Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting

Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, Philippas Tsigas
Department of Computing Science
Chalmers University of Technology and Göteborg University
412 96 Göteborg, Sweden
E-mail: {andersg,ptrianta,phs,tsigas}@cs.chalmers.se

Abstract

We present an efficient and practical lock-free implementation of a memory reclamation scheme based on reference counting, aimed for use with arbitrary lock-free dynamic data structures. The scheme guarantees the safety of local as well as global references, supports arbitrary memory reuse, uses atomic primitives which are available in modern computer systems and provides an upper bound on the memory prevented for reuse. To the best of our knowledge, this is the first lock-free algorithm that provides all of these properties. Experimental results indicate significant performance improvements for lock-free algorithms of dynamic data structures that require strong garbage collection support.

1. Introduction

Memory management is essential for building dynamic concurrent data structures. Concurrent algorithms for data structures and related memory management are commonly based on mutual exclusion. However, mutual exclusion causes blocking and can consequently incur serious problems as deadlocks, priority inversion or starvation. Researchers have addressed these problems by introducing non-blocking synchronization algorithms, which are not based on mutual exclusion. Lock-free algorithms are non-blocking, and guarantee that always at least one operation can progress, independently of the actions taken by the concurrent operations. Wait-free [3] algorithms are lock-free, and moreover guarantee that all operations can finish in a finite number of their own steps, regardless of the actions taken by the concurrent operations. The common consistency requirement for non-blocking algorithms is called linearizability [6].

In this paper we are focusing on practical and efficient memory management in the context of lock-free dynamic data structures. For an operation of an algorithm to be lock-free, all sub-operations must be at least lock-free. Consequently, lock-free dynamic data structures typically require lock-free memory management. The memory management problem is normally divided into the sub-problems of dynamic memory allocation versus garbage collection. Please note that we in this paper interpret the notion of garbage collection in a wider sense, to also include memory reclamation schemes that are guided by the applications. Moreover, as lock-free garbage collection implicitly and in a de-centralized manner would involve all concurrent participants in the garbage detection and reclamation process, this consequently rules out fully automatic garbage collection schemes (e.g. as used in Java).

Valois as well as Michael and Scott [16, 12] presented a lock-free memory allocation scheme for fixed-sized memory segments; this scheme has to be used in combination with the corresponding garbage collection scheme. Lock-free memory allocation schemes for general use have been presented by Michael [11] and Gidenstam et al. [2].

In the scope of lock-free garbage collection and memory reclamation, Michael [9, 10] proposed the hazard pointer algorithm that focuses on local references, and have been shown to be highly efficient for compatible data structures. A similar scheme has been proposed and patented by Herlihy et al. [5]; this scheme uses unbounded tags and is based on the double-width CAS[1] atomic primitive. As these schemes only guarantee the safety of local pointers from the threads, they cannot support arbitrary lock-free algorithms that might require to always being able to trust global references (i.e. pointers from within the data structure) to objects. This constraint can be strong and restrictive, and may force the algorithms to retry their traversals in the possibly large data structures, with resulting large performance

1 A compare-and-swap operation that can atomically update two adjacent memory words, which is available in some 32-bit and very few 64-bit architectures.

penalties that increase with the level of concurrency.

Garbage collection schemes that are based on reference counting can guarantee the safety of global as well as local references to objects. Valois et al. [16, 12] presented a lock-free reference counting scheme that can be implemented using available atomic primitives, though it is limited to be used only with the corresponding algorithm for memory allocation. Detlefs et al. [1] presented a scheme that allows also arbitrary reuse of reclaimed memory, but it is based on DCAS[2]. Herlihy et al. [4] presented and patented a modification of the previous scheme such that it only uses CAS (compare-and-swap) for the reference counting part. However, this scheme relies on another scheme that itself requires double-width CAS. It has been identified in [12] that reference counting techniques can potentially cause a reference from a thread to block (due to the ability of creating recursive references) arbitrarily number of nodes from being reclaimed.

In the context of wait-free memory management, a wait-free extension of Valois' scheme has been presented by Sundell [14]. Hesselink and Groote [7] have presented a wait-free memory management scheme that is restricted to the specific problem of sharing tokens.

