Automatic Memory Reclamation for Lock-Free Data Structures *

Nachshon Cohen
Technion Institute of Technology, Israel
nachshonc@gmail.com

Erez Petrank
Technion Institute of Technology, Israel
erez@cs.technion.ac.il

Abstract

Lock-free data-structures are widely employed in practice, yet designing lock-free memory reclamation for them is notoriously difficult. In particular, all known lock-free reclamation schemes are "manual" in the sense that the developer has to specify when nodes have retired and may be reclaimed. Retiring nodes adequately is non-trivial and often requires the modification of the original lock-free algorithm. In this paper we present an automatic lock-free reclamation scheme for lock-free data-structures in the spirit of a mark-sweep garbage collection. The proposed algorithm works with any normalized lock-free algorithm and with no need for the programmer to retire nodes or make changes to the algorithm. Evaluation of the proposed scheme on a linked-list and a hash table shows that it performs similarly to the best manual (lock-free) memory reclamation scheme.

Categories and Subject Descriptors D.4.2 [*Storage Management*]: Allocation/deallocation strategies; D.1.3 [*Programming Technique*]: Concurrent Programming

General Terms Algorithms, Design, Theory.

Keywords Memory Management, Concurrent Data Structures, Non-blocking, Lock-free, Hazard Pointers

1. Introduction

The rapid deployment of highly parallel machines has resulted in the acute need for parallel algorithms and their supporting parallel data-structures. In the last two decades, many *lock-free* data-structures (a.k.a. *non-blocking*) [8, 9] were proposed in the literature and their performance evaluated. Such data-structures offer progress guarantees for the programmer, which imply immunity to deadlocks and livelocks, and robustness to thread failures. However, a practitioner wanting to employ such a data-structure must also address the challenge of memory reclamation. In the presence of parallel execution, it is difficult to decide when an object will never be accessed by other threads.

Several memory reclamation schemes were proposed in the literature to allow lock-free memory reclamation [1, 3, 4, 10, 16]. All these schemes are manual in the sense that they require the programmer to install *retire* statements that notify the memory manager each time an object is unlinked from the data-structure and cannot be reached by subsequent threads computation. The latter requirement is not easy to determine and it sometimes requires a modification of the lock-free algorithm in order to be able to determine a point in the execution from which no thread can access an unlinked node. Two well-known lock-free data-structures: Harris's linked-list [6] and Herlihy & Shavit's skip-list [9] do not satisfy this property and need to be modified to allow memory reclamation [15]. Moreover, for data-structures that contain several links to a node (such as the skip-list) and in the presence of concurrent inserts and deletes, it is sometimes difficult to determine a safe point in time at which the node is completely unlinked. Similarly to inadequate deallocate operations in unmanaged program code, inserting retire statements at inadequate program locations may end up in notoriously hard-to-debug memory reclamation errors. Note that it is not possible to simply use a managed language and its built in garbage collector, as there is no garbage collection scheme available in the literature today that supports lock-free executions [19].

Interestingly, algorithmic modifications required for the insertions of the retire operations to lock-free data-structures may sometimes change the algorithms' properties and performance. For example, the linked-list of Herlihy and Shavit [9] allows wait-free searches. But modifying it to allow the retire operations [15] foils this property. An elaborate discussion on the linked-list and skip-list examples is provided in Appendix A.

In this paper we propose the Automatic Optimistic Access (AOA) scheme, a memory reclamation scheme for lock-free data-structures that does not require the programmer to install retire statements. This scheme is inspired by mark-

* This work was supported by the Israel Science Foundation grant No. 274/14.

OOPSLA'15, October 25–30, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3689-5/15/10...$15.00
http://dx.doi.org/10.1145/2814270.2814298

sweep garbage collection, but applied at a smaller scale – on a single data-structure with a structured algorithm, instead of over the entire heap with a general program. Recall that we do not know how to perform full-scale lock-free garbage collection over the entire heap [19]). The AOA scheme automatically determines (at run time) the set of unreachable data-structure nodes and reclaims them for future allocations. The obtained benefits are similar to the benefits of garbage collection for the heap: improve reliability, reduce debug time, and make the programmer life easier.

There has been much work on garbage collectors that obtain some partial guarantees for progress [2, 11, 12, 20–22]. However, none were able to eliminate the seemingly required synchronization barrier (e.g., handshake), when the garbage collector waits for all threads to acknowledge the beginning (or the end) of a collection cycle. A handshake foils lock-freedom because if a thread gets stuck, it does not respond to a handshake and eventually other threads run out of memory. We stress that lock-freedom requires making progress for any possible (and even a worst-case) scheduling. The AOA scheme, unlike all garbage collectors presented in the literature, strictly satisfies the lock-freedom property. In particular, it does not assume that all threads respond to a handshake.

While state-of-the-art garbage collectors can ensure the progress of the program itself, they all fail to guarantee the progress of the collector itself, causing an eventual failure of allocations and the entire program. Three major difficulties that arise in the full garbage collection setting need be dealt with here as well. First, there is the challenge of getting all threads' roots in a lock-free manner. Second, the collector must be able to function with no handshake, meaning that some threads may be executing under the assumption that a previous collection is still active. Finally, to prevent memory bloat, some form of lock-free compaction must be implemented. The last is a minor problem for us, because the nodes of the data-structure are of a fixed size. But we will deal with the first two challenges in the smaller scale of data-structure nodes under data-structure operations.

To read the roots of thread that does not respond, one needs access to its registers but modern CPUs do not allow reading other thread's registers. A naive solution is to make sure that each pointer held by a register also resides in the memory. But that requires a write and a memory fence for each register modification, making the overhead impractical. Building on the structured algorithms of lock-free data-structures, the AOA scheme solves this by making the threads record their roots at known interval and by making threads (in rare cases) return to execute from locations where their roots were known. A carefully merge of root snapshotting and execution restart allows avoiding the continuous recording of local roots. Restarting lock-free operations at various points requires some known structure of the algorithm. An adequate normalized algorithmic structure has been proposed for a different reason in Timnat and Petrank [24]. The AOA scheme is proposed for normalized lock-free algorithms. All lock-free data-structures we are aware of can be represented in an efficient normalized representation.

Finally, since a handshake is not allowed, we must deal with program threads or collector threads that wake up and execute operations without realizing that the garbage collection has advanced to a newer collection cycle and phase. To do that, we employ standard versioning schemes to protect sensitive collector and program shared data. Careful use of versioning provides a correct scheme that reclaims data-structure nodes properly.

The main contribution of this paper is the proposal of the AOA scheme, a lock-free memory reclamation scheme that relieves the programmer from manually inserting retire calls. The AOA scheme is applicable to any lock-free data-structure in the normalized form of [24]. It eliminates the need to modify algorithms to make them amendable to insertions of retire instructions, and it eliminates the need to find adequate locations for the retire instructions themselves. Both of the above tasks are non-trivial as exemplified in Appendix A. The AOA scheme allows a separation of concerns: it facilitates an independent algorithm design that is not memory-management-aware. Memory management can then be added automatically without any modifications to the original algorithm.

The AOA scheme deals with the main challenges of lock-free garbage collection on a small scale. We hope that the techniques and solutions proposed herein will be of use for future attempts on solving the full lock-free garbage collection problem.

Organization This paper is organized as follows. In Section 2 we specify assumptions and present the AOA interface. In Section 3 we present an overview of the AOA scheme. In Section 4 we provide some important background. In Section 5 we present the AOA scheme. In Section 6 we present the implementation and evaluation of the AOA scheme. We discuss related work in Section 7, and we conclude in Section 8.

2. Settings and Problem Statement

2.1 Shared Memory With TSO Execution Model

We use the standard computation model of Herlihy [8]. A shared memory is accessible by all threads. The threads communicate through memory access instructions on the shared memory, and a thread makes no assumptions about the status of any other thread, nor about the speed of its execution.

We also assume the TSO memory model, used by the common x86 architecture. For a formal definition of this model, see Owens et al. [18] and the references therein. Informally, write instructions are cached in a local write buffer and made visible either by a memory fence or by

an internal CPU action. Reads are not reordered with other reads, and writes are not reordered with other writes.

2.2 Lock-Freedom and Wait-Freedom

The AOA scheme provides memory management support for *lock-free* data structure. An data-structure is called *lock-free* (a.k.a. *non-blocking*) [8] if it guarantees that (at least one) operation will complete after the threads execute a finite number of steps. Lock-free data structures are fast, scalable and widely used. They eliminate deadlock, live-lock, and provide guaranteed system responsiveness. In the last two decades, many efficient lock-free implementations for almost all common data structure have been developed. While the lock-free property is important, it does not rule out starvation of threads. It is possible that some threads make progress but others do not. This paper focuses on memory management support for lock-free data-structures.

The *wait-free* guarantee is a stronger guarantee, ensuring that each and every operation terminates in a finite number of steps and, in particular, without dependence on the behavior of other processes in the system. Wait-free data-structures are more complicated to design and not many wait-free implementations were known until recently. A series of recent papers led to a mechanical simulation of normalized lock-free algorithms in a wait-free manner [13, 14, 24], which produced many efficient wait-free algorithms as well.

2.3 Problem Definition

The AOA scheme automatically finds unreachable objects and reclaims them. However, unlike a full garbage collection, the AOA scheme only reclaims objects of a predetermined data-structure and it requires that the data-structure algorithm be presented in the normalized form of [24]. Next we specify the data-structure's nodes and links and the interplay between the data-structure, the AOA scheme, and the underlying system.

We think of a data-structure as a set of objects that are connected via data-structure links (or pointers). The data-structure objects are denoted *nodes*. Each node may have *link* fields and *data* fields; link fields contain pointers to other nodes while data fields do not contain pointers to other nodes. A pointer in the node that references an (outside) object that is not a data-structure node (i.e., that the AOA is not responsible for reclaiming) is not called a link. It is a data field. In addition, there exists a set of global pointers from outside the data-structure to nodes in the data-structure, that allow access to the data-structure. These pointers are denoted *GlobalRoots* and their location must be supplied to the AOA scheme via the interface defined in Subsection 2.4. We assume that the location of GlobalRoots are not modified during the execution. For example, a head pointer, pointing to the head of a linked-list is a global pointer. Note that the content of a global root can be modified or nullified, but the location of the root cannot be modified. We assume that all pointers that allow access to nodes in the data-structure are registered as GlobalRoots using the proper interface. The pointers that are used by the data-structure operations to traverse the structure are not called global roots. They are *local* roots (local to each operating thread) and will be discussed below.

All pointers into the data-structure, including links, global roots, and local roots may contain *marked* (tainted) pointers that do not point to the beginning of a node. Specifically, many lock-free algorithms steal a bit or two of the address for other purposes, such as preventing the modification of the pointer. We assume a function that gets a marked pointer and computes the address of the node implied by it. We denote this function as *unmark*. In many implementation, using the function *unmark(P)* that simply clears the two least significant bits of *P* is adequate.

