Efficient Memory Management for Lock-Free Data Structures with Optimistic Access*

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 achieve high responsiveness, aid scalability, and avoid deadlocks and livelocks. But providing memory management support for such data structures without foiling their progress guarantees is difficult. Often, designers employ the hazard pointers technique, which may imposes a high performance overhead.

In this work we propose a novel memory management scheme for lock-free data structures called *optimistic access*. This scheme provides efficient support for lock-free data structures that can be presented in the normalized form of [23]. Our novel memory manager breaks the traditional memory management invariant which never lets a program touch reclaimed memory. In other words, it allows the memory manager to reclaim objects that may still be accessed later by concurrently running threads. This broken invariant provides an opportunity to obtain high parallelism with excellent performance, but it also requires a careful design. The optimistic access memory management scheme is easy to employ and we implemented it for a linked list, a hash table, and a skip list. Measurements show that it dramatically outperforms known memory reclamation methods.

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. *Lock-free* data structures (a.k.a. *non-blocking*) [10, 12] are immune to deadlocks and livelocks, fast, scalable and widely used in practice. However, when designing a dynamic non-blocking data structure, one must also address the challenge of memory reclamation. The problem arises when one thread attempts to reclaim an object while another thread is still using it. Accounting for all accesses of all threads before reclaiming an object is difficult and costly, especially when threads may be delayed for a while while still holding pointers to nodes in the shared memory.

One easy approach to this problem is to not reclaim memory at all during the execution. But this solution is only applicable to short-running programs. Another approach to reclaiming memory is to assume automatic garbage collection, which guarantees that an object is never reclaimed while it is being used. However, this only delegates the problem to the garbage collector. There has been much work on garbage collectors that obtain some partial guarantee for progress [13, 14, 19, 20, 2, 21], but current literature offers no garbage collection that supports lock-free execution [18].

A different approach is to coordinate the accessing threads with the threads that attempt reclamations. The programmer uses a memory management interface to allocate and reclaim objects and the reclamation scheme coordinates the memory recycling of reclaimed objects with the accessing threads. The most popular schemes of this type are *hazard pointers* and *pass the buck* [16, 11]. These (similar) methods require that each thread announces each and every object it accesses. To properly announce the accessed objects, a memory fence must be used for each shared memory read, which is costly. Employing one of these schemes for a linked list may slow its execution down by a factor of 5 [3]. To ameliorate this high cost, a recent extension by Braginsky et al. [3] proposed the *anchors* scheme, which is a more complex method that requires a fence only once per several accesses. The *anchors* scheme reduces the overhead substantially, but the cost still remains non negligible. Furthermore, the *anchors* scheme is difficult to design and it is currently available for Harris-Maged linked list [15] data structure only.

All known memory management techniques, including garbage collection and the above ad-hoc reclamation methods, provide a guarantee that a thread never accesses a reclaimed object. Loosely speaking, supporting this guarantee causes a significant overhead, because whenever a thread reads a pointer to an object, the other threads must become aware of this read and not reclaim the object. For an arbitrary program, this might mean a memory fence per each read, which is very costly. For more specialized programs or for specific lock-free data structures, better handling is possible, but a substantial performance penalty seems to always exist.

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

SPAA'15, June 13–15, 2015, Portland, OR, USA.

http://dx.doi.org/10.1145/2755573.2755579..

In this paper we propose to deviate from traditional methods in a novel manner by letting the program execute optimistically, allowing the threads to sometimes access an object that has been previously reclaimed. Various forms of optimistic execution have become common in the computing world (both hardware and software) as a mean to achieve higher performance. But optimistic access has never been proposed in the memory management literature due to the complications that arise in this setting. Optimistically accessing memory that might have been reclaimed requires careful checks that must be executed at adequate locations; and then, proper measures must be taken when the accessing of a reclaimed object has been detected. When a thread realizes that it has been working with stale values, we let it drop the stale values and return to a point where the execution is safe to restart.

Achieving such timely checks and a safe restart in this setting is quite difficult for arbitrary lock-free programs. Therefore, we chose to work only with lock-free data structures that can be presented in a *normalized form*. We used the normalized form proposed in [23]. This normalized form is on the one hand very general: it covers all concurrent data structure that we are aware of. On the other hand, it is very structured and it allows handling the checks and restarts in a prudent manner. As with other optimistic approaches, we found that the design requires care, but when done correctly, it lets the executing threads run fast with low overhead. We denote the obtained memory reclamation scheme *optimistic access*.

Measurements show that the overhead of applying the optimistic access scheme is never more than 19% compared to no reclamation, and it consistently outperform the hazard pointers and anchors schemes. Moreover, the application of the optimistic access method to a normalized lock-free data structure is almost automatic and can easily be applied to a given data structure. The optimistic access mechanism is lock-free and it may reclaim nodes even in the presence of stuck threads that do not cooperate with the memory reclamation process.

In order for the optimistic access to be possible at all, the underlying operating system and runtime are required to behave "reasonably". The specific required assumptions are detailed in Section 3. Loosely speaking, the assumption is that reading or writing a field in a previously allocated object does not trigger a trap, even if the object has been reclaimed. For example, a system in which a reclaimed object is returned to the operating system and the operating system unmaps its memory thereafter, is not good for us since reading a field of that object would create a segmentation fault and an application crash.[1] It is easy to satisfy an adequate assumption by using a user-level allocator. This may be a good idea in general, because a user-level allocator can be constructed to provide a better progress guarantee. For example, using an object pooling mechanism for the nodes of the data structure would be appropriate.

The main contribution of this paper is an efficient memory reclamation scheme that supports lock-freedom for normalized lock-free data structures. The proposed scheme is much faster than existing schemes and is easy to employ. We exemplify the use of the optimistic access scheme on a linked list, a hash table, and a skip list. The obtained memory recycling scheme for the **skip list** incurred an overhead below 12%, whereas the overhead of the hazard pointers scheme always exceeded a factor of 2. For the **hash** table, the optimistic access scheme incurred an overhead below 12%, whereas the overhead of the hazard pointers method was $16\% - 40\%$ for $1 - 32$ threads (and negligible for 64 threads). For **linked list**, the optimistic access method always outperforms the hazard pointers and the anchors mechanisms. The optimistic access method typically incurs an overhead of a few percents and at a worst setting it incurs an overhead of 19%. The hazard pointers mechanism typically incurs a large overhead of up to 5x. The anchors mechanism improves performance significantly over the hazard pointers but with short lists and high contention it incurs a significant overhead as well.

This paper is organized as follows. In Section 2 we present an overview of the optimistic access mechanism. In Section 3 we specify the assumption we make of the underlying system. Section 4 presents the optimistic access mechanism. In Section 5 we present the implementation and an evaluation. We discuss related work in Section 6 and conclude in Section 7.

2. OVERVIEW

Let us start with an intuitive overview of the optimistic access scheme. The main target of this scheme is to provide fast reads, as reads are most common. In particular, we would like to execute reads without writing to the shared memory. On the other hand, a lock-free memory reclamation scheme must be able to reclaim memory, even if some thread is stuck just before a read of an object that is about to be reclaimed. Thus, we achieve fast reads by allowing a thread to sometimes read an object after it was reclaim (and allocated to other uses).

The optimistic access scheme maintains correctness in spite of reading reclaimed objects using three key properties. First, a read *must not fault*, even when accessing a reclaimed memory. Second, the scheme *identifies* a read that accesses a reclaimed object immediately after the read. Third, when a read of such stale value is detected, the scheme allows a *rollback* of the optimistic read. We follow by describing how these three properties can be satisfied.

The first requirement is obtained by the underlying memory management system. We will require that accessing previously allocated memory will never cause a fault. This can be supported by using user-level allocators that allocate and de-allocate without returning pages to the system. Such allocators can be designed to support lock-free algorithms. (Typically, returning pages to the system foils lock freedom.)

Jumping to the third property, i.e., the roll back, we first note that the ability to roll back is (informally) made possible in most lock-free data structures. Such data structures handle races by simply restarting the operation from scratch. The same restarting mechanism can be used to handle races between data-structure operations and memory reclamation; indeed, such a roll-back mechanism is assumed and used in previous work (e.g., [16]). However, to formally define a roll-back (or restart) mechanism, we simply adopt the normalized form for lock-free data structures [23]. This normalized form is on one hand very general - it covers all data structure we are aware of. On the other hand, its strict structure provides a well-defined restart mechanism, which can be used for rolling back the execution when a stale value has been read.

