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

Anders Gidenstam, *Member, IEEE,* Marina Papatriantafilou, Håkan Sundell and Philippas Tsigas

Abstract— **We present an efficient and practical lock-free method for semi-automatic (application-guided) memory reclamation based on reference counting, aimed for use with arbitrary lock-free dynamic data structures. The method guarantees the safety of local as well as global references, supports arbitrary memory reuse, uses atomic primitives that are available in modern computer systems, and provides an upper bound on the amount of memory waiting to be reclaimed. To the best of our knowledge, this is the first lock-free method that provides all of these properties. We provide analytical and experimental study of the method. The experiments conducted have shown that the method can also provide significant performance improvements for lock-free algorithms of dynamic data structures that require strong memory management.**

Index Terms— **Memory Management, Memory Reclamation, Semi-Automatic, Garbage Collection, Data Structures, Lock-free, Shared Memory.**

I. Introduction

MEMORY MANAGEMENT is essential for building applications utilizing dynamic concurrent data structures. Support for memory management is available at various levels and strengths, ranging from operating system level to language level and from manual to automatic. Memory management involves two main dynamic tasks, *memory allocation* and *memory reclamation*. Memory allocation deals with the process of reserving memory and memory reclamation deals with the process of finding unused memory that can be *reclaimed*.

Concurrent algorithms for data structures and their related memory management are commonly based on mutual exclusion. However, mutual exclusion causes blocking and can consequently incur serious problems such as deadlocks, priority inversion or starvation. In addressing these problems, researchers have been working towards enabling concurrent access to data using algorithms which are not based on mutual exclusion. *Lock-free* algorithms [17], [7] fall within this effort; they guarantee that always at least one operation can progress, independently of the actions taken by the concurrent operations. Wait-free [6] algorithms satisfy the requirement of lock-free algorithms, and moreover, guarantee that all operations finish in a finite number of their own steps, regardless of the actions taken by the concurrent operations.

Lock-free and wait-free algorithms are also called *non-blocking*. They make use of atomic primitives provided by the hardware. These may be atomic read/write memory operations, such as *Compare-And-Swap* and *Fetch-And-Add*, or a pair of atomic read and atomic conditional write operations, such as *LoadLinked/StoreConditional*. It is important in non-blocking algorithms that the effects of the concurrent operations can be observed by the involved processes/threads in a consistent manner. The common consistency or safety requirement is *linearizability* [12]. An implementation of a shared object is linearizable if it guarantees that even when operations overlap in time, each of them appears to take effect at an atomic time instant that lies in its respective time duration, in a way that the effect of each operation is in agreement with a sequential execution of the operations on the object.

Dynamic data structures typically consist of a set of memory segments, called *nodes*, interconnected by referencing each other in some pattern. The references are typically implemented by *pointers* (a.k.a. *links*), that can identify each individual node by the means of its memory address. Each node may contain a number of pointers which reference other nodes. In a dynamic and concurrent data structure, arbitrary nodes can continuously and concurrently be added or removed. As systems have limited amount of memory, the memory occupied by these nodes needs to be dynamically allocated from and returned to the system. Hence, we need to have *dynamic memory allocation* (i.e., methods to allocate and free memory). Furthermore, we also need *dynamic memory reclamation* that can reclaim memory in a way that the problem of *dangling pointers* is avoided. If a memory block is reclaimed, while there still are pointers (either globally accessible or owned by particular threads) referencing that block, a subsequent access to the memory block(s) referenced by those pointers might be fatal to the application or even the system. The *strength* of a memory reclamation scheme is basically a measurement of its ability to avoid dangling pointers. As advanced data structures typically involve more pointers compared to simpler data structures, the need for stronger memory management also typically increases.

In the context of lock-free data structures there is an important distinction between a node being *logically deleted* (or *deleted* for short), i.e., no longer being part of the active part of the structure, and *reclaimed*, i.e., it's memory being released for reuse. A deleted node may still be accessed by concurrent operations while any access to a reclaimed node is erroneous.

We call a memory reclamation method that prevents dangling pointers in some type of references (when used correctly) *safe* with respect to that type of references. Further, depending on its *strength*, the method can guarantee the *safety* of local references

This is an extended version of the paper with the same title that appeared in the Proceedings of the 2005 International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN 2005), Las Vegas, USA, 7-9 December, 2005, pp. 202–207, IEEE Press.

A. Gidenstam is with Department 1: Algorithms and Complexity, Max-Planck-Institut für Informatik, 66123 Saarbrücken, GERMANY. E-mail: andersg@mpi-inf.mpg.de

M. Papatriantafilou and P. Tsigas are with the Department of Computer Science and Engineering, Chalmers University of Technology, SWEDEN. E-mail: {ptrianta|tsigas}@cs.chalmers.se

H. Sundell is with the School of Business and Informatics, University College of Borås, SWEDEN. E-mail: Hakan.Sundell@hb.se

and/or global references. Note that a memory reclamation method of low strength might be safe only with respect to local references.

