, tail;
10   void enqueue(V v) {
11     shared_ptr<Node> new_node = shared_ptr<Node>::make_shared(v);
12     critical_section_guard guard;
13     while (true) {
14       snapshot_ptr<Node> ltail = tail.get_snapshot();
15       new_node->prev.store(ltail);
16       // Help the previous enqueue set its next ptr
17       weak_snapshot_ptr<Node> lprev = ltail->prev.get_snapshot();
18       if (lprev && lprev->next == null) lprev->next.store(ltail);
19       if (tail.compare_and_swap(ltail, new_node)) {
20         ltail->next.store(std::move(new_node));
21         return; } } }
23   std::optional<V> dequeue() {
24     critical_section_guard guard;
25     while (true) {
26       snapshot_ptr<Node> lhead = head.get_snapshot();
27       snapshot_ptr<Node> lnext = lhead->next.get_snapshot();
28       if (!lnext) return {}; // Queue is empty
29       if (head.compare_and_swap(lhead, lnext)) {
30         return {lnext->value}; } } } };
```

Figure 10. Ramalhete and Correia's concurrent doubly-linked queue [26] implemented using our weak pointer interface (C++-like pseudocode).

always points to a live object, but the snapshot may return null if the object was replaced in between the acquire and the read of the reference count. Therefore, if the reference count is zero, the algorithm only returns a null pointer if the `atomic_weak_ptr` still manages the same acquired pointer. If not, the algorithm retries from the beginning. This retrying causes `get_snapshot` to be lock-free but not wait-free.

4.6 Example Usage

An example of how to apply our `weak_ptr` interface to Ramalhete and Correia's doubly-linked queue [26] is shown in Figure 10. The `prev` pointer of each node is stored in an atomic weak pointer, whereas the `next` pointers are stored in atomic shared pointers. The `critical_section_guard` (on lines 12 and 24) is only needed if generalized acquire-retire was implemented from a protected-region SMR technique. The `critical_section_guard` is responsible for calling `begin_critical_section` in its constructor and also `end_critical_section` in its destructor.

5 Experimental Evaluation

We implemented our techniques as a C++ library[4] and evaluated them on a series of benchmarks. Our experiments were run on a 4-socket 72-core machine (4× Intel(R) Xeon(R) E7-8867 v4, 2.4GHz) with 2-way hyperthreading, a 45MB L3 cache, and 1TB of main memory. Memory was interleaved across sockets using *numactl -i all*, and we used the *jemalloc* allocator [7]. Experiments were written in C++ and compiled with GCC 9.2.1 with 03 optimization. Our experiments vary the number of threads from 1 to 192, which allows us to measure the effect of oversubscription, as our hardware supports 144 threads.

5.1 Comparing Manual and Automatic Techniques

We applied the approach in Section 3 to three different manual SMR techniques, EBR [8], IBR (more specifically, 2GEIBR) [30], and Hyaline (more specifically, Hyaline-1) [22], to construct three new concurrent reference counting implementations, which we call RCEBR, RCIBR, and RCHyaline, respectively. The goal of this section is to understand the overhead of making manual techniques automatic as well as to compare the performance of RCEBR, RCIBR, and RCHyaline with the fastest existing reference counting algorithm. The two fastest existing reference counting algorithms that we are aware of are FRC [29] and CDRC [1]. We chose to compare with CDRC because FRC does not support marked pointers which are required in all of our benchmarks. For consistency, we rename CDRC to RCHP in the graphs as it is a combination of hazard-pointers and reference counting.

