OrcGC: Automatic Lock-Free Memory Reclamation

Andreia Correia
University of Neuchâtel
andreia.veiga@unine.ch

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Pascal Felber
University of Neuchâtel
pascal.felber@unine.ch

Abstract

Dynamic lock-free data structures require a memory reclamation scheme with a similar progress. Until today, lock-free schemes are applied to data structures on a case-by-case basis, often with algorithm modifications to the data structure.

In this paper we introduce two new lock-free reclamation schemes, one manual and the other automatic with user annotated types. The manual reclamation scheme, named *pass-the-pointer* (PTP), has lock-free progress and a bound on the number of unreclaimed objects that is linear with the number of threads.

The automatic lock-free memory reclamation scheme, which we named OrcGC, uses PTP and object reference counting to automatically detect when to protect and when to de-allocate an object. OrcGC has a linear bound on memory usage and can be used with any allocator. We propose a new methodology that utilizes OrcGC to provide lock-free memory reclamation to a data structure.

We conducted a performance evaluation on two machines, an Intel and an AMD, applying PTP and OrcGC to several lock-free data structures, providing lock-free memory reclamation where before there was none. On the Intel machine we saw no significant performance impact, while on AMD we observed a worst-case performance drop below 50%.

CCS Concepts • **Theory of computation** → **Distributed computing models**; • **Computing methodologies** → *Concurrent computing methodologies.*

Keywords memory reclamation, automatic memory management, lock-free

1 Introduction

Any application that dynamically allocates memory will eventually be confronted with the problem of when to return this memory to the allocator. For multi-threaded applications, it is often difficult to determine when exactly a memory block becomes permanently unreachable and therefore, be safely de-allocated, given that other threads may still hold references to this memory block. Furthermore, an object's destructor method cannot be called until the object becomes unreachable from other threads. An object is said to be *unreachable* when it cannot be accessed through a global or local reference. A global reference, also referred to as root pointer, is a reference stored in a global variable, accessible by more than one thread, while a local reference is only accessible by the thread that has created it.

Most automatic garbage collectors (GC) use lock-based approaches to scan through the allocated memory, identifying which memory blocks are reachable and which ones are not and can therefore be freed. Lock-free data structure algorithms cannot utilize lock-based GCs if they wish to keep the lock-free progress they were designed for. Until now, for most data structures, lock-free *manual* memory reclamation schemes [5, 11, 14, 19, 25] have been the only option. Nevertheless, incorporating these lock-free memory reclamation schemes on lock-free data structure algorithms remains a complex task, shun by most data structure designers. Moreover, several lock-free data structure algorithms are incompatible with these schemes. This is particularly troublesome for system allocated memory, given that an inadvertent access to a block of memory that has been returned to the system will cause a segmentation fault. Custom allocators avoid this problem because all allocated memory resides in the application's address space and accessing it does not generate a segmentation fault.

For a memory reclamation scheme to be lock-free, it must guarantee that all its operations are lock-free and only a finite amount of memory blocks, that are no longer reachable from the application's root pointers, can remain allocated. In other words, a lock-free scheme must guarantee that the number of retired objects is bounded. If we assume a system where some reclamation is to be done during the application's lifetime, then the memory reclamation operation must not be starved by any other operation, otherwise a memory reclamation scheme would not be needed.

In this paper we propose a novel automatic lock-free memory reclamation scheme that is compatible with any memory allocator. Our automatic memory reclamation, named OrcGC, combines per-object reference counting with a pointer-based reclamation scheme. Each dynamically allocated object (for example, a node in the data structure) contains an extra field, named `_orc`, which counts how many hard links currently exist to that object. A *hard link* is a reference stored

PPoPP '21, February 27–March 3, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8294-6/21/02...$15.00
https://doi.org/10.1145/3437801.3441596

in another object. The pointer-based reclamation scheme is used to track *local references* to objects shared among threads. These local references exist after reading a hard link, and they are kept in registers and on each thread's stack. By definition, a hard link is shared among all threads, while a local reference can only be accessed by the thread that has created it. An object is said to be *unreachable* when there are no hard links, no global references and no local references to it. This approach of combining reference counting with a pointer-based reclamation was previously pursued by Gidenstam et al. [11]. However, their solution requires explicit manual calls by the user to protect or retire an object. OrcGC automatically detects when an object is no longer reachable from other objects, and retires it. Also, an object can be momentarily taken out of the data structure and reinserted without actually being deleted.

As shown in the example for the Michael-Scott queue [20] in Algorithm 1, when deploying OrcGC, the user (i.e., the data structure implementer) must do type annotation (shown in blue), indicating what are the references that need to be tracked. Atomic hard links are annotated with the `orc_atomic` type, and local references with the `orc_ptr` type. The `orc_atomic` type implements all atomic instructions that modify a shared hard link. These instructions can potentially change the number of hard links to an object. The reference to the object is automatically published in an array of *hazardous pointers*, before the object reference counter, `_orc`, is incremented. The `orc_ptr` type contains metadata that identifies where the local reference is published in the array of hazardous pointers. The purpose of the `orc_ptr` type is to ensure that while an `orc_ptr` instance is alive, the user object which it references will not be deallocated. Most of the existing pointer-based reclamation schemes [14, 19, 24, 25] can be used by OrcGC to protect the local references of type `orc_ptr`.

Pointer-based reclamation schemes can provide a strong bound on memory usage. Existing implementations such as *hazard-pointers* (HP) [19] and *pass-the-buck* (PTB) [14] guarantee at most $t \times (H + 1)$ objects in the retired list of each thread, and therefore, at most $t^2 \times (H+1)$ retired objects waiting to be deleted, where t is the number of concurrent threads and H is the maximum number of hazardous pointers required by a data structure algorithm. To further improve the unreclaimed memory bound, we have developed a new lock-free pointer-based memory reclamation scheme that guarantees at most $H + 1$ objects waiting to be reclaimed by each thread and, consequently, at most $t \times (H + 1)$ objects waiting to be deleted. Our new pointer-base scheme, called *pass-the-pointer* (PTP), is the first to provide a linear bound on the number of objects to be deleted.

In PTP, each thread has two arrays whose size corresponds to the maximum amount of hazardous pointers H required by the lock-free algorithm. The protection of each hazardous pointer is done the same way as on HP or PTB. Although

Algorithm 1 — Michael-Scott queue with OrcGC

```
template<typename T> class MSQueueOrcGC {
  struct Node : orc_base {
    T* item;
    orc_atomic<Node*> next {nullptr};
    Node(T* it) : item{it} { }
  };

  orc_atomic<Node*> head;
  orc_atomic<Node*> tail;

  MSQueueOrcGC() {
    head = make_orc<Node>(nullptr);
    tail = head;
  }

  void enqueue(T* item) {
    orc_ptr<Node*> newNode = make_orc<Node>(item);
    while (true) {
      orc_ptr<Node*> ltail = tail.load();
      orc_ptr<Node*> lnext = ltail−>next.load();
      if (lnext == nullptr) {
        if (ltail−>next.cas(nullptr, newNode)) {
          tail.cas(ltail, newNode);
          return;
        }
      } else {
        tail.cas(ltail, lnext);
      }
    }
  }

  T* dequeue() {
    orc_ptr<Node*> node = head.load();
    while (node != tail.load()) {
      orc_ptr<Node*> lnext = node−>next.load();
      if (head.cas(node, lnext)) return lnext−>item;
      node = head.load();
    }
    return nullptr;
  }
};
```

HP, PTB and PTP share the same algorithm for protecting a pointer, they have distinct approaches for retiring an object. To retire an object in PTP, all published hazardous pointers are scanned and, if a pointer to the object is found, it will *pass* the responsibility to free that object to the thread protecting it. This operation of passing the responsibility to de-allocate the object's memory to the thread that is still using the object may be done at most $t \times H$ times, and the last thread that is using the object will be the one to actually de-allocate it. This approach of having a shared retired list among all threads was previously proposed in pass-the-buck [14]. The PTB `retire()` method scans the retired objects while gathering them to be freed at the end, leading to a quadratic bound on unreclaimed objects.