In this paper we focus on practical and efficient memory reclamation in the context of lock-free dynamic data structures. For an operation of an algorithm to be lock-free, all sub-operations must also be (at least) lock-free. Consequently, lock-free dynamic data structures typically require lock-free memory reclamation. Common automatic memory reclamation methods (e.g., *garbage collection*; cf. subsequent subsection for examples) may possibly work safely with the lock-free application, but they would not preserve the lock-free property of the latter, due to the typical use of stop-the-world and similar techniques. Our methodology is lock-free and *semi-automatic*, requiring some *application guidance*, namely on the nodes that should be put under surveillance for possible future reclamation. This guidance is to be given by the programmer of the lock-free data structure, who has commonly the appropriate level of insight to provide it. Before describing more precisely the contribution of this paper, let us give an overview of related work, to facilitate putting the new method into perspective.

Related work

There is a considerable body of research on the use of reference counting for memory reclamation or garbage collection, starting with classic non-deferred reference counting [2] and later deferred reference counting [4]. Several types of optimizations, both for compile time and for runtime have been proposed, often focused on identifying and removing nonessential reference count updates due to manipulation of local references [1], [24], [16]. As mentioned above, there are no general lock-free garbage collection methods for practical use. However, some of the optimization techniques for local references might be applicable also to lock-free memory reclamation methods.

Herlihy and Moss [11] presented a garbage collector which, as stated in the paper, is not fully lock-free (as used in our terminology) as it cannot handle halting processes. The algorithm was the first in the literature towards lock-free garbage collection and was designed in a quite high level. It uses excessive versioning and copying of objects for every update and this renders it not generally practical, as also mentioned in the same paper. Moreover, the scheme is not a fully shared collector since it requires the memory to be partitioned (not shared) among the processes.

Valois [32] followed by Michael and Scott [21] presented a memory allocation scheme for fixed-sized memory segments; this scheme has to be used in combination with the corresponding memory reclamation scheme. Lock-free memory allocation schemes for general use have been presented by Michael [20], Gidenstam et al. [5] and Schneider et al. [25]. In the context of wait-free memory management, a wait-free extension of Valois' reference counting scheme and memory allocator has been presented by Sundell [27]. Hesselink and Groote [13] have presented a wait-free memory management scheme that is restricted to the specific problem of sharing tokens.

Various lock-free memory reclamation schemes have been presented in the literature. Michael [18], [19] proposed the hazard pointer method, that focuses on local references: each thread maintains a list of pointers (hazard pointers) of the nodes the thread may later access; when a node is removed from the data structure, the hazard pointer lists of all threads need to be checked before the node is reclaimed. A similar scheme was independently proposed by Herlihy et al. [10]. In its original form the latter scheme uses unbounded tags and is based on the double-width CAS atomic primitive, a compare-and-swap operation that can atomically update two adjacent memory words. This operation is available in some 32-bit architectures, but only in very few of the current 64-bit architectures. More recently, in [9] Herlihy et al. showed how to remove the tags from their method, to allow it to be implemented using single-width compare-and-swap.

The aforementioned schemes only guarantee the safety of local pointers from the threads and not the safety of pointers inside dynamically allocated nodes. Hence, 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 nodes. This constraint can be strongly restrictive, and may force the data-structure algorithms to retry traversals in the possibly large data structures, with resulting large performance penalties that increase with the level of concurrency.

Memory reclamation schemes that are based on reference counting can guarantee the safety of global as well as local references to objects. Valois et al. [32], [21] 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 method for memory allocation (since the reference counter field of a node must remain intact even after it has been reclaimed). Detlefs et al. [3] presented a scheme that allows arbitrary reuse of reclaimed memory, but it is based on double-word CAS, which is a compare-and-swap operation that can atomically read and update two arbitrary memory words. This instruction is not available in common contemporary processor architectures. Herlihy et al. [8], [23], [9] presented a modification of the previous scheme by Detlefs et al. to only use single-word CAS (compare-and-swap) for the reference counting part.

A problem with reference counting techniques in concurrent environments, which was identified in [21], is that a single local reference from a slow thread could potentially prevent (due to the ability to create recursive references) an arbitrary number of nodes from being reclaimed. Consider for example a chain of nodes that has been removed from a singly linked list in order from the front to back, and consider also a slow thread holding a reference to the first deleted node: this node cannot (currently) be reclaimed and would still contain a reference to the next (subsequently) deleted node, preventing it, too, from being reclaimed and so on.

Contributions of this article

This paper combines the strength of reference counting with the efficiency of hazard pointers, into a general lock-free memory reclamation scheme, with the aim of keeping only the advantages of the involved techniques while avoiding the respective drawbacks. The new memory reclamation method, called BEWARE&CLEANUP, is lock-free and offers linearizable operations to the application, is compatible with arbitrary schemes for memory allocation, can be implemented using commonly available atomic primitives and guarantees the safety of local as well as global references. BEWARE&CLEANUP also guarantees that only a bounded amount of memory could be temporarily withheld from reclamation by any thread. Like the hazard pointers method, the new method is application-guided in the sense that it requires information from the application regarding which nodes should be put under surveillance for possible future reclamation. Table I summarizes the properties of the presented method in contrast with the other lock-free reclamation methods in the literature. These properties are explained in more detail in Section II.