Similarly to a garbage collector, the AOA scheme reclaims all nodes that are not reachable (by a path of links) from the global and local roots. However, the AOA scheme does not consider all the pointers that are directly available to the program as local roots. The AOA scheme may reclaim an object that is being read by the program. In other words, when the program reads an object without modifying it, the pointer to this object does not protect it from being reclaimed. Thus, local pointers, that reference objects that are only read and not modified, are not considered local roots for the reachability scan, and objects reachable only from such pointers can be reclaimed. So in the AOA scheme we only consider local pointers that participate in modifications of the shared memory as roots for the traversal of live objects. The interval during which a local pointer counts as a root for the automatic memory reclamation is defined and explained in Subsection 5.2.

Since nodes that the thread read can be reclaimed before the thread accesses them, an effort is made to make the access of reclaimed nodes safe from hitting a trap. This is achievable by using a user level allocator and not returning pages to the operating system. Alternatively, the signal handler can be configured to ignore the trap in such a case.

Throughout the rest of the paper we assume that enough space is allocated for the program. Otherwise, an out-of-memory exception will foil lock-freedom in a non-interesting manner. We also assume that data-structure nodes and the content of global roots are modified only via data-structure operations that are presented in the normalized form of [24].

An Example: the Harris-Michael Linked-List Let us exemplify the definitions above by considering the Harris-Michael linked-list [15]. Consider the following snippet of code: The data-structure node in this case is the `struct node` structure. This struct contains a single link at offset 0 (the next pointer) and two data fields. The void* field is considered a data field and it must not point to a node. This data-structure has two global roots: {&list1head,&list2head}, which are located at fixed locations in memory. Each global root contains a pointer to a node. Note that while the nodes

Listing 1. Harris-Michael linked-list

```
1 struct node{
2     struct node *next;
3     int key;
4     void *data;
5 };
6 struct node *list1head, *list2head;
```

referenced by list1head and list2head may change, the locations of the pointers list1head and list2head are fixed throughout the execution.

2.4 The AOA Interface

Let us now briefly describe the interface between the AOA scheme and the data structure it serves. A complete description of the interface is presented at Appendix C.

The data structure is responsible for providing a description of the data-structure node class to the AOA implementation. The description includes the class size, the number of pointers, and the location (offset) of each pointer. In addition, the program provides the location of all global roots to the AOA scheme. Before starting to execute data structure operations, the program initializes the AOA scheme by calling its initialization function.

Each read and write of the original algorithm operations must be modified to include a read- and write-barrier. This can be done automatically by a compiler, but in our simple implementation we let the programmer add these barriers manually. All allocations of data-structure nodes must be handled by the AOA functionality using a dedicated function.

The AOA scheme can be applied to multiple data structures in the same program. In this case, the AOA scheme interface is replicated. All components of the interface, including the allocation function, the write-barrier, etc. are implemented once for each data structure. A data-structure name prefix is used to distinguish the different interfaces.

3. Overview

In this section we provide an overview of the AOA scheme. The AOA scheme extends the optimistic access scheme [4], using it to reclaim nodes. The AOA scheme is a fast lock-free reclamation scheme and our goal is to keep the overhead of reclaiming nodes low. But while the (manual) optimistic access scheme assumes *retire* instructions were installed by the programmer at adequate locations to notify a disconnect of a node from the data-structure, the AOA scheme can work without any retire statements.

The AOA scheme design is inspired by the mark-sweep garbage collector. It marks nodes that are "reachable" by the program threads and it then reclaims all nodes that were not marked. On one hand, the task here is simpler, because we are only interested in reachability of nodes of the data-structure and not the reachability of all objects in the heap. However, the reclaimer must avoid scanning the roots of the threads (as some threads may not be responding), it must make sure that threads that wake up after a long sleep do not cause harm, and it must make sure that threads that stop responding while scanning reachable nodes do not make the entire tracing procedure miss a node.

Reachability, Roots, and Concurrence. The first step in the design is an important observation. The observation is that when running Cohen and Petrank's optimistic access scheme [4], all nodes that must not be reclaimed at any point in time are reachable from either global pointers (as defined in Section 2.3) or from hazard pointers used by that scheme. The AOA is built to preserve this temporal observation in the presence of concurrent execution. While the observation is correct for a snapshot of the execution, the AOA scheme works with it in a concurrent manner. First, the reclamation scheme is modified so that all active threads join the reclamation effort as soon as possible. Second, the setting of hazard pointers and the cooperation of other threads (by checking whether a reclamation has started and then moving to cooperate) is modified to ensure the correctness of the reclamation. These modifications ensure that active threads do not modify the reachability graph in a harmful way before joining the reclamation effort. Note that joining the reclamation effort does not block a thread's progress as the reclamation execution is lock-free and finite.

Protecting the Collector's Data-Structures. A classical mark-sweep garbage collector works in *cycles*, where in each cycle, one collection (mark and sweep) is executed. In this paper we use the term *phase* and not *cycle* to denote a reclamation execution in order to stress that the execution is asynchronous. Not all threads are assumed to cooperate with the currently executing reclamation and some threads may be (temporarily) executing operations that are relevant to previous phases. All garbage collectors we are aware of assume that the phases are executed in order; no thread starts a phase before all threads have finished all stages of the previous phase. This is not the case for us as we must deal with threads that do not respond.

A second challenge for the AOA is to deal with threads executing in an outdated reclamation phases and make sure that they do not corrupt the reclamation execution of the current phase. A proper execution is ensured using versioning of the reclaimers' shared data. Variables that can be corrupted by threads executing the wrong phase are put alongside a field that contains the current phase number. These variables are modified via a CAS that fails if the current phase number does not match the local phase (the phase observed by the thread that executes this modification). Before starting a new phase, all such *phase-protected* variables are updated to a new phase. Thus a late thread will never erroneously modify such phase-protected collector data.

An example of a phase-protected variable is the mark-bit table. The concern here is that a thread executing the wrong

phase may mark a spurious node, corrupting the currently executing marking stage. Since the mark-bits are phase-protected, a thread executing the incorrect phase will fail to mark a spurious node.

Similarly to standard garbage collection, we are able to ensure that in each phase, all nodes that were unreachable at the beginning of the phase are reclaimed and recycled for new allocations.

Completing Work for Failing Threads. The final challenge is the proper completion of a collection phase even if a thread fails in the middle of the phase. Before discussing the solution, we start by explaining the race condition that may harm the progress guarantee of the AOA algorithm. During the mark stage, it is customary to execute the following loop: pop an item from the mark-stack, attempt to mark this node, and if successful, scan the node's children. Suppose a thread drops dead after marking a node and before inserting its children to its mark-stack. Then the node's children must be scanned, but other threads have no way of knowing this. This race condition is traditionally handled by waiting for the thread to respond, which revokes the algorithm's progress guarantee.

To handle this challenge, we modified the mark stage as follows. First, we let the local mark-stack of each thread be readable by other threads. Second, we designed the mark procedure to satisfy the invariant that children of a marked node are either marked or are visible to other threads. Thus, even if a thread get stuck, other threads can continue to work on nodes that reside on its stack and no node is lost. This solution allows the AOA scheme to complete a garbage collection phase even when a thread crashes in the middle of marking a node.

A similar race may happen during the sweep phase, but with less drastic consequences. When a thread processing a chunk of memory for sweep purposes is delayed, we allow nodes that reside in that chunk to not be processed and not be handed to the allocator. The unprocessed chunk is a small fraction of the memory, making the harm negligible. Furthermore, these free spaces are recovered at the next phase.

4. Background

In this section we briefly describe the normalized representation and the optimistic access scheme, on which the AOA scheme relies. We also present the read- and write-barriers of the AOA scheme, which are inherited from the optimistic access scheme.

4.1 Normalized Data-Structures [24]

The AOA scheme assumes that the data-structure implementation is given in a normalized form. In this subsection we briefly describe what a normalized data-structure implementation looks like and explain why the normalized representation is required. Let us start with the latter.

The AOA scheme (infrequently) allows threads to access nodes whose underlying memory was reclaimed and recycled. Such an access may return an arbitrary value, which must not be relied on. When this happens, the execution is rolled back to a *safe point* where the stale value does not exist, and all references point to real (unreclaimed) nodes. In addition, a mark and sweep algorithm starts by gathering local references to nodes (denoted root collecting in the GC literature). To avoid considerable overhead, threads publish their local references only during safe points. However, additional references may be added to the threads' local state after the last encountered safe point; these references are not traced during the garbage collection phase. Thus, rolling back the execution to the safe point returns threads to the point where the thread's roots match the collected roots. The structure of a normalized form allows for easy definition of safe points and the rolling back mechanism. Next we informally explain what a normalized implementation looks like. The formal definitions are provided in Appendix B and in [24], but the discussion here is sufficient to understand the rest of the paper.

Loosely speaking, a normalized form partitions each data-structure operation into three routines. The first routine, called the *CAS generator* receives the input of the operation. It searches the data-structure and prepares a list of CAS instructions that would apply the required operation. The second routine, called the *CAS executor*, receives the prepared list of CASes and executes it, stopping at the first failure (or after all CASes were executed). The third routine, called *wrap-up*, receives the input of the operation, the CAS list prepared by the CAS generator, and the number of CAS executed by the CAS executor method. It determines whether the operation has completed, and (in case of success) also the returned value. Otherwise, the operation is started again using the *CAS generator*.

The partition into three routines needs to satisfy that the first routine (the CAS generator) and the third routine (the wrap-up) are *parallelizable*. An execution of a parallelizable routine depends only on its input, and not on the local state of the executing thread. Furthermore, at any point during the execution of a parallelizable routine it is possible to abandon the local state and return to the beginning of this routine without harming the correctness of the operation. Due to the special structure of parallelizable routines, it is possible to define a safe point at the beginning of such a routine. Consequently, at any point during execution, it is possible to abandon the execution and (re)start from such a safe point.

The simplest example of a parallelizable routine is a routine that starts from the data-structure root (e.g. head pointer) and then traverses part of the data-structure, never modifying shared memory. Such a routine depends only on the (unreclaimable) data-structure root, and restarting the traversal from scratch is always correct. More complex parallelizable routines may write to shared memory, if this modifica-

tion does not change data-structure semantics but only the underlying representation. A typical example is the physical delete of nodes that were previously marked (logically deleted) in Harris-Michael linked-list. In general, modifications of shared memory in normalized data-structures are allowed only via the CAS instruction.