This paper combines the strength of reference counting with the efficiency of hazard pointers, with the aim of keeping only the advantages of the involved techniques while avoiding the respective drawbacks. Our new lock-free memory reclamation scheme is lock-free and linearizable, is compatible with arbitrary schemes for memory allocation, can be implemented using commonly available atomic primitives, and guarantee the safety of local as well as global references. We also show how to bound the amount of unreclaimed memory that can be temporarily held by any thread.

The rest of the paper is organized as follows. In Section 2 we describe the type of systems that our implementation is aiming for. Section 3 describes the specifics of the problem of garbage collection we are focusing on. The actual algorithm is described in Section 4. We conclude the paper with Section 5.

2. System Description

Each node of the shared memory multi-processor system contains a processor together with its local memory. All nodes are connected to the shared memory via an interconnection network. A set of co-operating tasks is running on the system performing their respective operations. Each task is sequentially executed on one of the processors, while each processor can serve (run) many tasks at a time. The co-operating tasks, possibly running on different processors, use shared data objects built in the shared memory to coordinate and communicate. Tasks synchronize their operations on the shared data objects through sub-operations on top of a cache-coherent shared memory. The shared memory may not though be uniformly accessible for all nodes in the system; processors can have different access times on different parts of the memory.

The shared memory system should support atomic read and write operations of single memory words, as well as stronger atomic primitives for synchronization. In this paper we use the Fetch-And-Add (FAA) and the Compare-And-Swap (CAS) atomic primitives. These read-modify-write style of operations are available on most common architectures or can be easily derived from other synchronization primitives [13] [8].

3. Problem Description

In this paper we are aiming to solve the garbage collection problem in the context of dynamic lock-free data structures. Lock-free data structures typically consist of a set of memory segments, called *nodes* that each contain arbitrary data. These nodes are interconnected by referencing each other in an arbitrary pattern. The references are typically implemented by using *pointers* that can identify each individual node by the mean of memory addresses. Each node may contain an arbitrary number of pointers, called *links*, that reference other nodes. The operation to follow the referenced node through a link is called *dereferencing*. Some nodes are typically always part of the data structure, all others nodes are part of the data structure when they are referenced by a node that itself is a part of the data structure. In a dynamic and concurrent data structure, arbitrary nodes can continuously and concurrently be added or removed from the data structure. As systems have limited amount of memory, the occupied memory of these nodes needs to be dynamically allocated and reclaimed from/to the system.

In a sequential implementation of a data structure, the memory of a node is typically explicitly reclaimed to the system when the last reference to it has been removed, i.e. when the node has been *deleted*. In a concurrent environment this should also include possible local references to a node that any thread might have, as an access to the memory of a reclaimed node might be fatal to the correctness of the data structure and/or the whole system. The logical unit that correctly decides about reclaiming is called the *garbage collector* and should thus have the following property:

Property 1 *The garbage collector should only reclaim nodes that are not part of the data structure and for which future access by any thread is not possible.*

2 A double-word compare-and-swap operation that can atomically update two arbitrary memory words, which is not available on any modern architecture.

	Guarantees the safety of shared references (Property 5)	Bounded number of unreclaimed deleted nodes (Property 2)	Compatible with standard memory allocators (Property 4)	Suffices with single-word compare-and-swap
New algorithm	Yes	Yes	Yes	Yes
Detlefs et al. [1]	Yes	No [e]	Yes	No [a]
Herlihy et al. [5]	No	Yes	Yes	No [b]
Herlihy et al. [4]	Yes	No [e]	Yes	No [c]
Michael [9, 10]	No	Yes	Yes	Yes [d]
Valois et al. [16, 12]	Yes	No [e]	No	Yes

a The LFRC algorithm uses the double-word compare-and-swap (DCAS) atomic primitive.
b The pass-the-buck (PTB) algorithm uses the double-width compare-and-swap atomic primitive.
c The SLFRC algorithm is based on the pass-the-buck (PTB) algorithm, and thus uses double-width compare-and-swap.
d The hazard pointer algorithm uses only atomic reads and writes.
e These reference count-based schemes allow arbitrary long chains of deleted nodes that recursively reference each other to be created. In addition, deleted nodes that cyclically reference each other (i.e. cyclic garbage) will be not be reclaimed ever.

Table 1. Properties of different approaches to non-blocking memory management.