The rest of the paper is organized as follows. Section II describes the solution requirements for the problem we are focusing on, followed by a description of the system model, presented in Section III. The proposed method, BEWARE&CLEANUP, is described in Section IV. In Section V we show the correctness of BEWARE&CLEANUP by proving the lock-free and linearizability properties, as well as by proving an upper bound on the amount of memory that can be withheld from reclamation by the BEWARE&CLEANUP method . Section VI presents an experimental evaluation of BEWARE&CLEANUP in the context of use in a lock-free data structure. We conclude the paper with Section VII.

II. PROBLEM DESCRIPTION AND SOLUTION REQUIREMENTS

We focus on solving the memory reclamation problem in the context of dynamic lock-free data structures. As explained in the introduction, such data structures typically consist of *nodes*, i.e., memory segments, interconnected by referencing each other using *pointers* (to their memory addresses), also called *links*. The operation to gain access to a referenced node through a link is called *dereferencing*. Some nodes are typically permanent parts of the data structure, all other nodes are part of the data structure only 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 memory occupied by these nodes needs to be dynamically allocated from and returned to the system.

In a concurrent environment, before freeing a node, it should be checked that there are no remaining references to it; this should also include possible *local* references to a node that any *thread* might have, as a read or write access to the memory of a reclaimed node might be fatal to the correctness of the data structure and/or to the whole system. This is the *memory reclamation* problem. To correctly decide about reclaiming nodes, the memory reclamation method should thus satisfy the following property:

Property *1:* The memory reclamation method should only reclaim nodes that are not part of the data structure and for which there cannot be any future accesses by any thread.

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

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

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, memory reclamation systems typically need to interact with the involved threads and also get involved into how these threads access the nodes, e.g., by providing special operations for dereferencing links and by demanding that the data structure implementation explicitly calls the memory reclamation system when a node has been logically deleted.

Moreover, when used for supporting lock-free dynamic data structure implementations, the memory reclamation method also has to guarantee these features:

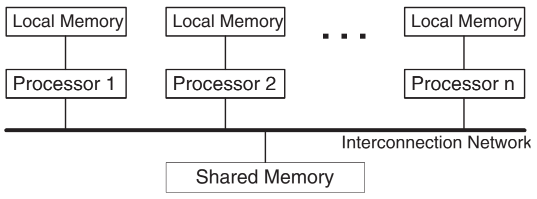


Fig. 1. Shared memory multiprocessor system structure.

Property *3:* All operations of the memory reclamation method for communication with the application (or data structure implementation) should be lock-free and linearizable.

In order to minimize the total amount of occupied memory for the various data structures, the following property is useful:

Property *4:* The memory that is reclaimed by the method should be accessible for any arbitrary future reuse in the system; i.e., the memory reclamation method should be compatible with the memory allocator (the system's default allocator or other allocator that can be used).

In a concurrent environment it may frequently occur that a thread is holding a local reference to a node that has been logically 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 still be able to use the links in the deleted node, e.g., in search procedures in large data structures:

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

The BEWARE&CLEANUP method proposed 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. As with the hazard pointers method, the method proposed here needs some application-guidance, namely information on which nodes have become logically deleted (i.e., may eventually cease to be *live*) and thus are targets for future reclamation. Table I shows a comparison with previously presented lock-free memory reclamation schemes with respect to these properties. All of the schemes fulfill properties 1 and 3, whereas only a subset of the other properties is met by each of the previous schemes.

III. SYSTEM MODEL

A typical abstraction of a shared memory multi-processor system configuration is depicted in Fig. 1. Each node of the 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 via multiprogramming. The co-operating tasks, possibly running on different processors, use shared data objects built in the shared memory to co-ordinate 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 is not necessarily uniformly accessible for all nodes in the system; processors can have different access times to different parts of the shared 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

	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 method: BEWARE&CLEANUP	Yes	Yes	Yes	Yes
LFRC [3]	Yes	No [e]	Yes	No [a]
Pass-the-buck [9], [10]	No	Yes	Yes	Yes [b]
SLFRC [9], [8]	Yes	No [e]	Yes	Yes [c]
SMR [18], [19]	No	Yes	Yes	Yes [d]
Valois et al. [32], [21]	Yes	No [e]	No	Yes

[a]The LFRC method uses the double-word compare-and-swap (DCAS) atomic primitive.

[b]The pass-the-buck (PTB) method originally used the double-width compare-and-swap atomic primitive.

[c]The SLFRC method is based on the LFRC and the pass-the-buck (PTB) method.

[d]The hazard pointers method 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 never be reclaimed.

TABLE I
PROPERTIES OF DIFFERENT APPROACHES TO LOCK-FREE MEMORY RECLAMATION.

```
procedure FAA(address:pointer to word, number:integer)
    atomic do
        *address := *address + number;

function CAS(address:pointer to word, oldvalue:word, newvalue:word):boolean
    atomic do
        if *address = oldvalue then *address := newvalue; return true;
        else return false;
```

Fig. 2. The Fetch-And-Add (FAA) and Compare-And-Swap (CAS) atomic primitives.

atomic primitives Fetch-And-Add (FAA) and Compare-And-Swap (CAS); see Fig. 2 for a description. These read-modify-write style operations are available on most common architectures or can be easily derived from other synchronization primitives with the same or higher consensus number [22], [15].

