Concurrent Deferred Reference Counting with Constant-Time Overhead

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

We present a safe automatic memory reclamation approach for concurrent programs, and show that it is both theoretically and practically efficient. Our approach combines ideas from referencing counting and hazard pointers in a novel way to implement concurrent reference counting with wait-free, constant-time overhead. It overcomes the limitations of previous approaches by significantly reducing modifications to, and hence contention on, the reference counts. Furthermore, it is safer and easier to use than manual approaches. Our technique involves using a novel generalization of hazard pointers to defer reference-count decrements until no other process can be incrementing them, and to defer or elide reference-count increments for short-lived references.

We have implemented the approach as a C++ library and compared it experimentally to several methods including existing atomic reference-counting libraries and state-of-the-art manual techniques. Our results indicate that our technique is faster than existing reference-counting implementations, and competitive with manual memory reclamation techniques. More importantly, it is significantly safer than manual techniques since objects are reclaimed automatically.

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

Keywords: automatic memory reclamation, concurrent algorithms, wait-free

ACM Reference Format:
Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. 2021. Concurrent Deferred Reference Counting with Constant-Time Overhead. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454060

[*]Authors are listed in alphabetical order.

1 Introduction

Memory reclamation, the problem of freeing allocated memory in a safe manner, is essential in any program that uses dynamic memory allocation. A block of memory is safe to reclaim only when it can not be subsequently accessed by any thread of the program. Determining exactly when this is the case is, however, a difficult problem, and even more so for mutlithreaded programs which could be sharing, copying, or modifying references to the same memory blocks concurrently. One solution is to rely on a fully garbage collected language [32], though it is not always possible, most-efficient, or most-flexible to use such languages. In non-garbage-collected languages, concurrent memory reclamation, often called *safe memory reclamation* (SMR), is a non-trivial and extensively studied problem with a long list of proposed solutions. A crucial difficulty in the concurrent setting is the possibility for read-reclaim races [27]. Such a race is between a process that reads and dereferences a pointer to an object and another that reclaims and reuses the corresponding memory, while it is being read.

Concurrent memory reclamation techniques can be broadly divided into two categories, manual and automatic. With manual techniques, the user is responsible for freeing objects. To protect against read-reclaim races, this is often performed with a *retire* operation, which defers the reclamation until it is safe, i.e., until no other thread is reading that object. Such techniques include read-copy-update (RCU) [25], epochs [22], hazard pointers [40], pass-the-buck [28], interval-based reclamation [50], hazard eras [47], and others [11].

Automatic techniques are similar to what can be found in garbage collectors, but without the ability to scan processor private root sets (registers, stacks, etc.). A common technique is reference counting [16, 28, 35, 41, 46, 48, 49], which consists in attaching a counter to each managed object that counts the number of pointers to it, and performing reclamation when the counter hits zero. Both manual and automatic techniques can be implemented as library interfaces, and both need to take care of read-reclaim races. Both can also have some advantages over fully collected languages, such as having more control over memory layout, guaranteeing lock-freedom, or other desirable properties.

In the context of concurrent data structures, manual techniques are often difficult to use and can lead to subtle and hard to reproduce bugs. As evidence, we note that the use of manual memory reclamation in several recent papers is incorrect (see Section 8). These errors can lead to memory leaks or even memory faults. Since these data structures and their use of memory reclamation were implemented and adopted by experts in the field, it would be difficult for common users to get them right.

Reference counting, on the other hand, requires very few modifications for programmers to integrate into their code, and provides memory safety and leak freedom automatically as long as the programmer either does not create reference cycles or breaks such cycles before they become unreachable. Owing to their ease of use, there has been an increase in interest in atomic reference-counted pointers, as evidenced by their inclusion in the most recent C++ standard (C++20). There also exists optimized open-source [21] and even commercial implementations [52]. However, for many concurrent data structures, reference counting can be expensive in practice due to the need to frequently increment and decrement shared counters [27].

A crucial challenge when designing a concurrent reference-counting scheme is dealing with read-reclaim races when the reference count hits zero. In particular, if one thread decrements the counter to zero, initiating reclamation, at the same time that another thread increments the counter, the object will appear to be live with a non-zero reference count, even though it is no longer safe to access. Various techniques have been developed to overcome this, including using tools from manual SMR to delay reclamation until there can no longer be any active reads [28, 48], and the split reference count technique [51, Chapter 7.2.4], which involves maintaining one internal reference count on the managed object itself and possibly several external reference counts, one on each pointer to the object.

In this paper, we propose an efficient approach to automatic memory reclamation based on a novel combination of reference counting and manual SMR. We make several advances to make library-based concurrent reference counting both theoretically efficient and more practical. Theoretically, we show the first solution with constant expected[*] time overhead using only single word compare-and-swap (CAS) and only delaying $O(P^2)$ decrements. Previous approaches are either only lock-free [14, 16, 28, 41, 49, 51], wait-free with $O(P)$ time [48] per operation, or use double-word fetch-and-add [35, 46], which is not available on modern machines.

Our approach is based on a new algorithm that generalizes hazard pointers to allow for multiple retires on the same object. Standard hazard pointers would not be efficient with multiple retires, requiring potentially much more space. This technique allows us to implement *deferred decrements* that protect an object's reference count, delaying decrements (and hence reclamation) while an increment is in progress. This contrasts with previous reference counting techniques [24, 28] that use hazard pointers to delay memory reclamation *after* a reference count hits zero. This is a subtle difference, but it has ramifications both in theory and practice. This generalization of hazard pointers, which we refer to as *acquire-retire*, could be of interest beyond reference counting. We further extend the approach by borrowing the idea of *deferred increments* from reference-counted garbage collectors [4, 5, 7, 17, 36]. When a reference to an object is short lived, it almost certainly doesn't need to modify the reference count. We can facilitate this using acquire-retire to protect the reference count during the reference's short lifetime. In the common case, this avoids both the increment and the decrement.

We have implemented our technique as a library for C++[†] and show that it is more efficient than existing optimized libraries for atomic reference-counted pointers [14, 21, 52]. Our experiments show that deferred decrements alone lead to improved performance over classic approaches, and that deferred increments can result in a substantial speedup–over an order of magnitude on highly contended workloads.

Lastly, we show that our scheme performs well against state-of-the-art manual SMR techniques from a recent benchmark suite [45, 50]. When applied to a range of concurrent data structures for which reference counting previously achieved no scaling whatsoever, our technique keeps up and scales alongside the fastest manual SMR techniques. Furthermore, it manages to achieve throughput rates within a factor of 1.2-2.5x of the fastest manual SMR techniques that use an unbounded amount of memory while consuming only a modest amount itself. Its memory consumption and performance is competitive with hazard pointers but, unlike hazard pointers, it is not constrained to a limited class of data structures. Last but not least, our scheme is automatic and hence easier and safer to use than manual ones. To summarize, the contributions of this paper are:

- A generalization of hazard pointers that supports constant-time acquire and allows multiple concurrent retires of the same handle, which we call *acquire-retire*,
- the design of a theoretically efficient scheme for automatic memory reclamation based on combining acquire-retire and reference counting,
- a practical implementation of the technique as a library for C++, evaluated on a comprehensive set of benchmarks which show that it outperforms existing reference-counting techniques, and is also competitive with manual SMR.

Model and Assumptions. We assume the standard concurrent shared memory model with P asynchronous processes and sequential consistency [30]. Appropriate fences 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 [30]. Roughly speaking, lock-freedom guarantees that some process makes

[*]The "expected time" bounds we give are purely due to hashing.

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

progress whereas wait-freedom guarantees that every process makes progress. Intuitively, linearizability means that each operation appears to take effect atomically at some point during its execution interval. Throughout, when we talk about the *time* of an operation, we mean the number of instructions (both local and shared) performed by that operation before it completes. By *pointer-width*, we mean a word that is just big enough for a pointer—i.e., not even an extra bit hidden somewhere. By space we mean number of words including both shared and local memory. Beyond reads and writes, we consider three other atomic read-modify-write primitives: compare-and-swap (CAS), fetch-and-store, and fetch-and-add. All three are supported by modern processors.

2 Related Work

Manual Memory Reclamation. Hazard pointers [40] is a widely used technique for protecting against read-reclaim races when using manual memory reclamation. The idea is to protect reads on a memory location by storing a pointer to it as a "hazard pointer" in a shared data structure. When a process wishes to free memory, the reclamation of that memory is deferred if it is currently protected by a hazard pointer. Checking if it is protected requires scanning the hazard pointers of all processes, but the cost can be amortized by only scanning every once in a while, buffering pointers that are ready to be freed until the next scan, and reclaiming the memory after the scan has determined which pointers are not protected by a hazard pointer. Protected pointers are checked again on the next scan.

In the terminology we will use, which is reasonably standard, protecting a resource by adding a pointer to the set of hazard pointers is an *acquire*, removing it is a *release*, marking a resource as ready to reclaim is a *retire*, and obtaining a pointer to a retired resource that is safe to reclaim (i.e. is not currently acquired) is an *eject*. Eject is often combined with retire [28, 40] such that a retire will eject and reclaim pointers that were retired earlier and are now safe. Since hazard pointers are meant to protect memory that is being freed, the interface and implementation of hazard pointers require that a memory block is retired at most once. A similar interface is used in pass-the-buck [28]. Acquiring a protected pointer using hazard pointers is lock-free but not wait-free since it might need to retry if the source pointer has been modified to point to a different memory location. The C++ community has proposed an interface for hazard pointers [42] that is implemented in Facebook's Folly library [21].