It should also always be possible to predict the maximum amount of memory that is used by the data structure, thus adding this requirement to the garbage collector:

Property 2 *At any time, there should be an upper bound on the number of nodes that is not part of the data structure, but not yet reclaimed to the system.*

In actual implementations of a garbage collector (GC) these properties can be very hard to achieve, as local references to nodes might not be accessible globally (e.g. they might be stored in processor registers). Therefore implementations of GC's typically need to interact with the involved threads and put restrictions on the access to the nodes, e.g. by providing special operations for dereferencing links and demanding that the data structure implementation explicitly calls the garbage collector when a node has been deleted.

Moreover, as the underlying data structures of interest are lock-free and typically also linearizable, the garbage collector also has to guarantee these features:

Property 3 *All operations of the garbage collector for communication with the underlying data structure implementation should be lock-free and linearizable.*

In order to minimize the whole system's total amount of occupied memory for the various data structures, we sometime would like to fulfill the following property:

Property 4 *The memory that is reclaimed by the garbage collector should be available for any arbitrary future reuse; i.e. the garbage collector should be compatible with the system's default memory allocator.*

In a concurrent environment it might frequently occur that a thread is holding a local reference to a node that has been deleted (i.e. removed from the data structure) by some other thread. In these cases it may be very useful for the first thread to be able to use the deleted node's links, e.g. in search procedures in large data structures:

```
structure Node
    mm_ref: integer /* Initially 0 */
    mm_trace: boolean /* Initially false */
    mm_del: boolean /* Initially false */
    ... /* Arbitrary user data and links follows */
    link[NR_LINKS_NODE]: pointer to Node /* initially NULL */
```

Figure 1. The Node structure

Property 5 *A thread that has a local reference to a node, should also be able to dereference all of the links that are contained in that node.*

The new algorithm in this paper fulfills all of these properties in addition to the property of only using atomic primitives that are commonly available in modern systems. Table 1 shows a comparison of the fulfilled properties with previously presented lock-free garbage collection schemes. All of the schemes fulfill properties 1 and 3, whereas only a subset of the other properties is met by the previously presented schemes.

4. The New Lock-Free Algorithm

In order to fulfill all of the requested properties in Section 3 as well as providing an efficient and practical method, our aim is to devise a reference counting method which can also employ the *hazard pointer* (HP) scheme of Michael [9, 10]. Roughly speaking, hazard pointers are used to guarantee the safety of local references and reference counts are used to guarantee the safety of internal links in the data structure. Thus, the reference count of each node should indicate the number of globally accessible links that reference that node. Figure 1 describes the node structure as it is used in our

algorithm. As in the HP scheme, each thread maintains a list of nodes that are deleted but not yet reclaimed, and this list is scanned for possible reclamation when its length has reached a certain threshold (i.e. THRESHOLD_2). Some of the deleted nodes might be prevented from reclamation because of a fixed number of hazard pointers, while some deleted nodes might be prevented because of a positive reference count adherent to links. Thus, it is important to keep the number of references to deleted nodes from links to a minimum. Before we continue with the techniques for bounding the size of the deletion lists, we introduce an assumption about what could be required by the lock-free data structure algorithm:

Assumption 1 *For each of the links in a deleted node that reference a deleted node, it should be possible to replace it with a reference to an active node, with retained semantics for any of the involved threads.*

The intuition behind this assumption lays behind an observation why links of a deleted node should be useful to dereference by a thread that has a local reference to it. The thread with a local reference to a deleted node surely wants to find an appropriate active node and therefore takes advantage of the links. If the corresponding reference also adheres to a deleted node, the previous step is repeated. From the point of view of the thread of interest, it would not make any difference if some other thread helped with the procedure and already made sure that the links of the deleted node all references active node. The procedure of replacing the links of a deleted node with references to active nodes is called *clean-up*.