IV. THE BEWARE&CLEANUP LOCK-FREE METHOD

We first describe the basic idea of the BEWARE&CLEANUP method and in subsequent subsections we give a more detailed description. In order to fulfill all the requested properties described in Section II as well as in order to provide an efficient and practical method, the aim of BEWARE&CLEANUP is to devise a reference counting method which can also employ the *hazard pointer* (HP) scheme of Michael [18], [19]. Roughly speaking, hazard pointers are used to guarantee the safety of local references while reference counts are used to guarantee safety of internal links in the data structure. Thus, the reference count of each node indicates the number of globally accessible links that reference that node.

Recall that according to the hazard-pointers method, each thread maintains a list of pointers to nodes the thread may later access. When a node is removed from the data structure (i.e., it becomes logically deleted), it is included in a *deletion list*. Before freeing the memory segment occupied by the deleted node, the hazard-pointer lists of all threads are checked for references to the node. If any hazard pointer references the node, the latter cannot be reclaimed at this time and will remain in the deletion list. The number of hazard pointers needed for each thread (i.e., the value of k) is estimated as in the original HP method [18], [19] by determining the maximal number of simultaneously used local references by any thread in any procedure and nesting of calls.

Program 1 describes the node structure used in BEWARE&CLEANUP. As in the HP scheme, each thread maintains a list of nodes that it has deleted but not yet reclaimed. This is called its *deletion list*. A thread scans its deletion list for nodes that can be reclaimed when the length of the list has reached a certain threshold (i.e., THRESHOLD_S); this is set as in the HP method, i.e., depending on the number of local references used per thread. Some of the deleted nodes might not be *reclaimable* (i.e., safe to reclaim) due to hazard pointers pointing to them, while some deleted nodes might not be reclaimable due to a positive reference count adherent to links. Thus, it is important to keep the number of references from links to deleted nodes to a minimum.

To bound the size of deletion lists (hence the amount of memory that is deleted and not yet returned to the system), BEWARE&CLEANUP performs a *clean-up procedure*. This procedure can update links in deleted nodes and, hence, the reference count of deleted nodes referenced by other deleted nodes, provided that the following property is satisfied by the lock-free algorithm that implements the data structure and uses the BEWARE&CLEANUP method:

Assumption *1:* Each link in a deleted node that references another deleted node, can be replaced with a reference to an active node, with retained semantics for all of the involved threads.

The intuition behind this assumption is the following: Consider a thread t holding a local reference to a deleted node d. Now, the most common reason for t to be interested in dereferencing any of the links in d is that t was traversing the linked data structure when the node d was deleted by another thread. In this situation t needs to find its way back to the active part of the data structure again. One option for t would be to restart the traversal from the/one of the structure's root pointers, but this makes traversing operations extremely inefficient in cases where there is a significant number of concurrent delete operations. To avoid this, many algorithms for lock-free data structures make use of the pointers in deleted nodes to search for an active node to continue the traversal from [14], [29], [30]. In most such algorithms any traversal through a chain of deleted nodes to reach an active node can be replaced by a shortcut to the first active

Program 1 The Node structure used in BEWARE&CLEANUP.

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

Program 2 BEWARE&CLEANUP: global and local variables.

```
/* Global variables */
HP[NR_THREADS][NR_INDICES]:pointer to Node; /* Initially NULL */
DL_Nodes[NR_THREADS][THRESHOLD_C]:pointer to Node; /* Initially
  NULL */
DL_Claims[NR_THREADS][THRESHOLD_C]:integer; /* Initially 0 */
DL_Done[NR_THREADS][THRESHOLD_C]:boolean; /* Initially 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_C]: integer;

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

Program 3 Reference counting interface functions and procedures.