Read Copy Update (RCU) [25] and epoch-based reclamation [22] also serve a similar purpose and have a somewhat similar interface. Instead of acquiring and releasing individual resources, however, they **read_lock** and **read_unlock** regions that are not paired with any particular resource. They guarantee that everything that is retired, by any process, during the protected (locked) region will be ejected after the region is finished. In some cases, this can make protection easier. On the other hand, it means a retired resource cannot be reclaimed until all protected regions that overlap the retire finish. Most of the regions could have nothing to do with the particular retire. This implies that the memory required by RCU and epochs can be unbounded. The linux implementation of RCU mitigates this problem by disabling interrupts during a locked region and asking the user to ensure that they hold the read lock very briefly.

There has also been a lot of work on combining different forms of memory reclamation. For example, many works have developed methods that combine HP and EBR [33, 47, 50]. Some of these are even wait-free [45], but the memory bound is very large: $O(PM)$ where M is the most memory that was live (allocated but not retired) at any time. Just like HP, these methods also require the programmer to make nontrivial modifications to their data structures. To avoid these modifications, beware and cleanup [24] uses a limited form of reference counting combined with hazard pointers, though their technique still requires manually calling retire.

There has been work on accelerating memory reclamation assuming specialized OS or hardware support. Such approaches including using signaling [2, 3, 11], memory barriers [6, 18], and hardware transactional memory [1, 19]. An interesting hybrid of manual and automatic reclamation involves using compiler support to automatically inject manual memory reclamation code into a data structure [12].

Some memory reclamation techniques have been developed for specific data structures, or data structures assumed to be in a specific form. These include optimistic access [13], which operates on so-called normalized data structures, and drop the anchor [10], which works on a list data structure.

Atomic Reference Counting. Perhaps the most widely used interface in non-garbage-collected languages for automatic reclamation is reference-counted pointers, as provided by the standard libraries of many languages, e.g., C++ and Rust. Atomically maintaining reference counts is a well studied problem. Valois, Michael and Scott [41, 49] first developed a lock-free approach, however, it can increment the counter of freed memory and requires a CAS to update the counter, which could fail and need to be repeatedly retried. Detlefs et al. [16] describe a lock-free method that avoids these two issues, but it requires a DCAS (a CAS on two independent words), which is not supported by current machine architectures. Herlihy et al. [28] are able to remove the DCAS assumption by using their pass-the-buck interface, leaving us with a lock-free solution to the problem with just single-word CAS. It still requires a CAS loop instead of a fetch-and-add due to the use of a sticky counter. All of these solutions are lock-free but not wait-free or constant time. In particular, a thread trying to read a pointer could retry indefinitely as other threads update or copy the pointer.

Sundell developed a wait-free solution [48]. However, like the Valois-Michael-Scott method, it can increment freed memory, making it inappropriate in many practical situations. Also, and perhaps more critically, the approach requires $O(P)$ time to retire each location, which is expensive.

In the practical world, the C++ standard recently added support for *atomic shared pointers* [38], which provide a thread-safe way for multiprocessor environments to share reference-counted pointers. Prior implementations of atomic operations on shared pointers use a small global hashtable of locks, and hence are not scalable in practice. We know of two external libraries that support lock-free solutions [21, 52]. Both are based on the split reference count technique [51, Chapter 7.2.4] and are lock-free, but not wait-free.

A similar technique was developed by Lee [35] and generalized by Plyukhin [46]. Their version is constant time but requires atomic double-word fetch-and-add on a location containing both a pointer and an unbounded sequence number. Unlike double-word CAS, double-word fetch-and-add is not supported by modern machine architectures.

In the preliminary version of this paper [8], we describe our reference-counting algorithm with deferred decrements but without deferred increments. It has the same theoretical time and space bounds. We also describe how the technique can be extended to enable safe atomic loads and stores of more general types other than reference-counted pointers.

Most recently Correia et al. developed OrcGC [14], an automatic memory reclamation scheme that is also based on atomic reference-counted pointers supported under the hood by a novel variant of hazard pointers. Their approach is lock-free and guarantees a linear bound on the number of unreclaimed objects. Similarly to previous work and unlike our work, they protect the managed object from unsafe reclamation when the reference count hits zero, rather than protecting the reference count from being decremented. Their algorithm also requires the use of an unbounded sequence number, which they store in the high-order bits of the reference count. Similarly to our work, they also take advantage of the fact that short lived references need not modify the reference count at all, and can instead temporarily protect the managed object with their hazard-pointer-like scheme.

Deferred Reference Counting. Deutsch and Bobrow [17] introduce *deferred reference counting* for garbage collected languages, which consists in eagerly counting references present in the heap, but ignoring those in registers and on the stack. Objects that reach a heap reference count of zero are placed in a "zero-count table". Periodically, the garbage collector then scans the stack and registers to determine which objects in the zero-count table are reachable, removing them from the zero-count table, or which are unreachable, and hence can be safely destroyed. Subsequent work by Bacon et al. [4], Levanoni and Petrank [36], and Blackburn and McKinley [7] further build on the idea of deferred reference counting, identifying additional situations in which reference-count updates can be deferred or elided entirely.

Unlike our method, all of this prior work focuses specifically on languages with automatic garbage collection and require pausing processes and hence are not lock-free. Although we borrow the name "deferred reference counting“ due to the high-level conceptual similarities, our techniques and methods are substantially different because they apply to manually memory-managed languages.

Single-Writer Atomic Copy. Our algorithm uses a recently proposed single-writer atomic copy primitive. More specifically, Blelloch and Wei [9] show how to implement a *Destination* object which supports reading from, writing to, and copying into with the restriction that only one process can by writing to or copying into a Destination object at a time. The copy into operation, *swcopy*, takes as input a pointer to a memory location and atomically copies the value from that memory location into the destination object. Their implementation supports all three operations in constant time and uses $O(M+P^2)$ space for M destination objects. Furthermore, it only uses single-word CAS and does not use unbounded sequence numbers. Their implementation achieves wait-free, constant-time swcopy by having read operations help copies that are in progress.

3 Overview of Our Approach

Recall that the difficulty of implementing safe concurrent reference counting is the possibility for a race between a decrement that sets the count to zero, initiating reclamation, and an increment, which increments the counter back above zero, giving the appearance that the managed resource is still live. Our idea is, intuitively, that if there is an increment racing with a decrement, to delay the decrement until after the increment has completed.

Our key insight is that this can be achieved by applying a hazard-pointers-like scheme where the resource being protected is neither a memory block nor a managed object, but rather the reference count itself that is attached to a managed object. This leads to simple algorithm for concurrent reference counting. To obtain a new pointer to a reference-counted object, our algorithm *acquires* the reference count of the object to protect it, then increments the counter and *releases* the protection. To discard a pointer, the reference count of the object is *retired*, which, when ejected (i.e., at some point in time when the reference count is not acquired by any increment), decrements the reference count, deleting the managed object and reclaiming the memory if it reaches zero. By delaying decrements until all the increments that started before it complete, we ensure that an object is safe to collect as soon as its reference count hits zero. This contrasts with previous techniques [24, 28] that perform decrements eagerly and use SMR to delay memory reclamation *after* a reference count hits zero.

Note that in our algorithm, a reference count could be retired multiple times before being ejected a single time. This could happen, for example, in an execution where three pointers to the same reference-counted object are discarded concurrently. Traditional hazard pointers interfaces [29, 40, 42] explicitly disallow resources from being retired more than once, which make sense in the SMR setting, but not when managing more general resources such as reference counts. To support these more diverse use cases, we define a generalization of hazard pointers called *acquire-retire* and show how to implement it efficiently, with all operations taking only constant time in expectation.

Lastly, we extend our reference-counting algorithm, which defers decrements, with what we call *snapshots*, which can be thought of as *deferred increments*. When a reference to an object is short lived, such as during the traversal of a linked data structure, a standard reference-counting scheme would have to increment and decrement the reference count in quick succession. Instead, we observe that we can apply acquire-retire to temporarily protect the reference count during the snapshot's lifetime. This avoids both the increment and the decrement, which we show substantially improves the practical performance of our scheme.

3.1 Our Reference-Counting Library

To illustrate and evaluate our techniques, we implemented them as a library for C++. Our implementation makes use of standards-compliant C++ features, including C++11 atomics and memory orderings, and uses no OS- or architecture-dependent code. In this section, we briefly discuss the interface of our library, compare it to the interfaces of other memory reclamation techniques, and discuss an important practical feature that allow us to efficiently implement a range of concurrent data structures.

Our library consists of three class templates, starting with `atomic_rc_ptr<T>`, which provides thread-safe management of an `rc_ptr<T>`, which manages a reference-counted pointer to an object of type `T`. The `atomic_rc_ptr<T>` interface is modelled after `atomic<shared_ptr<T>>` in the C++ standard, while `rc_ptr<T>` is designed to closely mimic `shared_ptr<T>`. Lastly, we provide `snapshot_ptr<T>`, which facilitates low-cost reads of an object managed by an `atomic_rc_ptr<T>` by protecting it with a deferred increment, rather than an explicit increment of the reference counter. We describe the usage of these types in more detail in the following sections.

atomic_rc_ptr. `atomic_rc_ptr<T>` is closely modelled after C++'s `atomic<shared_ptr<T>>`. It provides support for all of the standard operations, such as atomic load, store, and CAS.