Next we discuss how to satisfy the second property, i.e., noting that a stale read has occurred due to a race between the read and memory reclamation. The optimistic access scheme divides the memory reclamation into phases, which may be thought of as epochs, and poses the following restrictions. First, an object is never reclaimed at the same phase in which it is unlinked from the data structure. It can only be reclaimed at the next phase or later. Second, a thread that acknowledges a new phase does not access objects that were unlinked in previous phases. These two restrictions provide a lightweight mechanism to identify a potential read

[1] As an aside, we note that the implementation of unmap is typically not lock-free and it is not to be used with lock-free data structures. For example, in the Linux operating system, an unmap instruction both acquires a lock and communicates with other processes via an interprocess interrupt.

of a stale value. If a thread is not aware that a phase has changed, then his read may potentially be of a stale value. Otherwise, i.e., if the thread is aware of the current reclamation phase, then his read is safe.

To make the (frequent) read operation even lighter, we move some of the related computation work to the (infrequent) reclaiming mechanism. To this end, each thread is assigned with an associated warning flag that a phase has changed. This flag is called the *warning-bit*. This bit is set if a new phase had started without the thread noticing, and clear otherwise. During a phase change the warning bits of all threads are set. When a thread acknowledges a phase change it resets its bit. This way, checking whether a read might have read a stale value due to reclamation, is as simple as checking whether the flag is non-zero.

To summarize, reading of shared memory is executed as follows. First the shared memory is read. Next, the thread's warning-bit is checked. Finally, if the warning bit is set, a restart mechanism is used to roll back the execution to a safe point.

We now deal with program writes. We cannot allow an event in which a thread writes to an object that has previously been reclaimed. Such an event may imply a corruption of objects in use by other threads. Therefore, for writes we adopt a simplified version of the hazard pointers scheme that prevents writes to reclaimed objects. A thread declares a location it is about to write to in a hazard pointer. Reclamation is avoided for such objects. Since writes are less frequent, the overhead of hazard pointers for writes is not high. The warning flag allows a quick implementation, as explained in Section 4 below.

Finally, it remains to describe the the memory reclamation scheme itself. A simplified version of such an algorithm may work as follows. It starts by incrementing the phase number, so that it can identify objects that were unlinked before the reclamation started. It can then reclaim all objects that were unlinked in previous phases and are not pointed by hazard pointers.

The problem with the simple solution is that each thread that starts reclamation will increment the phase number and trigger restarts by all other threads. This should not happen too frequently. To reduce this overhead, we accumulate retired objects in a global buffer and let a reclaiming thread process objects unlinked by *all* threads. This reduces the number of phase changes and hence also the number of restarts. Even when using global pools, the optimistic access scheme naturally benefits from using temporary local pools that are used to reduce the contention on the global pools. Performance is somewhat reduced when space is limited and measurements of the tradeoff between space overhead and time overhead are provided in Section 5.

Advantage of the optimistic access scheme. Hazard pointers and anchors require an involved and costly read barrier that runs a verification process and a memory fence. In contrast, ours scheme works with a light-weight read barrier (that checks the warning bit). Hazard pointers are used for writes in a ways that is easy to install (practically, automatic), and being used for writes only, hazard pointers also incur a low overhead, as shown by the measurements.

3. ASSUMPTIONS AND SETTINGS

In this section we specify the assumption required for our mechanism to work and define the normalized representation of data structures. Finally, in Subsection 3.4 we present a running example: the delete operation of Harris-Maged linked list.

3.1 System Model

We use the standard computation model of Herlihy [10]. 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 [17].

Finally, as discussed in Section 2, we assume that accessing previously allocated memory does not trigger traps. Formally, we assume the following of the underlying system.

ASSUMPTION 3.1. *Suppose a memory address p is allocated at time t. Then, if the program at time $t' > t$ executes an instruction that reads from p, then the executing of this instruction does not trigger a runtime-system trap.*

3.2 Normalized Data Structures

The optimistic access scheme assumes that the data structure implementation is given in a normalized form. In this subsection we provide a motivating discussion and an overview over normalized representation. The formal definition following [23] is provided Appendix A. The memory management scheme proposed in this paper lets threads infrequently access reclaimed space. When this happens, the acting thread will notice the problem thereafter and it will restart the currently executing routine. The strict structure of the normalized algorithm provides safe and easily identifiable points of restart. Let us now informally explain how a normalized implementation looks like.

Loosely speaking, a normalized implementation of a data structure partitions each operation implementation into three parts. The first part, denoted the *CAS generator*, prepares a list of CASes that need to be executed for the operation. It may modify the shared data structure during this process, but only in a way that can be ignored and restarted at any point, typically these modifications improve the underlying representation of the data structure without changing its semantics[2]. The second part, denoted the *CAS executor*, attempts to execute the CASes produced by the CAS generator one by one. It stops when a CAS fails or after all have completed successfully. The third part, denoted the *wrap-up*, examines how many CASes completed successfully and decides whether the operation was completed or whether we should start again from the CAS generator. A particular interesting property of the CAS generator and the wrap-up methods, is that they can be restarted at any time with no harm done to the data structure.

Very loosely speaking, think, for example, of a search executed before a node is inserted into a linked list. This search would be done in a CAS generator method, which would then specify the CAS required for the insertion. For reasonable implementations, the search can be stopped at any time and restarted. Also, when the wrap-up method inspects the results of the (single) CAS execution and decides whether to start from scratch or terminate, it seems intuitive that we can stop and restart this examination at any point in time. The normalized implementation ensures that the CAS generator and the wrap-up methods can be easily restarted any time with no noticeable effects.

In contrast, the actual execution of the CASes prepared by the CAS generator is not something we can stop and restart because they have a noticeable effect on the shared data structure. Therefore, the optimistic access scheme must make sure that the CAS executor method never needs to restart, i.e., that it does not access reclaimed space. Here, again, thanks to the very structured nature of the executed CASes (given in a list), we can design the protection automatically and at a low cost.

[2]A typical example is the physical delete of nodes that were previously logically deleted in Harris-Maged linked list implementation.

One additional requirement of a normalized data structure is that all modifications of the structure are done in a CAS operation (and not a simple write). Efficient normalized representations exist for all lock-free data structures that we are aware of.

The formal definition of the normalized method is required for a proof of correctness. These details appear in the Appendix A and also in the original paper that defined this notion, but the informal details suffice to understand the optimistic access scheme as described below.

3.3 Assumptions on the Data Structure

Here, we specify assumptions that we make about the data structure to which memory management is added. Most of these assumptions are assumed also by all previous memory reclamation schemes.

Many lock-free algorithms mark pointers by modifying a few bits of the address. The programmer that applies the optimistic access scheme should be able to clear these bits to obtain an unmarked pointer to the object. Given a pointer *O*, the notation *unmark(O)* denotes this unmarked pointer. This is one issue that makes our scheme not fully automatic.

Second, we assume that the data structure operations do not return a pointer to a reclaimable node in the data structure. Accessing a node can only happen by use of the data structure interface, and a node can be reclaimed by the memory manager if there is no possibility for the data structure interface functions to access it.

The data structure's functions may invoke the memory management interface. Following the (standard) interface proposed for the hazard pointers technique of [16], two instructions are used: *alloc* and *retire*. Allocation returns immediately, but a *retire* request does not immediately reclaim the object. Instead, the retire request is buffered and the object is reclaimed when it is safe. Deciding where to put the (manual) *retire* instructions (by the programmer) is far from trivial. It sometimes require an algorithmic modification [15] and this is the main reason why the optimistic access scheme is not an automatic transformation.

The third assumption is a proper usage of *retire*. We assume that retire is operated on a node in the data structure only after this node is unlinked from the data structure, and is no longer accessible by other threads that traverse the data structure. For example, we can properly retire a node in a linked list only after it has been disconnected from the list. We further assume that only a single thread may attempt to retire a node.

We emphasize that an object can be accessed after it has been properly retired. But it can only be accessed by a method that started before the object was retired. Nevertheless, because of this belated access, an object cannot be simply recycled after being retired.