As main contributions of this paper, we introduce:

- a new manual memory reclamation scheme, pass-the-pointer, with linear bound on unreclaimed memory;

- a new automatic memory reclamation scheme, OrcGC, which only requires annotation of shared objects and local references to shared objects; and
- a methodology to apply the OrcGC scheme to any data structure, using type annotation.

Both our memory reclamation schemes are compatible with any allocator, require no compiler modifications, and are implemented as a single C++ header.

The rest of the paper is organized as follows. We first discuss related work in §2. We then present our manual memory reclamation PTP in §3. In §4 we describe our automatic memory reclamation scheme OrcGC. We perform an evaluation of OrcGC in §5. Finally we conclude in §6.

2 Related Work

On a multi-threaded application, the *memory reclamation scheme* is the component responsible for de-allocating objects in a safe manner. It executes two distinct operations, named *protect* and *retire*. Before de-referencing a pointer, the object must be protected from de-allocation using the *protect* functionality. Retiring an object implies determining that there are no other threads holding pointers to the object.

In blocking concurrent data structures, the reclamation scheme can itself be blocking, however, using a blocking reclamation scheme on a lock-free data structure negates the lock-free progress, given that any call to a data structure method that executes a retire operation will block.

Memory reclamation schemes can be split into two distinct families: *manual* schemes and *automatic* schemes.

Incorporating a manual scheme into a lock-free data structure implies protecting the object (node) before accessing it, and retiring the object when it is no longer reachable. In lock-free schemes [5, 7, 11, 14, 19, 24, 25], this implies calling a protection method for a pointer so as to read it safely from a given memory address, `get_protected(addr)`, before de-referencing the pointer to access variables in the object, and calling the `retire(ptr)` when the object is no longer reachable from the root pointers of the data structure. Determining *where* to call these two methods on a lock-free data structure, can be a challenging task, requiring significant expertise.

Deploying an automatic scheme into a lock-free data structure implies little to no intervention from the data structure's developer. An object will be automatically retired when it is not longer reachable from a root pointer of the data structure, nor accessible from the local pointers of any thread. Garbage collection schemes are an example of automatic reclamation. Another example are C++'s `std::unique_ptr` and `std::shared_ptr` which work only for sequential code. These two impose type annotation, but require no explicit pointer protection nor call to retire.

Another important distinction between reclamation schemes is whether or not a scheme supports the system allocator. When an object is de-allocated with the system allocator, the page where the object is located may be returned to the operating system and, subsequently, to another process. Any access to that page, even for reading, will trigger a segmentation fault and consequent application crash. Certain schemes [7, 8] rely on specialized allocators to function correctly. Those allocators will not return pages to the operating system and will enforce type-safe allocation, so that re-used objects maintain their structure. The PTP and OrcGC algorithms we describe in §3.1 and §4.1 have no such limitations and can be used safely with any allocator.

Atomic-reference-counters [29] was the first manual lock-free reclamation scheme. It utilizes one word in each dynamically allocated object to keep track of how many references currently exist to the object, whether they are local or global. It does not support the system allocator.

Epoch-based-reclamation (EBR) [10, 13] and RCU [3] are *quiescent* based memory reclamation schemes where a single timestamp (epoch) protects a large number of objects. In EBR/RCU algorithms, the protect functionality is typically wait-free and the retire is always blocking, meaning that EBR schemes are not lock-free. They are however, applicable to most data structures (though not automatically).

Pass-the-buck (PTB) [14] is a scheme with lock-free progress for protect and wait-free for retire. PTB uses a double-word-compare-and-swap (DWCAS) atomic instruction. During a retire, each thread creates a list of retired objects which may be proportional to the number of threads, implying that if all threads are attempting a retire operation, the total number of objects retired but not yet de-allocated is $O(Ht^2)$.

Hazard-pointers (HP) [19] is similar to PTB in progress but does not require the usage of DWCAS, using solely atomic loads and stores. Its bound is also $O(Ht^2)$.

Hazard-eras (HE) [25] is a scheme with lock-free protect and wait-free retire. A recent alternative has been proposed (WFE) [24] where both the protect and retire are wait-free. The bound on memory usage for HE and WFE is proportional to the number of *live* objects at any given time $\#L$, times the number of threads squared, $O(\#LHt^2)$. HE and WFE combine ideas from pointer-based reclamation and EBR to provide a protect function that reduces the number of sequentially consistent stores called in the protect function, thus improving performance at the cost of a significantly larger bound on memory usage. A dynamic object that is used in HE or WFE must contain two words that define an interval of when the object was visible to other threads (it was *alive*).

Drop-the-anchor (DTA) [5] extends HP and also improves the performance of the protect functionality. Applying DTA to a lock-free data structure may require non-trivial modifications to the data structure's algorithm.

Interval-based-reclamation (IBR) uses the same protect algorithm as HE. IBR has several variants with only one

Scheme	Progress	Supports any allocator	Bound on memory usage	Atomic instructions	Extra words per object
Atomic-reference-counters [29]	lock-free	no	$O(t)$	FAA	1
EBR/RCU [3, 10, 13]	blocking	yes	∞	loads/stores	0
Pass-the-buck (PTB) [14]	lock-free	yes	$O(Ht^2)$	DWCAS	0
Hazard-pointers (HP) [19]	lock-free	yes	$O(Ht^2)$	loads/stores	0
Beware-and-cleanup (B&C) [11]	lock-free	yes	$O(Ht^2)$	CAS + FAA	2
Drop-the-anchor (DTA) [5]	lock-free	yes	$O(Ht^2)$	loads/stores	0
Free-access [7]	lock-free	no	n/a	loads/stores	1
Automatic-optimistic-access [8]	lock-free	no	n/a	loads/stores	1
Hazard-eras (HE) [25]	lock-free	yes	$O(\#LHt^2)$	FAA	2
2GEIBR [30]	lock-free	yes	$O(\#LHt^2)$	FAA	2
Hyaline-S/1S [23]	lock-free	yes	$O(\#LHt^2)$	FAA	2
Wait-free-eras (WFE) [24]	wait-free	yes	$O(\#LHt^2)$	DWCAS + FAA	2
Pass-the-pointer (PTP)	lock-free	yes	$O(Ht)$	CAS or exchange	0
OrcGC	lock-free	yes	$O(Ht)$	CAS + exchange	1

Table 1. Comparison of reclamation schemes. FreeAccess and OrcGC are automatic schemes, all others are manual.

of them (2GEIBR) providing lock-free and bounded memory usage, and only when applied to data structures in the normalized formulation by Timnat-Petrank [28].

Hyaline-S and Hyaline-1S [23] use a protect algorithm similar to HE but a significantly different retire method. Like 2GEIBR, when applied to data structures in the Timnat-Petrank formulation, these reclamation schemes provide a higher memory bound when compared with HE. The higher memory bound is due to protecting a range of eras comprised between the smallest and the highest protected eras.

Beware-and-cleanup (B&C) [11] is a manual memory reclamation that combines HP with atomic-reference-counters. Each tracked object (node) in the data structure contains two words, one for `mm_ref` a counter of hard link references and another word for `mm_trace` and `mm_del`, used internally by the B&C scheme. B&C utilizes HP and therefore has a quadratic bound on memory usage.