- **load**`()`. Atomically creates an `rc_ptr` to the currently managed object, returning the `rc_ptr`.
- **get_snapshot**`()`. Atomically creates a `snapshot_ptr` to the currently managed object, returning the `snapshot_ptr`.
- **store**`(desired)`. Atomically replaces the currently managed pointer with `desired`, which may be either an `rc_ptr` or a `snapshot_ptr`.
- **compare_and_swap**`(expected, desired)`. Atomically compares the managed pointer with expected, and if they are equal, replaces the managed pointer with desired. The types of `expected` and `desired` may be either `rc_ptr` or `snapshot_ptr`, and need not be the same.
- **compare_exchange_weak**`(expected, desired)`. Same as `compare_and_swap` but, if the managed pointer is not equal to expected, loads the currently managed pointer into expected. This operation may spuriously return false, i.e. it is possible that the value of expected does not change.

The most interesting point of the interface is that it supports two flavors of load operations, **load** and **get_snapshot**, which return `rc_ptr` and `snapshot_ptr` respectively.

rc_ptr and snapshot_ptr. The `rc_ptr` type is closely modelled after C++'s standard library `shared_ptr`. It supports all pointer-like operations, such as dereferencing, i.e. obtaining a reference to the underlying managed object, and assignment of another `rc_ptr` to replace the current one. It is safe to read/copy an `rc_ptr` concurrently from many threads, as long as there is never a race between one thread updating the `rc_ptr` and another reading it. Such a situation should be handled by an `atomic_rc_ptr`.

The `snapshot_ptr` type supports all of the same operations as `rc_ptr`, except that `snapshot_ptr` can only be moved and not copied. Additionally, while `rc_ptr` can safely be shared between threads, `snapshot_ptr` should only be used locally by the thread that created it. The use of `snapshot_ptr` should result in better performance than `rc_ptr` provided that each thread does not hold too many `snapshot_ptr` at once. If a thread exceeds the soft limit on `snapshot_ptr` (see Section 5.2), their performance will degrade to similar to or slightly worse than `rc_ptr`. Therefore, `snapshot_ptr` is ideal for reading short-lived local references, for example, reading nodes in a data structure while traversing it.

To illustrate our library and the three types, we refer to an implementation of a concurrent stack in Figure 1a, which we elaborate on in the next section. The `head` node of the stack is stored in an `atomic_rc_ptr` because it may be modified and read concurrently by multiple threads. Each node of the stack stores its `next` pointer as a non-atomic `rc_ptr`. This is safe, because although multiple threads may read the same pointer concurrently, the internal nodes of the stack are never modified, only the head is. Lastly, we can use a `snapshot_ptr` while performing `pop_front`, since reading the head is a short-lived local reference that will never be shared with another thread.

Support for Marked Pointers. A common optimization in concurrent data structures is to steal some of the unused bits from a pointer to mark links in the data structure as pending deletion. Since our reference-counted pointer algorithm uses plain single-word pointers and does not internally steal any bits, it is possible to expose those redundant bits to the programmer for them to use in this fashion. Our pointer types therefore include a customization point that allows a markable pointer type to be used in place of raw pointers internally, and allows custom behavior to be added via a policy class. We have used this to implement *markable* versions of our types that offer `get_mark`, `set_mark`, and `compare_and_set_mark`, which require no manual bit twiddling from the programmer, allowing them to easily and efficiently implement data structures with marked links.

3.2 Usability Comparison to Manual SMR

Our interface is closely modeled after and designed to be as easy to use as the standard C++ types. In Figure 1, we depict three implementations of a concurrent stack, using our library, hazard pointers, and RCU. Our code avoids the potential pitfalls of manual SMR, as it is impossible to read the value stored in `head` without protecting it automatically, and no manual retires are necessary. Although calling retire is quite simple in this example, it is not always so easy. Figure 2 depicts a snippet of code from an implementation of the Natarajan and Mittal tree [44]. This code cleans up deleted nodes from the tree by swinging a pointer from a node to one of its descendants. It is a subtle but important detail to notice that in the presence of concurrent updates, this operation may delete multiple nodes, and hence may be required to retire many nodes, not just *successor*. In Section 8, we discuss how this bug and others have appeared in the artifacts of several published papers written by memory management experts.

4 Defining the Acquire-Retire Interface

We propose a generalization of hazard pointers for resource management called *acquire-retire*. As with hazard pointers, it supports four operations: acquire, release, retire, and eject. The generalization is that it allows multiple retires of the same handle, which is critical in our reference-counting implementation. An *acquire* takes a pointer to a location containing a resource handle, reads the handle and *protects* the resource, returning the handle. A later paired *release*, releases the protection. A *retire* is used to indicate the resource is no longer needed. A later paired *eject* will return the resource handle indicating it is no longer protected and safe to destruct. We say a retire, or its corresponding destruct, is *delayed* between the retire and when its handle is ejected. For our time and space bounds, we require that every retire is followed by at least one eject. All operations are linearizable [31], i.e., must appear to be atomic.

We describe a constant-time implementation of acquire-retire, that only requires single-word memory instructions, and for P processors and K protected resources has $O(PK)$ memory overhead. Describing an efficient implementation of acquire-retire requires two insights. The first is that hazard pointers can be combined with a recent result on atomic copy [9] to ensure constant-time acquire. The second insight is that multiple concurrent retires of the same handle can be supported by appropriately keeping track of multiplicity.

Allowing multiple retires of the same handle makes defining the behavior of **retire** and **eject** more subtle. The high-level approach is to associate **acquire** operations with **retire** operations rather than handles. In our interface, **acquire**(ptr, ann) takes as input a pointer to a memory location (ptr) storing a resource handle, and a pointer to an announcement slot (ann). It returns the handle stored at the memory location. The **release**(ann) operation takes as input an announcement slot, and has no return value. In a sequential execution, we say that an **acquire**(ptr, ann) operation is *active* between its execution and the execution of either the next **acquire**($*$, ann) operation or the next **release**(ann) operation, whichever comes first. After this point, the **acquire** is said to be *inactive*. The **retire**(h) operation takes as input a handle and the **eject** operation either returns $\perp$ or a handle.

Our implementation requires that **acquire**/**release** operations on the same announcement slot are never concurrent with each other. Typically, each process will have its own private set of announcement slots. Announcement slots can either be allocated statically, or dynamically as threads are created and retired, in the same way as hazard pointers [40]. We formally specify the behaviour of the interface below.

Definition 4.1 (Acquire-Retire)**.** *Any proper, concurrent execution can be linearized to a sequential history with the following guarantees:*

1. *Each* **acquire**(ptr, $*$) *returns the handle currently stored in the memory location pointed to by* ptr.
2. *Let* f *be a function that maps each* **acquire** *returning* h *to either a later* **retire**(h) *or* $\perp$. *Let* g *be an injective (one-to-one) function that maps each* **eject** *returning* h *to an earlier* **retire**(h). *For all* f, *there is a* g *such that whenever* $f(A) = g(E)$, *the* **acquire** A *is inactive by the time* **eject** E *is executed.*

We note that Definition 4.1 captures our intuition of what the interface is supposed to protect against. In particular, it ensures that any destruct of a resource placed after the **retire** and **eject** will happen after all processes **release** that resource. If there are multiple retires on the same handle, it ensures that each is mapped to at most one **eject**.

Definition 4.1 never forces **eject** operations to return a handle, so for an implementation of acquire-retire to be useful, it has to provide some guarantees on how often **retire**s are ejected. We say a **retire** is ejected if there is an **eject** mapped to the **retire**. Assuming each call to **retire** is always followed by a call to **eject**, our algorithm ensures that there are always no more than $O(KP)$ **retire**s that have not been

```
struct Node { T t; rc_ptr<Node> next; }
atomic_rc_ptr<Node> head;

void push_front(T t) {
  rc_ptr<Node> p = make_rc<Node>(t, head.load());
  while (!head.compare_exchange_weak(
          p->next, p)) {}
}

optional<T> pop_front() {
  snapshot_ptr<Node> p = head.get_snapshot();
  while (p != nullptr &&
         !head.compare_exchange_weak(
         p, p->next)) { }
  if (p != nullptr) return {p->t};
  else return {};
}
```

(a) Our Library

```
struct Node : rcu_obj_base<Node> {
  T t; Node* next; };
atomic<Node*> head;

void push_front(T t) {
  auto p = new Node{{}, t, head.load()};
  while (!head.compare_exchange_weak(
          p->next, p)) {}
}

optional<T> pop_front() {
  rcu_reader guard;
  auto p = head.load();
  while (p != nullptr &&
         !head.compare_exchange_weak(
         p, p->next)) { }
  if (p != nullptr) {
    p->retire();
    return {p->t};
  }
  else return {};
}
```