3.4 A Running Example: Harris-Michael delete operation

We exemplify the optimistic access scheme throughout the paper by presenting the required modifications for the delete operation of Harris-Maged linked list. In Listing 1 we present the delete operation in its normalized form and including a retire instruction for a node that is removed from the list. In its basic form (and ignoring the constraints of the normalized representation), Harris-Michael delete operation consists of three steps: (1) *search*: find the node to be deleted and the node before it, (2) *logical delete*: mark the node's next pointer to logically delete it, and (3) *physical delete*: update the previous node's next pointer to skip the deleted node. During the search of the first stage, a thread also attempts to physically delete any node that is marked as logically deleted.

The normalized form of the operation is written in the three standard methods. The first method is the CAS generator method which performs the search and specifies the CAS that will logically delete the node by marking its next pointer. If the key is not found in the linked list then a list of length zero is returned from the CAS generator. The CAS executor method (not depicted in Listing 1) simply executes the CAS output by the CAS generator, and thus performs the logical deletion. The wrap-up method checks how many CASes were on the CAS list and how many of them were executed to determine if we need to return to the CAS generator, or the operation is done. The wrap-up interprets an empty CAS list as an indication that the key is not in the structure and then FALSE can be returned. Otherwise, if the CAS succeeded, then a TRUE is returned. If the CAS failed, the wrap-up determines a restart.

Note that the third step of the basic algorithm, physical delete, is not executed at all in the normalized form. The reason is that in its strict structure the wrap-up method does not have access to the pointer to the previous node and so it cannot execute the physical delete. However, this is not a problem because future searches will physically delete this node from the list and the logical delete (that has already been executed) means that the key in this node is not visible to the contains operation of the linked list. Another difference between the original implementation and the normalized one is that the original implementation may simply return FALSE upon failing to find the key in the structure. The normalized implementation creates an empty CAS list that the wrap-up method properly interprets and returns FALSE.

Finally, we added a retire instruction to Listing 1 after any physical delete. This is proper reclamation because new operations will not be able to traverse it anymore. Using a retire after the logical deletion is not proper because it is still accessible for new list traversals.

Listing 1: Harris-Michael linked list delete operation: normalized form with retire instructions

```
1  bool delete(int sKey, Node *head, *tail);
2  descList CAS_Generator(int sKey, Node *head, *tail){
3      descList ret;
4  start:
5      while(true) {/*Attempt to delete the node*/
6          Node *prev = head, *cur = head->next, *next;
7          while(true) { /*search for sKey position*/
8              if(cur==NULL){
9                  ret.len=0;
10                 return ret;
11             }
12             next = cur->next;
13             cKey = cur->key
14             if(prev->next != cur)
15                 goto start;
16             if(!is_marked(next)){
17                 if(cKey>=sKey)
18                     break;
19                 prev=cur;
20             }
21             else{
22                 if( CAS(&prev->next, cur, unmark(next)) )
23                     retire(cur);
24                 else
25                     goto start;
26             }
27             cur=unmark(next);
28         }
29         if(cKey!=sKey){
30             ret.len=0;
31             return ret;
32         }
```

```
33          ret.len=1;
34          ret.desc[0].address=&cur->next;
35          ret.desc[0].expectedval=next;
36          ret.desc[0].newval=mark(next);
37          return ret; /*Return to CAS executor*/
38      }
39  }
40  int WRAP_UP(descList exec, int exec_res,
41                              int sKey, Node *head, *tail){
42      if(exec.len==0) return FALSE;
43      if(exec_res==1) /*CAS failed*/
44          return RESTART_GENERATOR;
45      else return TRUE;
46  }
```

4. THE MECHANISM

In this section we present the optimistic access mechanism, which adds lock-free memory recycling support to a given data structure implementation with a memory management interface (i.e., alloc and proper retire instructions).

The mechanism uses a single bit per thread, denoted *thread.warning*. The warning bit is used to warn a thread that a concurrent recycling had started. If a thread reads *true* of its warning bit, then its state may contain a stale value. It therefore starts the CAS generator or wrap-up methods from scratch. On the other hand, if the thread reads *false* from its warning bit, then it knows that no recycling had started, and the thread's state does not contain any stale values. We assume that a thread can read its warning bit, clear its warning bit, and also set the warning bits of all other threads (non-atomically).

Modification of the shared memory is visible by other threads, and modifying an object that has been recycled is disastrous to program semantics. Therefore, during any modification of the shared data structure we use the hazard pointers mechanism [16] to mark objects that cannot be recycled. Each thread has a set of three pointers denoted $thread.HP_1$, $thread.HP_2$, $thread.HP_3$ that are used to protect all parameters of any CAS operation in the CAS generator or the wrap-up methods.

In addition, the CAS executor method and the wrap-up method can access all the objects mentioned in the CASes list that is produced by the CAS generator. The optimistic access scheme prevents these objects from being recycled by an additional set of hazard pointers. Let C be the maximum number of CASes executed by the CAS executor method in any of the operations of the given data structure. Each thread keeps an additional set of $3 \cdot C$ pointers denoted $thread.HP_1^{owner}$, ..., $thread.HP_{3C}^{owner}$. These hazard pointers are installed in the end of the CAS generator method and are cleared in the end of the wrap-up method. A thread may read the hazard pointers of all threads but it writes only its own hazard pointers.

Modifications to the Data Structure Code.

In Algorithm 1 we present the code for reading from shared memory (the read-barrier for shared memory). This code is used in the CAS generator and wrap-up methods. (There are no reads in the CAS executor method.)

As a read from the shared memory may return a stale value, when using the optimistic access memory recycling scheme, checking the warning bit lets the reading thread identify such an incident. If the warning bit is false, then we know that the read object was not recycled, and the read value is not stale. If, on the other hand, the warning bit is true, the read value may be arbitrary. Furthermore, pointers previously read into the thread's local variables may point to stale objects. Thus, the thread discards its local state, and restarts from a safe location: the start of the CAS generator or wrap-up method.

ALGORITHM 1: Read shared memory (var = *ptr)

```
1: temp = *ptr
2: if thread.warning == true then
3:     thread.warning:= false
4:     restart
5: end if
6: var = temp
```

The code in Algorithm 1 resets the warning bit before restarting. This can be done because the recycler will only recycle objects that appear in the recycling candidates list when it starts. This means that the warning bit is set after the data structure issued a retire instruction on the objects in the list. Since the retire instruction is *proper* in the data structure implementation, we know that all these objects are no longer accessible from the data structure and we will not encounter any such object after we restart the method from scratch. Therefore, upon restarting, we can clear the warning bit.

In order to exemplify such a read on our running example, consider, for example, Line 12 from Listing 1. It should be translated into the following code (COMPILER-FENCE tells the compiler to not change the order during compilation).

Listing 2: Algorithm 1 for Listing 1 Line 12

```
1  next = cur->next;
2  COMPILER-FENCE;
3  if(thread->warning){thread->warning=0;goto start;}
```

Next we define the write-barrier for all instructions that modify the shared memory. By the properties of normalized representation, this only happens with a CAS instruction. Algorithm 2 is applied to all modifications, except for the execution of the CASes list in the CAS executor, which are discussed in Algorithm 3. A simplified version of the hazard pointer mechanism [16] is used to protect the objects whose address or body is accessed in this instruction. If a CAS modifies a non-pointer field then only one hazard pointer is required for the object being modified. Recall that *unmark* stands for removing marks embedded in a pointer to get the pointer itself.

ALGORITHM 2: An observable instruction res=CAS($\&O.field, A_2, A_3$)

```
1: thread.HP_1 = unmark(O)
2: if A_2 is a pointer then thread.HP_2 = unmark(A_2)
3: if A_3 is a pointer then thread.HP_3 = unmark(A_3)
4: memoryFence (ensure HP are visible to all threads)
5: if thread.warning == true then
6:     thread.warning:= false
7:     thread.HP_1=thread.HP_2=thread.HP_3 = NULL
8:     restart
9: end if
10: res=CAS(&O.field, A_2, A_3)
11: thread.HP_1=thread.HP_2=thread.HP_3 = NULL
```

Running Example. The only case where Algorithm 2 is used in the running example is in Line 22 of Listing 1, which is translated to the following code in Listing 3.

Listing 3: Algorithm 2 for Line 22 of Listing 1

```
1  HP[0]=prev;
2  HP[1]=cur;
3  HP[2]=unmark(next);
4  _memory_fence();
5  if(thread->warning){thread->warning=0;goto start;}
6  if( CAS(&prev->next, cur, unmark(next)) ){
7      HP[0]=HP[1]=HP[2]=NULL;
8      ...
9  else{
10     HP[0]=HP[1]=HP[2]=NULL;
11     ...
```

We stress that *prev* and *cur* are unmarked. If, for example, *prev* was possibly marked, Line 1 would contain HP[0]=unmark(prev);.