When the AOA scheme executes parallelizable methods, it uses their nice properties and it returns to a safe point when needed. In contrast to the CAS generator and the wrap-up routines, the *CAS executor* is not parallelizable and cannot be (safely) restarted. Thus, the AOA scheme ensures that references in the thread's local state are always made visible during the execution of the CAS executor. By definition of the CAS executor method, it never reads from shared memory, and so the references in the thread's local state never change during the execution of the CAS executor. The references in the thread's local state are made visible during the execution of this routine by recording them publicly once before the CAS executor begins.

Finally, we note that efficient normalized representations exist for all lock-free data-structures of which we are aware. Further details appear in Appendix B and in [24].

4.2 The Optimistic Access Scheme [4]

The AOA scheme extends the optimistic access scheme, and uses some of the mechanisms it supplies. In this subsection we sketch the design of the original (manual) optimistic access scheme and specify some details that the AOA scheme inherits.

The optimistic access scheme was designed to provide fast manual lock-free memory reclamation for data-structures in normalized form. The optimistic access scheme was able to allow fast reads from the shared memory without the high overhead incurred by earlier schemes. To this end, it allowed optimistically reading a node even if this node might have been concurrently reclaimed. When a thread discovered that the read value might be stale, it used the rolling-back mechanism of the normalized representation to abandon the local state (including the stale value) and restart from a safe point.

To discover when a value is potentially stale, the optimistic access scheme works in phases. In each phase, it attempts to reclaim only nodes that were unlinked and manually retired at the previous phase. Each thread has a flag, called the *warning bit*, and at the beginning of a phase, the warning bits of all threads are set. When a thread notices that its warning bit is set, the thread clears its warning bit and rolls back the execution to a safe point. Therefore, we get the invariant that a thread may read a stale value only when its warning bit is set.

The optimistic access scheme assumes that memory is never returned to the operating system, so that accessing recycled nodes does not trigger traps. The AOA scheme follows the optimistic access scheme by allowing optimistic accesses and prohibiting the returning of pages to the operating system.

Writes cannot be executed optimistically since an optimistically executed write may corrupt a node used by other threads. Thus, each thread has a set of three globally visible "hazard" pointers, called *WriteHPs*. The optimistic access scheme does not reclaim nodes pointed to by hazard pointers. Before the write begins, the thread writes the three participating pointers (the modified node, the expected pointer, and the new pointer) to its WriteHPs. Then it checks whether the warning bit has been set. If it has, the thread clears its warning bit and rolls back the execution to a safe point, and if it hasn't, then the CAS to the shared memory can be executed safely. These hazard pointers can be cleaned immediately after the CAS completes. However, there are some hazard pointers that need to be kept for longer than the execution of a single CAS as explained below.

The optimistic access scheme occasionally makes threads roll back the execution to a safe point. The scheme maintains an invariant that after returning to a safe point, a thread's local state does not contain references to reclaimed nodes. This means that local pointers into the data-structures (links) that are held at a safe point must be protected with hazard pointers along the execution starting at a safe point and as long as the thread might return to this safe point.

Safe points are clear and simple for normalized data-structures. Two such safe points are defined by the optimistic access scheme. The first safe point is at the beginning of the CAS generator routine and the second safe point is at the beginning of the wrap-up routine. The inputs to the CAS generator are the inputs of the entire operation and so they do not contain a reference to a node except for global roots. Thus there is no need to protect any local pointer following this safe point. But the second safe point, at the beginning of the wrap-up routine, does have local pointers that reference nodes in the data-structure. The inputs to the wrap-up routine are again the inputs to the operation, but also the CAS list generated by the CAS generator. In order to make sure that any thread that returns to this safe point has local pointers into nodes that were not reclaimed, all pointers in the CAS list are protected by hazard pointers. These hazard pointers are set just before starting to execute the wrap-up method and are only cleared at the end of the wrap-up method.

In addition to protecting safe points and their subsequent instructions, the optimistic access scheme never rolls back the CAS executor because it writes to shared space and these actions cannot be undone. Therefore, all the nodes the CAS executor accesses during its execution must not be reclaimed and they are also protected throughout its execution. The inputs to the CAS executor is the CAS list generated by the CAS generator, with exactly the same roots as is the safe point at the beginning of the wrap-up routine (discussed in the previous paragraph). Thus, the roots in the CAS list are published immediately before the CAS generator finishes

Listing 2. Reading shared memory

```
1  //Original instruction: var=*ptr
2  var=*ptr
3  if(thread->warning){
4      thread->warning=0;
5      helpCollection(); // different from the OA scheme.
6      restart; // from the beginning of currently
7               // executing routine
8  }
```

Listing 3. Write shared memory

```
1  // Original instruction: res=CAS(O.field, A, B)
2  WriteHPs[0]=UNMARK(O);
3  if (A is a node pointer) WriteHPs[1]=UNMARK(A);
4  if (B is a node pointer) WriteHPs[2]=UNMARK(B);
5  __memory_fence();
6  if(thread->warning){
7      thread->warning=0;
8      helpCollection();// different from the OA scheme.
9      restart; // from the beginning of currently
10             // executing routine
11 }
12 res=CAS(O.field, A, B);
13 WriteHPs[0]=WriteHPs[1]=WriteHPs[2]=NULL;
```

and are preserved during the execution of the CAS executor and the wrap-up routines.

4.2.1 Adopting Optimistic Access Scheme Code into the AOA Scheme

Let us specify how the above description of the optimistic access scheme gets adopted into the AOA scheme. In Listings 2, 3, and 4 we present code executed in the AOA scheme. Listing 2 is executed with each read from shared memory (a.k.a. a read-barrier). The code in Listing 3 is executed whenever one writes to the shared memory during the execution of the parallelizable CAS generator method or the parallelizable wrap up method. The code in Listing 4 is executed at the end of the CAS generator method. These algorithms are the same algorithms as in the optimistic access scheme, except for one difference that is crucial for correctness: threads that find out that a phase has changed (their warning bit is set) move to helping the reclamation procedure. As mentioned in Section 2.3, we sometimes need to access a pointer that was marked by the original lock-free algorithm. We use the notation UNMARK(O) for a routine that returns a pointer to the beginning of the node O, without any mark that the algorithm might have embedded in the pointer.

At the end of the wrap-up routine, all SafePointHPs are cleaned (i.e., set to NULL).

Listing 4. End of CAS generator routine

```
1  //Assume output of the CAS generator has been set to
2  //a sequence of CASes: CAS(O_i.field_i,A_i,B_i), 0 ≤ i < k
3  SafePointHPs = all node pointers in the CAS sequence
4  __memory_fence();
5  if(thread->warning){
6      thread->warning=0;
7      helpCollection();// different from the OA scheme.
8      restart; // from the beginning of currently
9               // executing routine (CAS generator)
10 }
11 // End of CAS generator method
```

5. The Mechanism

In this section we present the details of the AOA scheme. We start by describing the memory layout, and then describe the three stages of the algorithm: the root collecting, the mark, and the sweep. We assume that the program has the cooperation code described in Listings 2, 3, and 4 described above.

Recall that we use the term *phase* (and not *cycle*) to denote a reclamation execution in order to stress that the execution is asynchronous. Not all threads are assumed to be cooperating with the currently executing reclamation. Each phase is further partitioned into three *stages*: root collection, mark, and sweep. In the first stage of a phase all global pointers plus all references residing in threads' hazard pointers are collected. In the second stage of a phase we mark all nodes reachable from the set of root pointers collected in the first stage. In the third stage of a phase we sweep, i.e, reclaim all non-marked nodes and remove the marks from marked nodes, cleaning them to prepare for the mark of the next phase.

5.1 Memory Layout

The AOA scheme uses four globally visible variables: a phase counter, an allocation pool, a mark-bit table, an additional (small) mark-bit table for each array, and a sweep chunk index.

The phase counter is the first variable that is incremented when a new phase starts. It holds the current collection phase number and is used to announce the start of a new phase.

The allocation pool contains the nodes ready to be allocated during the current phase. It is emptied at the beginning of a phase, refilled during the sweep phase, and then used for new allocations. The allocation pool is phase-protected, meaning that adding or removing items can succeed only if the attempting thread is updated with the current phase number. The allocation pool is implemented as a lock-free stack, where the head pointer is put alongside a phase-number field. Modifications of the head pointer are executed by a wide CAS, which succeeds only if the phase number is correct. Every item in the pool is an array containing 126 en-

tries; thus a thread accesses the global pool only once per 126 allocations. Like the global pool, the local pools are also emptied at the beginning of a phase.

The mark-bit table is used for the mark stage. To avoid spuriously marking nodes by threads operating on previous phases, every 32 mark-bits are put alongside a 32-bit phase number. The marking procedure marks a node with a 64-bit CAS that sets the respective bit while checking that the phase number was not modified. If necessary, it is possible to extend the phase number to 64-bits by using a wide CAS instruction of 128 bits. However, even for frequent triggering in which a new reclamation phase starts after allocation of 1000 nodes, wraparound of the 32-bits phase number occurs only once per $2^{32} \cdot 1000 \approx 4.3 \cdot 10^{12}$ allocations.

The sweep chunk index is a phase-protected index used to synchronize the sweeping effort. It contains a 32-bit index that represents the next sweep page that should be swept. This variable is phase protected, put alongside a 32-bit phase number, and modified only via a 64-bit CAS.

5.1.1 Load Balancing for Tracing Arrays of Pointers

The AOA scheme employs an additional mark-bit table for each large arrays of links. (For example, for a hash table.) An array may create a problem for load balancing if not partitioned. It is preferable that each thread will trace a different part of the array and not compete with the other threads on tracing the entire array. Thus the AOA scheme divides each array into *chunks* and associates a mark bit for each chunk. These mark-bits are phase-protected; similarly to the mark-bit table, every 32 bits are put alongside a 32-bits phase number. The mark bit is set if the respective chunk was traced and all references already appear in the mark-stack. In addition, the AOA scheme associates an additional counter for each array that is used for synchronizing the tracing efforts. This counter is used to divide the chunks between threads. If a chunk is treated as a simple node, threads may continuously compete for tracing a chunk, reducing efficiency. The counter is used to optimistically divide chunks into threads such that each thread will start by tracing a different chunk. But at the end, when the counter goes beyond the array limits, threads make sure all chunks have been marked before moving on to trace the next node.

5.2 Root Collecting

Collecting the roots for the AOA collection phase amounts to gathering all hazard pointers and global roots. This is a huge simplification over general garbage collection for which efficient lock-free root scanning is a major obstacle. Note that we do not need thread cooperation to obtain these roots, and we can gather the roots even if threads are inactive. The code for the root collection procedure appears in Listing 5.

Root collection is done by every thread participating in the collection phase. In fact, if a node is unreachable from the roots collected by some thread, then this node is guaranteed to be unreachable. But threads that observed previous roots may declare it as reachable (denoted floating garbage). It is possible to reach a consensus about the roots, or even to compute the intersection of the collected roots, but a simpler solution is to let every thread to collect roots on its own and to trace them.