Automatic-optimistic-access (AOA) [8] is a lock-free reclamation scheme that requires a customized allocator and the data structure methods to be restartable, for example, be written in a normalized form [28].

Up until now, Free Access [7] was the only known automatic reclamation scheme to provide lock-free progress for pointer protection and object retire. Previous GC either have stop-the-world pauses, or they guarantee lock-free progress for the *protect* functionality or for the data structure itself, at the cost of starving *retire* operations, thus having an unbounded number of unreclaimed objects. Free Access addresses this issue by having a retry mechanism for the operations on the data structure, such that the retire operation does not have to retry and therefore is not starved. Free Access uses some of the ideas of AOA, requiring data structures to be able to be transformed in the Timnat-Petrank normalized form or some other restartable form and, in addition, compiler modifications are needed to deploy this scheme. There are limitations to the code executed inside the data structure operations, as this code can be re-executed multiple times. The code of the operations must not modify any global variables, thread-local variables or stack-allocated variables defined outside of the retry loop. Also, it must not contain any heap allocation or de-allocation and it must not execute I/O calls, including writes to files, handling of network packets or syscalls. Furthermore, any data structure algorithm that executes a `fetch_add()` or an `exchange()` operation, is not supported by FreeAccess nor AOA, seen as such data structures are not compatible with the a restartable form in [28].

Table 1 displays important properties of several lock-free reclamation schemes, such as the overall progress of the scheme (first column), whether it supports any allocator or requires a customized allocator (second column), which atomic instructions it utilizes (third column), and how many words it needs to reserve in each dynamically created object (fourth column).

Limitations of existing schemes. Previous work has shown that most lock-free manual reclamation schemes are not generally applicable [7]. There are three main obstacles preventing the majority of these schemes from being deployed to a larger set of data structures.

The first obstacle concerns data structures where nodes have multiple incoming hard links which may be unlinked in different orders at run-time, depending on specific operation interleaving. This prevents the call to `retire()` from being placed in a particular place in the code, because the retire method can be called on an object only *after* the object becomes unreachable. If there is no simple way of determining when the object becomes unreachable, then there is no way to call `retire()` safely. An example of a data structure with this characteristic is the Kogan-Petrank MPMC wait-free queue [17]. Neither HP nor the other lock-free reclamation

Obstacle I

schemes shown in Table 1 can be deployed with the original algorithm for this queue. OrcGC or Free Access are the only schemes that can provide safe memory reclamation to this data structure.

obstacle II

The second obstacle pertains to a class of data structures where the pointers of a node must not be modified even after being retired, so as to allow safe traversal of these objects while maintaining correctness or progress guarantees. Examples of these are, the linked list with wait-free lookups by Herlihy-Shavit [15], whose wait-free progress implies such a guarantee, and the original lock-free linked list by Harris [12], whose correctness is lost when integrated with most reclamation schemes. The complexity of applying a memory reclamation to this last data structure has been detailed in previous work [7]. B&C, Free Access and OrcGC are capable of solving this issue and, therefore, can be safely used with this class of data structures.

obstacle III

The third obstacle occurs for data structures where one thread may temporarily unlink a node from the data structure while another thread subsequently re-inserts it. One example of a data structure with this behavior is the lock-free skip list by Herlihy and Shavit [15], in which a half-inserted node may be removed by another thread, becoming temporarily unreachable from any of the data structure's global references, and later become reachable again by the insertion operation, upon its completion. Although in this case the node is logically marked when re-inserted, this behavior creates a problem for manual techniques, given that a call to `retire()` is made after the temporarily unlinking of the node. However, this retired node will be made reachable by the insertion operation. Subsequently, threads executing a `contains()` will potentially cause a heap-use-after-free fault. Out of the lock-free methods shown in Table 1, OrcGC and Free Access are the only ones capable of correctly dealing with this *re-insertion* behavior.

Many other reclamation schemes exist in the literature with different tradeoffs in performance and memory usage [1, 2, 4, 6, 9, 16]. In this paper, we focus our attention on the ones that provide lock-free progress using only atomic instructions, without relying on OS infrastructure (such as signals) nor on hardware functionality (such as hardware transactional memory), seen as these functionalities are not guaranteed to be lock-free in practice.

3 Manual Reclamation

All existing manual memory reclamation schemes with lock-free progress support at least two operations, namely, *protect* to prevent an object from being de-allocated while it is being de-referenced, and *retire* to delete an object when the object is no longer reachable. The delete of an object implies a call to its destructor method and de-allocation of the memory block where the object is located. Although not needed for correctness, we define a third API as *clear*, which clears the protection for a given pointer.

We now introduce a novel algorithm for manual memory reclamation with lock-free progress and a linear bound of $O(Ht)$ of unreachable objects.

3.1 Pass-the-pointer

In PTP, the protection procedure is similar to HP and PTB, where each published hazardous pointer protects a single object. However, reclaiming an object is done in a different way. PTP's approach for retiring an object is to pass the responsibility of its de-allocation to the thread that is still using it. The `retire()` procedure executes `handoverOrDelete()` which scans all the published `hp` in search of a matching pointer. If it finds an hazardous pointer that is protecting the pointer p_1 from being deleted (line 27), it atomically replaces whatever pointer p_2 is in the corresponding `handover` entry (line 28). In case there was another pointer p_2 published in that same position, it is guaranteed that the corresponding hazardous pointer position is no longer protecting p_2. From this point on, the thread is now responsible to continue the search of a matching protected pointer, but this time for pointer p_2. Notice that each object may be retired a single time and that a pointer is extracted from an entry in `handover` only to be placed further down another entry, or to be deleted. This allows `handoverOrDelete()` to continuously *push* pointers further down in the `handovers` array until the end is reached (the last `hp` index of the last thread). To prevent objects from being left on the `handover` array, the method *clear* can be called when an hazardous pointer is no longer being used. The thread clearing the hazardous pointer will take the corresponding handover pointer and call `handoverOrDelete()`. Notice that without lines 16 - 19 of Algorithm 2, objects may be left indefinitely in a given slot of the `handovers` array if the thread never calls `get_protected()` or `clear()` or `retire()` again, however, this does not affect correctness nor the memory usage bound.

The maximum amount of objects in the `handover` array is $t \times H$. Each thread may have at most one single pointer to an object that is not in the handovers array. Two scenarios can occur to the object: either it will be deleted at the end of the scan of published pointers because no other thread can de-reference it; or it will `handover` its de-allocation, replacing the position of another object pointer (or `nullptr`). At any given time, the total amount of retired but not yet deleted objects is at most $t \times (H + 1)$, thus imposing a linear bound. PTP does not utilize a thread-local list of retired objects.