(b) RCU

```
struct Node : hazptr_obj_base<Node> {
  T t; Node* next; };
atomic<Node*> head;

void push_front(T t) {
  auto p = new Node{{}, t, head.load()};
  while (!head.compare_exchange_weak(
          p->next, p)) {}
}

optional<T> pop_front() {
  Node* p;
  hazptr_holder h;
  do {
    p = h.get_protected(head);
    if (p == nullptr) return {};
  } while (!head.compare_exchange_weak(
           p, p->next)) { }
  if (p != nullptr) {
    p->retire();
    return {p->t};
  }
  else return {};
}
```

(c) Hazard Pointers

Figure 1. C++ implementations of an ABA-safe, concurrent stack using our library, RCU, and hazard pointers. The syntax for hazard pointers and RCU is based on a C++ standards proposal [43], and is implemented in Folly [21].

```
void cleanup() {
  ...
  /* Update the left child of ancestor to point to sibling */
  if(ancestor.left->compare_and_swap(successor, sibling)) {
    /* retire nodes on path from successor to sibling */
    for(Node* n = successor; n != subling;) {
      Node* tmp = n;
      if(getFlag(n->left)) {
        retire(n->left);
        n = n->right;
      } else {
        retire(n->right);
        n = n->left;
      }
      retire(tmp);
    }
    return true;
  } else return false; }
```

Figure 2. Manually calling retire is easy to forget and it sometimes adds non-trivial code. The highlighted portion of the code is not needed in our library.

ejected, where K is the total number of announcement slots. We defer the description of our algorithm for acquire-retire until Section 6.

5 Deferred Reference Counting

Armed with the acquire-retire technique, we now describe our algorithms for reference counting with deferred decrements and increments. The interface supports atomically storing to, loading from, and CASing into a mutable reference-counted pointer in a shared location. Our algorithms support these operations with constant-time overhead, have $O(P^2)$ memory overhead, and defer at most $O(P^2)$ reference-count decrements (see Theorem 1). We note that deferred increments is just a practical optimization which does not affect these bounds.

Both algorithms also have the useful property that references are implemented as raw pointers, which means two things. First, that a reference occupies just a single word, unlike some implementations [52] which use a double-word representation and require a double-word CAS. Second, that we do not "steal" any bits of the pointer representation, as is done by some libraries [21]. This is important in some applications, since it leaves unused bits of the pointer representation for the user to utilize, which is necessary in many common implementations of lock-free data structures that "mark" pointers. For example, the Harris linked list [26] or Natarajan and Mittal's binary search tree [44].

5.1 Deferred Decrements

Recall that the race we are trying to avoid when designing a scheme for concurrent reference counting occurs when one thread removes a reference, decrementing the corresponding counter to zero, at the same time that another thread creates a new reference, incrementing the counter. Such races in which a location can be simultaneously read by one thread and updated by another can occur in just about any lock-free data structure. Our approach solves this problem by using acquire-retire to protect the reference count and defer decrements from being applied while there is a potential increment in progress. We say a decrement is *deferred* if a reference has been overwritten or otherwise deleted, but the count on the corresponding managed object has not yet been decremented. The eject operation on the reference count corresponds to decrementing the count and, if it goes to zero, reclaiming the managed object.

Algorithms and Analysis. Figure 3 depicts our algorithm using a reference-counting interface similar to the one used by Herlihy et al. [28] and Detlefs et al. [16]. We assume each reference-counted object has a counter attached that can be atomically incremented or decremented with **addCounter**, which returns the old value.

The **load** operation atomically loads a pointer from a shared memory location into a local pointer and returns

```
using ref = Object*;

AnnouncementSlot announcement[P];

ref load(ref* A) {
  ref ptr = acquire(A, &announcement[pid]);
  if (ptr != nullptr) increment(ptr);
  release(&announcement[pid]);
  return ptr; }

void store(ref* A, ref desired) {
  if (desired != nullptr) increment(desired);
  ref current = fetch_and_store(A, desired);
  if (current != nullptr) {
    retire_and_eject(current); }

bool cas(ref* A, ref expected, ref desired) {
  ref ptr = acquire(&desired, &announcement[pid]);
  if (compare_and_swap(A, expected, desired)) {
    if (desired != nullptr) increment(desired);
    if (expected != nullptr) {
      retire_and_eject(expected); }
    release(&announcement[pid]);
    return true;
  } else {
    release(&announcement[pid]);
    return false; } }

void destruct(ref ptr){
  if (ptr != nullptr) {
    decrement(ptr); } }

void retire_and_eject(ref ptr) {
  retire(ptr);
  optional⟨ref⟩ e = eject();
  if (e != ⊥) decrement(e); }

void increment(ref ptr) {
  ptr->addCounter(1); }

void decrement(ref ptr){
  if (ptr->addCounter(-1) == 1) {
    delete ptr; } }
```

Figure 3. Operations for atomic reference-counted pointers with deferred decrements. pid is the unique id of the current processor, $0 \le \texttt{pid} < P$.

it. Since **load** creates a new reference to the object, it increments the reference count. To protect against a potential race between this increment and a decrement setting the count to zero, the increment is surrounded by an **acquire** and **release**.

The **store** operation atomically copies a local pointer into a shared memory location. Since this creates an additional reference to `desired`, it first increments the reference count. Note the subtle detail that unlike in **load**, this increment does not need to be protected by an **acquire** and **release**. This is because the existence of the argument `desired` guarantees that the reference count is at least one, and hence can not race to zero during this operation. Our implementation writes into the shared memory location using a fetch-and-store operation so that it can decrement the reference count of the pointer that was overwritten. Decrementing the reference count immediately would introduce a race, so instead, we defer the decrement by retiring the pointer. Each **retire** is always followed by an **eject** of a previously retired pointer, which is then decremented. Recall that pairing each **retire** with an **eject** is what allows acquire-retire to yield efficient time and space bounds. Notice that a process might have the same pointer in its retired list multiple times, which is why we need the more general acquire-retire interface rather than hazard pointers.

The **cas** operation works similarly to **store**, except that it only modifies the reference counts if the underlying CAS succeeds. Note that for safety reasons, **cas** must first protect `desired` with an **acquire** before performing the CAS. If it did not, the CAS could succeed right before another thread stored to `A`, which could cause the reference count of `desired` to be decremented. If this decrement took the count to zero, initiating reclamation, the object would be unsafely destroyed before the **cas** had a chance to increment the reference count.

The **destruct** operation takes as input a reference-counted pointer that is no longer needed and destroys it. In an object-oriented language, such as C++ or Rust, this would be handled automatically by the reference-counted pointer's destructor. Note that since it would be an unsafe race to read from a pointer while it was being destroyed, **destruct** does not have to call **retire** but can instead eagerly decrement.

The **retire_and_eject**, **increment** and **decrement** operations are used internally and are not part of the interface. The **decrement** operation is responsible for initiating reclamation if the counter is decremented to zero. In our pseudocode, by **delete**, we mean to destroy the underlying `Object`, which includes recursively calling **destruct** on any reference-counted pointers it owns, and reclaiming the memory it occupies.

Theorem 1 (Deferred Reference Counting): *On P processes, any number of reference-counted objects with references stored in shared mutable locations supporting atomic load, store, and CAS can be implemented safely with:*

1. *references as just pointers (i.e., single-word addresses),*
2. *$O(1)$ time for load,*
3. *$O(1)$ expected time for store and CAS excluding the cost of any call to* **delete** *resulting from a* **decrement**
4. *$O(P^2)$ space overhead and $O(P^2)$ deferred decrements,*
5. *only single-word read, write, CAS, fetch-and-store, and fetch-and-add.*

This implies constant-time overhead since the deletion of the retired objects is required by any non-trivial reclamation scheme. The proof of Theorem 1 is deferred to the full paper.

Copy versus Move Semantics. Our algorithms in Figure 3 implement **store** and **cas** with *copy semantics*. That is, since they effectively create a new reference to `desired`, they increment the corresponding reference count. In many practical situations however, the caller may have no subsequent use for their copy of `desired`, which may be soon to be destructed, leading to a decrement of the reference count. In this situation, it is favorable to implement versions of **store** and **cas** that have *move semantics*, i.e., that *consume* the copy of `desired` passed as an argument. This removes the need to increment the reference count since the caller gives up their count. Our C++ library implements this optimization.

5.2 Deferred Increments / Snapshots

A big performance bottleneck that appears when implementing concurrent data structures using pure reference counting occurs when traversing linked nodes. On a node-based concurrent data structure, a safe traversal requires temporarily incrementing and decrementing the reference counts of all the nodes encountered to prevent them from being deleted while being read. This is inefficient for multiple reasons; increments and decrements must be performed with an atomic fetch-and-add instruction, and these may contend if multiple processors are operating on the same node concurrently.

This contrasts with other SMR techniques such as hazard pointers, which just perform a write to a single-writer location for each node traversed, or epoch-based methods, which perform a single write before beginning the traversal. Neither of these methods experience any contention. Furthermore, due to cache coherency protocols, incrementing the reference count of a node reserves the cache line in exclusive mode, causing the other processes to experience a cache miss the next time they access this node.