Listing 5. Root collecting

```
1  void collectRoots(LocalRoots){
2      For each thread T do
3          LocalRoots += T.WriteHPs
4          LocalRoots += T.SafePointHPs
5      For each root in GlobalRoots
6          LocalRoots += *root
7  }
```

Let us say a few words of intuition on why it is enough to trace only nodes accessible by hazard pointers (in addition to the global roots). In a sense, the hazard pointers represent local roots for the garbage collection of the data-structure. A memory reclamation phase starts with raising the warning flags of all threads. Suppose that threads could immediately spot the warning flag and could immediately respond to it. In this case all threads immediately start helping the collection and then they return to a safe point in which the hazard pointers actually represent all the local roots that they have into the data-structure. However, the threads do not respond immediately. They may perform a few instructions before detecting the warning flag. The barriers represented in Listings 2, 3, and 4 ensure that if a thread modifies a link during a collection phase (after the flag is set), then the modified node, the old value, and the new value are stored in hazard pointers (and thus are considered roots). Thus no harm happens during this short execution until the flag is detected.

5.3 The Mark Stage

In the mark phase we start from the data-structure root pointers and traverse inner-data-structure pointers to mark all reachable data-structure nodes. This procedure is executed while other threads may be executing operations on the data-structure but in a very limited way. Each thread that discovers that reclamation is running concurrently joins the reclamation effort immediately.

One of the things we care about is that if a thread fails, then other threads can cover for it, completing the work on its plate. To do this, we let each thread's mark-stack be public and also a thread keeps a public variable called *curTraced* which holds a pointer to the node it is currently working on. This allows all other threads to pick up the tracing of a failed thread.

Next, let us describe the basic tracing routing that traces one single node that currently resides on the top of the mark-stack. The reclaiming thread starts by peeking at the top of its local mark-stack, reading the first node. We stress that this

Listing 6. Mark a node

```
1  //Attempt to process (mark) a single node from the
2  //markstack. Return LOCAL_FINISH if local markstack
3  //is exhausted, 0 otherwise.
4  int markNode(){
5      if(markstack.is_empty())
6          return LOCAL_FINISH;
7      obj = markstack.peek();
8      if(is_marked(obj) ||
9                  phase(markbit(obj))!=localPhase){
10         markstack.pop();
11         return 0;
12     }
13     curTraced=obj;
14     pos = --markstack.pos;//popping obj.
15     for each child C of obj do
16         markstack.push(C);
17     if(mark(obj))//successfully modified mark-bit table
18         return 0;
19     else{
20         //undo pushing children.
21         markstack.pos=pos;
22         delete entries above pos (set to NULL);
23         return 0;
24     }
25 }
```

node is not popped, since popping it makes it "invisible" to other threads and then they cannot help with tracing it. Second, the thread checks that the node is still unmarked and the phase is correct. If the node is already marked it is popped from the mark-stack and we are done. If the global phase was incremented, we know that we have fallen behind the reclamation execution, so tracing terminates immediately. It is fine to terminate reclamation at this point because the reclamation phase that we are in the middle of has already been completed by concurrent threads and we can attempt to continue allocating after updating to the current phase. Third (if the node is unmarked and the phase is correct), the mark procedure saves the traced node in *curTraced*. Now there is a global reference to the traced node and we pop it from the mark-stack. Next, the mark procedure traces the node, pushing the node's children to the mark-stack. Finally, the thread attempts to mark the traced node by setting the relevant bit in the mark-bit table. This mark operation succeeds if the node was unmarked and the local phase matches the global phase. If the node is successfully marked, we are done. If we fail to mark the node, then the node's children are removed from the mark-stack and we return. The code of the tracing routine appears in Listing 6. This procedure returns "LOCAL_FINISH" if the local mark-stack has been exhausted, and the constant 0 otherwise.

The markNode routine is invoked repeatedly until the local mark-stack is empty. But an empty (local) mark-stack of a thread does not imply the termination of the marking stage. Determining global termination in a lock-free manner is our next challenge. The tracing stage completes when all mark-stacks and the *curTraced*s of all threads contain only marked nodes. However, a thread cannot simply read all mark-stacks, because the mark-stacks can be modified during the inspection. This challenge is solved by noting a special property of the *curTraced* variable. A thread T modifies its curTraced variable when it discovers an unmarked object. If a thread inspects T's mark-stack, finds everything marked (and also T's curTraced is marked) and if and during the inspection period T's curTraced variable is not modified and T's local phase does not change, then T's mark-stack was also not modified during this period. If, however, something did change, or something was not marked, then T either finds a unmarked node and helps marking it or it simply starts the inspection from scratch.

Checking whether the marking stage completed starts by recording the current phase and the curTraced variables of all threads. Then it inspects mark-stacks of threads executing the current phase. Finally it reinspects the curTraced variable and the local phase of all threads. If no local phase was modified, no curTraced variable was modified, and all mark-stacks and all curTraced variables contained only marked nodes, then tracing is complete. If the thread observes an unmarked node and the thread is executing the current phase, it helps with tracing it. If the curTraced variables were modified and the inspecting thread observed no unmarked node, then the thread restarts the inspection. The code that checks whether the marking effort completed and the code that executes the marking stage are presented in Listing 7.

5.4 Sweep

The sweep stage is simpler than the previous stages. The sweep is done in granularity of pages, denoted *sweep pages*. Each thread synchronously grabs a sweep page and places all unmarked nodes in this page in the allocation pool. Synchronization is done by a phase-protected atomic counter, so that threads executing an incorrect phase will not grab a sweep page. A thread executing the sweep of an incorrect phase will not corrupt the allocation pool since the allocation pool is phase protected as well. A thread that drops dead at a middle of a sweep causes all unmarked nodes on this sweep page to be abandoned for the current phase. However, at most a single sweep page is lost per unresponsive thread.

Typically, sweep is also used to clear the marks from all nodes in preparation for the next collection phase. This is not done here. Instead, the mark clearing is executed at the beginning of a new phase. During a new phase initiation we need to protect all mark-bit words with the new phase number. While creating this protection we can clear the marks simultaneously with no additional cost.

The sweep algorithm is presented in Listing 8. It is divided into two functions. The first function, sweepPage, processes a single page, moving all unmarked nodes to the al-

Listing 7. mark-finish-or-progress

```
bool finish_or_progress(){
    int curPhases[numThreads];
    void *curTraces[numThreads];
    for each thread T at index i do
        curPhases[i]=T.localPhase;
        curTraces[i]=T.curTraced;
        if(curPhases[i]==thread.localPhase &&
                        !is_marked(curTraces[i]))
            {help(curTraces[i]); return false;}
    for each thread T at index i do{
        if( curPhases[i] != thread.localPhase ) continue;
        for each markstack record R of T do{
            if(!is_marked(R)){
                help(R);
                return false;
            }
        }
    }
    for each thread T at index i do
        void *ct = T.curTraced;
        if(curTraces[i]!=ct)
            return false;//progress made by other thread.
        if(curPhases[i]!=T.localPhase)
            return false;//thread i joined the mark.
    return true;//finished.
}
void help(void *node){
    if(thread.localPhase == phase)
        markstack.push(node);
    else
        markstack.clear(); //finish mark stage.
}
void markStage(){
    do{
        while(markNode()!=LOCAL_FINISH){}
    }while(finish_or_progress()==false);
}
```

Listing 8. Sweep

```
sweepPage(int pageNumber){
    void *page = getPage(pageNumber);
    for each node O residing in page do{
        if(mark−bit(O)==false)
            localPool.insert(O);
        if(localPool.size ==LOCAL_POOL_SIZE){
            allocationPool.insert(localPool, localPhase);
            localPool.reset();
        }
    }
}
sweep(){
    while(true){//while sweep not finished.
        do{//obtain a single page for sweep
            int64 old = sweep_chunk_index;
            if(index(old)>=numSweepPages)
                return;//sweep finished
            if(phase(old)!=localPhase)
                return;//entire phase finished
            int64 new=old+1;//same phase, index+1
        }while(!CAS(&sweep_chunk_index, old, new));
        //Sweep nodes on obtained page.
        sweepPage(index(old));
    }
}
```

location pool. The second function, sweep, synchronizes the sweeping effort between all participating threads.

5.5 Phase Triggering

Similarly to triggering of general garbage collection, triggering a reclamation is more of an art than science. Various triggering mechanisms can be used. In our simple implementation we used a very simple heuristic. Before the measurement began we allocated a fixed number of nodes and inserted them to the allocation pool. This can be thought of as analogue to a memory manager that allocates a heap for its use in an execution. A new phase was triggered when the allocation pool was exhausted. During the collection phase all unreachable nodes were returned back to the allocation pool.

This implementation ensures that everything is lock-free. If one uses an underlying allocator to extend the allocation pool dynamically, then one should make sure that the underlying allocator is lock-free and does not allow unmapping of previously allocated pages (note that unmapping pages generally foils lock-freedom), then it is possible to consider other triggering schemes that adapt to the runtime behavior of the data-structure. In Listing 9 we present the code for allocating a node and the code for triggering a new collection phase.

6. Methodology and Results

We used the AOA reclamation mechanism with two widely used data-structures: Harris-Michael linked-list and a hash table. We compared the execution of these data-structures with the AOA mechanism to executions of these data-structures with the basic manual optimistic access mechanism. This scheme represents the most efficient manual reclamation available today. We also compared to executions with no reclamation and to executions with a reference counting mechanism. Although not stated by its author, the reference counting method of Valois [25], enhanced by Michael [17], can be applied automatically by the compiler if there is no cyclic garbage. Since cyclic garbage is not common in many lock-free data-structures, reference counting can be considered an alternative to the AOA method proposed in this paper. It offers comparable guarantees.