In our implementation, the hazardous pointers are published on a bi-dimensional array of atomic pointers, indexed by thread id and hazardous pointer index (`hp[tid][idx]`), while the handovers are placed on separate bi-dimensional array, so as to reduce contention and avoid false-sharing. An entry in the `hp[tid][idx]` can be written only by the thread whose id is `tid`, but may be read by any other thread during a

Algorithm 2 — Pass The Pointer

```
1 std::atomic<T*> hp[maxThreads][maxHPs];   // array size is [t][H]
2 std::atomic<T*> handovers[maxThreads][maxHPs];

4 T* get_protected(std::atomic<T*>* addr, int idx) {
5   T *pub, *ptr = nullptr;
6   while ((pub = addr->load()) != ptr) {
7     hp[tid][idx].store(pub);                 // or .exchange(pub)
8     ptr = pub;
9   }
10  return pub;
11 }

13 void clear(int idx) {
14   hp[tid][idx].store(nullptr, memory_order_release);
15   // The following lines are optional
16   if (handovers[tid][idx].load() != nullptr) {
17     T* ptr = handovers[tid][idx].exchange(nullptr);
18     if (ptr != nullptr) handoverOrDelete(ptr, tid);
19   }
20 }

22 void retire(T* ptr) { handoverOrDelete(ptr, 0); }

24 void handoverOrDelete(T* ptr, int start) {
25   for (int it = start; it < maxThreads; it++) {
26     for (int idx = 0; idx < maxHPs; ) {
27       if (hp[it][idx].load() == ptr) {
28         ptr = handovers[it][idx].exchange(ptr);
29         if (ptr == nullptr) return;
30         // Check it is not the new ptr
31         if (hp[it][idx].load() == ptr) continue;
32       }
33       idx++;
34     }
35   }
36   delete ptr;                   // destroy and de-allocate the object
37 }
```

retire. Each position in the handovers bi-dimensional array is logically associated with an hazardous pointer entry. An entry in the `handovers[tid][idx]` can be written or read by any thread and, therefore, this is done with an atomic `exchange()`.

PTP has the same constraints as HP, PTB, HE and other manual lock-free reclamation schemes: a call to `retire(ptr)` implies that `ptr` is a reference to an object that is no longer reachable from any other object or global reference.

In PTP the protect function (lines 4 - 11 of Algorithm 2) is similar to HP and PTB, with a loop that retries until the pointer published in `hp[tid][idx]` is the same as the one read from `addr`. The hazardous pointer will only fail to be published if a data structure operation makes progress, implying lock-free progress for the protect function.

4 Automatic Reclamation

A memory reclamation scheme is said to provide *automatic memory reclamation* if and only if it automatically deletes an object when the object becomes unreachable, and it protects any pointer to an object before de-referencing the object. In other words, automatic reclamation schemes do not need to have explicit calls to `retire()` nor `protect()` in the user code. C++ `unique_ptr` and `shared_ptr` are two examples of automatic object disposal when the pointer goes out of scope, which can be used in sequential code. Garbage collector (GC) algorithms, such as mark-and-sweep, are another example.

Prior work has claimed to have GC algorithms that provide lock-free progress. However, for a scheme to be lock-free it must fulfill three vital requirements: protecting an object must be done with lock-free progress, determining when it is safe to de-allocate an object must be done with lock-free progress, and the memory usage must be bounded. So far, no general-purpose GC presented in the literature has simultaneously provided these three characteristics.

We will now describe OrcGC, a memory reclamation scheme that achieves these three requirements for all lock-free acyclic algorithms and is compatible with any allocator. It can also be used in cyclic algorithms as long as unreachable objects do not form cycles between themselves, or in other words, these cycles must be broken before becoming unreachable. In addition, these acyclic algorithms must guarantee that, objects can not form chains of unbounded size, as soon as these objects become unreachable from the algorithms global references. Concretely, an unreachable object should not be left with an hard link to a reachable object.

4.1 OrcGC

OrcGC is able to automatically detect if an object is no longer reachable by keeping a counter per object, `_orc`, of how many hard links from other objects exist. As soon as the number of hard links to the object reaches zero, this object is potentially unreachable. A hard link is always modified through one of three atomic instructions: store, compare-and-swap (cas) or exchange. These will internally trigger an update of the `_orc` counter. If a thread accesses the object through a load, the counter is not updated however, an hazardous pointer will be published. The `_orc` counter serves as an indicator that the object is potentially no longer reachable. In any case, even when the counter reaches zero, it does not mean the object is unreachable, because a thread may have a local reference to the object which can be used to link the object back to the data structure. OrcGC protects local references using a pointer-based scheme. The actual access to the object, be it a write or read access, can only occur after having its reference published in the thread's array of hazardous pointers. Once the reference is published and re-validated, it is then safe to access the object. For the object to be deleted there must be a point in time where, simultaneously, no hazardous pointer to the object is published *and* its `_orc` counter is zero.

In Algorithm 3 we show the classes used by the OrcGC scheme. The class `orc_base` is where the object reference counter `_orc` is kept (line 8), and all shared object types must

Algorithm 3 — The 4 OrcGC classes and make_orc()

```
1  static const uint64_t SEQ = (1ULL << 24);
2  static const uint64_t BRETIRED = (1ULL << 23);
3  static const uint64_t ORC_ZERO = (1ULL << 22);
4  #define ocnt(x) ((SEQ−1) & (x))

6  // Base type which all tracked objects must extend
7  struct orc_base {
8    std::atomic<uint64_t> _orc {ORC_ZERO};
9  };

11 // User must declare all shared atomic variables as orc_atomic
12 template<typename T> class orc_atomic : std::atomic<T>;

14 // All raw pointers T* must be replaced with orc_ptr<T*>
15 template<typename T> class orc_ptr;

17 class PassThePointerOrcGC{
18   struct TLInfo {
19     std::atomic<orc_base*> hp[maxHPs];
20     std::atomic<orc_base*> handovers[maxHPs];
21     int usedHaz[maxHPs];
22     bool retireStarted {false};
23     std::vector<orc_base*> recursiveList;
24   };
25   TLInfo tl[maxThreads];
26 };
27 PassThePointerOrcGC g_ptp {};
28 thread_local int tid;

30 // Allocating a new object must be done through make_orc<T>()
31 template <typename T, typename... Args>
32 orc_ptr<T*> make_orc(Args&&... args) {
33   T* ptr = new T(std::forward<Args>(args)...);
34   g_ptp.tl[tid].hp[0].store(ptr, memory_order_release);
35   return orc_ptr<T*>(ptr, 0);
36 }
```

extend `orc_base`. In addition, pointers to shared objects must be declared as `orc_atomic` instead of `std::atomic`. The `orc_atomic` class is in fact a `std::atomic` with all its methods overwritten, as shown in Algorithm 4.

The overwritten methods automatically protect a local reference and update the `_orc` variable if necessary, allowing all execution on a shared object to be safe, i.e., protecting the object from being deleted. For example when an atomic `compare_exchange_strong` (`cas`) is issued, the new value corresponds to a new hard link to the object it will refer to, meaning that the `_orc` of the object it will refer to, will be incremented (line 71) and the previous value corresponds to one less hard link to the previous object it referred to (line 72). In addition, any modification done to the `_orc` variable, requires the object to be protected beforehand.

► **Proposition 1.** *Before updating the* `_orc` *variable of a shared object, this object must be published on the list of hazardous pointers.*

Proof. The only two methods that modify the `_orc` counter are `incrementOrc()` and `decrementOrc()`. These methods

Algorithm 4 — OrcGC: orc_atomic class

```
37 template<typename T> class orc_atomic : std::atomic<T> {
38   void incrementOrc(T ptr) {
39     if (ptr == nullptr) return;
40     uint64_t lorc = ptr−>_orc.fetch_add(SEQ+1) + SEQ + 1;
41     if (ocnt(lorc) != ORC_ZERO) return;
42     if (ptr−>_orc.cas(lorc, lorc + BRETIRE)) g_ptp.retire(ptr);
43   }

45   void decrementOrc(T ptr) {
46     if (ptr == nullptr) return;
47     g_ptp.tl[tid].hp[0].store(ptr, memory_order_release);
48     uint64_t lorc = ptr−>_orc.fetch_add(SEQ−1) + SEQ − 1;
49     if (ocnt(lorc) != ORC_ZERO) return;
50     if (ptr−>_orc.cas(lorc, lorc + BRETIRE)) g_ptp.retire(ptr);
51   }

53   orc_atomic(T ptr) {
54     incrementOrc(ptr);
55     std::atomic<T>::store(ptr);
56   }

58   ~orc_atomic() {
59     T ptr = std::atomic<T>::load();
60     decrementOrc(ptr);
61   }

63   void store(T ptr) {
64     incrementOrc(ptr);
65     T old = std::atomic<T>::exchange(ptr);
66     decrementOrc(old);
67   }

69   bool compare_exchange_strong(T old, T ptr) {
70     if (!std::atomic<T>::cas(old,ptr)) return false;
71     incrementOrc(ptr);
72     decrementOrc(old);
73     return true;
74   }

76   orc_ptr<T> load() {
77     T ptr = g_ptp.get_protected(this, 0);
78     return orc_ptr<T> {ptr, 0};
79   }
```

are always called from `orc_atomic`'s `cas()`, `store()` or `exchange()`. The `incrementOrc()` method acts on a local reference which was previously protected through a `load()` or a `make_orc()`. On the other hand, `decrementOrc()` may be called on a pointer that has not yet been protected, for example when executing `store()`, line 66. This is the reason the pointer is saved in the hazardous pointer array, line 47, before changing the `_orc` variable in line 48. □