Finally, the optimistic access scheme also protects all the objects that are accessed during the execution of the CASes list. Recall that this list is generated by the CAS generator method and executed thereafter by the CAS executor method. We need to protect all these objects so that no CAS is executed on reclaimed memory. To that end, we protect the relevant objects by hazard pointers at the end of the CAS generator method. The protection is kept until the end of the wrap-up method, because these objects are available to the wrap-up method and can be written by it. All these hazard pointers are nullified before the wrap-up method completes. The code for this protection is presented in Algorithm 3.

ALGORITHM 3: End of CAS Generator.
Input: A list of ℓ CASes S= CAS(&O_1.field, $A_{1,2}$,$A_{1,3}$),...,CAS(&O_C.field, $A_{C,2}$,$A_{C,3}$).

1: **for** i $\in 1 \ldots \ell$ **do**
2: $thread.HP^{owner}_{3\cdot i+1} = unmark(O_i)$
3: **if** $A_{i,2}$ is a pointer **then** $thread.HP^{owner}_{3\cdot i+2} = unmark(A_{i,2})$
4: **if** $A_{i,3}$ is a pointer **then** $thread.HP^{owner}_{3\cdot i+2} = unmark(A_{i,3})$
5: **end for**
6: memoryFence (ensure HP are visible to all threads)
7: **if** thread.warning == true **then**
8: thread.warning:= false
9: **for** i $\in 1 \ldots \ell$, $j \in 1,2,3$ **do** $thread.HP^{owner}_{i,j}$=NULL
10: restart
11: **end if**
12: **return** S (finish the CAS generator method)

A basic optimization. A trivial optimization that we have applied in our implementation is to not let two hazard pointers point to the same object. Algorithm 3 guards objects until the end of the wrap-up method. So all these objects need not be guarded (again) during this interval of execution. Moreover, in case this optimization eliminates all assignment of hazard pointers in Algorithm 3 or 2, then the memory fence and the warning check can be elided as well.

Running Example. There are three places where the CAS generator method finishes: Line 10, Line 31, and Line 37. For Lines 10 and 31 there is no need to add any code because, in this case, the CAS generator method returns a CAS list of length zero and there is no object to protect. Thus there is no need to execute the memory fence or the warning check. For Line 37 we add the following code:

Listing 4: Algorithm 3 for Listing 1 Line 37

```
1  //Original code before the return (Lines 34 - 36).
2  ret.desc[0].address=&cur->next;
3  ret.desc[0].expectedval=next;
4  ret.desc[0].newval=mark(next);
5  //Algorithm 3 added instructions
6  HP[3]=cur;
7  HP[4]=next;
8  //No need to set HP[5] (equal to HP[4])
9  _memory_fence();
10 if(thread->warning)
11     {thread->warning=0; HP[3]=HP[4]=NULL;goto start;}
12 //End of Algorithm 3
13 return ret;
```

The Recycling Mechanism.

Having presented code modifications of data structure operations, we proceed with describing the recycling algorithm itself: Algorithms 4-6.

The recycling is done in *phases*. A phase starts when there are no objects available for allocation. During a phase, the algorithm attempts to recycle objects that were retired by the data structure until the phase started. The allocator can then use the recycled objects until exhaustion, and then a new phase starts.

The optimistic access scheme uses three shared objects pools, denoted *readyPool, retirePool*, and *processingPool*. The readyPool is a pool of ready-to-be-allocated objects from which the allocator allocates objects for the data structure. The retirePool contains objects on which the retire instruction was invoked by the data structure. These objects are waiting to be recycled in the next phase. In the beginning of the phase, the recycler moves all objects from the retirePool to the processingPool. The processingPool is used during the recycling process to hold objects that were retired before the current phase began and can therefore be processed in the current phase. Note that while the recycling is being executed on objects in the processingPool, the data structure may add newly retired objects to the retirePool. But these objects will not be processed in the current phase. They will wait for the subsequent phase to be recycled. During the execution of recycling, each object of the processingPool is examined and the algorithm determines whether the object can be recycled or not. If it can, it is moved to the readyPool, and if not, it is moved back to the retirePool and is processed again at the next phase.

Since we are working with lock-free algorithms, we cannot wait for all threads to acknowledge a start of a phase (as is common with concurrent garbage collectors). Since no blocking is allowed and we cannot wait for acknowledgements, it is possible that a thread is delayed in the middle of executing a recycling phase r and then it wakes up while other threads are already executing a subsequent recycling phase $r' > r$. The optimistic access scheme must ensure that threads processing previous phases cannot interfere with the execution of the current phase. To achieve this protection, we let a modification of the pools only succeed if the phase number of the local thread matches the phase number of the shared pool. In fact, we only need to protect modifications of the retirePool and the processingPool. Allocations from the readyPool do not depend on the phase and also once a thread discovers that an object can be added to the readyPool, this discovery remains true even if the thread adds the object to the readyPool much later.

There are various ways to implement such phase protection, but let us specify the specific way we implemented the pools and the matching of local to global phase numbers. Each pool is implemented as a lock-free stack with a version (phase) field adjacent to the head pointer. The head pointer is modified only by a wide CAS instruction that modifies (and verifies) the version as well. When adding an object or popping an object from the pool fails due to version mismatch, a special return code VER-MISMATCH is returned. This signifies that a new phase started, and the thread

should update its phase number. Each thread maintains a local variable denoted *localVer* that contains the phase that the thread thinks it is helping. The thread uses this variable whenever it adds or removes objects to or from the pool.

A phase starts by moving the content of the retirePool to the processingPool (which also empties the retirePool), and increasing the version of both pools. This operation should be executed in an atomic (or at least linearizable) manner, to prevent a race condition where an object resides in both the retirePool and the processingPool. We use a standard trick of lock-free algorithms to accomplish this without locking (in a lock-free manner). The versions (localVer and the versions of the pools) are kept even (i.e., they represent the phase number multiplied by 2) at all times except for the short times in which we want to move the items from the retirePool to the processingPool. Swapping the pools starts by incrementing the retirePool version by 1. At this point, any thread attempting to insert an object to the retirePool will fail and upon discovering the reason for the failure, it will help swapping the pools before attempting to modify the retirePool again. Then the thread attempting to recycle copies the content of the retirePool into the processingPool while incrementing its version by 2. This can be done atomically by a single modification of the head and version. Finally, the retirePool version is incremented by 1 (to an even number), and the pool content is emptied. Again, these two operations can also be executed atomically.

When the data structure calls the retire routine, the code in Algorithm 4 is executed. It attempts to add the retired object to the retirePool (and it typically succeeds). If the attempt to add fails due to unequal versions (VER-MISMATCH), the thread proceeds to the next phase by calling Recycling (Algorithm 6), and then retries the operation.

ALGORITHM 4: Reclaim(obj)

```
1: repeat
2:    res=MM.retirePool.add(obj,localVer)
3:    if res==VER-MISMATCH then
4:       Call Recycling (Algorithm 6)
5:    end if
6: until res!=VER-MISMATCH
```

The allocation procedure appears in Algorithm 5. It attempts to pop an object from the readyPool. If unsuccessful, the thread calls Recycling (Algorithm 6), which attempts to recycle objects. Then it restarts the operation.

ALGORITHM 5: Allocate

```
1: repeat
2:    obj = MM.readyPool.pop()
3:    if obj==EMPTY then
4:       Call Recycling (Algorithm 6)
5:    end if
6: until obj!=EMPTY
7: memset(obj, 0); //Zero obj
8: return obj
```

The recycling procedure of the optimistic access scheme is presented in Algorithm 6. It starts by moving the content of the retirePool into the processingPool in a linearizable lock-free manner as described above, and it then increments the local phase counter. In Line 10, the thread checks if the (new) retirePool version matches the thread version. If not, the current phase was completed by other threads and the thread returns immediately. Note that the thread is unable to access the retirePool or the processingPool until it calls the recycling procedure again. The thread then sets the warning bits of all threads at Line 12. This tells the threads that objects for which the retire procedure was invoked before the current phase are candidates for recycling, and accessing them may return stale values. Finally, the thread collects the hazard pointer records of all threads. Objects that are pointed to by an hazard pointer are potentially modified, and should not be recycled in the current phase.