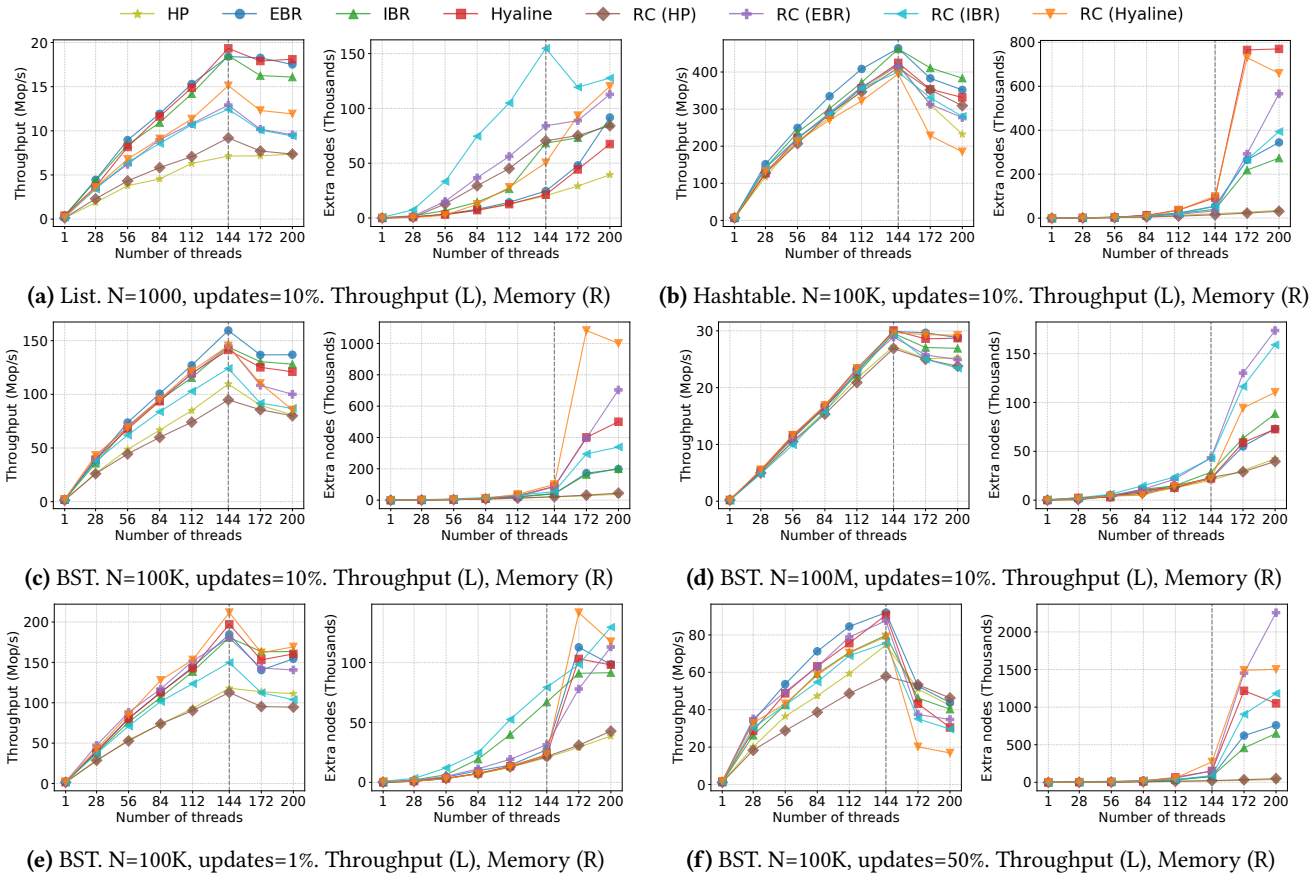
As for manual techniques, we compare with HP, EBR, IBR, and Hyaline. An important parameter to tune when using EBR and IBR is how often the global epoch gets incremented. Incrementing too often could bottleneck scalability whereas incrementing infrequently would increase memory usage. For EBR and RCEBR, we found a good rate to be one increment every 10 allocations and for IBR and RCIBR, we found this to be one increment every 40 allocations.

For both HP and RCHP, we found that prefetching appropriately significantly increased throughput. In particular, before announcing a pointer in the hazard array, we prefetch the cache line that it points to because there is a good chance we will dereference the pointer after succeeding in announcing it. The benefit of this is that we can start loading the cache line before the memory barrier, which is an expensive operation. Note that due to this prefetching optimization, our throughput reported here for HP and RCHP is greater than the throughput of the same schemes in the CDRC paper.

To benchmark performance, we applied these memory reclamation techniques to three different lock-free data structures: Harris-Michael list [11, 18], Michael hash table [18], and Natarajan-Mittal tree [21].