In the previous section, we gave algorithms for reference counting with deferred decrements. Although they achieve constant-time overhead, they are still prone to the practical performance hit of frequent increments. To improve our scheme in practice, we therefore introduce the notion of *deferred increments*. Specifically, if an algorithm needs to briefly protect an object, such as during the traversal of a linked data structure, but does not need to keep a long-lasting reference, we observe that there is no need to eagerly increment the reference count. Instead, the algorithm can use the existing infrastructure of acquire-retire to temporarily prevent any decrements from being applied while the reference is held. We refer to such a protected reference as a *snapshot*. Snapshots prevent deferred decrements from being applied while they are held, and when they are released the protection can be cleared, resulting in no change to the reference counter.

By using reference counting to protect long-lived references, such as links inside the data structure, and snapshots to protect short-lived references, we obtain the best of both worlds – the ability to traverse the data structure without introducing contention, without the burden of having to manually retire nodes that are no longer reachable. This is not possible with a pure reference counting or pure SMR (e.g., hazard pointers) approach.

Snapshot Algorithm. We show the algorithms for snapshots in Figure 4. The **get_snapshot** operation is similar to **load**, except that it returns a `Snapshot`, which is a protected local reference coupled with an `AnnouncementSlot`. When a snapshot is no longer needed, it can be released with the **release_snapshot** method. Note that the same processor that acquired the snapshot must release it.

Since multiple snapshots may need to be held by a single processor, our implementation allocates seven additional

```
using Snapshot = pair⟨ref, AnnouncementSlot*⟩;
int MAX_SNAPSHOTS = 7;

AnnouncementSlot snapshots[P][MAX_SNAPSHOTS];
thread_local int next;

Snapshot get_snapshot(ref* A) {
  AnnouncementSlot* slot = get_slot()
  ref ptr = acquire(A, slot);
  return {ptr, slot}; }

void release_snapshot(Snapshot S) {
  auto [ptr, slot] = S;
  if (ptr != nullptr) {
    if (slot->read() == ptr) release(slot);
    else decrement(ptr); } }

AnnouncementSlot* get_slot() {
  for (int i = 0; i < MAX_SNAPSHOTS; i++)
    if (snapshots[pid][i].read() == ⊥)
      return &snapshots[pid][i];
  AnnouncementSlot* slot = &snapshots[pid][next];
  increment(slot->read())
  next = (next + 1) % MAX_SNAPSHOTS;
  return slot; }

void destruct(ref ptr){
  if (ptr != nullptr) {
    retire_and_eject(ptr); } }
```

Figure 4. Interface and algorithm for snapshots. This algorithm is compatible with the reference-counting algorithm of Figure 3, except that the **destruct** operation from Figure 3 must be replaced with the one given here.

announcement slots per processor. This means that the eight total announcement slots of a process fit on a single cache line on common architectures. By packing them into a single cache line, the **ejectAll** method of acquire-retire does not suffer any noticeable performance loss.

When a process wishes to acquire a snapshot, the algorithm scans its announcement slots and selects the first empty slot it finds. If no slots are available, it selects one of the existing slots and eagerly increments the reference count on the protected object (i.e., it applies the deferred increment) and takes over the slot for itself. In our implementation, the slot to take over is selected in a round-robin fashion. When a snapshot is released, it checks whether its announcement slot has been reused, and if so, correspondingly decrements the reference count. Otherwise, no decrement is necessary, and the announcement can simply be released.

Lastly, to safely hold snapshots, we need to slightly modify the **destruct** operation for references. If a snapshot is taken from a shared reference, and that reference is subsequently updated, the reference count can not be eagerly decremented, or the object protected by the snapshot might be destroyed. Instead, the decrement must be deferred by calling **retire**.

6 Acquire-Retire Algorithm

We now describe how to implement constant-time **acquire**, **release**, **retire**, and expected constant-time **eject**. This algorithm uses techniques from hazard pointers [40] and pass-the-buck[28] with some changes to support the more general acquire-retire interface.

```
1  using AnnouncementSlot = Destination⟨optional⟨T⟩⟩;

3  thread_local list⟨T⟩ rlist;
4  thread_local list⟨T⟩ flist;

6  T acquire(T* ptr, AnnouncementSlot* ann) {
7    ann->swcopy(ptr);
8    return ann->read(); }

10 void release(AnnouncementSlot* ann) {
11   ann->write(⊥); }

13 void retire(T t) {
14   rlist.add(t); }

16 optional⟨T⟩ eject() {
17   perform steps towards ejectAll(rlist);
18   if (!flist.is_empty())
19     return flist.pop();
20   return ⊥; }

22 void ejectAll(list⟨T⟩ rl) {
23   list⟨T⟩ plist = empty;
24   // loop through all existing AnnouncementSlots
25   for each AnnouncementSlot* ann {
26     optional⟨T⟩ a = ann->read();
27     if(a != ⊥) plist.add(a); }
28   list⟨T⟩ freed = multiSetDiff(rl, plist);
29   flist.add(freed);
30   rlist.remove(freed); }
```

Figure 5. Implementing acquire-retire. `Destination` is a destination object supporting atomic copies [9]. `slots` is a list of all of the announcement slots owned by all processors.

The standard lock-free version of **acquire** from hazard pointers executes a loop in which the pointer to be protected is read from a shared location and written into a local announcement slot. Each iteration, the pointer is re-read from the shared location to check whether it still matches the one that was announced. To reduce the complexity of **acquire** to constant time, we leverage a recently proposed primitive called `swcopy` [9], which atomically copies from one location to another location, but requires that the destination location is only written to by a single process. Note that making the read of the shared location and write to the announcement slot appear to happen atomically is precisely the purpose of the lock-free acquire loop, and hence, by replacing it with a `swcopy`, we can implement **acquire** in constant time. To use this primitive, we would have to implement announcement slots using the *Destination* object described in Section 2. Blelloch and Wei [9] present an implementation of M *Destination* objects using $O(M + P^2)$ space such that *read*, *write* and *swcopy* all take constant time.

A **release**(*ann*) operation unprotects by simply clearing the announcement slot *ann*, and **retire**(x) simply adds x to a process-local retired list called *rlist*. To determine which handles are safe to eject, the **ejectAll**(*rl*) method loops through all the announcement slots and makes a list of all the handles that it sees. We call this list of handles *plist* for "protected list". If a handle is seen multiple times in A, then it will also appear that many times in *plist* (this differs from standard hazard arrays). Next, **ejectAll** computes a multi-set difference between *rl* and *plist*. This step can be implemented in $O(|rl| + K)$ expected time using a local hash table, where K is the total number of announcement slots. The result of this multi-set difference are handles that can be safely ejected without violating the specifications of acquire-retire. It is important that we keep track of multiplicity and perform multi-set difference because when a handle is retired multiple times, each occurrence of this handle in the announcement slots might be associated with a different **retire**. So if a handle appears in the retired list s times and the announcement slots t times, it is safe to eject only $s - t$ copies of this handle.

An **eject** is essentially a deamortized version of **ejectAll**. Every time it is called, it performs a small constant number of steps towards **ejectAll**(*rlist*), where each hash table operation counts as a single step. Thus **eject** takes expected constant time. When **ejectAll** returns a list of handles, they get removed from *rlist* and added to a local free list to be returned one at a time by the following **eject**s.

Pseudocode for this implementation appears in Figure 5 and its properties are summarized in Theorem 2.

Theorem 2 (Acquire-Retire): *For an arbitrary number of resources and locations, P processes, and at most K resources protected at any given time, the acquire-retire interface can be supported with:*

1. *$O(1)$ time for acquire, release, and retire,*
2. *$O(1)$ expected time for eject,*
3. *$O(KP)$ deferred retires,*
4. *$O(KP)$ space overhead assuming $K \geq P$, and*
5. *only single-word read, write and CAS*

Proof (outline). To show that Algorithm 5 is a linearizable implementation of the acquire-retire interface, we need to prove both properties in Definition 4.1. Here we outline the main ideas of the proof. The full proof of correctness is left for the full paper. The first property is reasonably straightforward since we use an atomic copy, returning the value copied. For the second property we first identify the linearization points of the four operations, which are at Lines 7, 11, 14, and 19. We then need to consider all functions f that map **acquire**s to later **retire**s. For any such function we construct an injective function g from **eject**s to **retire**s that satisfies the required property, i.e., whenever $f(A) = g(E)$, the **acquire** A is inactive by the time **eject** E is executed. The idea here is to tag (for proof purposes) all elements of the announcement slots for acquires A with the **retire** $f(A)$. Similarly we tag all elements in the local retired list with the **retire** that added it. When taking the multiset difference from the retired list to the announcement slots, we can pair, one-to-one, elements in the retired list with ones that match in the announcement slots. By the algorithm, any that match will not be placed on the free list by **ejectAll**. The remaining will be, and when later ejected, the constructed function g will map the eject to the **retire** tag on the handle. Hence, since all active **acquire**s, and corresponding tagged **retire**s, are in the announcement slots, for an **acquire** A and eject E, with $f(E) = g(A)$, A is no longer active at the eject. □

7 Evaluation

In this section, we provide an experimental evaluation of our C++ library across two benchmark setups. First, we compare its performance to other implementations of reference-counted pointers. Second, we compare our approach to the performance of manual SMR techniques. We performed some preliminary experiments using the wait-free **acquire** algorithm, and found that it was as fast as the lock-free one after applying a fast-path slow-path methodology [34], but since the performance was mostly determined by the fast path, we decided to use the simpler lock-free implementation for the rest of the experiments.