On commodity hardware, reading the node reference and incrementing the reference count is not atomic. If the node is

Listing 9. allocation and phase triggering

```
1  void *alloc_DS(threadData t){
2      if(t.localPool is empty){
3          t.localPool = allocationPool.pop(t.localPhase);
4          if(t.localPool is empty)
5              TriggerNewPhase(t);
6      }
7      return t.localPool.pop();
8  }
9
10 void TriggerNewPhase(threadData t){
11     CAS(&phase, t.localPhase, t.localPhase+1);
12     t.localPhase = phase;
13     //Initialize phase−protected variables
14     For each phase−protected variable V do
15         tmp = V
16         while(tmp.version<t.localPhase){
17             CAS(&V, tmp, ⟨0,t.localPhase⟩);
18             tmp = V
19         }
20     Set warning flag for all threads.
21     collectRoots(t.markstack);
22     markStage();
23     sweep();
24 }
```

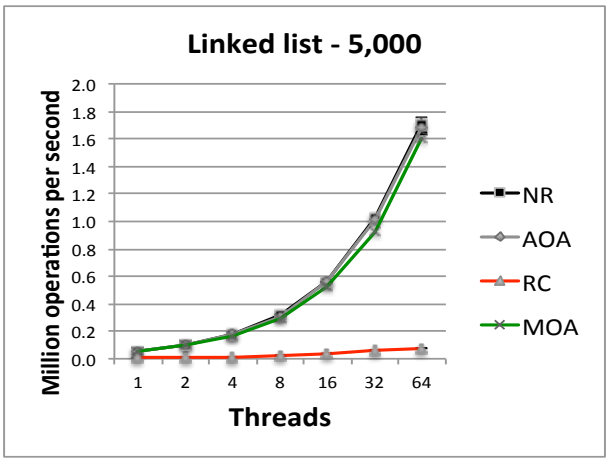

Figure 1. Comparing the throughput of the proposed AOA scheme, reference counting, and no reclamation with Harris-Michael linked-list with 5,000 items. The x-axis is the number of participating threads. The y-axis is the throughput of the presented scheme.

reclaimed after reading the reference and before incrementing its counter, the counter of the non-existing node is modified, possibly corrupting memory. Thus, Valois [25] assumed type-persistency: once a node is allocated at a memory location, the memory location will occupy only instances of the same class type as the first occupying node. Thus a belated thread modifies only a valid reference count field. The reference counting scheme is denoted RC. The manual optimistic access scheme is denoted MOA and the baseline algorithm that performs no memory reclamation is denoted NR.

Methodology. Following the tradition in previous work, we evaluated the data-structure performance by running a stressful workload that executes the data-structure operations repeatedly on many threads. In all our tests, 80% of the operations were read-only. For the linked-list data-structure we tested two configurations: one where the list length was 5000 denoted LinkedList5000, and another with list length 128 denoted LinkedList128. The hash table size was 10000, and the load factor was 0.75. These micro-benchmarks cover a wide range of behaviors. The LinkedList5000 has low contention as each operation is relatively long. The LinkedList128 has higher contention as the operations are of medium length. The hash has low contention, but extremely fast operations. Similar settings were used in previous work (e.g., [1, 4]).

To check the scalability of the proposed scheme, each micro-benchmark was executed with a varying number of threads, all of which were power-of-2's ranging between 1 and 64. Each execution lasted 1 second, and the total number of executed operations was recorded (throughput). Longer execution times (e.g., 10 seconds) produce similar results. Each execution was repeated 10 times, and the average throughput and 95% confidence levels are reported. For the AOA scheme we initialized the allocation pool with 32,000 nodes before the measurements began; a new collection phase is triggered when the allocation pool is exhausted.

The code was compiled using the GCC compiler version 4.8.2 with the -O3 optimization flag, running on an Ubunto 14.04 (kernel version 3.16.0) OS. The machine featured 4 AMD Opteron(TM) 6376 2.3GHz processors, each with 16 cores (64 threads overall). The machine used 128GB RAM, an L1 cache of 16KB per core, an L2 cache of 2MB for every two cores, and an L3 cache of 6MB per processor.

Results. In Figure 1 we compare the performance of the proposed AOA scheme, the manual optimistic access scheme (MOA), the reference counting scheme, and no reclamation, with Harris-Michael linked-list of length 5,000 nodes. It can be seen that the overhead of the AOA scheme, compared to no reclamation, is very low, and at max reaches 3%. Compared to the reference counting scheme, the AOA scheme improves the performance by $3x - 31x$. The reason is that reading memory is very lightweight in the AOA scheme, whereas the reference counting scheme requires two atomic operations per read. The overhead of the reference counting scheme is much higher as the number of threads grows, due to the contention on the reference counting field. The AOA scheme performed similar or slightly better than the manual optimistic access scheme.

In Figure 2 we compare the performance of the proposed AOA scheme, reference counting, manual optimistic access reclamation, and no reclamation, when used with the Harris-

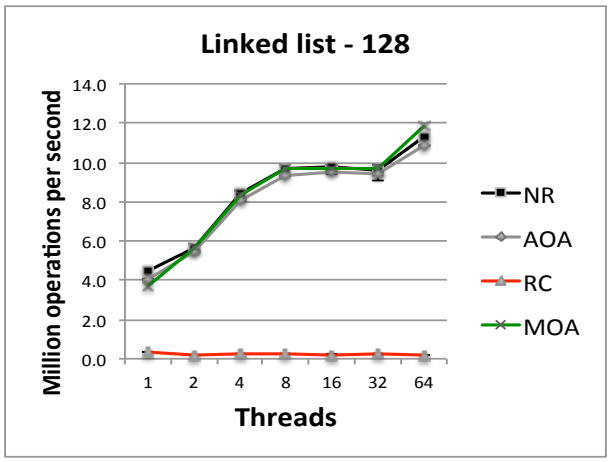

Figure 2. Comparing the throughput of the proposed AOA scheme, reference counting, and no reclamation, with Harris-Michael linked-list with 128 items. The *x*-axis is the number of participating threads. The *y*-axis is the throughput of the presented scheme.

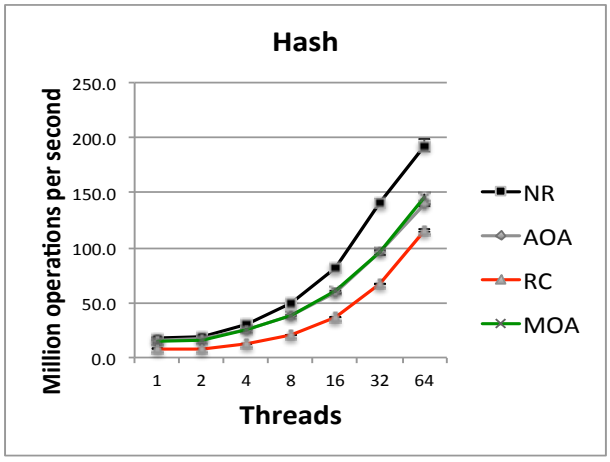

Figure 3. Comparing the throughput of the proposed AOA scheme, reference counting, and no reclamation, with the hash table. The *x*-axis is the number of participating threads. The *y*-axis is the throughput of the presented scheme.

Michael linked-list of length 128 nodes. The AOA scheme incurs an overhead of $0-10\%$ compared to the no reclamation scheme. Compared to the reference counting scheme, the AOA scheme improves performance by $6x-25x$. The AOA scheme performed similar to the manual optimistic access scheme: Comparing the AOA scheme with the manual optimistic access scheme shows that it was better by 9% for a single thread and slower by 9% for 64 threads. For a single thread the additional retire calls had some overhead, while for 64 threads these local computations probably reduced contention and improved scalability.

In Figure 3 we compare the performance of the proposed AOA scheme, reference counting, manual optimistic access reclamation, and no reclamation, when used with a hash table of size 10,000 nodes. The overhead of the AOA scheme, compared to the no reclamation, is $14\%-33\%$, which slightly increases as the number of threads increases. Compared to the reference counting scheme, the AOA scheme improves performance by $30\%-54\%$ for $1-32$ threads, and by 16% for 64 threads. With the hash micro-benchmark, the fraction of CAS instructions (vs. read instructions) is greater than in previous micro-benchmarks, so the reference counting cost of 2 atomic instructions per read is somewhat reduced. Furthermore, the read operations do not contend since each thread picks a random bucket, reducing the reference counting overhead in this case. The AOA scheme performed almost exactly like the manual optimistic access scheme. In fact it is difficult to see the gray line as it is almost completely hidden by the green one.

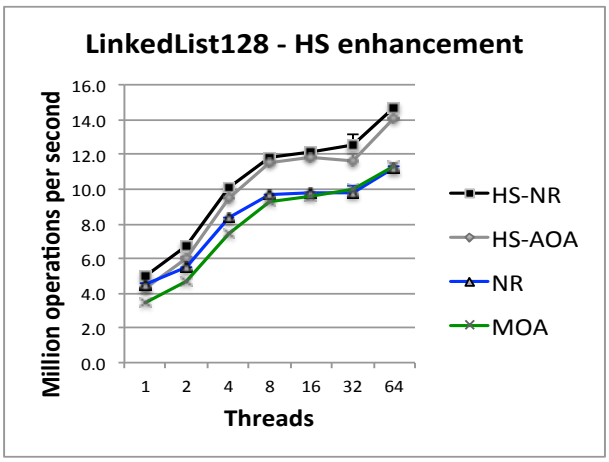

Figure 4. Studying the benefit of applying Herlihy-Shavit enhancement. NR and MOA stands for the baseline implementation with no reclamation (NR) and the manual optimistic access scheme. HS-NR and HS-AOA are the implementation enhanced by Herlihy-Shavit wait free searches with no reclamation and the AOA scheme. This enhancement does not satisfy the requirements needed to apply a manual memory reclamation scheme.

In all of the measurements so far, we used one data-structure implementation with different memory reclamation schemes. But some algorithms cannot be used with manual reclamation, yet, they can be used with automatic reclamation (as discussed in Appendix A). So we also compared the execution of the Harris-Michael linked-list which is amendable to manual memory reclamation to executions of the linked-list with wait-free searches of Herlihy and Shavit [9]. The latter cannot be used with manual reclamation. Each of these algorithms was also run with no memory reclamation for comparison. For the Herlihy-Shavit algorithm we considered NoRecl (HS-NR) and AOA (HS-AOA). And for the Harris-Michael linked-list we used NoRecl (NR) and the manual optimistic access scheme (MOA). The results are de-

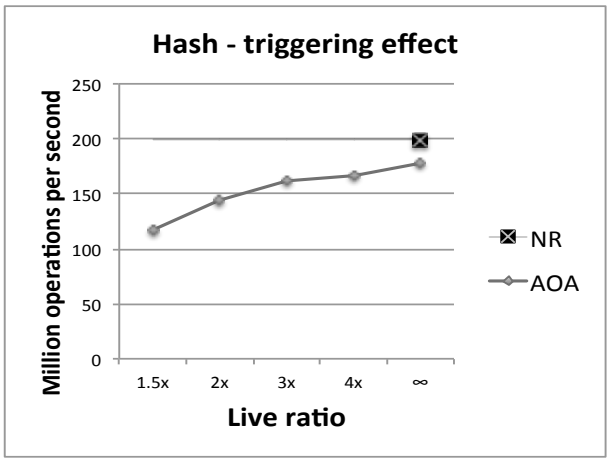

Figure 5. Studying the effect of triggering on performance. The *x*-axis is the number of nodes in the allocation pool divided by the number of live nodes. The *y*-axis is the throughput.

picted in Figure 4. The Herlihy-Shavit algorithm that cannot be used with manual reclamation performed better than the Harris-Michael algorithm, to which manual reclamation can be added. With no reclamation the difference is 9%-23%. Furthermore, the faster algorithm with the AOA scheme outperforms the slower algorithm even when the slower algorithm does not spend any time on memory reclamation.