As described earlier, besides hazard pointers, nodes in the deletion lists are possibly prevented from reclamation by links in other deleted nodes. These nodes might be in the same deletion list or in some other thread's deletion list. For this reason, all threads' deletion lists are accessible for reading by any thread. When the length of the deletion list reaches a certain threshold (THRESHOLD_1) the thread performs a clean-up of all the nodes in its deletion list. If all of the nodes are still prevented from reclamation, this must be due to nodes in some other thread's deletion list, and thus the thread tries to perform a clean-up of all the other threads' deletion lists as well. As this procedure is repeated until the length of the deletion list is below the threshold, the amount of deleted nodes that are not yet reclaimed is bounded. The actual calculation of THRESHOLD_1 is described in Section 4.2. The threshold THRESHOLD_2 is set according to the HP scheme and is less than or equal to THRESHOLD_1.

4.1. Application Programming Interface

Figure 2 describes the functions for safe handling of the reference counted nodes.

```
/* Global variables */
HP[NR_THREADS][NR_INDICES]: pointer to Node;
DL_Nodes[NR_THREADS][THRESHOLD_1]: pointer to Node;
DL_Claims[NR_THREADS][THRESHOLD_1]: integer;
DL_Done[NR_THREADS][THRESHOLD_1]: boolean;
/* the above matrixes should be initialized to the values of
NULL, NULL, 0 respective false */

/* Local static variables */
threadId: integer; /* Unique and fixed number for each thread
  between 0 and NR_THREADS-1 */
dlist: integer; /* Initially ⊥ */
dcount: integer; /* Initially 0 */
DL_Nexts[THRESHOLD_1]: integer;

/* Local temporary variables */
node, node1, node2, old: pointer to Node;
thread, index, new_dlist, new_dcount: integer;
plist: array of pointer to Node;

function DeRefLink(link:pointer to pointer to Node):
  pointer to Node
D1    Choose index such that HP[threadId][index]=NULL
D2    while true do
D3        node := *link;
D4        HP[threadId][index] := node;
D5        if *link = node then
D6            return node;

procedure ReleaseRef(node:pointer to Node)
R1    Choose index such that HP[threadId][index]=node
R2    HP[threadId][index]:= NULL;

function CompareAndSwapRef(link:pointer to pointer to Node,
  old: pointer to Node, node: pointer to Node): boolean
C1    if CAS(link,old,node) then
C2        if node ≠ NULL then
C3            FAA(&node.mm_ref,1);
C4            node.mm_trace:=false;
C5        if old ≠ NULL then FAA(&old.mm_ref,-1);
C6        return true;
C7    return false;

procedure StoreRef(link:pointer to pointer to Node,
  node: pointer to Node)
S1    old := *link;
S2    *link := node;
S3    if node ≠ NULL then
S4        FAA(&node.mm_ref,1);
S5        node.mm_trace:=false;
S6    if old ≠ NULL then FAA(&old.mm_ref,-1);

function NewNode : pointer to Node
NN1  node := Allocate the memory of node (e.g. using malloc)
NN2  node.mm_ref := 0;
NN3  node.mm_del := false;
NN4  Choose index such that HP[threadId][index]=NULL
NN5  HP[threadId][index] := node;
NN6  return node;

procedure DeleteNode(node:pointer to Node)
DN1  ReleaseRef(node);
DN2  node.mm_del := true; node.mm_trace := false;
DN3  Choose index such that DL_Nodes[threadId][index]=NULL
DN4  DL_Done[threadId][index]:=false;
DN5  DL_Nodes[threadId][index]:=node;
DN6  DL_Nexts[index]:=dlist;
DN7  dlist := index; dcount := dcount + 1;
DN8  while true do
DN9       if dcount = THRESHOLD_1 then CleanUpLocal();
DN10      if dcount ≥ THRESHOLD_2 then Scan();
DN11      if dcount = THRESHOLD_1 then CleanUpAll();
DN12      else break;
```

Figure 2. Reference counting functions

The function *DeRefLink* safely de-references a given link, and sets a hazard pointer to the de-referenced node, thus guaranteeing the future safety to access the returned node. The procedure *ReleaseRef* should be called when a given node will not be accessed by the current thread anymore. It will clear the corresponding hazard pointer.

To update a link for which there might be concurrent updates to the link, the function *CompareAndSwapRef* should be used, which gives result whether the update was successful or not. The procedure will make sure that any thread that calls *DeRefLink* on the link can safely do so, if the thread has a hazard pointer reference to the node which contains the link. The requirements are that the calling thread of *CompareAndSwapRef* should have a hazard pointer to the given node that should be stored.