Setup. We ran our experiments on a 4-socket machine with 72 physical cores in total (Intel(R) Xeon(R) E7-8867 v4, 2.4GHz), 2-way hyperthreading, and 45MB L3 cache. The machine's interconnection layout is fully connected meaning that all four sockets are equidistant from each other. We interleaved memory across sockets using *numactl -i all*. For scalable memory allocation we used the jemalloc library [20]. All of our experiments were written in C++ and compiled with g++ version 9.2.1 on optimization level `O3`. Our experiments vary the number of threads from 1 to 200, which serves to also measure the effect of oversubscription since our hardware supports up to 144 with hyperthreading.

7.1 Comparison of Reference-Counting Techniques

We compare with implementations from four different libraries: the `atomic_` free functions for `shared_ptr`[‡] from libstdc++ (The GNU C++ library [37]), Anthony William's just::thread library [52], Facebook's Folly library [21], and OrcGC [14]. The implementation in libstdc++ is lock-based whereas the others are lock-free. Both just::thread and Folly use something similar to the split reference count technique described in [51, Chapter 7.2.4]. We also implemented two reference-counted pointers based on Herlihy et al. [28]. The first follows their approach as closely as possible, while the second is an improved version that we optimized. Specifically, we replaced some of the CAS loops in the original algorithm with fetch-and-add and fetch-and-store instructions where applicable to improve performance.

Microbenchmark #1: Load/Store Throughput. We maintain an array of N shared memory locations, each storing an atomic reference counted pointer to a 32-byte object. The array is padded so that each pointer is on a different cache line. Each thread picks a memory location uniformly at random and performs either a **load** or a **store**. Threads perform a **store** with probability p_s and a **load** with probability $1 - p_s$. Before a **store**, the thread allocates a new reference-counted pointer to a new object to be stored. After a thread performs a **load**, which increments the reference count, it reads the value being pointed to, and then destructs the loaded pointer. We show results for $N = 10$, a highly contended workload, and $N = 10M$, a workload with almost no contention. We run each experiment for 5 seconds, which was sufficient for reaching steady state performance, and report the total throughput of **load**s and **store**s averaged across 5 runs.

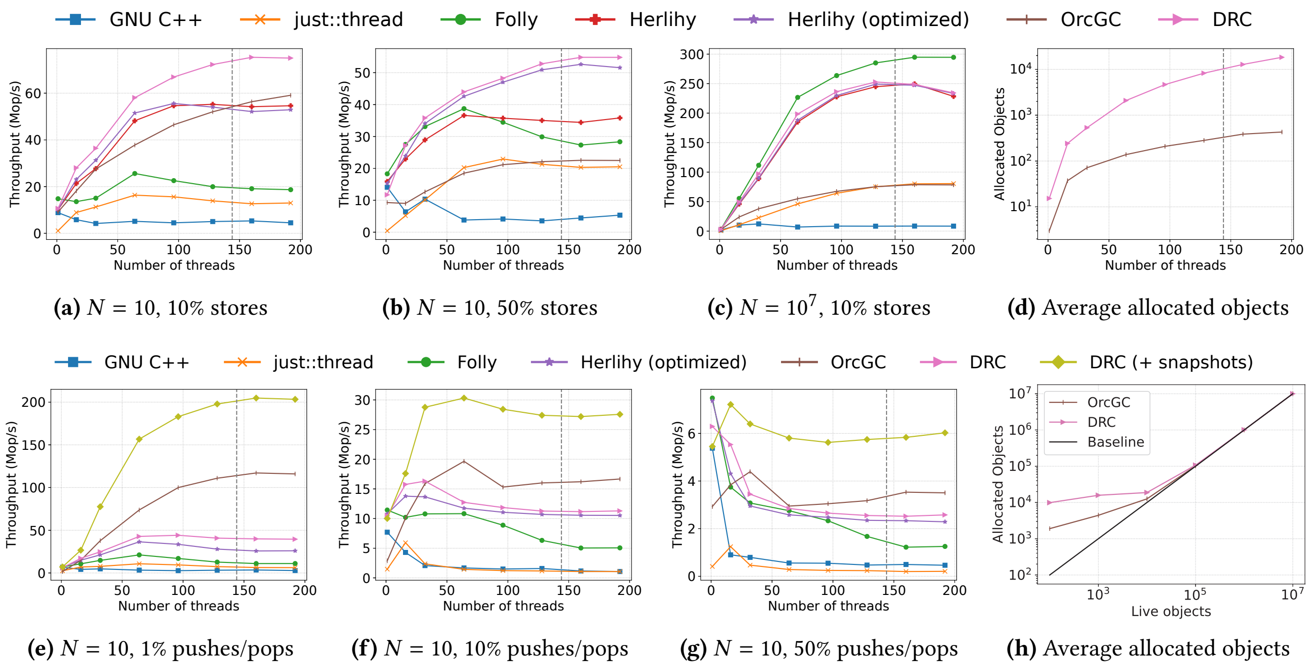
Results. The results of these experiments are depicted in Figures 6a–6c. In the high-contention workloads (6a–6b) our implementation (DRC) consistently outperforms the others, particularly on the load-heavy workload. Though Folly and just::thread use similar a similar technique, we found that Folly's implementation consistently outperforms just::thread. This is because Folly's implementation is highly optimized. For example, they pack a 48-bit pointer and a 16-bit counter into a single word to avoid the double-word-width CAS used by just::thread. The libstdc++ implementation achieves little if any observable speed up after 16 threads because it uses a set of 16 global locks. The second-best overall competitor is Herlihy's algorithm, our improved version of which comes close to the performance of our algorithm on the store-heavy workload. Although it does not exhibit the strongest throughput, OrcGC shows consistent scaling. On the read-heavy workload, it catches up to the performance of Herlihy at 144 threads, and takes over second place once oversubscription is entered. On the store-heavy workload, however, OrcGC is consistently outperformed by both Folly and Herlihy.

On the low contention workload, Folly is the winner, while Herlihy and our algorithm come in second. just::thread and OrcGC trail behind, and libstdc++ exhibits no scaling at all. Folly's performance is attributable to the fact that, under low contention, the work performed by the deferred algorithms to acquire and protect the pointer is almost always unnecessary. OrcGC's performance on the store-heavy and low-contention workloads compared to its stronger earlier performance on the load-heavy workload suggest that its store operation is particularly expensive. This can be explained by the fact that its retire operation, which will be invoked on each store, performs $O(P)$ work, while ours and Herlihy perform constant expected work.

The tradeoff is that our approach and that of Herlihy use more memory. They may defer up to $O(P^2)$ reclamations, while OrcGC defers at most $O(P)$ reclamations, and the other schemes perform no deferred reclamation and always reclaim immediately. In Figure 6d, we show the average memory usage in terms of the number of allocated objects against the number of threads. The average number of objects allocated for our algorithm is approximately $0.5P^2$, while the number allocated by OrcGC is approximately $3P$, which matches the theoretically expected bounds.

Microbenchmark #2: Concurrent Stack. We implemented a concurrent stack using the code shown in Figure 1a, but also supporting a *find* operation that takes as input, a value, and searches the stack, returning true if that value is present.

[‡]At the time of writing, the latest C++ standard has deprecated these free functions and replaced them with specializations of `std::atomic`. However, neither libstdc++ or libc++ have yet provided an implementation.



(a) $N = 10$, 10% stores **(b)** $N = 10$, 50% stores **(c)** $N = 10^7$, 10% stores **(d)** Average allocated objects

(e) $N = 10$, 1% pushes/pops **(f)** $N = 10$, 10% pushes/pops **(g)** $N = 10$, 50% pushes/pops **(h)** Average allocated objects

Figure 6. Benchmark results comparing reference-counted pointer implementations. Figures 6a–6d compare raw load/store throughput and memory usage. Figures 6e–6h compare throughput and memory when used to implement a concurrent stack.

Implementations that do not support snapshots perform a **load** instead. We maintain an array of $N = 10$ concurrent stacks, each padded to its own cache line. Every stack initially has 20 elements. Threads perform a find on a uniformly random stack with probability p_f, or, with probability $1 - p_f$, a pop from a uniformly random stack followed by a push of the popped value onto another uniformly random stack (possibly the same one). If the popped stack was empty, nothing is pushed. We show results for $p_f = 0.01, 0.1, 0.5$, indicating read-heavy, read-mostly, and update-mostly workloads.

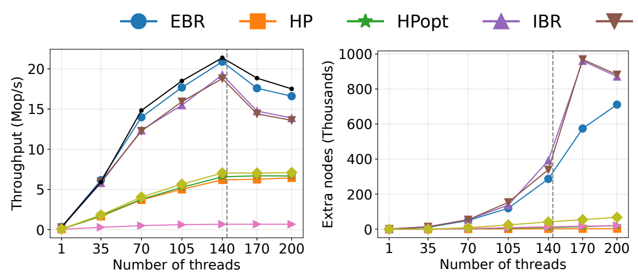
Results. The results are depicted in Figures 6e–6g. We test both our non-snapshotting algorithm (DRC) and the full version with snapshots (DRC + snapshots). The clearest takeaway from these experiments is that snapshotting provides tremendous benefits, particularly on read-heavy workloads. Recall that OrcGC also employs a technique similar to our snapshots, which is why it, too, outperforms the other methods. Our non-snapshotting algorithm outperforms the remaining implementations, but by a smaller margin. At 128 threads, the snapshotting algorithm improves the throughput of the read-heavy workload by 1.7x compared to OrcGC, 5x compared to the non-snapshotting algorithm, 7x compared to our optimized implementation of Herlihy's algorithm, and 16x compared to Folly. On the update-mostly workload, our algorithm still outperforms the other implementations by at least 2x, due to finds not creating contention with updates.