In our implementation, the `_orc` variable is composed of a counter on the first 22 bits, followed by a bit that indicates if the object is marked to be deleted, named `BRETIRED`, and the remaining bits are used to store a sequence that increments every time the counter is changed. Given that the counter has 22 bits, with one bit reserved for the sign, this means in our implementation an object may have at most 2^{21} hard

Algorithm 5 — OrcGC - PassThePointerOrcGC class

```
80  void clear(T ptr, const int idx, const bool reuse) {
81    if (!reuse && idx!=0) {
82       if (--tl[tid].usedHaz[idx]!=0) return;
83    }
84    if (ptr != nullptr) {
85      uint64_t lorc = ptr->_orc.load();
86      if (ocnt(lorc) == ORC_ZERO) {
87        if (ptr->_orc.cas(lorc, lorc+BRETIRED)) retire(ptr);
88      }
89    }
90  }

92  void retire(T ptr) {
93    if (tl[tid].retireStarted) {
94      tl[tid].recursiveList.push_back(ptr);
95      return;
96    }
97    tl[tid].retireStarted = true;
98    for(int i=0; ; i++) {
99      while (ptr != nullptr) {
100       auto lorc = ptr->_orc.load();
101       if (ocnt(lorc) != (BIT_RETIRE|ORC_ZERO))
102         if ((lorc = clearBitRetired(ptr))==0) break;
103       if (tryHandover(ptr)) continue;
104       uint64_t lorc2 = ptr->_orc.load();
105       if (lorc2 != lorc) {
106         if (ocnt(lorc) != (BIT_RETIRE|ORC_ZERO))
107           if (clearBitRetired(ptr)==0) break;
108         continue;
109       }
110       delete ptr;               // may add objects in recursiveList
111       break;
112     }
113     if (tl[tid].recursiveList.size() == i) break;
114     ptr = tl[tid].recursiveList[i];
115   }
116   tl[tid].recursiveList.clear();
117   tl[tid].retireStarted = false;
118 }
```

Algorithm 6 — Helper functions of PassThePointerOrcGC

```
119 int getNewIdx(int start_idx=1) {
120   for (int idx = start_idx; idx < MAX_HAZ; idx++) {
121     if (tl[tid].usedHaz[idx] != 0) continue;
122     tl[tid].usedHaz[idx]++;
123     uint64_t curMax = maxHPs.load();
124     while (curMax <= idx) { maxHPs.cas(curMax, idx+1); }
125     return idx;
126   }
127 }

129 void usingIdx(const int idx) {
130   if (idx == 0) return;
131   tl[tid].usedHaz[idx]++;
132 }

134 bool tryHandover(orc_base*& ptr) {
135   int lmaxHPs = maxHPs.load();
136   for (int it = 0; it < maxThreads; it++) {
137     for (int idx = 0; idx < lmaxHPs; idx++) {
138       if (ptr == tl[it].hp[idx].load()) {
139         ptr = tl[it].handovers[idx].exchange(ptr);
140         return true;
141       }
142     }
143   }
144   return false;
145 }

147 uint64_t clearBitRetired(orc_base* ptr) {
148   tl[tid].hp[0].store(ptr);
149   uint64_t lorc = ptr->_orc.fetch_add(-BRETIRED)-BRETIRED;
150   if (ocnt(lorc) == ORC_ZERO &&
151       ptr->_orc.cas(lorc, lorc+BRETIRED)) {
152     tl[tid].hp[0].store(nullptr);
153     return lorc + BRETIRED;
154   } else {
155     tl[tid].hp[0].store(nullptr);
156     return 0;
157   }
158 }
```

links to it. Although we know of no data structure algorithm where this limit can be reached, a higher capacity can be reserved for the counter of hard links, at the cost of reducing the number of bits for the sequence.

The value of the counter on the first 22 bits can be positive or negative because the `cas` instruction executes the `_orc` increment after making sure the instruction was successful (line 71). The object is made accessible before the `_orc` is incremented, which allows other threads to possibly remove the hard link and decrement the counter before the increment is executed. The `_orc` increment could be done before the `cas` (line 70), but it would unnecessarily increase contention on the `_orc` variable because it would have to be decremented if the `cas` failed.

The bit `BRETIRED` can be set by one of the threads that leave `_orc` value at `ORC_ZERO`, i.e., as soon as there are no hard links to the object. The thread that marks `_orc` to `BRETIRED` is responsible to call the `retire()` method. Before deleting the object, the retire method validates that the object is unreachable (line 100 to 109). The `_orc` variable may transition from `ORC_ZERO` to either a positive or negative value, because a thread holding a local reference to the object can at any time create a hard link to the object, thus making it accessible from the global references. This would mean that an object would be incorrectly retired, however, it can only be deleted when there are no hard links nor local references to the object. A previously retired object may leave the retired list if a hard link is added to the object. This happens when the thread responsible for deleting the object detects that the `_orc` counter is no longer `ORC_ZERO` and calls `clearBitRetired()`, returning a non-zero value. After clearing the bit `BRETIRED`, if the `_orc` value is once more at `ORC_ZERO`, the thread must ensure the object is again retired (line 150).

The purpose of the sequence in `_orc` is to validate if the object is unreachable at a specific point in time. During execution, the `_orc` counter of an object can be at `ORC_ZERO` multiple times, and the necessary condition of unreachability, simultaneously imposes that `_orc` counter is at `ORC_ZERO` and no references to this object are present in the list of hazardous pointers. This validation must be atomic, implying we must be able to detect if the `_orc` variable changed during the traversal of the hazardous pointers. Attaching a sequence to the counter allows for an easy and fast detection.

▸ **Lemma 1.** *An object to be deleted must have, at a point in time, no hazardous pointer to the object published and its* `_orc` *counter is* `ORC_ZERO`.

Proof. During the hazardous pointers traversal, the `_orc` variable, that is initially at `ORC_ZERO` (line 101), must not change. Otherwise, it would be possible to publish an hazardous pointer on a position previously traversed, because an hard link could be temporarily available. The sequence attached to the `_orc` variable guarantees its value has not changed during the traversal (line 105). In case the traversal did not find any hazardous pointer protecting the object, then from Proposition 1 there isn't any ongoing `incrementOrc()` or `decrementOrc()` executing and the object's `_orc` value can not be changed. If at the end of the hazardous pointers traversal, no pointer is protecting the object and the `_orc` value has not changed, this indicates that the `_orc` value will remain at `ORC_ZERO` from this point on and the object can be safely deleted. □

When an object is deemed safe to be destroyed and its memory block returned to the allocator (free'ed), the destructor of the object is first called (line 110). If the object contains pointers to other objects, annotated as `orc_atomic`, the C++ runtime will call the destructors for those `orc_atomic` instances. Then, the `orc_atomic` destructor will decrement the orc counter of the object it was pointing to (line 60 of Algorithm 4), which in turn may trigger a retire of that object (line 50) and possibly its destruction. As such, a deletion of the first node on a large list of nodes, may trigger the deletion of the entire list recursively, as long as none of the nodes on that list are being protected by other threads. To prevent program stack explosion due to the possibly large number of recursive calls, we create a temporary `recursiveList` and traverse this list one object at a time (line 113), thus ensuring the program stack depth calls `retire()` recursively at most one time.