It has been noted that HP and IBR are not safe to use with the Natarajan-Mittal tree directly [1]. This is because traversals in the Natarajan-Mittal tree can continue through marked nodes. We still include these numbers in our experiments for reference, even though these experiments occasionally crash. Modifying the Natarajan-Mittal tree to work with HP and IBR would likely make it slower. Note that an advantage of RCHP and RCIBR is that they work with Natarajan-Mittal tree without any such modifications.

[4] Available at https://github.com/cmuparlay/concurrent_deferred_rc



(a) List. N=1000, updates=10%. Throughput (L), Memory (R)

(b) Hashtable. N=100K, updates=10%. Throughput (L), Memory (R)

(c) BST. N=100K, updates=10%. Throughput (L), Memory (R)

(d) BST. N=100M, updates=10%. Throughput (L), Memory (R)

(e) BST. N=100K, updates=1%. Throughput (L), Memory (R)

(f) BST. N=100K, updates=50%. Throughput (L), Memory (R)

Figure 11. Benchmark comparing manual and automatic SMR techniques. Figure 12a shows results for a Harris-Michael list, Figure 12b for a Michael hash table, and Figures 12c–12f for various workloads on a Natarajan-Mittal tree.

Range query workload. We begin by analyzing the experiment shown in Figure 11. In this workload, we initialized the Natarajan-Mittal tree with 100K keys randomly selected from the key range $[0, 200K)$, and then performed update operations (half insert, half delete) and range queries. We use a sequential range query algorithm, which is not linearizable. with equal probability. Each update operation selects a uniform random key from $[0, 200K)$ to insert/delete and each range query selects a uniform random key k from the same range and queries for all keys in the interval $[k, k{+}64)$. In this experiment, we found that RCEBR, RCIBR, and RCHyaline outperform RCHP by more than 7x on 144 threads. This is because during a range query, the entire path from the current node to the root needs to be protected by `snapshot_ptrs`, so RCHP eventually runs out of announcement locations and starts relying on reference count increments, which is significantly more expensive. RCEBR, RCIBR, and RCHyaline also performs similarly to their manual counterparts, performing within 10-15% at 144 threads.

Other workloads. Figure 12 shows the throughput and memory usage of these SMR technique on a wide variety of workloads. These workloads only contain updates and single point lookups. For example, Figure 12c shows a workload where the Natarajan-Mittal tree is initialized with 100K keys, and each process performs 10% update operations and 90% lookups. Again, all keys are chosen uniformly randomly from a key range twice the initial size of the data structure. For the hash table experiments, we initialized the number of buckets so that the average load factor is 1.

When update frequency is low (Figure 12e), RCEBR has almost the exact same throughput as EBR and RCHyaline is actually slightly faster than Hyaline. However, RCIBR ends up being about 20% slower than IBR and this overhead comes from two main factors. First, RCIBR adds both a reference count and a birth epoch to each node, and this increase in size accounts for about half of the performance difference. Second, each `try_acquire` in RCIBR requires reading a thread local variable storing the process id and this access is surprisingly slow, accounting for the other half of the performance difference. Overall, on the BST experiments with 144 threads, RCEBR performs within 10% of EBR (in terms of throughput) and RCHyaline performs within 15% of Hyaline. Also, RCEBR is up to 1.7x faster than RCHP in Figure 12c.

In the non-oversubscribed scenarios, the automatic version of each memory reclamation scheme tends to use a similar amount of memory to the manual version. However in the

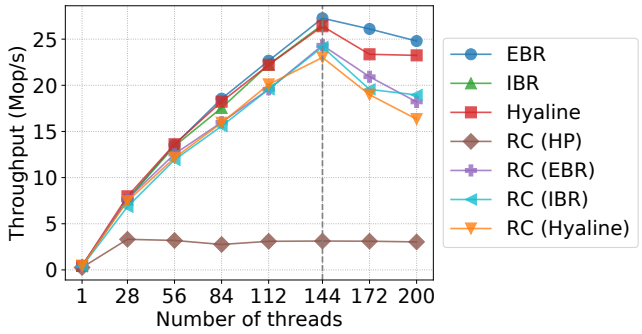


Figure 12. Natarajan-Mittal tree - Range query experiments: 50% updates, 50% range queries of size 64.