Lastly, in Figure 6h, we show the memory usage in terms of the number of allocated nodes with respect to the number of live nodes (the total number of nodes in all of the stacks). The number of threads in this experiment was fixed at 128. As the number of live nodes increases, the number of allocated nodes is asymptotic to the number of live nodes, indicating that the memory overhead of the schemes is indeed additive, and not proportional to the number of live nodes.

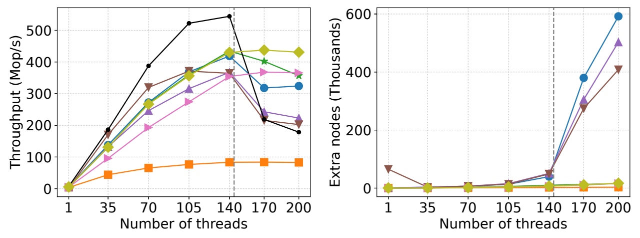
7.2 Comparison to Manual SMR Techniques

We compare our reference-counting technique with four different manual SMR techniques, hazard pointers (HP) [40], hazard eras (HE) [47], two-global-epoch IBR [50] and epoch-based reclamaion (EBR) [22] applied to three different lock-free data structures: Harris-Michael list [26, 39], Michael hash table [39], and Natarajan-Mittal tree [44]. When applying our technique, we use snapshot pointers for the short-lived references that processes hold onto while traversing the data structure. In the Natarajan-Mittal tree, each process holds onto at most five snapshot pointers at a time, and in the list and hash table, each process holds onto at most three. To measure the benefits of snapshot pointers, we also benchmark our implementation without them, using only `rc_ptr`, which increments reference counts eagerly. As a baseline, we also measure the performance of each data structure when no memory reclamation is performed, meaning that nodes are never freed at all.

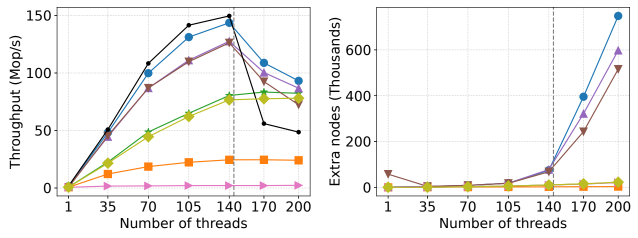
Benchmarks. We leveraged the IBR benchmark suite [50] which contains implementations of HP, HE, IBR and EBR applied to the three data structures. In Section 8, we identify some bugs in the IBR benchmarking suite related to incorrectly applying these memory reclamation techniques. For our benchmarks, we fixed all of them except the last one, which only applies to the Natajaran-Mittal tree when used



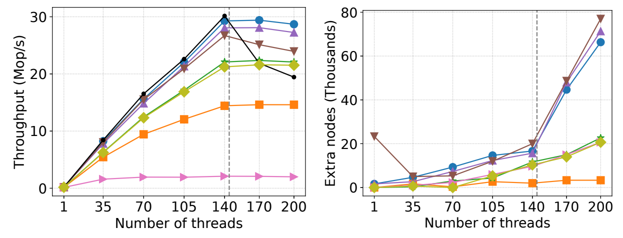
(a) List. N=1000, updates=10%. Throughput (L), Memory (R)



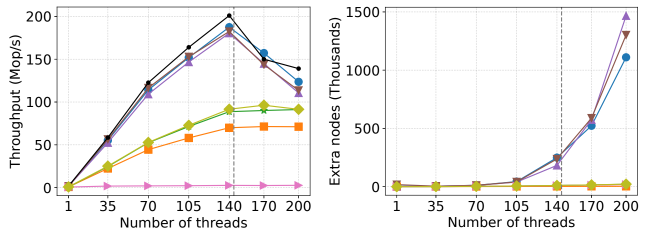
(b) Hashtable. N=100K, updates=10%. Throughput (L), Memory (R)



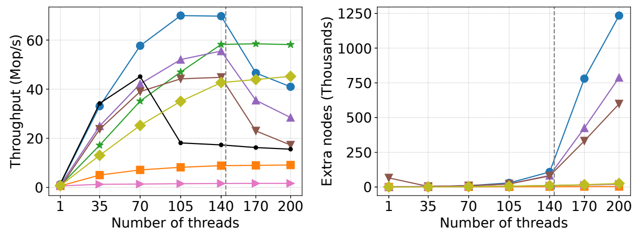
(c) BST. N=100K, updates=10%. Throughput (L), Memory (R)



(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 7. Benchmark results comparing deferred reference counting with manual SMR techniques. Figure 7a shows results for a Harris-Michael list, Figure 7b for a Michael hash table, and Figures 7c–7f for various workloads on a Natarajan-Mittal tree.

with HP, HE, or IBR. Fixing this would required significant modifications to the data structure, and would only slow down the performance of these SMR techniques due to the extra restarts. Therefore, these experiments depict a generous estimate of how HP, HE, and IBR would perform when correctly applied to the Natarajan-Mittal tree. We also optimized the throughput of the HP implementation by reducing the number of times the announcement array is scanned. While this significantly improves throughput in some cases, it does so at the cost of a slight increase in memory.

For each data structure, we tried various sizes and update frequencies. For example in Figure 7c, we initialized the BST with 100K keys and each process performed 10% update operations (half insert, and half delete) using a key chosen uniformly randomly from the range $[0, 200K)$. The remaining 90% of operations were lookups. For the hash table experiments, we initialized the number of buckets so that the average load factor is 1.

Results. The results of these experiments are shown in Figure 7. In each pair of graphs, throughput is plotted on the left and space overhead is plotted on the right. Space overhead is measured by calculating the number of nodes that were removed from the data structure and not yet freed.

We found that using snapshot pointers is crucial for getting reference counting to scale on many of these lock-free data structures. It improves performance by up to 40× in Figure 7e and a minimum of 1.2× in Figure 7b. This optimization is what allows automatic reference counting to be competitive with manual SMR. Overall, our reference-counting technique tends to closely match the throughput and space usage of optimized hazard pointers (HPopt). One exception is in the update-heavy workload of Figure 7f where the cost of reference-count increments and decrements during updates causes a 38% performance overhead.

We found that our technique generally performs very well on hash table workloads (one of which is shown in Figure 7b) because on average, each lookup acquires one snapshot pointer, which is about as cheap as acquiring a HP or announcing an epoch during EBR. In this workload, for thread counts of 140 or higher, our technique actually outperforms all of the manual SMR techniques.

In general, our technique does not seem to be slowed down by over-subscription whereas HE, IBR, and EBR are often severely impacted. The memory usage of HE, IBR and EBR spike upwards during over-subscription because one stalled thread can prevent a lot of nodes from being collected.

In most cases, our throughput is 1.2-2.5× slower than EBR, but we experience 3-61× less memory overhead. The only exception is the linked list workload in Figure 7a, where we are up to 5.1× slower than EBR, but in exchange, we waste 210× less memory on 200 threads, and 6.5× less on 140 threads. Our memory usage is always within a factor 3 of HPopt, which indicates that having P^2 delayed decrements usually translates to holding onto about P^2 extra nodes for these data structures.

These results show that automatic reference counting, when implemented efficiently, can perform competitively with manual memory reclamation techniques. Furthermore, whenever manual techniques outperform our algorithm, our algorithm uses significantly less space.

8 Usability Difficulties of Manual SMR

Applying manual memory reclamation techniques to concurrent data structures can be non-trivial and difficult to get right, even for expert users, often leading to bugs that are not caught for a long time. In this section, we will discuss some recurring bugs that we have discovered in research code while working on memory reclamation. We emphasize that these bugs exist in the applications of these memory reclamation techniques, not in the techniques themselves.

Correctly Calling Retire. While some manual techniques are more difficult to apply than others, one thing that they have in common is the need for the user to determine when an object is no longer reachable from the shared data structure and explicitly call `retire` on this object. This can be challenging in a concurrent setting. For example, if there are two pointers in shared memory to an object and the pointers are concurrently cleared by two different processes, it is not clear which process should be the one to call `retire` or how the process even learns that the other pointer has been cleared. Another issue is that it is easy to forget to retire a node, especially when there are concurrent operations involved. For example, in the Natarajan and Mittal tree [44], the delete operation marks an internal node for deletion, and then calls a cleanup procedure which performs a CAS removing the node. A common mistake is to only retire a single internal node after this CAS. However, in the presence of concurrent deletes, this CAS can potentially remove a long chain of marked nodes, all of which need to be retired. This exact memory leak can be found in the artifacts of several papers [14, 15, 23, 45, 50], some of which are specifically about concurrent memory reclamation.