4.1.1 Automatic pointer protection

A memory reclamation scheme must ensure that, before de-referencing a pointer to a shared object, the object cannot be deleted by a concurrent thread. For manual reclamation schemes, the pointer protection is written explicitly

Algorithm 7 — OrcGC: orc_ptr class

```
template<typename T>
class orc_ptr {
  T ptr;                                            // the raw pointer
  int idx;                  // 'ptr' is published in g_ptp.tl[tid].hp[idx]

  orc_ptr() {
    idx = g_ptp.getNewIdx();
    ptr = nullptr;
  }

  ~orc_ptr() { g_ptp.clear(ptr, idx, false); }

  orc_ptr(const orc_ptr& other) {
    idx = other.idx;
    ptr = other.ptr;
    if (idx == 0) {
      idx = g_ptp.getNewIdx();
      g_ptp.tl[tid].hp[idx].store(ptr, memory_order_release);
    } else {
      g_ptp.usingIdx(idx);
    }
  }

orc_ptr& operator=(const orc_ptr& other) {
  bool reuseIdx = ((other.idx < idx) &&
                   (g_ptp.getUsedHaz(idx) == 1));
  g_ptp.clear(ptr, idx, reuseIdx);
  if (other.idx < idx) {
    if (!reuseIdx) idx = g_ptp.getNewIdx(other.idx+1);
    g_ptp.tl[tid].hp[idx].store(other.ptr, mem_order_release);
  } else {
    g_ptp.usingIdx(other.idx);
    idx = other.idx;
  }
  ptr = other.ptr;
  return *this;
}        // for brevity, we don't display the move or copy constructors
```

on the user code. However, an automatic scheme must detect, without user intervention, if a reference is to be protected before returning it to the user code. Also, it must keep track of how many local references to the shared object are available to the user and detect when a reference no longer needs to be protected. To this end, OrcGC utilizes two classes: a class called `orc_ptr` that stores the information related to the pointer protection (an instance of this class is returned to the user code); and the `PassThePointerOrcGC` class where the actual pointer protection is done. `orc_ptr` is composed of a reference to the shared object, `ptr`, and the index, `idx`, of the thread's hazardous pointers array of the `PassThePointerOrcGC` class where the reference `ptr` is published. `PassThePointerOrcGC` class keeps a shared list of hazardous pointers per thread, referred as `hp`. As soon as one of the `atomic_orc`'s methods is called, any shared object reference used by the method is protected by the thread in its hazardous pointers array. The array index where the pointer is stored is saved in the `orc_ptr` instance.

A shared object must be created using the global method `make_orc<T>()`, which will return an object of type `orc_ptr<T*>` with the shared object protected on the `PassThePointerOrcGC` class. Another method that returns `orc_ptr` is `orc_atomic.load()`. Both `orc_atomic.load()` and `make_orc()` protect the reference to the shared object publishing the pointer on index 0 of the thread's hazardous pointers array. In case an `orc_ptr` is stored locally, then an actual index is attributed to it and the hazardous pointer is copied from index 0 to the new position. Typically, `orc_atomic.load()` and `make_orc()` return a temporary `orc_ptr` that must be stored in a local `orc_ptr` if it is to be de-referenced later. When a local `orc_ptr` is assigned to another local `orc_ptr`, the assignment operator (lines 182 to 194) ensures the copy of the pointer in the hazardous pointers array is done in the same direction of the hazardous pointers traversal in the retire method. In Algorithm 7 we present some of the `orc_ptr` methods that support the protection of the reference managed by the `orc_ptr`.

The methodology to deploy OrcGC on a data structure is as follows:

1. Make all dynamic types (nodes) extend `orc_base`.
2. Create new instances of these dynamic types using the `make_orc<T>()` method instead of the `new()` operator.
3. Replace the usage of `std::atomic<T*>` with `orc_atomic<T*>`, where `T` is a dynamic type.
4. Use an `orc_ptr<T*>` to save the return value from `orc_atomic<T*>::load()` and `make_orc<T>()`, or to pass protected pointers across functions.

5 Evaluation

We now present an evaluation of PTP and OrcGC and compare them with other state-of-the-art schemes when applied to lock-free data structures, using synthetic benchmarks. We executed these microbenchmarks on two different machines. The first was a dual-socket 2.50 GHz Intel Xeon 5215 with a total of 20 hyper-threaded cores (40 HW threads). The second was a dual-socket 2.70 GHz AMD EPYC 7281 with a total of 32 hyper-threaded cores (64 HW threads). Both machines were running Ubuntu LTS and using gcc 8.3 with the `-O2` optimization flag.

We have applied our scheme to a total of 11 lock-free data structures and collected them in an open source library available at https://doi.org/10.5281/zenodo.4362581. In this library we have deployed OrcGC on the following data structures: a lock-free queue by Michael and Scott [20]; a lock-free queue by Morrison and Afek [21] (LCRQ); a wait-free queue by Kogan and Petrank [17]; a wait-free queue by Correia and Ramalhete [26] (TurnQueue); Harris original lock-free linked list [12]; Michael's modification to Harris lock-free linked list [18] (Michael); Herlihy and Shavit's modification to Harris linked list with wait-free lookups [15] (HS); a wait-free linked list by Timnat, Braginsky, Kogan and Petrank [27] (TBKP); a lock-free binary search tree by Natarajan and Mittal [22] (NM-tree); Herlihy and Shavit modification to Fraser's skip list [15] (HS-skip); We also implemented a new lock-free skip list (CRF-skip) based on Herlihy and Shavit's skip list.

Figures 1 and 2 display the normalized throughput for a micro-benchmark where we execute 10^7 pair of enqueue and dequeue operations on each queue. Results vary depending on the queue specific algorithm. Queues are in general very sensitive to back-off strategies because of the high contention on the head and tail of the queue. In the Michael-Scott queue, performance using OrcGC memory reclamation almost doubles when 4 threads are concurrently executing. This improvement is due to the natural back-off added by the code execution necessary to update the reference count variable of each enqueued or dequeued node. This back-off effect is no longer relevant in higher thread counts, where contention is unavoidably high. Typically, queues annotated with OrcGC perform worse when running in single thread. This is explained by the extra code execution that automatically protects an object and retires an object that is no longer accessible.

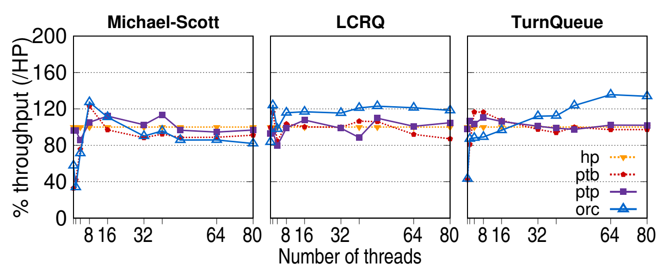

Figure 1. Lock-free and wait-free queues with 10^7 pairs of enqueues/dequeues on Intel machine.