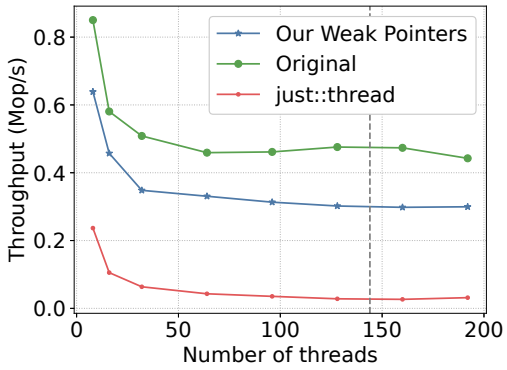


Figure 13. Benchmark results for atomic weak pointers. Original is the optimized doubly linked queue of Ramalhete and Correia [26] that uses a custom manual memory management technique. Our algorithm uses atomic weak pointers powered by the hazard pointer implementation of acquire-retire. `just::thread` is a commercial library of atomic shared and weak pointers.

linked list experiment and also in oversubscribed cases, the automatic version tends to have several times more memory overhead. This is because in reference counting techniques, each retired pointer could recursively prevent the collection of many nodes beyond the one it directly points to.

5.2 Evaluation of Atomic Weak Pointers

We compare our implementation of atomic weak pointers with the best known existing lock-free implementation, the just::thread library [32], and against a manually memory-managed data structure. For our comparison we use the doubly linked queue of Ramalhete and Correia [26]. This queue is a good candidate since it uses back pointers that can be represented using weak pointers. For this comparison, we use our reference counting library powered by the hazard pointer implementation of acquire-retire. We found that the main bottleneck of the throughput of the data structure is the contention on the CAS operations, and hence the different choices of acquire-retire implementation only made minor differences to the performance.

The original implementation of the data structure does not use a general purpose memory management scheme, but actually uses a customized version of hazard pointers specifically engineered for it. This modified hazard pointers scheme allows announced nodes to protect not only themselves, but also the nodes adjacent to them. This reduces the number of memory barriers required by the algorithm. For this reason, it is not likely that a general purpose memory management scheme would outperform it.

In our experiment, we initialize a single queue with P elements, and have P threads. Each thread repeatedly pops an element from the queue and then reinserts it. We then measure the number of such operations that were performed per second. Each benchmark is repeated five times for stability. The results of this experiment are depicted in Figure 13.

The biggest difference in performance occurs at $P = 1$ (not depicted on the plot due to scale), where the original implementation is 4.5x faster than our weak pointers, and 67x faster than just::thread. At $P = 8$ threads, our weak pointer implementation is just 19% slower than the manual approach, and 4.2x faster than just::thread. This trend roughly continues to $P = 192$, where our weak pointers are 33% slower than the manual approach, but 10x faster than just::thread. Given that the original implementation uses a memory management approach that is customized to the data structure at hand, these results are very promising for a completely automatic approach. Furthermore, we substantially outperform the best existing automatic approach at all thread counts.

6 Related Work

6.1 Manual SMR

Manual SMR techniques can be broadly classified as either protected-pointer-based or protected-region-based.

Protected-pointer-based methods. These methods work by identifying specific objects or memory locations that are currently in use and hence should not be destroyed/freed. The collection part of the algorithm is responsible for ensuring that it never frees something that is currently in use. Hazard-pointers [19] is one of the most widely used protected-pointer-based techniques. The main idea is that every process has some globally visible array of "hazard pointers". When a process wishes to read a mutable shared pointer, it *announces* its intention to do so by writing the pointer into one of the hazard pointers. This may require a retry if the value of the pointer changes before the announcement is complete. When the process has finished reading or manipulating the shared object, it *releases* the hazard pointer by clearing the announcement. When a process removes a node from the data structure and wishes to free it, it instead *retires* the node, which places it in a *retired list* of nodes pending deletion. A process that wishes to reclaim memory must scan the hazard array of every process to ensure that

it does not reclaim anything currently announced. Nodes in the retired list that are not announced are safe to free.