In Figure 5, we study the effect of triggering on performance under infrequent recycling. The frequency of recycling is determined by the space reserved for allocation, which determines how much allocation can take place before recycling is needed. Similarly to the garbage collection literature, we set the available space for allocation as a multiplicative factor of the live space (nodes used by the data-structure). For this benchmark, we let the live space be 30,000 nodes and we vary the live ratio in the range $1.5x, 2x, 3x, 4x$ and ∞. Note that when we use ∞, we don't reclaim at all and so the overhead originates only from the read- and write-barriers. We compared these runs to runs with no reclamation (for which triggering is not relevant). In Figure 5 it can be seen that the AOA scheme is sensitive to the frequency of triggering, and performance improves if the triggering is infrequent. Looking at ∞ live ratio (no reclamation), we see that the overhead that the AOA scheme incurs on the execution without reclamation is 11%. A ratio of 2 compared to a ratio of ∞ demonstrates that the overhead of the reclamation itself in this case is 19%.

For the AOA scheme the memory overhead is a constant which our parameters determine to be around $32,000$ objects. For the NoRecl method the space overhead is of-course the largest in the long run. It is simply determined by the number of allocations because there is no reclamation. For the data structures at hand, allocations only happen during insertions, when the key to be inserted is not found in the data-structure. The number of such allocations is about 5% of the overall number of operations executed, because in our workload 10% of the operations are insertions and for approximately half the key is not found in the data structure.

The RC method exhibits a small memory overhead in most of the cases as it reclaims space with no delay. However, for the hash table with 64 threads the overhead was approximately 20,000 nodes. The reason is that in the implementation we used (of [17]), if the thread's local list of reclaimed nodes is empty, it immediately allocates a new node, even though other lists are full. While it is possible to improve the memory overhead for the RC scheme, we did not investigate this avenue further. The MOA memory overhead was similar to the AOA scheme almost everywhere except when running the hash benchmark with 64 threads. There, the reclamation overhead of the MOA scheme was high when the memory overhead was set to 32,000 nodes. The reason for this reduced performance for the manual scheme is that this scheme allocates a local pool for retired objects to each thread and also a local pool for local allocations. This reduces the number of objects that are available for allocation (for all threads) and results in excessive reclamation cycles. In addition, during a reclamation cycle, there is a high contention on the shared pool of retired objects. Thus, for this case we let the memory overhead be approximately 50,000 nodes.

We also recorded the number of memory reclamation phases performed by the AOA scheme. For the hash micro-benchmark, the number of phases varies between 32 for a single thread and 375.8 for 64 threads. For the LL5K micro-benchmark, the number of phases varies between 0 and 3. This is a long list and the number of operations that execute in one second with a single thread did not require memory reclamation. We did verify (for all benchmarks) that executions of 10 seconds show similar throughput behavior with 10 times the number of collections. For the LL128 micro-benchmark, the number of phases varies between 5 and 18.

To summarize the performance evaluation, the AOA scheme outperforms the reference count scheme on almost all configurations, and many times in a drastic manner. For moderate workloads, it demonstrates a very low overhead over the baseline algorithm that reclaims no nodes. The AOA performs similarly to the manual optimistic access scheme but the automatic scheme is easier to apply and it can be applied in cases where the manual reclamation methods cannot be applied.

7. Related Work

The first proposed lock-free memory reclamation scheme is the reference counting scheme [25]. Each node is associated with a count of the threads that access it and a node can be reclaimed if its reference count is dropped to 0. Race conditions may lead to a node being reclaimed immediately

before its reference count is incremented. Solving this race requires either a type persistence assumption [23, 25] or the use of the double compare-and-swap (DCAS) primitive [5] which is not always available. The performance overhead is high as this method requires (at least) two atomic operations per node read [7].

Sundell [23] extended the reference counting scheme to support wait-free execution. Wait-freedom is achieved by requesting help whenever a thread reads a pointer from shared memory. The proposed scheme guarantees that the read completes after finite number of instructions. The scheme requires at least 4 atomic instructions per read and no measurements were provided.

The most popular lock-free reclamation schemes are the *hazard pointers* and *pass the buck* mechanisms of Michael [16] and Herlihy et al. [10]. In these schemes every thread is associated with a set of pointers, denoted *hazard pointers* or *guards*, which mark nodes the thread is accessing. These schemes defined the manual memory reclamation interface. Loosely speaking, the following is required: first, a node can be accessed only if it is linked to the data-structure or the thread holds a hazard pointer to the node. Second, the programmer may retire a node only after it is unlinked from the data-structure. These two properties implies that a node can be reclaimed after it was retired and there is no hazard pointer that points it. Except for the complexity of using the manual memory reclamation interface, the runtime cost of these methods is also non-negligible. In these schemes a node is accessed only after writing its address to a hazard pointer; on the TSO (or weaker) memory model, this requires a write and a memory fence per read node. Still, the hazard pointers scheme is scalable since each thread writes only its own hazard pointers so concurrent readers do not contend.

Braginsky et al. [3] proposed the *anchor* scheme as an improvement to the hazard pointer scheme. The anchor scheme writes to hazard pointers only once per k reads, and the hazard pointer implicitly prevents the next $O(k)$ nodes from being reclaimed. They use time stamps and a freezing mechanism to recover from stuck threads. The anchor method significantly reduces the overhead of memory fences. However, it is required to design an anchor version for each data-structure, and the authors only provided an example implementation for the linked-list.

Alistarh et al. [1] proposed the StackTrack method, which utilizes hardware transactional memory to solve the memory reclamation problem. This method breaks each operation to a series of transactions and writes its hazard pointers only once per transaction. It is shown that successfully committed transaction cannot interfere with a memory reclamation. The method is automatically applicable by a compiler, but the programmer is still required to provide retire statements.

Cohen and Petrank [4] have recently proposed the OA scheme. The OA scheme allows threads to optimistically read reclaimed nodes, thus avoiding the need to write hazard pointers for every read. A thread writes its hazard pointers only when the thread writes to shared memory. The OA scheme is faster than both the hazard pointers and the anchor methods, and is slightly simpler to apply than the hazard pointers method. Still, the programmer is required to retire nodes.

8. Conclusions

In this paper we presented the AOA automatic lock-free memory reclamation scheme. Unlike *manual* memory reclamation schemes, the AOA scheme does not require the programmer to manually install *retire* instructions for nodes after unlinking them from the data-structure. As exemplified for the linked-list and the skip-list, installing retire instructions is non-trivial. It requires changing the original algorithms and a very subtle understanding of the original concurrent algorithms. Therefore, using the automatic scheme proposed in this paper significantly simplifies programmer efforts for memory reclamation of lock-free data-structures. Measurements show that the AOA scheme significantly outperforms the competing reference-counting scheme, it is comparable to the best known manual lock-free memory reclamation and it incurs moderate overhead when compared to executions that do not reclaim memory at all.

References

[1] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *EuroSys*. ACM, 2014.

[2] J. Auerbach, D. F. Bacon, P. Cheng, D. Grove, B. Biron, C. Gracie, B. McCloskey, A. Micic, and R. Sciampacone. Tax-and-spend: Democratic scheduling for real-time garbage collection. In *EMSOFT*, pages 245–254, 2008.

[3] A. Braginsky, A. Kogan, and E. Petrank. Drop the anchor: lightweight memory management for non-blocking data structures. In *SPAA*, pages 33–42. ACM, 2013.

[4] N. Cohen and E. Petrank. Efficient memory management for lock-free data structures with optimistic access. In *SPAA*. ACM, 2015.

[5] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele Jr. Lock-free reference counting. *DISC*, pages 255–271, 2002.

[6] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, pages 300–314. Springer, 2001.

[7] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *JPDC*, pages 1270–1285, 2007.

[8] M. Herlihy. Wait-free synchronization. *TOPLAS*, 1991.

[9] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[10] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *TOCS*, 23(2):146–196, 2005.

[11] M. P. Herlihy and J. E. B. Moss. Lock-free garbage collection for multiprocessors. *TPDS*, 1992.

[12] R. L. Hudson and J. E. B. Moss. Sapphire: Copying GC without stopping the world. In *Joint ACM-ISCOPE Conference on Java Grande*, pages 48–57, 2001.

[13] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *PPoPP*, pages 223–234. ACM, 2011.

[14] A. Kogan and E. Petrank. A methodology for creating fast wait-free data structures. In *PPoPP*, pages 141–150. ACM, 2012.

[15] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82. ACM, 2002.

[16] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *TPDS*, 15(6):491–504, 2004.

[17] M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical report, DTIC Document, 1995.

[18] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-tso. In *Theorem Proving in Higher Order Logics*, pages 391–407. Springer, 2009.

[19] E. Petrank. Can parallel data structures rely on automatic memory managers? In *MSPC*, pages 1–1. ACM, 2012.

[20] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgard. Stopless: A real-time garbage collector for multiprocessors. In *ISMM*, pages 159–172, 2007.

[21] F. Pizlo, E. Petrank, and B. Steensgaard. A study of concurrent real-time garbage collectors. In *PLDI*, pages 33–44, 2008.

[22] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *PLDI*, pages 146–159, 2010.

[23] H. Sundell. Wait-free reference counting and memory management. In *IPDPS*, pages 24b–24b. IEEE, 2005.

[24] S. Timnat and E. Petrank. A practical wait-free simulation for lock-free data structures. In *PPoPP*, pages 357–368. ACM, 2014.

[25] J. D. Valois. Lock-free linked lists using compare-and-swap. In *PODC*, pages 214–222. ACM, 1995.

A. Motivating Examples

In this section we demonstrate the difficulty of applying manual memory management with *retire* instructions in order to motivate automatic memory management. We look at two basic and well-known lock-free data-structure algorithms: Harris's linked-list [6] and Herlihy and Shavits' skip-list [9]. For both data-structures we show that non-trivial algorithmic modifications are required in order to place retire instructions and apply a manual memory reclamation method. We then discuss possible solutions for making these algorithms amendable to manual memory reclamation.

A.1 Harris's Linked-List [6]

Harris's linked-list [6] was innovative when introduced and is still considered one of the most simple and efficient lock-free data-structures. Harris also proposed the Epoch-Base Reclamation (EBR) scheme for reclaiming nodes in the list, but using Epoch-Base reclamation foils lock-freedom as it is not robust to thread failures.