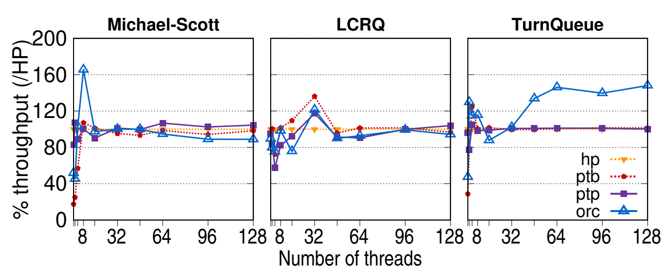

Figure 2. Lock-free and wait-free queues with 10^7 pairs of enqueues/dequeues on AMD machine.

Figures 3 and 4 display the throughput for the Michael-Harris lock-free linked list when using different reclamation schemes. On the leftmost plots, the workload is made of

50% random insertions and 50% removals. The central plots have 5% insertions, 5% removals and 90% lookups. The rightmost plots executes lookups exclusively. Each data point represents the mean over 5 runs of 20 seconds.

OrcGC has no significant cost on the Intel machine (Figure 3) and a performance penalty which for high thread counts and write-intensive workloads can go to 50% on the AMD machine (Figure 4). The linked list execution is dominated by the search procedure, where a significant cost is added due to pointer protection. For every hazard pointer that is published, there is a synchronization fence. In our implementation we use the `exchange` instruction to publish the pointer and we believe that the relative cost of this instruction is architecture dependent. We experimented replacing the `exchange` instruction with an `mfence` and results on the AMD where similar to the ones presented for Intel. This behavior indicates that the `exchange` instruction is faster than an `mfence` in the AMD architecture. On Intel the `exchange` instruction is the dominant factor performance wise. On the other hand, for the AMD architecture, the performance of the manual memory reclamation schemes surpasses OrcGC, for which other execution costs become more relevant.

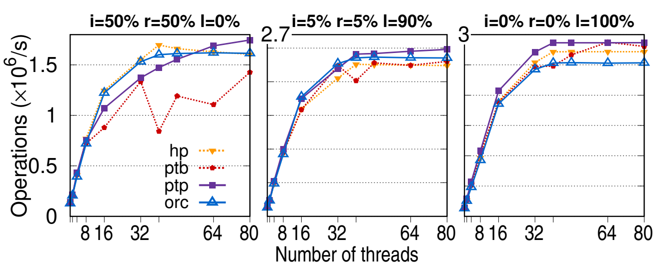

Figure 3. Michael-Harris lock-free list, 10^3 keys, Intel.

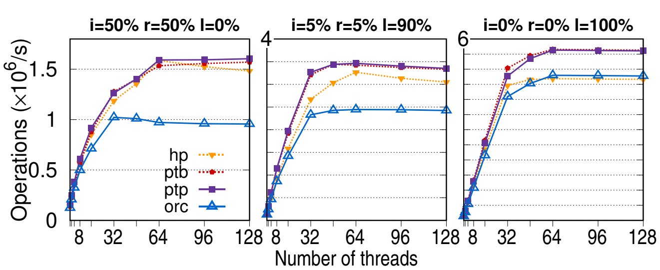

Figure 4. Michael-Harris lock-free list, 10^3 keys, AMD.

Compared with HP and PTB, the PTP scheme has little to no impact on performance, while ensuring a lower bound on the number of unreclaimed objects.

Figures 5 and 6 show the results for OrcGC applied to four different lock-free data structures on two different machines. The data structures are the following: Harris is the original algorithm described by Harris [12], Michael is the modified lock-free linked list by Michael [18], HS is the list by Herlihy and Shavit [15] based on Harris but without restarts for lookups, and TBKP is the wait-free linked list by Timnat et al. [27]. No algorithmic modification were made on these data structures, only type annotation, as described in our methodology. Apart from Michael, these linked lists serve as an example of lock-free data structures on which manual memory reclamation could not be applied. Using OrcGC, we can now compare the performance of these algorithms on equal terms.

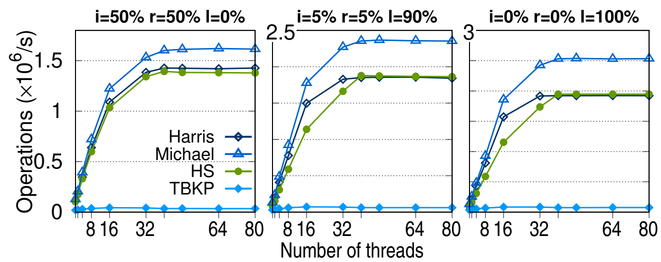

Figure 5. Lock-free linked lists with OrcGC, 10^3 keys, Intel.

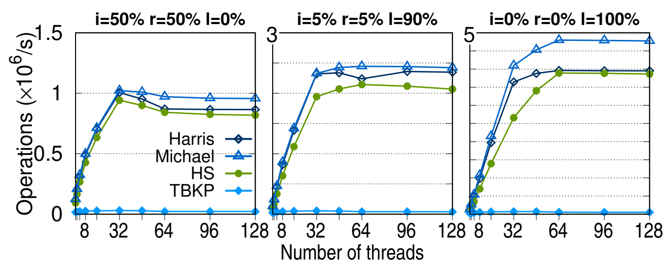

Figure 6. Lock-free linked lists with OrcGC, 10^3 keys, AMD.

Figures 7 and 8 show lock-free tree-shaped data structures with automatic or manual memory reclamation, whenever the data structure algorithm allows it. The lock-free tree by Nataran and Mittal displays the same trend as the lock-free linked list based data structures. Our automatic memory reclamation OrcGC can have a drop in performance of at most 50% on the write intensive scenario.

We ported the Java implementation of Herlihy and Shavit's skip list to C++ and integrated it with OrcGC. The `contains()` method of this skip list was designed to search for the key, traversing the skip list starting at the top level and descending until the the bottom level is reached, without ever restarting the search from the top level. This search can encounter marked nodes that are ignored, nevertheless, these nodes must be reachable and must remain linked to the data structure. This particular design can potentially create chains of nodes of key-bounded size, implying this data structure can lead to a number of unreclaimed objects bounded by the key size, even when using OrcGC memory reclamation.

To guarantee linear bound memory reclamation with OrcGC, the underlying data structure must not form chains between objects removed (unreachable) from the data structure. We have implemented a new lock-free skip list (CRF-skip) which allows the `contains()` method to restart whenever a node is poisoned. A poisoned node is a node that can no longer reach the data structure. This new design, where removed nodes are completely isolated from the data structure, reduces the progress of the `contains()` method to be lock-free, given that the search procedure can no longer continue when a poisoned node is found. Performance-wise, CRF-skip typically outperforms HS-skip in all tested scenarios and in addition has a much smaller memory footprint. In our experiments, memory usage for HS-skip is about 19 GB, while CRF-skip uses less than 1 GB.

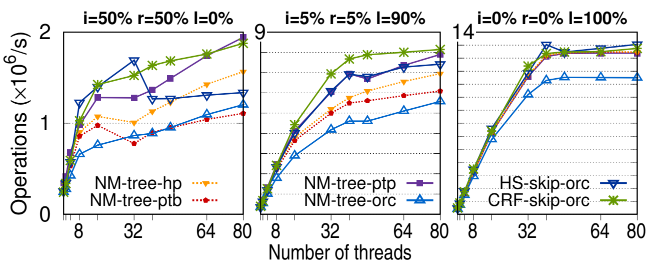

Figure 7. Lock-free tree and skip lists, 10^6 keys, Intel.