```
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, new: pointer to Node): boolean
C1    if CAS(link,old,new) then
C2        if new ≠ NULL then
C3            FAA(&new.mm_ref,1);
C4            new.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; node.mm_trace := false; node.mm_del := false;
NN3   Choose index such that HP[threadId][index]=NULL
NN4   HP[threadId][index] := node;
NN5   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_C then CleanUpLocal();
DN10      if dcount ≥ THRESHOLD_S then Scan();
DN11      if dcount = THRESHOLD_C then CleanUpAll();
DN12      else break;
```

node in the chain without affecting the effect of the traversal. This property is used by BEWARE&CLEANUP to *clean-up* links in deleted nodes.

The clean-up works as follows: as described earlier, besides hazard pointers, nodes in the deletion lists are possibly unavailable for reclamation due to links from other deleted nodes. These nodes might be in the same deletion list or in some other thread's deletion list. For this reason, the deletion lists of all threads are accessible for reading by any thread. When the length of a thread's deletion list reaches a certain threshold (THRESHOLD_C) the thread performs a clean-up of all nodes in its deletion list. If none of these nodes become reclaimable after the clean-up, this must be due to references from nodes in the deletion lists of other threads, and thus the thread tries to perform a clean-up of all 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_C is described in Section V-B. THRESHOLD_S is set according to the HP scheme and is less than or equal to THRESHOLD_C.

A. Overview of Application Programming Interface

The following functions and procedures are defined for safe handling of the reference counted nodes in a user program or data structure:

function DeRefLink(link:**pointer to pointer to** Node): **pointer to** Node

The function *DeRefLink* safely dereferences the given link, setting a hazard pointer to the dereferenced node, thus guaranteeing the safety of future accesses to the returned node. In particular the calling thread can safely dereference and/or update any links in the returned node subsequently.

procedure ReleaseRef(node:**pointer to** Node)

The procedure *ReleaseRef* should be called when the given node will not be accessed by the current thread anymore. It clears the corresponding hazard pointer.

function CompareAndSwapRef(link:**pointer to pointer to** Node,old:**pointer to** Node, new:**pointer to** Node): **boolean**

The function *CompareAndSwapRef* is used to update a link for which there might be concurrent updates. It returns true if the update was successful and false otherwise. The thread calling *CompareAndSwapRef* should have a hazard pointer to the node that is to be stored in the link.

procedure StoreRef(link:**pointer to pointer to** Node, node:**pointer to** Node)

The procedure *StoreRef* should be used when updating a link where there cannot be any concurrent updates. The requirements are that the thread calling *StoreRef* has a hazard pointer to the node that should be stored, and that no other thread could possibly write concurrently to the link (in that case *CompareAndSwap-*

Ref should be invoked instead).

function NewNode:**pointer to** Node

The function *NewNode* allocates a new node. A hazard pointer is set to the node to guarantee the safety of future accesses to it.

procedure DeleteNode(node:**pointer to** Node)

The procedure *DeleteNode* should be called when a node has been logically removed from the data structure and its memory should eventually be reclaimed. The user operation that called *DeleteNode* is responsible[1] for removing all references to the deleted node from the active nodes in the data structure. This is similar to what is required when calling a memory allocator directly in a sequential data structure. However, independently of the state at the call of *DeleteNode* and of when concurrent operations eventually remove their (own or created) references to the deleted node, BEWARE&CLEANUP will not reclaim the deleted node until it is safe to do so, i.e., when there are no threads that could potentially access the node anymore, thus completely avoiding the possibility of dangling pointers.

In Section IV-D we give an example of how these functions can be used in the context of a lock-free queue algorithm based on linked lists.

Callbacks: The following functions are callbacks that have to be defined by the designer of each specific data structure. They are called by BEWARE&CLEANUP.

procedure CleanUpNode(node:**pointer to** Node)

The procedure *CleanUpNode* makes sure that all references from the links in the given node point to active nodes, thus removing redundant reference chains passing through an arbitrary number of deleted nodes.

procedure TerminateNode(node:**pointer to** Node, concurrent:**boolean**)

TerminateNode makes sure that none of the node's contained links 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.[2] When the argument `concurrent` is false BEWARE&CLEANUP guarantees that there cannot be any concurrent updates of the node, thereby allowing *TerminateNode* to use the cheaper *StoreRef* instead of *CompareAndSwapRef* to update the node's links.

B. Auxiliary procedures

Auxiliary operations that are defined for internal use by the reference counting algorithm:

procedure Scan()

It searches through all not yet reclaimed nodes deleted by this thread and reclaims those that do not have any matching hazard pointer and do not have any counted references adhering from any links contained in any of the nodes in the system.

procedure CleanUpLocal()

It aims at updating nodes deleted by this thread so that all their links referencing other deleted nodes are replaced with links

[1]After the call of *DeleteNode*, concurrent operations may still use references to the deleted node, and might even temporary add links to it, although concurrent operations that observe a deleted node are supposed to eventually remove all references and links to it.

[2]In principle this procedure could be provided by the memory manager, but in practice it is more convenient to let the user decide the memory layout of the node records. All node records would still be required to start with the `mm_ref`, `mm_trace` and `mm_del` fields.

Program 4 Example of user defined callback functions.

```
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 for all x where link[x] of node is reference-counted do
TN5          repeat node1 := node.link[x];
TN6          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);
```

Program 5 Internal operations.