ALGORITHM 6: Recycling

```
1: //Start a new phase
2: localRetire=MM.retirePool
3: localProcessingPool=MM.processingPool
4: if localRetire.ver==localVer or
       localRetire.ver==localVer+1 then
5:    MM.retirePool.CAS(<localRetire, localVer>,
      <localRetire, localVer+1>)
6:    MM.processingPool.CAS(<localProcessingPool,
      localVer>, <localRetire, localVer+2>)
7:    MM.retirePool.CAS(<localRetire, localVer+1>,
      <Empty, localVer+2>)
8: end if
9: localVer=localVer+2
10: if MM.retirePool.ver > localVer then return
11:                          //Phase already finished
12: for each thread T do
13:    Set T.warning = true
14: end for
15: memoryFence (ensure warning bits are visible)
16: for each HP record R do
17:    Save R in a LocalHParray.
18: end for
19: //Processing the objects
20: while res = MM.processingPool.pop(localVer) is not empty
    do
21:    if res!=VER-MISMATCH then
22:       if res does not exist in LocalHParray then
23:          MM.readyPool.add(n)
24:       else
25:          res=MM.retirePool.add(res, localVer)
26:       end if
27:    end if
28:    if res==VER-MISMATCH then return
29:                                //Phase already finished
30: end while
```

Next, the processingPool is processed. For each candidate object in the processingPool, if it is not referenced by a hazard pointer, then it is eligible for recycling and therefore it is moved to the readyPool. Otherwise, the object cannot be recycled and it is returned to the retirePool, where it will be processed again in the next phase. Accesses to the processingPool and to the retirePool are successful only if the phase number is correct.

In order to determine if a given object is protected by a hazard pointer it is advisable to sort the hazard pointers to make the search faster or to insert the hazard pointers into a hash table (which is what we did).

5. METHODOLOGY AND RESULTS

To evaluate the performance of the optimistic access reclamation scheme with lock-free data structures, we have implemented it with

three widely used data structures: Harris-Maged linked list and hash table [15], and Herlihy and Shavit's skip list [12]. The optimistic access memory management scheme (and additional schemes as well) were applied to the baseline algorithm in a normalized form, which performs no memory recycling. The baseline algorithm, denoted *NoRecl*, serves as a base for performance comparison. The proposed optimistic access method is denoted *OA* in the measurements.

To obtain an allocator that does not unmap pages that were previously allocated (as specified in Assumption 3.1), we use object pooling for allocation. The pool is implemented using a lock-free stack, where each item in the stack is an array of 126 objects. To allocate, a thread obtains an array from the stack (using the lock-free stack pop operation) and then it can allocates 126 times locally with no synchronization. To fairly compare the memory management techniques and not just the underlying allocator, we converted all implementations to use the same object pool allocation for all allocations of objects of the data structure (except for the EBP method, discussed below, which uses its own allocator). As a sanity check, we verified that the object pooling method performed similarly (or better) than malloc on all measured configurations.

Additional Memory Management Schemes compared. We discuss related memory management techniques for lock-free data structures that are available in the literature in Section 6. Let us now specify which methods were compared per data structure.

For Harris-Maged linked list [15], a comprehensive comparison of memory management techniques was done by [3]. We used their baseline implementation (*NoRecl*), their hazard pointers implementation (*HP*), and their *anchors* implementation[3]. We also compare the Epoch Base Reclamation (*EBR*), proposed by Harris [8]; we took an implementation of this method by Fraser [7], which uses its integrated allocator. Namely, We did not replace the allocator. The latter method is not lock-free, but is sometimes used in practice to implement lock-free algorithms (to reduce the overhead associated with lock-free reclamation methods). Its disadvantage is that it does not deal well with threads failures, which is a major concerns for the lock-free methods. Finally, we implemented the optimistic access technique proposed in this paper. All implementations were coded in C.

For the hash table, each bucket was implemented as a linked list of items and the above linked-list implementations were used to support each of the buckets. For the hash table size, we used a load factor of 0.75. While both the linked list test and the hash table test use Harris-Maged linked list, the list length differed greatly for the two tests. In the linked list test all items reside on a single (relatively long) linked list, while in the hash table test the average length of a linked list is below one item. A hash table is probably a better example of a widely used data structure. We did not implement the *anchors* version of a hash table because the anchors improve accesses to paths of pointer dereferencing, while the lists in the hash table implementation are mostly of size 1, i.e, contain only a single entry.

For Herlihy and Shavit's skip list [12], we ported the Java implementation of [12] into C, and then converted it into a normalized form. We implemented the hazard pointers scheme for the skip list; the implementation uses $2 \cdot MAXLEN + 3$ hazard pointers. Finally, we implemented the optimistic access technique. The CAS generator method of the delete operation generates at most $MAXLEN + 1$ CASes to mark the next fields of the deleted node. This implies that $3 \cdot MAXLEN + 6$ hazard pointers are needed by the *OA* implementation. However, many of the protected objects are the same, and so the actual number of hazard pointers required is $MAXLEN + 5$: all CASes executed by the CAS executor method share one single modified object, and for each level the expected and new object pointers are the same. An *anchor* version for the skip list was not used because it is complex to design and no such design appears in the literature.

[3]Loosely speaking, the *anchors* implementation installs a hazard pointer once every every K reads. We picked $K = 1000$ for best performance results, as thread failures are rare.

Methodology. It is customary to evaluate data structures by running a stressful workload that runs the data structure operations repeatedly on many threads. Similarly to Alistarh et al. [1], in all our tests, 80% of the operations were read-only. Additional test configurations are reported in Appendix B. The hash table and the skip list were initialized to 10,000 nodes before the measurement began. The linked list was initialized to 5,000 nodes (thus denoted LinkedList5K). We also measure a short linked list which is initialized to 128 nodes (thus denoted LinkedList128), which creates reasonable high contention. Each micro-benchmark was executed with a varied number of threads being power-of-2 numbers between from 1 and 64 to check the behavior in different parallel settings. Each execution was measured for 1 seconds, which captures the steady-state behavior. We ensure that a 10-seconds test behave similarly to a 1-second test.

The code was compiled using the GCC compiler version 4.8.2 with the -O3 optimization flag. We ran the experiments on two platforms. The first platform featured 4 AMD Opteron(TM) 6272 2.1GHz processors, each with 16 cores (64 threads overall). The second platform featured 2 Intel Xeon(R) CPU E5-2690 2.90GHz processors, each with 8 cores with each core running 2 hyper-threads (32 threads overall). Measurements for the Intel Xeon platform are provided in Appendix B.

For each micro-benchmark we tested, we depict the ratio of the throughput between the evaluated memory management mechanism and the baseline algorithm (*NoRecl*), across different numbers of threads. A high number is better, meaning that the scheme has higher throughput. E.g., a result of 90% means the throughput was 0.9 of the baseline's throughput. A figure depicting the actual throughput is provided in Appendix B. Each test was repeated 20 times and the ratio of the average throughput is reported with error bars that represent 95% confidence level. The x-axis denotes the number of threads, and the y-axis denotes the average throughput for the evaluated method divided by the average throughput for the *NoRecl* method.

Results. In Figure 1 we compare the running time of the measured data structures with the various memory management methods. In this test, reclamation is triggered infrequently, once every 50,000 allocations, to capture the base overhead of the reclamation schemes. For the LinkedList5K micro-benchmark, operations have long execution time that is mostly spent on traversals. The *OA* has a very low overhead in most configurations and at max it reaches 4%. The EBR also has very low overhead, and in some cases it even ran slightly faster (recall that it uses a different allocator). However, for 64 threads its overhead was 12%. The HP overhead always exceeds $3x$. The Anchors has an overhead of $3\% - 52\%$, and the overhead increases as the number of threads increases.