Michael [15] discusses the difficulty of node reclamation for Harris' list and argues that it is not possible to apply the hazard pointers method to this data-structure implementation. The reason is that in order to apply hazard pointers or a similar reclamation method a retire operation must satisfy a condition that cannot be satisfies by Harris' list. The condition is that one issues a retire operation on node N only after N has been disconnected from the list and no thread will obtain access to it from now on. It is possible that a thread is looking at N simultaneously with the execution of the retire instruction and this case is fine because hazard pointers will be used to delay the actual reclamation of N. But no thread should be able to gain access to N after the retire instruction was issued. This is important because N may be now reclaimed and future accesses may view stale values.

Let us review why the condition is not satisfied in Harris' algorithm and then present Michael's fix. Note that this understanding and modification requires a subtle (i.e., deep) understanding of the algorithm and of the memory reclamation scheme.

Deleting a node N in Harris's linked-list happens in two stages. First, N's next pointer is marked (to prevent further modifications). Then, the node is physically deleted by rerouting the next pointer of N's preceding node to point to the node that follows N in the list. Concurrence may cause several subsequent nodes to be marked before physical deletion happens to any of them. Handling a series of consecutive marked nodes is a major issue for the installation of retire operations. Let us explain why. In Harris's implementation, if two (or more) consecutive nodes are marked, then they are simultaneously deleted physically by rerouting the next pointer of the node that precedes all these consecutive marked nodes to point to the node that follows the series of marked nodes. This seems a natural way to deal with a series of marked nodes and it is also a very efficient solution. But the simultaneous physical deletion makes the installation of hazard pointers fail because it foils the condition needed to apply the hazard pointers scheme. Suppose a thread is currently accessing one of the nodes in the sequence of marked nodes. Then this thread is able to access other nodes in this sequence even after they are physically deleted from the list. This is bad news for the installation of retire instructions. This thread may access a node that has been retired.

Let us describe specific race condition. Suppose that A,B,C,D are consecutive nodes in the list $A \rightarrow B \rightarrow C \rightarrow D$. T_1 starts a search, and reaches B. T_2 marks C, and then T_3 marks B. Next, T_4 notices that B and C are marked and physically deletes B and C by connecting $A \rightarrow D$. At this point B and C are retired. Then, both T_2 and T_3 discovered that B and C were unlinked and restart the operation, cleaning their

own hazard pointers to the nodes B and C. Since C was retired and no hazard pointer references it, it is next reclaimed. Now T_1 wakes up, reads B's next field (C), protects it with a hazard pointer, revalidates that B still points to C, and then accesses C in spite of the fact that C has been reclaimed and may contain stale values for T_1.

This race condition is rather tricky and non-expert developers may attempt to apply hazard pointers to this scheme. The resulting bug reveals itself only under rare conditions, which complicates debugging.

Herlihy and Shavit [9] proposed a variant of the Harris-Michael algorithm where the *search* (a.k.a. *contains*) operations do no participate in physical deletion. This yields faster and wait-free searches. But their variant also does not enable the installation of hazard pointers due to similar races.

Michael's solution to this problem is to physically delete one node at the time, starting at the first one. In addition, Michael makes sure that a thread does not traverse through a marked pointer without attempting to handle the physical deletion and starting from scratch if the delete fails. Michael's solution allows memory reclamation, and prior to this work there was no way around Michael's condition for inserting retire statements. But as shown in our measurements, Michael's solution also has a performance cost due to threads starting searches from scratch.

A.2 Herlihy and Shavits' [9] Skip-List

Skip-list is a widely used data-structure as it allows a simple implementation that (probabilistically) achieves a logarithmic execution time for the insert, the delete and the search operations. The lock-free skip-list algorithm of Herlihy and Shavit is much simpler than the competing lock-free tree algorithms.

The skip-list is composed of a sequence of sorted linked-lists. In the lock-free implementation [9] these lists are implemented according to the Harris-Michael linked-list [15]. The lowest linked-list contains all the elements in the skip-list. Higher linked-lists contain only a subset of the elements, where on average, the size of the i^{th} linked-list is half the size of the $i-1^{\text{th}}$ linked-list. Each element of the skip-list has a height, and an element of height ℓ participates in the first ℓ linked-lists. Element insertion begins with the thread inserting the element to the bottom level and then proceeding with the insertions to the following levels, one level at a time. Deleting an element works the other way, starting from the linked-list of the element's height and finishing at the bottom level. As in Harris-Michael linked-list, the element is first logically deleted, and then (after all levels are logically deleted) it is disconnected from the list.

This implementation does not satisfy the conditions needed to apply the hazard pointers (or any other similar) manual memory reclamation scheme. Consider an element with a height 2, but any height higher than 1 will do. The main difficulty here is that such an element is inserted *twice* to the data-structure. Once into the bottom level and then into the second level. If a thread is delayed before installing the last reference to this node, then it is unsafe to retire this node, because a reference to this node might be later installed into the skip-list. If one attempts to retire such a node, than there is a scenario in which a retired node is reached by a thread after it was reclaimed.

Let us describe a specific race condition. Suppose a thread gets delayed in the middle of inserting an element E of value X and height 2 into the data-structure. The thread has already inserted E to the bottom level but gets delayed before linking it to the second level. At this time, another thread starts a delete operation of value X, finds E, deletes it from the skip-list, then unlinks it. At this point it is natural to put a retire statement, with which the deleting thread notifies the reclamation system about E, and the reclamation system may in turn recycle the node and allocate it again as E'. Next, the inserting thread wakes up and installs the reference to E, which actually references E', into the skip-list. This corrupts the data-structure. Such a corruption is very difficult to debug.

Let us now describe a possible modification of the algorithm that enables the installation of retire statements, and so also the application of the hazard pointers [16] or the OA [4] memory schemes. The inserting thread prevents recycling of the currently inserted item using some methodology (a hazard pointer or some other coordination method) until the insertion operation is complete. When it is complete, the inserting thread checks whether the node was concurrently deleted. If it was, the inserting thread unlinks it from the data-structure. Thus, when the node becomes reclaimable, it is guaranteed not to be reachable from the data-structure. This solution is delicate and requires expertise and care about possible races. The AOA reclamation scheme is automatic and does not require such modifications.

B. Normalized Representation of Data Structures

In this section we provide the formal definition of normalized data-structure implementations [24]. Similar to [4], we relax the original definition a bit, because we can also work with data-structures that do not satisfy all the constraints of the original definition. Of-course, the optimistic access memory scheme will work with the original definition of [24] as well, but the relaxed restrictions that we specify below suffice.

Definition 4.1 of [24] (slightly relaxed): A lock-free algorithm is provided in a normalized representation if:

- Any modification of the shared data-structure is executed using a CAS operation.

- Every operation of the algorithm consists of executing three methods one after the other and which have the following formats.
  1. CAS Generator: its input is the operation's input, and its output is a list of CAS descriptors. CAS descriptors are tuples of the form (address, expectedVal, newVal). This method is parallelizable (see Definition 3.4 of [24] below).
  2. CAS Executor, which is a fixed method common to all data-structures and all algorithms. Its input is the list of CAS descriptors output by the CAS generator method. The CAS executor method attempts to execute the CASes in its input one by one until the first one fails, or until all CASes complete. Its output contains the list of CAS Descriptors from its input and the index of the CAS that failed (which is zero if none failed).
  3. Wrap-Up, whose input is the output of the CAS Execution method plus the operation's input. Its output is either the operation result, which is returned to the owner thread, or an indication that the operation should be re-executed from scratch (from the Generator method). This method is parallelizable (see Definition 3.4 of [24] below).

We remark that in [24] there is an additional requirement for a contention failure counter and for versioning that we do not need for our construction. This makes the above definition more relaxed. Of-course, the proposed memory management scheme will work well also when the data-structure representation adheres to the full (stricter) definition of [24].

As discussed in [24], any data-structure has a normalized lock-free implementation, but not necessarily efficient. However, interesting data structures had a small (often negligible) overhead.

We provide the definition of parallelizable methods below. An important property of parallelizable methods is that at any point during their execution, it is correct to simply restart the method from its beginning (with the same input). We will use this fact by starting the CAS generator and the wrap-up methods from the beginning whenever we detect stale values that were read optimistically following a concurrent reclamation.

The algorithm we propose is optimistic. Optimistic algorithms typically execute instructions even when it is not clear that these instructions can be safely executed. In order to make sure that the eventual results are proper, optimistic algorithms must be able to roll back inadequate execution, or they stop and check before performing any visible modification of the shared memory that cannot be undone. In the AOA scheme, rolling back is done by restarting from the beginning of the method (Generator or Wrap-Up). In the rest of the paper, the instruction *restart* refers to restarting from the beginning of the currently executed method (which will always be the Generator or the Wrap-Up).

To simplify the discussion of a restart, we assume that the CAS generator and wrap-up methods do not invoke methods during their execution. The reason is that if a method is invoked by the CAS generator (for example), restarting the CAS generator from scratch, when a check in the invoked method detects stale values, implies returning from all invoked methods, removing their frames from the runtime stack without executing them further. Note first that this is achievable by letting the invoked routines return with a special code saying that a restart is required and the calling method should act accordingly. Note also that inlining can be applied to create a method that does not invoke other methods if no recursion is used and no virtual calls exist.

Additionally, we assume that the executions of the original data structure (with no memory management) do not trigger a trap. Adding checks and handling restarts in a trap code are more involved and are outside the scope of the current paper.

Before starting with the definition of parallelizable methods, we start by defining an avoidable execution of a method (3.3 of [24]). An avoidable execution of a method is an execution that can be rolled back. Such an execution should not perform any modification visible by other threads, otherwise it cannot be rolled back. For completeness we also provide Definition 3.1 and Definition 3.2 of [24] below.

Definition 3.1 of [24] (Futile CAS) *A futile CAS is a CAS in which the expected value and the new value are identical.*

Definition 3.2 of [24] (Equivalent Executions) *Let I_1 and I_2 be two (different) implementations of a data-structure D. Let E_1 be an execution over I_1 and let E_2 be an execution over I_2. Then the executions are* equivalent *if the following hold:*

Results: *In both executions all threads execute the same data-structure operations and receive identical results.*

Relative Operation Order: *The order of invocation points and return points of all data-structure operations is the same in both executions.*

Definition 3.3 of [24] (Avoidable Method Execution) *A run of a method M by a thread T on input I in an execution E is avoidable if each CAS that T attempts during the execution of M is avoidable in the following sense. Let S_1 denote the state of the computation right before the CAS is attempted by T. Then there exists an equivalent execution E' for E such that both executions are identical until reaching S_1, and in E' the CAS that T executes in its next step (after S_1) is either futile or unsuccessful. Also, in E' the first execution step from S_1 is executed by a thread who is the owner of an ongoing operation.*

We now recall the definition of parallelizable methods, i.e., Definition 3.4 of [24] and then explain why it implies that restarting from the beginning of the method is fine.