Several variants of hazard pointers exist, many of them designed to help implement other memory reclamation schemes. Herlihy *et al.* [13] develop Pass The Buck (PTB), which is used to implement their algorithm for lock-free reference counting. Correia *et al.* [4] develop pass-the-pointer (PTP), which improves on the memory bounds of traditional hazard pointers and is used to implement their own lock-free reference counting algorithm, OrcGC. Anderson *et al.* [1] introduce acquire-retire, the first constant-time implementation of hazard-pointers, which also allows a pointer to be retired multiple times concurrently.

Protected-region-based methods Rather than protecting specific objects/memory locations, protected-region-based methods protect groups of objects. This generally results in lower synchronization cost (fewer memory barriers) and hence higher throughput, but at the cost of wasting more memory, since many objects will be protected even when they do not need to be. Epoch-based reclamation (EBR) [8] and Read-copy-update (RCU) [10] are the most widely used protected-region-based techniques. In EBR, the algorithm maintains a global timestamp called the epoch. Whenever a memory location is retired, it is placed in a retired list corresponding to the current epoch. When the user wishes to begin an operation that will access or modify shared state, the executing thread announces the value of the current epoch. When every thread has announced the value of the current epoch, the retired list from the previous epoch can be freed and the epoch can advance to the next value. Note that this is safe because if an object is retired at epoch e and every process has subsequently announced epoch $e + 1$, then any thread that was performing an operation at the time of the retire has since completed. DEBRA [2] is an optimized implementation of EBR with better practical performance.

Hazard Eras (HE) [23, 27] is a combination of protected-pointer- and protected-region-based methods. In HE, acquired pointers do not announce the pointer itself, but rather the epoch on which it was read. If the epoch changes infrequently, this results in fewer memory barriers than a full-blown protected-pointer-based scheme. In HE and Interval-based Reclamation (IBR) [30], each allocated object is tagged with a birth epoch. In IBR, a retired object is safe to reclaim when no announced epoch intersects its birth-death interval.

Hyaline [22, 25] is a variant of EBR that tags each retired object with a counter corresponding to the number of active operations. When an operation completes, it can decrement one from every object that retired during it. The operation that brings a counter to zero is responsible for freeing it. Crystalline [24] extends Hyaline with wait-freedom.

6.2 Lock-Free Reference Counting

Lock-free reference counting (LFRC) was first described by Detlefs *et al.* [6], but their algorithm requires a DCAS operation (a CAS on two independent words), which is not supported by any current architecture. Herlihy *et al.* [13] use their PTB technique to obtain an algorithm for single-word lock-free reference counting (SLFRC). The idea is to use PTB to protect the reference count of the object from being freed while a process is attempting to increment it. Sundell [28] developed the first wait-free algorithm for reference counting, however, some of their operations cost $O(P)$ time.

The split reference count technique [31] is a non-SMR-based lock-free solution for atomic reference counting. It involves splitting the reference count into an internal count, and an external count on each mutable shared reference. Loads from shared references increment the corresponding external count, while local releases decrement the internal count instead. When a shared reference is discarded, its accumulated external count minus one is added to the internal count. While this technique doesn't rely on SMR, it tends to scale poorly in practice since loads must be performed with a double-word CAS to increment the external count.

The major performance drawback of reference counting is the necessity to increment the reference count each time an object is read. Recent work has addressed this by developing solutions for reference counting that allow safe reads without incrementing the reference count. Tripp *et al.* [29] implement Fast Reference Counter (FRC). FRC uses deferred reference counting and a per-thread root set (equivalent to an announcement array of hazard pointers) to achieve low contention and enable safe reads of managed objects without incrementing the reference count. Correia *et al.* [4] develop OrcGC, which uses their PTP technique to implement reference-counted pointers that can also be safely read without incrementing. Finally, Anderson *et al.* [1] develop Concurrent Deferred Reference Counting (CDRC), which uses the acquire-retire technique to defer reference count decrements and also enable safe reads without increments.

7 Conclusion

In this work, we showed that an automatic memory reclamation technique can compete with the best manual techniques, and showed that such a technique can also support atomic weak pointers. Though perhaps it is not yet time to completely retire manual memory reclamation, we believe that these results show, even more strongly than previous results, that we are getting close, and that automatic memory management should be preferable in a majority of situations.

Acknowledgments

We thank the anonymous referees for their comments and suggestions. This research was supported by NSF grants CCF-1901381, CCF-1910030, CCF-1919223, and CCF-2119069.