For the LinkedList128 micro-benchmark, operations are shorter, and traversals do not take over the execution. Also, the baseline reaches maximum throughput at 16 threads, and then throughput decreases due to contention. For higher numbers of threads, memory reclamation methods behave better and even slightly improve performance of over no-reclamation by reducing contention. (This was previously reported by [5, 1].) The *OA* has an overhead of $-1\% - 19\%$ and the overhead is lower for a high number of threads. The EBR has an overhead of $-2\% - 26\%$, again lower for high

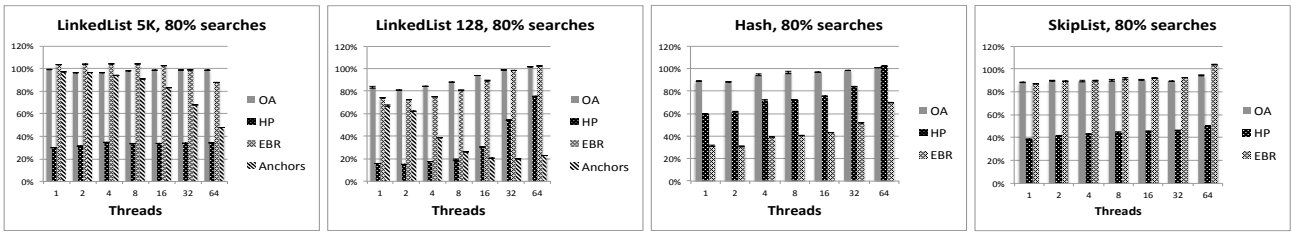

Figure 1: Throughput ratios for the various memory management techniques and various data structures. The x-axis is the number of participating threads. The y-axis is the ratio between the throughput of the presented scheme and the throughput of NoRecl.

number of threads. The overhead of the HP method is above $3x$ for up to 16 threads. For $32-64$ threads it behaves better with an overhead of $25\%-46\%$. The Anchors responds poorly to the shorter linked-list length, and incurs an overhead of $2x-5x$. For a large number of threads (and higher contention), it became even slower than the hazard pointers method.

For the hash micro-benchmark, operations are extremely short, and modifications have a large impact on insert and delete operations. Contention has noticeable effect for 64 threads, letting executions with memory reclamation demonstrate low overheads and sometimes even slightly improved performance. The *OA* has an overhead of $-1\%-12\%$. EBR responds poorly to the short operation execution times, and it demonstrates an overhead of $2x-3x$ for $1-32$ threads. For 64 threads it slightly improves with an overhead of 30%. HP has an overhead of $16\%-40\%$ for $1-32$ thread and -2% for 64 threads.

For the skip list micro-benchmark, operations take approximately the same time as LinkedList128, but face less contention. Moreover, operations are significantly more complex (executes more instructions). The *OA* has an overhead of $8\%-12\%$. the EBR has overhead of $8\%-13\%$ for $1-32$ threads, but slightly improved performance for 64 threads. The HP has overhead of $2x-2.5x$.

To summarize, the *OA* overhead is at most 19% in all measured configurations, which is significantly faster than currently state-of-the-art lock-free reclamation methods. The optimistic access method has comparable performance to the EBR method (which is not lock-free), and is significantly better than EBR for the hash micro-benchmark.

Next, we study how the various choice of parameters affects performance. The impact of choosing the size of the local pools is depicted in Figure 2. Measurements for the LinkedList5K and the Hash micro benchmarks are depicted, showing the behavior with long and short operations time. All tests were executed with 32 threads. We started a new reclamation phase approximately every $16,000$ allocations. We later show that this choice is immaterial. It can be seen that the choice of local pool size has minor effect on the LinkedList5K micro-benchmark. For the hash micro-benchmark, all methods suffer a penalty for small local pools, but the *OA* scheme suffers a penalty also for a medium sized local pool. Using the *perf* Linux tool, we found that Algorithm 6 was the source of this slow-down. The reason is that local pools are popped from the *processingPool* and pushed into the *readyPool* in a tight loop, so contention becomes noticeable for medium sizes. Reasonably sized pools of 126 objects are sufficiently large to obtain excellent performance for *OA*.

Next, we study how the frequency of reclamation phases affects performance; the results are depicted in Figure 3. All tests were executed with 32 threads. The *OA* triggers a new phase when no object is available for allocation. We initialized the number of entries available for allocation to be the data structure size plus an

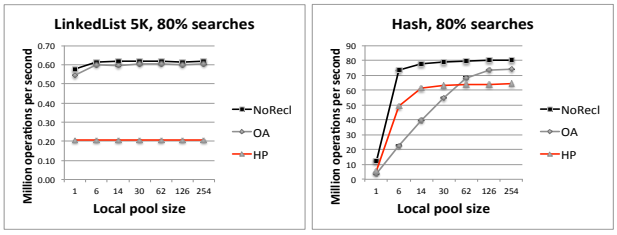

Figure 2: Throughput as a function of the size of the local pools.

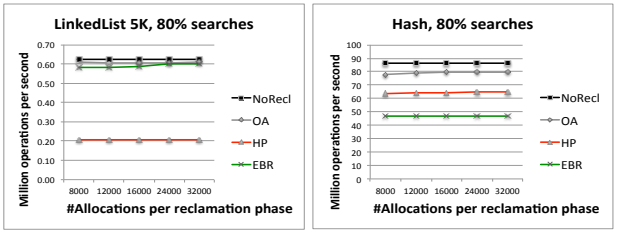

Figure 3: Throughput as a function of collection phases frequency.

additional δ, for δ equals 8000, 12000, 16000, 24000, or 32000. Thus a new phase is triggered approximately every δ allocations. We started with $\delta = 8000$ because allocations are counted system-wide. For example, $\delta = 32000$ means that each thread allocated approximately $32000/32 = 1000$ objects before a phase begin. Each thread has two local pools: one for allocation and one for retired objects. The local pool size is 126. Thus $\delta = 8000 \approx 32 \cdot 126 \cdot 2$ is the minimum size where threads do not starve. With $\delta < 7600$ performance indeed drops drastically, due to thread starvation. In Figure 3 it is possible to see that different frequencies have a low impact on the performance of *OA*.

The other schemes do not use a global counting mechanisms. Therefore, measuring same effect for other schemes might mean a significant change in their triggering. Instead, we made an effort to compare the reclamation schemes without changing the algorithmic behavior. In the HP scheme we picked $k = \delta/32$, where each thread starts a reclamation phase *locally* after it retired k objects. In the EBR scheme, a reclamation starts *locally* after q *operations* started. We picked $q = (\delta/32) \cdot 10$ since deletions are about 10% of the total operations. We also made sure that threads do not starve in the other schemes for these settings of δ.

6. RELATED WORK

The basic and most popular lock-free reclamation schemes are the *hazard pointers* and *pass the buck* mechanisms of [16] and

[11]. In these schemes every thread has a set of *hazard pointers* or *guards*, which mark objects the thread is accessing. Before accessing a data structure node, the object's address is saved in a thread's hazard pointer. A validation check is executed to verify that the object was not reclaimed just before it became guarded by a hazard pointer. If the validation fails, the thread must restart its operation. A node can be reclaimed only if it is not guarded by any thread's hazard pointers. The main disadvantage of these schemes is their cost. Each access (even a read) of a node requires a write (to the hazard pointer), a memory fence (to make sure that the hazard pointer value is visible to all other threads), and some additional reads for computing the validation test. Fence elision techniques [6] can be used to ameliorate this overhead, but such methods foil lock-freedom.

Braginsky et al. [3] proposed the *anchor* scheme as an improvement to the hazard pointer scheme. The anchor scheme allows a hazard pointer to be registered only once per each k reads of the data structure. The anchor method significantly reduces the overhead, but not to a negligible level (see measurements in Section 5). Moreover, the complexity of designing an anchor reclamation scheme for a given data structure is high, and the authors only provided an example implementation for the linked list.

Alistarh et al. [1] have recently proposed the StackTrack method, which utilizes hardware transactional memory to solve the memory reclamation problem. This method breaks each operation to a series to transactions, such that a successfully committed transaction cannot be interfered with a memory reclamation.

Another method for manual object reclamation that supports lock-freedom is reference counting. Each object is associated with a count of threads that access it and a node can be reclaimed if its reference count is dropped to 0. Correctness requires either a type persistence assumption [24, 22] or the use of the double compare-and-swap (DCAS) primitive [4] which is not always available. The performance overhead is high as these methods require (at least) two atomic operations per object read [9].

If lock-freedom is not required, then in most cases the epoch-based reclamation method is a high performant solution [8, 15]. Before an operation starts, the executing thread reports the timestamp it reads, and upon operation completion it clears the published timestamp. An object A can be reclaimed when every thread that started an operation before A was retired completes its operation.