Restarts. An important detail that is sometimes missed is that many of these manual reclamation techniques (HP, HE, WHE, IBR) often require significant changes to the original concurrent data structure in order to be applicable. To protect an object using one of these techniques, a process has to first announce either a pointer to the object or an epoch, and then verify that the object has not been retired. If there is no way to verify this, then the object could have already been freed before the announcement happened, so it is not safe to access. In this case, some sort of fall back plan is needed and this usually involves aborting and restarting the operation. The IBR and WHE benchmark suites applied HP, HE, WHE, and IBR to the original Natarajan and Mittal tree without additional restarting, which leads to unsafe memory accesses.

We conclude this section by reiterating our premise that manual SMR techniques are easier to misuse than automatic ones, so most users should prefer to rely on automatic memory reclamation.

9 Discussion and Conclusion

In this work, we designed, analyzed, and evaluated a new technique for automatic memory reclamation for non-garbage-collected languages based on a novel combination of SMR techniques and reference counting. We showed that our technique is theoretically more efficient than existing methods, and demonstrated that it is also practical by implementing it as a library for C++ and comparing it to a range of existing schemes, both automatic and manual. Our method performs strongly against existing automatic techniques, improving performance by up to a factor of 16 when compared against state-of-the-art open source and commercial reference-counted pointers. Against manual SMR techniques, it remains competitive, achieving similar throughput and memory consumption to hazard pointers, and usually performing within a factor of 1.2-2.5× against the fastest manual techniques that consume unbounded amounts of memory.

A limitation of our automatic memory reclamation technique that we inherit from reference counting is that an object cannot be collected while it is part of a reference cycle. There are many approaches to deal with cycles (e.g. weak pointers) and it would be interesting to explore incorporating those into our technique.

Lastly, although we have applied the acquire-retire framework specifically to reference counting, we believe that the framework on its own is also important. By considering resources in general, and supporting multiple retires on the same resource, our interface generalizes previous ones, which focused mostly on memory-reclamation. We believe it will find a range of applications beyond reference counting and possibly even beyond memory reclamation.

Acknowledgments

We thank the anonymous referees for their comments and suggestions. Thanks to Andreia Correia, Pedro Ramalhete, and Pascal Felber for pointing out that copying snapshot pointers is non-trivial. This research was supported by NSF grants CCF-1901381, CCF-1910030, and CCF-1919223, and an NSERC PGSD Scholarship.

References

[1] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. 2014. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *European Conference on Computer Systems (EUROSYS)*. https://doi.org/10.1145/2592798.2592808

[2] Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. 2017. Forkscan: Conservative memory reclamation for modern operating systems. In *European Conference on Computer Systems (EUROSYS)*. https://doi.org/10.1145/3064176.3064214

[3] Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. 2018. Threadscan: Automatic and scalable memory reclamation. *ACM Trans. Parallel Comput.* 4, 4 (2018), 1–18. https://doi.org/10.1145/3201897

[4] David F Bacon, Clement R Attanasio, Han B Lee, VT Rajan, and Stephen Smith. 2001. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/378795.378819

[5] Henry G Baker. 1994. Minimizing reference count updating with deferred and anchored pointers for functional data structures. *SIGPLAN Not.* 29, 9 (1994), 38–43. https://doi.org/10.1145/185009.185016

[6] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. 2016. Fast and robust memory reclamation for concurrent data structures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. https://doi.org/10.1145/2935764.2935790

[7] Stephen M Blackburn and Kathryn S McKinley. 2003. Ulterior reference counting: Fast garbage collection without a long wait. In *ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*.

[8] Guy E. Blelloch and Yuanhao Wei. 2020. Concurrent Reference Counting and Resource Management in Wait-free Constant Time. arXiv:2002.07053 [cs.DC]

[9] Guy E Blelloch and Yuanhao Wei. 2020. LL/SC and Atomic Copy: Constant Time, Space Efficient Implementations using only pointer-width CAS. In *34rd International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.DISC.2020.5

[10] Anastasia Braginsky, Alex Kogan, and Erez Petrank. 2013. Drop the anchor: lightweight memory management for non-blocking data structures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. https://doi.org/10.1145/2486159.2486184

[11] 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)*. https://doi.org/10.1145/2767386.2767436

[12] Nachshon Cohen. 2018. Every data structure deserves lock-free memory reclamation. *ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications (OOPSLA)* (2018). https://doi.org/10.1145/3276513

[13] Nachshon Cohen and Erez Petrank. 2015. Efficient memory management for lock-free data structures with optimistic access. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. https://doi.org/10.1145/2755573.2755579

[14] 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)*. https://doi.org/10.1145/3437801.3441596

[15] 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. https://doi.org/10.1145/2786763.2694359

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

[17] L Peter Deutsch and Daniel G Bobrow. 1976. An efficient, incremental, automatic garbage collector. *Commun. ACM* 19, 9 (1976), 522–526. https://doi.org/10.1145/360336.360345

[18] Dave Dice, Maurice Herlihy, and Alex Kogan. 2016. Fast Non-Intrusive Memory Reclamation for Highly-Concurrent Data Structures. *SIGPLAN Not.* 51, 11 (2016), 36–45. https://doi.org/10.1145/2926697.2926699

[19] Aleksandar Dragojević, Maurice Herlihy, Yossi Lev, and Mark Moir. 2011. On the power of hardware transactional memory to simplify memory management. In *ACM Symposium on Principles of Distributed Computing (PODC)*. https://doi.org/10.1145/1993806.1993821

[20] 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.

[21] Facebook. 2020 (accessed June 5, 2020). *Facebook Open Source Library.* https://github.com/facebook/folly.

[22] Keir Fraser. 2004. *Practical lock-freedom.* Technical Report. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-579

[23] 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)*. https://doi.org/10.1145/3385412.3386031

[24] Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. 2009. Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting. *IEEE Trans. Parallel Distrib. Syst.* 20, 8 (2009). https://doi.org/10.1109/TPDS.2008.167

[25] 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 Syst. J.* 47, 2 (2008), 221–236. https://doi.org/10.1147/sj.472.0221

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

[27] 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

[28] 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). https://doi.org/10.1145/1062247.1062249

[29] 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)*. https://doi.org/10.1007/3-540-36108-1_23

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

[31] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

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

[33] Jeehoon Kang and Jaehwang Jung. 2020. A marriage of pointer-and epoch-based reclamation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3385412.3385978

[34] Alex Kogan and Erez Petrank. 2012. A methodology for creating fast wait-free data structures. *SIGPLAN Not.* 47, 8 (2012), 141–150. https://doi.org/10.1145/2370036.2145835

[35] Hyonho Lee. 2010. Fast local-spin abortable mutual exclusion with bounded space. In *International Conference on Principles of Distributed Systems (OPODIS)*. https://doi.org/10.1007/978-3-642-17653-1_27

[36] Yossi Levanoni and Erez Petrank. 2001. An on-the-fly reference counting garbage collector for Java. In *ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*. https://doi.org/10.1145/504282.504309

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

[38] Alisdair Meredith. 2017. Revising atomic_shared_ptr for C++20. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0718r2.html.

[39] 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)*. https://doi.org/10.1145/564870.564881

[40] Maged M Michael. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (2004), 491–504. https://doi.org/10.1109/TPDS.2004.8

[41] M. M. Michael and M. L. Scott. 1995. *Correction of a memory management method for lock-free data structures*. Technical Report. Computer Science Department, University of Rochester.

[42] Maged M. Michael, Michael Wong, Paul McKenney, Geoffrey Romer, Andrew Hunter, Arthur O'Dwyer, David S. Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, and Mathias Stearn. 2019. Hazard Pointers: Proposed Interface and Wording for Concurrency TS 2. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1121r1.pdf.

[43] Maged M. Michael, Michael Wong, Paul McKenney, Geoffrey Romer, Andrew Hunter, Arthur O'Dwyer, David S. Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, and Mathias Stearn. 2017. Proposed Wording for Concurrent Data Structures: Hazard Pointer and Read-Copy-Update (RCU). http://open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0566r3.pdf.

[44] Aravind Natarajan and Neeraj Mittal. 2014. Fast concurrent lock-free binary search trees. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/2555243.2555256

[45] Ruslan Nikolaev and Binoy Ravindran. 2020. Universal Wait-Free Memory Reclamation. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/3332466.3374540

[46] Dan Plyukhin. 2015 (accessed November 5, 2019). Distributed Reference Counting for Asynchronous Shared Memory. http://rucs.ca/theory-of-computation/distributed-reference-counting-for-asynchronous-shared-memory.

[47] Pedro Ramalhete and Andreia Correia. 2017. Brief announcement: Hazard eras-non-blocking memory reclamation. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. https://doi.org/10.1145/3087556.3087588

[48] Håkan Sundell. 2005. Wait-Free Reference Counting and Memory Management. In *International Parallel and Distributed Processing Symposium (IPDPS)*. https://doi.org/10.1109/IPDPS.2005.451

[49] John D. Valois. 1995. Lock-free Linked Lists Using Compare-and-swap. In *ACM Symposium on Principles of Distributed Computing (PODC)*. https://doi.org/10.1145/224964.224988

[50] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H Alan Beadle, and Michael L Scott. 2018. Interval-based memory reclamation. *SIGPLAN Not.* 53, 1 (2018), 1–13. https://doi.org/10.1145/3200691.3178488

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

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