```
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_C-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
SC7              node.mm_trace := false;
SC8      index := DL_Nexts[index];
SC9  plist := ∅; new_dlist:=⊥; new_dcount:=0;
SC10 for thread := 0 to NR_THREADS-1 do
SC11     for index := 0 to NR_INDICES-1 do
SC12         node := HP[thread][index];
SC13         if node ≠ NULL then
SC14             plist := plist + node;
SC15 Sort and remove duplicates in array plist
SC16 while dlist ≠ ⊥ do
SC17     index := dlist;
SC18     node:=DL_Nodes[threadId][index];
SC19     dlist := DL_Nexts[index];
SC20     if node.mm_ref = 0 and node.mm_trace and node ∉ plist then
SC21         DL_Nodes[threadId][index]:=NULL;
SC22         if DL_Claims[threadId][index] = 0 then
SC23             TerminateNode(node,false);
SC24             Free the memory of node
SC25             continue;
SC26         TerminateNode(node,true);
SC27         DL_Done[threadId][index]:=true;
SC28         DL_Nodes[threadId][index]:=node;
SC29     DL_Nexts[index]:=new_dlist;
SC30     new_dlist := index;
SC31     new_dcount := new_dcount + 1;
SC32 dlist := new_dlist;
SC33 dcount := new_dcount;
```

that reference live nodes instead. It invokes the callback function *CleanUpNode* on every node in the thread's deletion list.

procedure CleanUpAll()

It invokes the callback function *CleanUpNode* on every node in the deletion lists of all threads. This updates all links in deleted nodes that reference other deleted nodes, to reference live nodes instead.

C. Detailed description of the method

DeRefLink (Program 3), first reads the pointer to a node stored in `*link` at line D3. Then at line D4 it sets one of the thread's hazard pointers to point to the node. At line D5 it verifies that the link still points to the same node as before. If `*link` still points to the node, it knows that the node is not yet reclaimed and that it cannot be reclaimed until the hazard pointer now pointing to it is released. If `*link` has changed since the last read, it retries.

ReleaseRef (Program 3), removes this thread's hazard pointer pointing to `node`. Note that if `node` is deleted, has a reference count of zero, and no other hazard pointers are pointing to it, then the node can be reclaimed by *Scan* (invoked by the thread that called *DeleteNode* on the node).

CompareAndSwapRef (Program 3), performs a common CAS on the link and updates the reference counts of the respective nodes accordingly. Line C4 notifies any concurrent *Scan* that the reference count of `new` has been increased. Note that the node `new` is safe to access during *CompareAndSwapRef* since the thread calling *CompareAndSwapRef* is required to have a hazard pointer pointing to it. At line C5 the reference count of `old` is decreased. The previous reference count must have been greater than zero since `*link` referenced the node `old`.

StoreRef (Program 3), is valid to use only when there are no concurrent updates of `*link`. After updating `*link` at line S2 *StoreRef* increases the reference count of `node` at line S4. This is safe since the thread calling *StoreRef* is required to have a hazard pointer to `node`. Line S5 notifies any concurrent *Scan* that the reference count of `node` is non-zero. At line S6 the reference count of `old` is decreased. The previous reference count must have been greater than zero since `*link` referenced the node `old`.

NewNode (Program 3), allocates memory for the new node from the underlying memory allocator and initializes the header fields each node must have. It also sets a hazard pointer to the node.

DeleteNode (Program 3), marks the node `node` as logically deleted at line DN2. Then at the lines DN3 to DN7 the node is inserted into this thread's deletion list. By clearing `DL_Done` at line DN4, before writing the pointer at line DN5, concurrent *CleanUpAll* operations can access the node and tidy up its references.

If the number of deleted nodes in this thread's deletion list is larger than or equal to THRESHOLD_S a *Scan* is performed. It will then reclaim all nodes in the list that are not referenced by other nodes or threads.

If the thread's deletion list is now full, that is, it contains THRESHOLD_C nodes, the thread will first run *CleanUpLocal* at line DN9 to make sure that all of its deleted nodes only point to nodes that were alive when *CleanUpLocal* started. Then, it runs *Scan* at line DN10. If *Scan* is unable to reclaim any node at all then the thread will run *CleanUpAll*, which cleans up the nodes in all threads' deletion lists.

Callbacks: ***TerminateNode*** (Program 4), should clear all links in the node `node` by writing NULL to them. This is done by using either *CompareAndSwapRef* or *StoreRef*, depending on whether there might be concurrent updates of these links or not (as indicated by the argument `concurrent`).

CleanUpNode (Program 4), should make sure that none of the links in the node `node` points to nodes that were deleted before this invocation of *CleanUpNode* started.

Auxiliary procedures: ***Scan*** (Program 5), reclaims all nodes deleted by the current thread that are not referenced by any other node or any hazard pointer. To determine which of the deleted nodes can safely be reclaimed, *Scan* first sets the `mm_trace` bit of all deleted nodes that have reference count zero (lines SC1 to SC8). The check at line SC6 ensures that the reference count was indeed zero when the `mm_trace` bit was set.

Then *Scan* records all active hazard pointers of all threads in `plist` (lines SC9 to SC15). In the lines SC16 to SC31 *Scan* traverses all not yet reclaimed nodes deleted by this thread. For each of these nodes the tests at line SC20 determine if (i) the reference count is zero, (ii) the reference count has consistently been zero since before the hazard pointers were read (indicated by the `mm_trace` bit being set) and (iii) the node is not referenced by any hazard pointer. If all three of these conditions are true, the node is not referenced and *Scan* checks if there may be concurrent *CleanUpAll* operations working on the node at line SC22. If there are no such *CleanUpAll* operations, *Scan* uses *TerminateNode* to release all references the node might contain and then reclaims the node (lines SC23 and SC24). In case there might be concurrent *CleanUpAll* operations accessing the node, *Scan* uses the concurrent version of *TerminateNode* to set all the node's contained links to NULL. By setting the `DL_Done` flag at line SC27, before the node is reinserted into the list of unreclaimed nodes at line SC28, subsequent *CleanUpAll* operations cannot prevent any subsequent *Scan* from reclaiming this node.

CleanUpLocal (Program 5), traverses the thread's deletion list and calls *CleanUpNode* on each of the nodes in it to make sure that their contained links do not reference any node that was already deleted when *CleanUpLocal* started.

CleanUpAll (Program 5), traverses the `DL_Nodes` arrays (i.e., the deletion lists) of all threads and tries to make sure that none of the nodes it finds contain links to nodes that were already deleted when *CleanUpAll* started. The tests at line CA4 prevent *CleanUpAll* from interfering with *Scan* for nodes that have no references left. The test at line CA6 prevents *CleanUpAll* from accessing a node that *Scan* has already reclaimed. If the node is still present in `DL_Nodes[thread][index]` at line CA6 then a concurrent *Scan* accessing this node must be before line SC21 or be after line SC28 without having reclaimed the node.

D. Example application

The BEWARE&CLEANUP method can be applied for memory reclamation in lock-free algorithms for dynamic data structures in a straightforward manner, in a similar way to previously presented lock-free memory reclamation schemes. Program 6 shows the lock-free queue algorithm by Valois et al. [31], [21] as it would be integrated with the BEWARE&CLEANUP memory reclamation method. A further and more extensive example of a lock-free algorithm using BEWARE&CLEANUP is the lock-free doubly linked list data structure in [30].

Program 6 Example of a queue algorithm using the BEWARE&CLEANUP memory reclamation method.