7. CONCLUSIONS

This paper presents the *optimistic access* lock-free memory management mechanism. Unlike previous memory managers, this algorithm allows threads (infrequent) access to reclaimed objects. This optimistic mechanism obtains a drastic reduction in the reclamation overheads compared to other schemes available in the literature. It is also simpler to implement. In order to preserve correctness in spite of potentially reading stale values, the algorithm implements a low-overhead cooperation mechanism, where a thread that reclaims objects warns other threads that a reclamation has taken place, and accessing threads check for this warning at appropriate locations. As far as we know this is the first memory management scheme that allows accessing reclaimed objects in order to obtain (considerable) performance improvement.

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

[5] D. Dice, M. Herlihy, Y. Lev, and M. Moir. Lightweight contention management for efficient compare-and-swap operations. In *EuroPar*. ACM, 2013.

[6] D. Dice, H. Huang, and M. Yang. Techniques for accessing a shared resource using an improved synchronization mechanism, 2010. Patent US 7644409 B2.

[7] K. Fraser. http://www.cl.cam.ac.uk/research/srg/netos/lock-free/src/temp/lockfree-lib/.

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

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

[10] M. Herlihy. Wait-free synchronization. *TOPLAS*, pages 124–149, 1991.

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

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

[13] M. P. Herlihy and J. E. B. Moss. Lock-free garbage collection for multiprocessors. *TPDS*, pages 304–311, 1992.

[14] 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.

[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] 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.

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

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

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

[21] 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.

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

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

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

APPENDIX

A. A FORMAL DEFINITION OF NORMALIZED DATA STRUCTURES

In this section we provide the formal definition of normalized data structure implementations [23]. In fact, 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 [23] as well, but the relaxed restrictions that we specify below suffice.

Definition 4.1 of [23] (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 [23] 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 [23] below).

We remark that in [23] 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 [23].

As discussed in [23], 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 optimistic access 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 [23]). 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 [23] below.

Definition 3.1 of [23] (Futile CAS) *A futile CAS is a CAS in which the expected value and the new value are identical.*

Definition 3.2 of [23] (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 [23] (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 [23] and then explain why it implies that restarting from the beginning of the method is fine.

Parallelizable method (Definition 3.4 of [23]). *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 methodï£¡s CAS operations. In particular, the execution does not depend on the executing threadï£¡s local state prior to the invocation of the parallelizable method. Second, at the point*

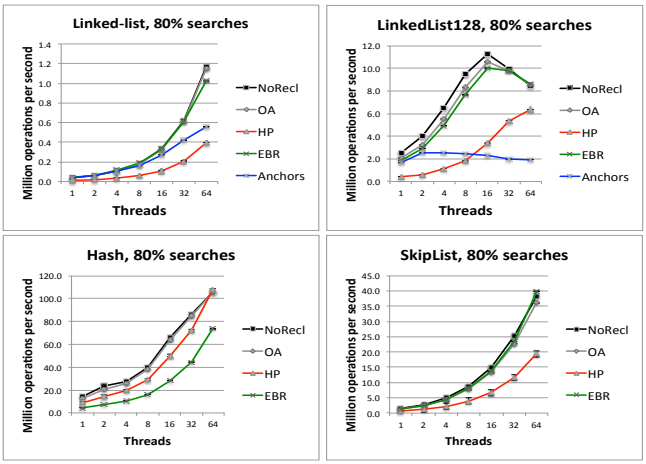

Figure 4: Execution time for the various memory management techniques and various data structures. The x-axis is the number of participating threads. The y-axis is the throughput in million operations per second.

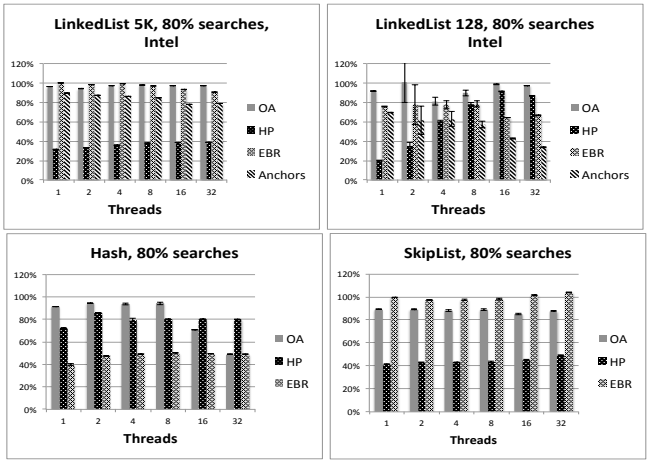

Figure 5: Repeating Figure 1 for an Intel Xeon machine.

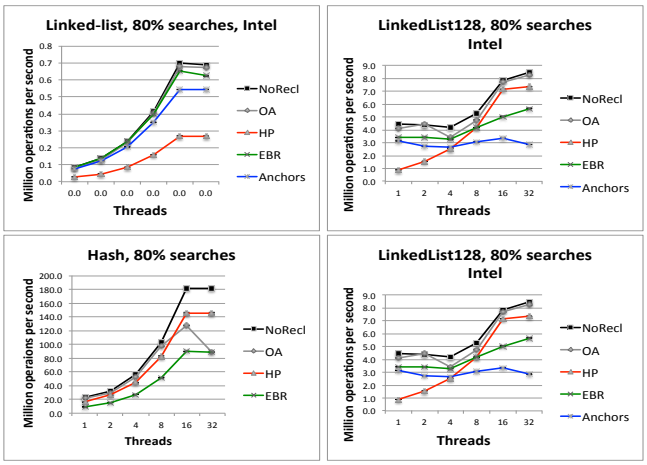

Figure 6: Repeating Figure 1 for an Intel Xeon machine with actual throughput.

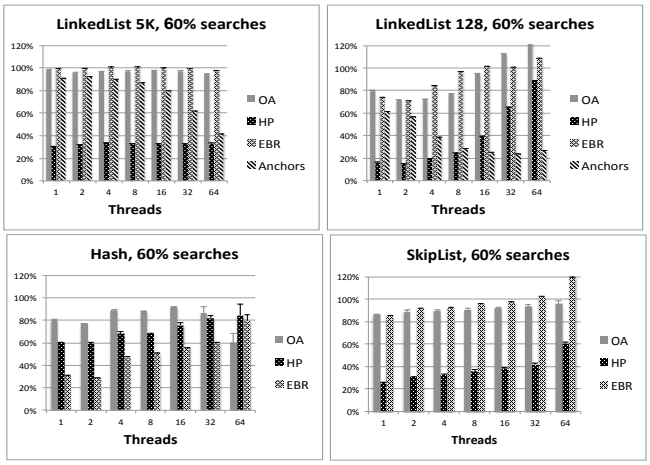

Figure 7: Throughput ratios of various memory management techniques with 40% mutation rates (60% searches).

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.

B. ADDITIONAL MEASUREMENTS

In Figure 4 we depict the actual throughput of the memory reclamation schemes depicted in Figure 1.

In Figure 5 we repeat the experiments of Figure 1 with the Intel Xeon platform and in Figure 6 we depict the actual throughput. On most configurations we see similar behavior to the AMD machine. One notable exception is the hash micro-benchmark, where for 16 and 32 threads the performance of the OA scheme drops drastically. We were not able to find the source of this slow-down.

We also evaluated the optimistic access scheme with a mutation rate higher than the standard 20% (i.e., 80% searches). The results for 40% mutation rate appear in Figure 7 and the results for $^2/_3$ mutation rate appear in Figure 8. As expected, the optimistic access incurs higher overhead when the mutation rate is higher due to the more costly write-barrier. Nevertheless, it outperform both the anchors and the hazard pointers schemes, except for the one case of the hash table run with 64 threads. Measurements revealed that Algorithm 6 was a major source for this slowdown, possibly due to the high contention on the shared pools.

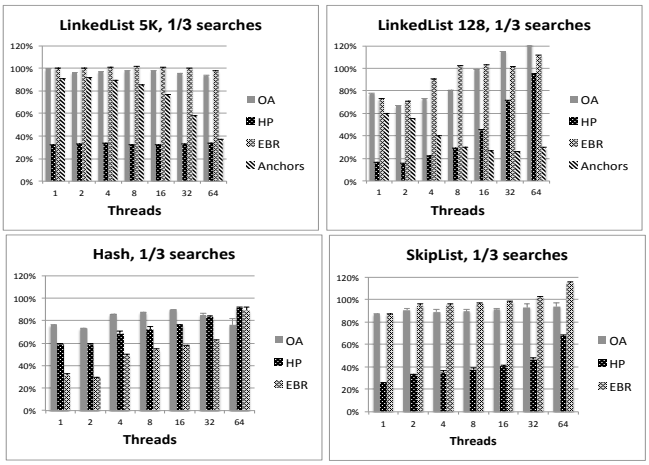

Figure 8: Throughput ratios of various memory management techniques with $^2/_3$ mutation rates ($^1/_3$ searches).