Parallelizable Method (Definition 3.4 of [24]). *A method M is a parallelizable method of a given lock-free algorithm, if for any execution in which M is called by a thread T with an input I the following two conditions hold. First, the execution of a parallelizable method depends only on its input, the shared data-structure, and the results of the method's CAS operations. In particular, the execution does not depend on the executing thread's local state prior to the invocation of the parallelizable method. Second, at the point where M is invoked, if we create and run a finite number of parallel threads, each one executing M on the same input I concurrently with the execution of T, then in any possible resulting execution, all executions of M by the additional threads are avoidable.*

Now we show that a parallelizable method can be restarted from scratch. Consider an execution where a thread T restarts a method M a finite number of times, each time with the same input. The method execution does not depend on the thread's local state, so there exists an equivalent execution where each invocation is executed by a different (auxiliary) thread. Only the last invocation (which never restarts) is considered executed by T, which is the owner thread for the ongoing operation. The auxiliary threads do not finish to execute the method. Thus, consider these threads as crashing at the point where a restart is executed. By definition of parallelizable method, there exists an equivalent execution where the auxiliary threads execution of M is avoidable. But then the auxiliary threads execute no modification observable by other threads, which is equivalent to having T execute the method alone with no restarts.

C. The AOA Interface

Let us now describe the interface between the AOA scheme and the data-structure it serves. The AOA scheme is implemented as a header file in C. The rest of this paper explains how to implement the functionalities included in this header file. But before including the header file in the data-structure's source code, several #define statements must be provided by the programmer. These #define statements are discussed below.

The programmer starts by providing a name space for the data-structure to be used by the AOA scheme for this data-structure and to avoid collision of AOA actions on different data-structures. This name space will be used as a suffix for all the AOA function names. The name space (denoted DS) is provided using a #define statement. The second definition provided by the programmer is the NODE_SIZE parameter that contains the size of the data-structure node. The third define statement specifies NCHILDREN that contains the number of link fields in the data-structure node. In our simple implementation NCHILDREN must be between 1 and 3, but it is easy to support any constant number of links. The fourth set of (NCHILDREN) definitions is CHILD_OFFSET_$\{1,2,3\}$. The definition CHILD_OFFSET_i specifies the offset of the i^{th} link. The fifth (optional) definition specifies a function UNMARK(ptr) that gets a pointer and returns an unmarked pointer. If undefined, the implementation used the default of clearing the two least significant bits.

Listing 10. Harris-Michael linked-list

```
1  #include "hmll.h" //the data−structure definitions
2  ---------------- AOA memory manager
3  #define DS HMLL //Harris−Michael linked−list
4  #define NODE_SIZE sizeof(struct node)
5  #define NCHILDREN 1
6  #define CHILD_OFFSET_1 offsetof(struct node, next)
7  //default (unneeded) definition
8  //#define UNMARK(ptr) ((void*)((((long)(ptr))&(~3ul)))
9  #include "aoa.h" // supplied by the AOA implementation
10 ---------------- Implementation
11 void insert(...){
12     ...
13     struct node *newnode=alloc_HMLL(threadData[tid]);
14 }
15 void init(){
16     //initialize HMLL memory manager
17     init_allocator_HMLL(threadData, numThreads);
18     //add global roots
19     addGlobalRoot_HMLL(&list1head, 1);
20     addGlobalRoot_HMLL(&list2head, 1);
21     ...
```

Each read and write of the original algorithm operations must be modified in a way that is named read- and write-barrier in the memory management nomenclature. This can be done automatically by a compiler, but in our simple implementation we let the programmer apply these additions. All allocations of data-structure nodes must be handled by the AOA functionality and this should be done using the alloc_DS function (where DS is the name space of the said data-structure). The alloc_DS function returns a newly allocated node.

An initiation function denoted init_allocator_DS must be called during the initiation of the data-structure to initiate the memory manager. The destroy_DS function may be invoked to de-allocate the AOA internal data-structure when it is no longer needed. The function addGlobalRoot_DS is used to declare the location of global roots. It is possible to declare an array of roots by calling this function once and providing the length of the array. (That may be useful for a hash table.)

Harris-Michael Linked-List Example Let us now exemplify the user of the AOA interface on the Harris-Michael linked-list. The example is provided in Listing 10.

Applicability to Multiple Data-Structures It is possible to apply the AOA scheme to multiple instances of a data-

structure by providing the global roots of all instances via the AOA interface. It is also possible to apply the AOA scheme to multiple data-structures. In this case, each data-structure must be compiled separately (on a different source file) and each data-structure must provide a unique name space. In our implementation there is no sharing between multiple instances of the AOA scheme, even though such sharing may be possible (e.g. using a single set of hazard pointers). Of course, the AOA scheme does not prohibit using different memory reclamation schemes on other data-structures. But if the data-structure uses the AOA interface, it is incorrect to allocate an instance without registering it properly.

D. Some Words on Correctness

D.1 Root Collection (Subsection 5.2)

In this subsection we further explain why it is sufficient to collect the hazard pointers instead of the local roots. To this end, we relax the responsibility of the root collecting stage in two ways. Suppose a thread has a local reference to a node R. First, consider the case that after root collecting the thread executes several instructions and rolls back to a safe point, and during this period (between root collecting and rolling back) R does not participates in accesses to shared memory. If R is not referenced at the safe point, the root collection may ignore this root so it can be reclaimed. Second, suppose that the thread reads the node R, but immediately drops the read value and rolls back the execution to a safe point. If the read instruction does not trigger a trap or the trap is caught, the read can be considered a no-operation (specifically, an operation that does not access shared memory). Next we argue that collecting the hazard pointers and the global roots provides the same set of roots as collecting all node pointers in thread's local state after the two relaxations above.

Root collecting starts by setting the warning flag for all threads. By Listings 2, 3 and 4, the threads execute the data-structure operation until before writing to shared memory or after reading shared memory. After this point the thread helps with the collection efforts. Combining the two relaxations discussed above, a root R should be considered only if

1. R is still referenced at the safe point;
2. R participates in a memory write (i.e. CAS) operation.

Let us start with the latter. Before writing to shared memory during the CAS generator or the wrap-up routines, all local roots participating in the write (i.e. the written node, the expected value, and the new value) are exposed via a set of hazard pointers. The write is executed only after verifying that the roots were exposed before a memory reclamation phase begin. This set of hazard pointers is called *WriteHPs*. Similarly, before writing to shared memory in the CAS executor, all local roots participating in the write are exposed via the SafePointHPs set of hazard points.

Next we consider roots that are referenced from safe points. We define two safe points in each operation. The first is at the beginning of the CAS generator method and the second is at the beginning of the wrap-up method. The CAS generator method depends only on its input, which is passed from outside the data-structure. Since the input cannot contain a node reference unless it is protected by a global root, the thread is not required to publish anything via hazard pointers. The safe point at the beginning of the CAS generator is applicable (can be jumped into) during the entire execution of the CAS generator.

The second safe point is the beginning of the wrap-up method; it is applicable during the entire execution of the wrap-up routine. The thread is required to publish all roots referenced from this safe point during the execution of the wrap-up routine. The inputs to the wrap-up routine are the CAS list generated by the CAS generator and the operation input. As discussed above, the operation input does not contain roots. So during the execution of the wrap-up routine, all references in the CAS list must be published via the SafePointHPs set of hazard pointers. Next note that during the execution of the CAS executor it is not allowed to roll back the execution, so all roots must be published during the entire execution of the CAS executor. Since the CAS executor does not read any pointer from shared memory, the set of roots are not modified and are equal to the set of roots in the method input, which is the CAS list generated by the CAS generator routine. To summarize, during the execution of the CAS executor and the wrap-up routine, the roots in the CAS list generated by the CAS generator are public via the SafePointHPs set of hazard pointers.

D.2 The Mark Stage (Subsection 5.3)

Before discussing the correctness of the presented mark stage, let us first describe some challenges that the mark stage handles. The first challenge in the marking stage is to avoid missing nodes even when a thread drops dead in the middle of marking a node. The second challenge is to avoid tracing corrupted addresses when a tracing thread wakes up and is acting in the presence of a subsequent marking phase that it is not yet aware of.

To show that the tracing routine handles the first challenge, we declare the following invariant: during the execution of the mark stage, the children of a marked node are either marked, resides on some (maybe another thread's) mark-stack, or resides on some curTraced variable. This way, nodes are never lost during a trace. Let us explain why the tracing procedure (Listing 6) satisfies the declared invariant. We start by assuming that threads do never modify links or global roots during the mark stage and handle this case later. Then the invariant follows from three simple properties of the tracing algorithm. First, a node is popped from the mark-stack only after it is inserted to the curTraced position in the mark-stack. Second, a node is removed from the curTraced position only after it was marked. Third, a node is

marked (by a thread T) only after all its children resides on T's mark-stack. These three properties implies that a node is removed from the mark-stack only after all its children resides on a mark-stack of some thread. The argument is finished by noting again that the children are removed from a mark-stack only after being marked.

Next we describe why data-structure operations cannot foil the invariant. To this end we need to declare another invariant regarding modification of shared memory (links and global roots) during ongoing reclamation phase. The invariant is as follows: if a thread T' modifies shared memory after T started the mark stage of the current phase, then T observed both the old value and the newly modified value when collecting T' hazard pointers. This invariant implies that if a link is modified during the marking stage the new value is a root, thus satisfying the tracing invariant.

Let us now explain why this second invariant holds. Root collection starts by raising the threads warning flags, then all hazard pointers are collected, and eventually global roots and links are traversed. Before a write began, the thread installs all participating pointers into its hazard pointers (including the new and old values). Then it checks whether the warning flag was set or not; if the warning flag is set the thread does not execute the write but rather joins the collection efforts. If the thread T' modifies a link during the time T executes the mark stage, then surely T''s warning flag is set and it was not set after T' finished to install the hazard pointers and check the flag. Since T collected T' hazard pointers after the warning flag of T' was set, it is guaranteed that T observed both the old and the newly modified value.

Finally let us explain why a corrupted address is never traced. If a node is traced, then the node address was obtained and then the parent was marked. The key observation here is that the mark-bit table is phase protected, so marking a node implies that the thread observes the correct phase. Since the node was obtained before the parent was marked, it surely points to a valid node during the current phase. By our assumption on the underlying allocator, the node content can be accessed even after arbitrarily number of phases. Note that it does not imply that the node location contains a valid node; during the time the node is traced it may contain unrelated data (i.e. corrupted children). But the thread will fail to mark the node and will remove the node's children from its mark-stack. In the node is referenced from a global root, the argument follows since global roots always points to valid nodes. When helping other threads, the argument follows since a helper thread checks whether the helped thread executes the current phase and whether the helping thread executes the correct phase. If the phase is incorrect, helping is not provided as the address may be corrupted.