To update a link for which there cannot be any concurrent updates the procedure *StoreRef* should be called. The procedure will make sure that any thread that calls *DeRefLink* on the link can safely do so, if the thread has a hazard pointer reference to the node which contains the link. The requirements are that the calling thread of *StoreRef* should have a hazard pointer to the given node that should be stored, and that no other thread will possibly write concurrently to the link (otherwise *CompareAndSwapRef* should be invoked instead).

The function *NewNode* allocates a new node, sets a free hazard pointer to it for guaranteeing the future safety for access, and then returns it. The procedure *DeleteNode* should be called when a node is removed from the data structure and which memory should be possible to reclaim for reuse. The user operation that called *DeleteNode* is responsible for removing all references to the deleted node from the active nodes in the data-structure. This is similar to what is required when using a memory allocator in a sequential data-structure. The memory manager will not reclaim the deleted node until it is safe to do so.

Callbacks Figure 3 outlines the callbacks that have to be defined by the designer of each specific data structure. The procedure *TerminateNode* will make sure that none of the links in the given node will have any claim on any other node. *TerminateNode* is called on a deleted node when there are no claims from any other node or thread to the node. The procedure *CleanUpNode* will make sure that all claimed references from the links of the given node will only point to active nodes, thus removing redundant passages through an arbitrary number of deleted nodes.

Auxiliary Procedures Figure 4 describes auxiliary functions for internal use by the reference counting scheme. The procedure *Scan* will search through all not yet reclaimed nodes deleted by this thread and reclaim only those that does not have any matching hazard pointer and do not have any counted references from any links inside of nodes.

```
procedure TerminateNode(node:pointer to Node,concurrent:boolean)
TN1  if not concurrent then
TN2      for all x where link[x] of node is reference-counted do
TN3          StoreRef(node.link[x],NULL);
TN4  else
TN5      for all x where link[x] of node is reference-counted do
TN6          repeat node1 := node.link[x];
TN7          until CompareAndSwapRef(&node.link[x],node1,NULL);

procedure CleanUpNode(node:pointer to Node)
CN1  for all x where link[x] of node is reference-counted do
     retry:
CN2      node1:=DeRefLink(&node.link[x]);
CN3      if node1 ≠ NULL and node1.mm_del then
CN4          node2:=DeRefLink(&node1.link[x]);
CN5          CompareAndSwapRef(&node.link[x],node1,node2);
CN6          ReleaseRef(node2);
CN7          ReleaseRef(node1);
CN8          goto retry;
CN9      ReleaseRef(node1);
```

Figure 3. Callback functions

The procedure *CleanUpLocal* will try to remove redundant claimed references from links in deleted nodes that has been deleted by this thread. The procedure *CleanUpAll* will try to remove redundant claimed references from links in deleted nodes that have been deleted by any thread.

4.2. Algorithm Correctness and Bounds on Unreclaimed Memory

Theorem 1 *The algorithm implements a lock-free and linearizable scheme for garbage collection.*

Theorem 2 *For each thread p_i the maximum number of deleted but not reclaimed nodes in the dlist for p_i is at most $N \cdot (k + l_{max} + \alpha + 1)$, where N is the number of threads in the system, k is the number of hazard pointers per thread, l_{max} is the maximum number of links a node can contain and α is the maximum number of links in live nodes that may transiently point to a deleted node. (The number depends on the application.)*

Corollary 1 *The cleanup threshold, THRESHOLD_1, used by the algorithm should be set to $N \cdot (k + l_{max} + \alpha + 1)$.*

Corollary 2 *The number of deleted but not yet reclaimed nodes in the system is bounded from above by $N^2 \cdot (k + l_{max} + \alpha + 1)$*

Due to space restrictions, the corresponding proofs of the above theorems are left to the full version of the paper.