C. RUNNING EXAMPLE

In this section we complete applying the optimistic access scheme on the running example. Now, let us provide the complete applying the optimistic access scheme on Listing 1. For simplicity of presentation and ease of reading, we wrap the warning bit check inside a macro.

```
1  #define CHECK {__compiler_fence(); \
2      if(unlikely(*warning)) \
3          {*warning=0; goto start;}}
4  //A CAS serves as a memory barrier in x86
5  #define FENCE_CHECK { \
6      if(unlikely(CAS(warning,1,0)))\
7          goto start;}
8
9  bool delete(int sKey, Node *head, *tail);
10 descList CAS_Generator(int sKey, Node *head, *tail){
11     descList ret;
12     char *warning=&thread.warning;
13     void **HP = thread.HP;//array of hazard pointers
14 start:
15     while(true) {/*Attempt to delete the node*/
16         Node *prev = head, *cur = head->next, *next;
17         CHECK //protect reading cur
18         while(true) { /*search for sKey position*/
19             if(cur==NULL){
20                 ret.len=0;
21                 return ret;
22             }
23             next = cur->next;
24             cKey = cur->key
25             node *tmp=prev->next;
26             CHECK //protect reading next,cKey,tmp
27             if( tmp != cur)
28                 goto start;
29             if(!is_marked(next)){
30                 if(cKey>=sKey)
31                     break;
32                 prev=cur;
33             }
34             else{
35                 HP[0]=prev;//Algorithm 2
36                 HP[1]=cur;
37                 HP[2]=unmark(next);
38                 FENCE_CHECK
39                 if( CAS(&prev->next, cur, unmark(next)) )
40                     retire(cur);
41                 else
42                     goto start;
43             }
44             cur=unmark(next);
45         }
46         if(cKey!=sKey){
47             ret.len=0;
48             return ret;
49         }
50         ret.len=1;
51         ret.desc[0].address=&cur->next;
52         ret.desc[0].expectedval=next;
53         ret.desc[0].newval=mark(next);
54         HP[3]=cur;//Algorithm 3
55         HP[4]=next;
56         FENCE_CHECK
57         return ret; /*Return to CAS executor*/
58     }
59 }
60 int WRAP_UP(descList exec, int exec_res,
61                         int sKey, Node *head, *tail){
62     if(exec.len==0) return FALSE;
63     if(exec_res==1) /*CAS failed*/
64         return RESTART_GENERATOR;
65     else return TRUE;
66 }
```

Listing 5: Applying the optimistic access scheme

D. SKETCH OF CORRECTNESS PROOF

THEOREM 1. *Let I be an implementation of a data structure D with proper reclaim instructions and suppose that Assumption 3.1 holds. Then the memory managed implementation of I is a linearizable implementation of D with memory reclamation support.*

Given any data structure implementation I, we denote the memory managed implementation obtained by the optimistic access scheme by I_{OA}. In order to show Theorem 1, i.e., that I_{OA} is a linearizable implementation of the same data structure, we will show a stronger result, namely that for every execution of I_{OA}, there exists an equivalent execution over I. Recall that if two executions are equivalent and one of them is linearizable, then the second is also linearizable. The formal definition of equivalent execution appears in [23], and is stated in Subsection 3.2.

Let us now sketch the proof of Theorem 1. The proof is built by starting with any execution of I_{OA} and showing an equivalent execution of I. This is done by several steps in which we modify the execution and show that we maintain equivalence to the original execution. Eventually, we obtain an equivalent execution of I, which is equivalent to the original execution of I_{OA} by transitivity, and we are done.

So we start with an execution E over I_{OA} and let us first build an execution E' which is equivalent to E but never accesses stale values of reclaimed objects. Recall that the optimistic access scheme optimistically accesses objects, but restarts the execution if a recycling started. Loosely speaking, the execution E restarts a method whenever it detects accessing stale values. In the final execution of the method, it does not restart because the entire execution of the method did not detect any use of stale values.

The execution E' is obtained from E by replacing the stale values with correct values that would have been read if the object had not been recycled. This can be done by "saving" the content of each recycled object before recycling it, and then using the saved information to respond to subsequent accesses of the object. After reading a stale value, the thread immediately checks its warning

bit and restart the operation, discarding the read value. In the full proof, we show that the obtained execution E' is equivalent to E.

Next we move to creating a third equivalent execution E''. In this execution we separate all retries of a method into parallel executions of new auxiliary threads. Namely, if a method restarts 4 times and finishes properly in the 5th attempt, then we will have 4 new auxiliary threads, each executing one of the failed attempts and the original thread will only execute the final run, in which all tests of Algorithms 1, 2, or 3 passed with no need to restart. Execution E'' cannot happen in the real world because the implementation never starts a method in a separate thread without executing a full operation. In E'' there is a thread that may execute a CAS generator method (with the proper inputs) as a standalone. Note that by Definition 3.4 of [23] we can execute this method with no dependence on the thread's local state. We further make a small modification to each of the auxiliary threads and make them stop executing before the check of the warning bit rather than after. Although this execution is imaginary, it is well defined and we show that it is equivalent to Execution E'.

In the building of E'' we further make additional modifications that do not adhere to the behavior of I_{OA}. Particularly, we remove all checks that find the warning bit false. Note that in these cases, the original instructions of the original implementation are run. More precisely, every execution of Algorithm 1 with a read instruction *inst* or Algorithm 2 with a CAS instruction *inst*, in which the warning bit is found false, is replaced by executing *inst*. Every execution of Algorithm 3 with a sequence of CAS instructions S, in which the warning bit is found false, is replaced by executing S. In the full proof it is shown that E'' is equivalent to E'.

Next, we move further into creating an execution E''', which comes closer to an execution of the original implementation I. Due to our interference with the memory values returned in response to memory accesses, we know that no thread accesses a recycled object. Therefore, we can modify the allocator to never reuse an address. Namely, each object can be assigned a unique location in the memory, and the execution will continue to be equivalent. Memory recycling is redundant for this case, but it exists in the execution as before. We further continue and modify the *reclaim* instruction to be a no-operation. This means that our interference with memory is no longer needed. At this point, accessing a recycled memory brings the correct values of the object with no recycling. The full proof will build on the fact that an object is never recycled until every CAS that modifies it is completed. The execution E''' is equivalent to E'' and is in fact a possible execution of I, except that we have strange auxiliary threads that execute methods partially and then make a check of the warning flag and drop dead.

We now move to execution E^*. We first note that E''' is an execution of the original implementation, and since the methods that are executed by the auxiliary threads are parallelizable, we may eliminate them. The way this is done, is by creating a series of equivalent executions, each making a CAS of the auxiliary thread avoidable, until all CAS instructions of the auxiliary threads are avoidable. We then simply drop the execution of the auxiliary threads and obtain the final execution E^* which completely matches the I implementation and is equivalent to the original execution E of I_{OA} and we are done.

E. OPTIMIZATIONS

Whenever possible, we considered the following optimizations to the optimistic access scheme. First, whenever the program accessed an object that was known to be unreclaimable during the read, we omitted the warning bit check after reading it. Example for this optimizations are reading from a sentinel node, or reading from a node that is protected by a hazard pointer.

Second, when there were two independent reads (e.g. reading the key and the next pointer of a node), we executed both reads and checked the warning bit only once after both reads completed. The correctness follows since the reads are independent and hence for each of these reads, it holds that the warning bit is checked before the read values are used.

Third, we computed the address of the warning bit once in the beginning of the operation and not repeatedly. We notified the compiler that the warning bit is unlikely to be true, so it may optimize the code for this case. Finally, we let the warning bit be of type char (which contains zero or one values) because it produced slightly faster results than the int type.

Finally, we set the warning bit by a CAS that succeeded only once per phase changed. This reduce the number of restarts to once per thread per phase; using a simple write may results in n restarts per thread per write, where n stands for the number of participating threads. We used a (standard) quad-word CAS, where the warning bit uses 8 bits, and the phase number uses the remaining 56 bits.