```
structure QNode
        mm_ref: integer
        mm_trace: boolean
        mm_del: boolean
        next: pointer to QNode
        value: pointer to Value
/* Global variables */
head, tail:pointer to QNode

procedure InitQueue()
IQ1   node := NewNode(); node.next := NULL;
IQ2   head := NULL; StoreRef(&head,node);
IQ3   tail:= NULL; StoreRef(&tail,node);

function Dequeue():pointer to Value
DQ1   while true do
DQ2       node := DeRefLink(&head);
DQ3       next := DeRefLink(&node.next);
DQ4       if next = NULL then
DQ5           ReleaseRef(node); return NULL;
DQ6       if CompareAndSwapRef(&head,node,next)
DQ7       then break;
DQ8       ReleaseRef(node); ReleaseRef(next);
DQ9   DeleteNode(node);
DQ10  value := next.value; ReleaseRef(next);
DQ11  return value;

procedure Enqueue(value:pointer to Value)
EQ1   node := NewNode(); node.next := NULL; node.value := value;
EQ2   old := DeRefLink(&tail);
EQ3   prev := old;
EQ4   repeat
EQ5       while prev.next ≠ NULL do
EQ6           prev2 := DeRefLink(&prev.next);
EQ7           if old ≠ prev then ReleaseRef(prev);
EQ8           prev := prev2;
EQ9   until CompareAndSwapRef(&prev.next,NULL,node);
EQ10  CompareAndSwapRef(&tail,old,node);
EQ11  if old ≠ prev then ReleaseRef(prev);
EQ12  ReleaseRef(old); ReleaseRef(node);
```

E. Algorithm extensions

For simplicity reasons, the BEWARE&CLEANUP method in this paper is described with a fixed number of threads. However, the method can easily be extended to a dynamic number of threads using a similar extension to the one that extents HP to dynamic number of threads [19]. The global matrix of hazard pointers (*HP*) can be turned into a linked list of arrays. The deletion lists can also be linked into a global chain, and as the size of the deletion lists changes, old redundant deletion lists can be safely reclaimed by using an additional HP scheme for memory reclamation.

V. CORRECTNESS PROOF

In this section we present the correctness proof of the BEWARE&CLEANUP method. The outcome of the correctness proof are the following theorems that state the most important properties of the BEWARE&CLEANUP method.

Theorem *1:* BEWARE&CLEANUP is a lock-free and linearizable method for memory reclamation.

Theorem *2:* The number of deleted but not yet reclaimed by BEWARE&CLEANUP nodes in the system is bounded from above by

$$N^2 \cdot (k + l_{max} + \alpha + 1),$$

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 above theorems are proved below using a series of lemmas. We first prove that BEWARE&CLEANUP does not reclaim memory that could still be accessed; then we prove an upper bound on the amount of such deleted but unreclaimed garbage there can be accumulated; last we prove that the method is linearizable and lock-free [12]. A set of definitions that will help us to structure and shorten the proof is given first.

Definition *1:* Let F_t denote the state of the pool of free nodes at time t. We interpret $n \in F_t$ to be true when n has been freed as per line SC24 in *Scan*. Any linearizable, preferably lock-free, memory allocator can be used to manage the free pool.

Definition *2:* Let $n \in HP_t(p)$ denote that thread p has a verified hazard pointer set to point to node n at time t. A verified hazard pointer is one that has been or will be returned by a *DeRefLink* or *NewNode* operation. The array of hazard pointers in the implementation, the array HP, may also contain pointers temporarily set by *DeRefLink* operations that will be forced to retry, but these are not considered as part of the $HP_t(p)$ sets.

Definition *3:* Let $n \in DL_t(p)$ denote that node n is deleted and is awaiting reclamation in the dlist of thread p at time t.

Definition *4:* Let $Del_t(n)$ denote that the node n is marked as logically deleted at time t. Note that the deletion mark is not removed until the node is returned to the free pool.