References

[1] Daniel Anderson, Guy E Blelloch, and Yuanhao Wei. 2021. Concurrent deferred reference counting with constant-time overhead. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 526–541.

[2] Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way. In *ACM Symposium on Principles of Distributed Computing (PODC)*. 261–270.

[3] Andreia Correia and Pedro Ramalhete. 2018. Strong trylocks for reader-writer locks. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[4] Andreia Correia, Pedro Ramalhete, and Pascal Felber. 2021. OrcGC: automatic lock-free memory reclamation. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 205–218.

[5] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 631–644.

[6] David Detlefs, Paul Alan Martin, Mark Moir, and Guy L. Steele Jr. 2002. Lock-free reference counting. *Distributed Computing* 15, 4 (2002), 255–271.

[7] J. Evans. 2019 (accessed November 5, 2019). *Scalable memory allocation using jemalloc*. https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919.

[8] Keir Fraser. 2004. *Practical lock-freedom*. Technical Report. University of Cambridge, Computer Laboratory.

[9] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E Blelloch, and Erez Petrank. 2020. NVTraverse: in NVRAM data structures, the destination is more important than the journey. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 377–392.

[10] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. 2008. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal* 47, 2 (2008), 221–236. https://doi.org/10.1147/sj.472.0221

[11] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *International Symposium on Distributed Computing (DISC)*. 300–314. https://doi.org/10.1007/3-540-45414-4_21

[12] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.* 67, 12 (2007), 1270–1285. https://doi.org/10.1016/j.jpdc.2007.04.010

[13] 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).

[14] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2002. The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures. In *International Symposium on Distributed Computing (DISC)*. 339–353. https://doi.org/10.1007/3-540-36108-1_23

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

[16] Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management* (1st ed.). Chapman & Hall/CRC.

[17] The GNU C++ Library. 2019 (accessed November 5, 2019). *The GNU C++ Library*. https://gcc.gnu.org/onlinedocs/libstdc++/.

[18] Maged M Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

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

[20] Microsoft. 2021 (accessed November 17, 2021). Microsoft's C++ Standard Library. https://github.com/microsoft/STL.

[21] Aravind Natarajan and Neeraj Mittal. 2014. Fast concurrent lock-free binary search trees. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[22] Ruslan Nikolaev and Binoy Ravindran. 2019. Hyaline: fast and transparent lock-free memory reclamation. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[23] Ruslan Nikolaev and Binoy Ravindran. 2020. Universal Wait-Free Memory Reclamation. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 130–143.

[24] Ruslan Nikolaev and Binoy Ravindran. 2021. Brief Announcement: Crystalline: Fast and Memory Efficient Wait-Free Reclamation. In *International Symposium on Distributed Computing (DISC)*.

[25] Ruslan Nikolaev and Binoy Ravindran. 2021. Snapshot-free, transparent, and robust memory reclamation for lock-free data structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 987–1002.

[26] Pedro Ramalhete and Andreia Correia. [n. d.]. DoubleLink - A Low-Overhead Lock-Free Queue. http://concurrencyfreaks.blogspot.com/2017/01/doublelink-low-overhead-lock-free-queue.html.

[27] Pedro Ramalhete and Andreia Correia. 2017. Brief announcement: Hazard eras-non-blocking memory reclamation. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 367–369.

[28] Håkan Sundell. 2005. Wait-Free Reference Counting and Memory Management. In *International Parallel and Distributed Processing Symposium (IPDPS)*.

[29] Charles Tripp, David Hyde, and Benjamin Grossman-Ponemon. 2018. FRC: a high-performance concurrent parallel deferred reference counter for C++. *Acm Sigplan Notices* 53, 5 (2018), 14–28.

[30] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H Alan Beadle, and Michael L Scott. 2018. Interval-based memory reclamation. *ACM SIGPLAN Notices* 53, 1 (2018), 1–13.

[31] Anthony Williams. 2012. *C++ concurrency in action: practical multithreading*. Manning Publ.

[32] Anthony Williams. 2019 (accessed November 5, 2019). *just::thread Concurrency Library*. https://www.stdthread.co.uk.