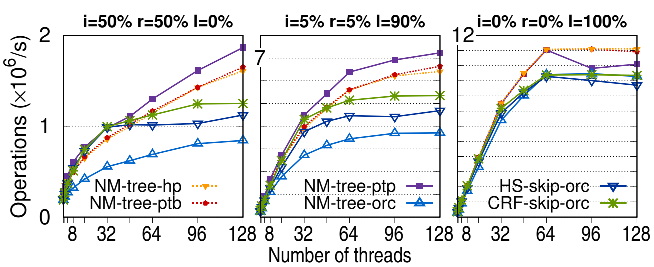

Figure 8. Lock-free tree and skip lists, 10^6 keys, AMD.

Depending on the CPU architecture and the particular data structure, the cost of doing automatic reclamation with OrcGC can be high, to nearly non-existent. Lock-free data structure implementers now have a choice of an easily deployable reclamation scheme to start from and, if the throughput is not satisfactory, they can then alter their data structure's algorithm to fit with existing manual reclamation schemes. Moreover, there are data structure algorithms which are non-trivial to modify, at least not without an important re-design of the algorithm. For those, OrcGC is now a simple (and sometimes the only) option.

6 Conclusion

For many existing data structures, incorporating lock-free memory reclamation has been until now a hard challenge for developers to tackle. OrcGC is the first automatic memory reclamation scheme with lock-free progress and linear bound on the number of unreclaimed objects, automatically protecting an object and detecting when it is no longer reachable in order to safely de-allocate it.

In this paper we applied our OrcGC methodology to multiple data structures, for which previously no lock-free memory reclamation scheme was successfully integrated. We believe OrcGC can contribute to make many more lock-free data structures available to general practitioners.

Acknowledgments

This work is supported in part by the Swiss National Science Foundation (SNSF) under project number 200021-178822/1.

References

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

[2] Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. 2018. ThreadScan: Automatic and Scalable Memory Reclamation. *ACM Trans. Parallel Comput.* 4, 4, Article 18 (May 2018), 18 pages. https://doi.org/10.1145/3201897

[3] Andrea Arcangeli, Mingming Cao, Paul E. McKenney, and Dipankar Sarma. 2003. Using Read-Copy-Update Techniques for System V IPC in the Linux 2.5 Kernel. In *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*. 297–309. http://www.usenix.org/events/usenix03/tech/freenix03/arcangeli.html

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

[5] Anastasia Braginsky, Alex Kogan, and Erez Petrank. 2013. Drop the Anchor: Lightweight Memory Management for Non-Blocking Data Structures. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '13)*. Association for Computing Machinery, New York, NY, USA, 33–42. https://doi.org/10.1145/2486159.2486184

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

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

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

[9] Dave Dice, Maurice Herlihy, and Alex Kogan. 2016. Fast Non-Intrusive Memory Reclamation for Highly-Concurrent Data Structures. *SIGPLAN Not.* 51, 11 (June 2016), 36–45. https://doi.org/10.1145/3241624.2926699

[10] Keir Fraser. 2004. *Practical lock-freedom.* Ph.D. Dissertation. University of Cambridge, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.599193

[11] 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), 1173–1187. https://doi.org/10.1109/TPDS.2008.167

[12] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings.* 300–314. https://doi.org/10.1007/3-540-45414-4_21

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

[14] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2002. The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures. In *Distributed Computing, 16th International Conference, DISC 2002, Toulouse, France, October 28-30, 2002 Proceedings.* 339–353. https://doi.org/10.1007/3-540-36108-1_23

[15] Maurice Herlihy and Nir Shavit. 2011. *The art of multiprocessor programming.* Morgan Kaufmann.

[16] Jeehoon Kang and Jaehwang Jung. 2020. A Marriage of Pointer- and Epoch-Based Reclamation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020).* Association for Computing Machinery, New York, NY, USA, 314–328. https://doi.org/10.1145/3385412.3385978

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

[18] Maged M. Michael. 2002. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '02).* Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/564870.564881

[19] 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

[20] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '96).* Association for Computing Machinery, New York, NY, USA, 267–275. https://doi.org/10.1145/248052.248106

[21] Adam Morrison and Yehuda Afek. 2013. Fast Concurrent Queues for X86 Processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13).* Association for Computing Machinery, New York, NY, USA, 103–112. https://doi.org/10.1145/2442516.2442527

[22] Aravind Natarajan and Neeraj Mittal. 2014. Fast concurrent lock-free binary search trees. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014.* 317–328. https://doi.org/10.1145/2555243.2555256

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

[24] Ruslan Nikolaev and Binoy Ravindran. 2020. Universal Wait-Free Memory Reclamation. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '20).* Association for Computing Machinery, New York, NY, USA, 130–143. https://doi.org/10.1145/3332466.3374540

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

[26] Pedro Ramalhete and Andreia Correia. 2017. POSTER: A Wait-Free Queue with Wait-Free Memory Reclamation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17).* Association for Computing Machinery, New York, NY, USA, 453–454. https://doi.org/10.1145/3018743.3019022

[27] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. 2012. Wait-Free Linked-Lists. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12).* Association for Computing Machinery, New York, NY, USA, 309–310. https://doi.org/10.1145/2145816.2145869

[28] Shahar Timnat and Erez Petrank. 2014. A Practical Wait-Free Simulation for Lock-Free Data Structures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14).* Association for Computing Machinery, New York, NY, USA, 357–368. https://doi.org/10.1145/2555243.2555261

[29] John D. Valois. 1995. Lock-Free Linked Lists Using Compare-and-Swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995.* 214–222. https://doi.org/10.1145/224964.224988

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

A Artifact Appendix

A.1 Abstract

The artifact contains the source code for a manual and an automatic memory reclamation scheme, respectively named PTP and OrcGC. It also contains multiple benchmarks used in our evaluation. Compilation requires a C++ compiler with C++17 support. Building and running the benchmarks will result in plots similar to the ones in Figures 1, 2, 3, 4, 5, 6, 7 and 8.

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

- **Algorithm:** PTP, OrcGC, HP, PTB
- **Program:** microbenchmarks
- **Compilation:** g++ 8.3
- **Binary:** g++-8
- **Run-time environment:** Ubuntu 18.10
- **Hardware:** any CPU with a C++ compiler supporting C++17
- **Execution:** sole user execution for 30 hours
- **Output:** text data files with the results of the throughput for each execution
- **Experiments:** makefile and manual steps
- **Disk space required (approximately):** 19 MB
- **Time needed to prepare workflow (approximately):** 1 minute
- **Time needed to complete experiments (approximately):** 30 hours
- **Publicly available:** yes
- **Code licenses:** MIT License

A.3 Description

A.3.1 How delivered

Available as open source under the MIT software license: http://doi.org/10.5281/zenodo.4300080

A.3.2 Hardware dependencies

None.

A.3.3 Software dependencies

We have tested building and running on a clean install of Ubuntu 18.10. This required a C++ compiler with support for the C++17 language and the make tool. It should work with any Linux distribution.

A.3.4 Data sets

None.

A.4 Installation

The source code can be built using the `make` command.

A.5 Experiment workflow

The benchmarks can be executed by using the `make ./run-all.py` commands. This in turn will execute the following binaries: `q-ll-enq-deq`, `set-ll-1k`, `set-tree-1m` and `set-skiplist-1m`.

A.6 Evaluation and expected result

OrcGC is expected to perform worse for the AMD architecture when compared with manual reclamation schemes. On Intel, OrcGC performance should be similar to the compared manual schemes. PTP performed as well or better than comparable manual reclamation schemes.

A.7 Experiment customization

Additional parameters can be explored, such as different number of threads, running times, larger or smaller key sets and different write/read ratios. Changing this parameters requires modifying the `.cpp` files in the `graphs/` folder and rebuilding with `make`.

A.8 Notes

None.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html