5. Conclusions

To the best of our knowledge, we have presented the first lock-free algorithmic implementation of a lock-free garbage collection scheme based on reference counting that has all the following features: i) guarantees the safety of local as

```
procedure CleanUpLocal()
CL1  index := dlist;
CL2  while index ≠ ⊥ do
CL3      node:=DL_Nodes[threadId][index];
CL4      CleanUpNode(node);
CL5      index := DL_Nexts[index];

procedure CleanUpAll()
CA1  for thread := 0 to NR_THREADS-1 do
CA2      for index := 0 to THRESHOLD_1-1 do
CA3      node:=DL_Nodes[thread][index];
CA4      if node ≠ NULL and not DL_Done[thread][index] then
CA5          FAA(&DL_Claims[thread][index],1);
CA6          if node = DL_Nodes[thread][index] then
CA7              CleanUpNode(node);
CA8          FAA(&DL_Claims[thread][index],-1);

procedure Scan()
SC1  index := dlist;
SC2  while index ≠ ⊥ do
SC3      node:=DL_Nodes[threadId][index];
SC4      if node.mm_ref = 0 then
SC5          node.mm_trace := true;
SC6          if node.mm_ref ≠ 0 then node.mm_trace := false;
SC7      index := DL_Nexts[index];
SC8  plist := ∅; new_dlist:=⊥; new_dcount:=0;
SC9  for thread := 0 to NR_THREADS-1 do
SC10     for index := 0 to NR_INDICES-1 do
SC11         node := HP[thread][index];
SC12         if node ≠ NULL then
SC13             plist := plist + node;
SC14 Sort and remove duplicates in array plist
SC15 while dlist ≠ ⊥ do
SC16     index := dlist;
SC17     node:=DL_Nodes[threadId][index];
SC18     dlist := DL_Nexts[index];
SC19     if node.mm_ref = 0 and node.mm_trace and node ∉ plist then
SC20         DL_Nodes[threadId][index]:=NULL;
SC21         if DL_Claims[threadId][index] = 0 then
SC22             TerminateNode(node,false);
SC23             Free the memory of node
SC24             continue;
SC25         TerminateNode(node,true);
SC26         DL_Done[threadId][index]:=true;
SC27         DL_Nodes[threadId][index]:=node;
SC28     DL_Nexts[index]:=new_dlist;
SC29     new_dlist := index;
SC30     new_dcount := new_dcount + 1;
SC31 dlist := new_dlist;
SC32 dcount := new_dcount;
```

Figure 4. Internal functions.

well as global references, ii) provides an upper bound of deleted but not yet reclaimed nodes, iii) is compatible with arbitrary memory allocation schemes, and iv) uses atomic primitives which are available in modern architectures.

We have performed experiments with applying the new scheme on the lock-free deque algorithm by Sundell and Tsigas [15]. Results indicate that our new lock-free garbage collection scheme can significantly improve the performance and reliability of implementations of lock-free dynamic data structures that require the safety of global references. We believe that our implementation is a powerful complement to Michael's hazard pointer scheme in the aim of designing highly efficient dynamic data structures.

References

[1] D. Detlefs, P. Martin, M. Moir, and G. Steele Jr. Lock-free reference counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, Aug. 2001.

[2] A. Gidenstam, M. Papatriantafilou, and P. Tsigas. Allocating memory in a lock-free manner. In *Proceedings of the 13th Annual European Symposium on Algorithms*, pages 329–242. LNCS vol. 3669, Springer Verlag, 2005.

[3] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, Jan. 1991.

[4] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Dynamic-sized lock-free data structures. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 131–131. ACM Press, 2002.

[5] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structure. In *Proceedings of 16th International Symposium on Distributed Computing*, Oct. 2002.

[6] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[7] W. H. Hesselink and J. F. Groote. Wait-free concurrent memory management by create and read until deletion (CaRuD). *Distributed Computing*, 14(1):31–39, Jan. 2001.

[8] P. Jayanti. A complete and constant time wait-free implementation of cas from ll/sc and vice versa. In *DISC 1998*, pages 216–230, 1998.

[9] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, pages 21–30, 2002.

[10] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(8), Aug. 2004.

[11] M. M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, June 2004.

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

[13] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing*, Aug. 1997.

[14] H. Sundell. Wait-free reference counting and memory management. In *Proceedings of the 19th International Parallel & Distributed Processing Symposium*. IEEE, Apr. 2005.

[15] H. Sundell and P. Tsigas. Lock-free and practical doubly linked list-based deques using single-word compare-and-swap. In *Proceedings of the 8th International Conference on Principles of Distributed Systems*, pages 240–255. LNCS vol. 3544, Springer Verlag, Dec. 2004.

[16] J. D. Valois. *Lock-Free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, 1995.