Definition *5:* A *shared link* is either a global shared variable visible to the application or a pointer variable residing inside a node. Specifically, the elements in the per thread arrays of hazard pointers, HP, and the per thread arrays of deleted nodes, DL_Nodes, are *not* considered as shared links, since these are internal to BEWARE&CLEANUP.

Definition *6:* Let $Links(n)$ denote the set of shared links (pointers) present in node n.

Definition *7:* Let $l_x \mapsto_t n_x$ denote that the shared link l_x points to node n_x at time t and $l_x \mapsto_t \perp$ that the shared link l_x contains null at time t.

Definition *8:* Let $Ref_t(n)$ denote a set containing the shared links that point to node n at time t.

Definition *9:* A node n is said to be *reclaimable* at time t iff $Ref_t(n) = \emptyset$, $Del_t(n)$ is true and $\forall p : \; n \notin HP_t(p)$.

Definition *10:* A memory reclamation method is *safe* iff it never reclaims a node that is not *reclaimable* at the time of reclamation.

The API operations of the memory reclamation method need to be shown that are linearizable (atomic); namely, this applies to operations *DeRefLink* (DRL), *ReleaseRef* (RR), *NewNode* (NN), *DeleteNode* (DN), *CompareAndSwapRef* (CASR) and *StoreRef* (SR). For the safety and correctness of the memory reclamation the internal operations are also involved directly; for brevity, we introduce shorter variants of their names, namely: TN for *TerminateNode*, SCAN for *Scan*, CUN for *CleanUpNode*, CUL for *CleanUpLocal* and CUA for *CleanUpAll*.

In the following expressions which define the sequential semantics of our operations, the syntax is $S_1 : O_1, S2$, where S_1 is the conditional state before the operation O_1 and S_2 is the resulting state after the operation has been performed.

Definition *11:* The sequential semantics of the operations:
DeRefLink

$$\exists n_1 . l_1 \mapsto_{t_1} n_1 : \mathbf{DRL(l_1)} = n_1, n_1 \in HP_{t_2}(p_{curr}) \quad (1)$$

$$l_1 \mapsto_{t_1} \perp : \mathbf{DRL(l_1)} = \perp, \quad (2)$$

ReleaseRef

$$n \in HP_{t_1}(p_{curr}) : \mathbf{RR(n)}, n \notin HP_{t_2}(p_{curr}) \tag{3}$$

NewNode

$$\begin{gathered}\exists n_1 . n_1 \in F_{t_1} : \mathbf{NN}() = \mathbf{n_1}, n_1 \notin F_{t_2} \wedge Ref_{t_2}(n_1) = 0 \\ \wedge \neg Del_{t_2}(n_1) \wedge n_1 \in HP_{t_2}(p_{curr})\end{gathered} \tag{4}$$

DeleteNode

$$\begin{gathered}n_1 \in HP_{t_1}(p_{curr}) : \mathbf{DN(n_1)}, Del_{t_2}(n_1) \\ \wedge n_1 \in DL_{t_2}(p_{curr}) \wedge n_1 \notin HP_{t_2}(p_{curr})\end{gathered} \tag{5}$$

CompareAndSwapRef

$$\begin{gathered}l_1 \mapsto_{t_1} \perp \wedge n_2 \in HP_{t_1}(p_{curr}) : \mathbf{CASR(l_1, \perp, n_2) = True}, \\ l_1 \mapsto_{t_2} n_2 \wedge l_1 \in Ref_{t_2}(n_2) \wedge n_2 \in HP_{t_2}(p_{curr})\end{gathered} \tag{6}$$

$$\begin{gathered}\exists n_1 \,.\, l_1 \mapsto_{t_1} n_1 \wedge n_2 = n_1 : \mathbf{CASR(l_1, n_2, \perp) = True}, \\ l_1 \mapsto_{t_2} \perp \wedge l_1 \notin Ref_{t_2}(n_2)\end{gathered} \tag{7}$$

$$\begin{gathered}\exists n_1 \,.\, l_1 \mapsto_{t_1} n_1 \wedge n_2 = n_1 \wedge \\ n_3 \in HP_{t_1}(p_{curr}) \wedge l_1 \in Ref_{t_1}(n_2) : \\ \mathbf{CASR(l_1, n_2, n_3) = True}, l_1 \mapsto_{t_2} n_3 \wedge l_1 \notin Ref_{t_2}(n_2) \\ \wedge l_1 \in Ref_{t_2}(n_3) \wedge n_3 \in HP_{t_2}(p_{curr})\end{gathered} \tag{8}$$

$$\begin{gathered}\exists n_1 \,.\, l_1 \mapsto_{t_1} n_1 \wedge n_1 \neq n_2 \wedge n_3 \in HP_{t_1}(p_{curr}) : \\ \mathbf{CASR(l_1, n_2, n_3) = False}, \\ l_1 \mapsto_{t_2} n_1 \wedge n_3 \in HP_{t_2}(p_{curr})\end{gathered} \tag{9}$$

Scan

$$\begin{gathered}: \quad \mathbf{Scan}(), \forall n_i \in DL_{t_1}(p_{curr}).(n_i \in F_{t_2} \wedge \\ (\forall n_x \text{ s.t. } l_x \mapsto_{t_1} n_x \wedge l_x \in Links(n_i). \\ l_x \notin Ref_{t_2}(n_x)) \vee (\exists p_j . n_i \in HP_{t_1}(p_j)) \vee \\ (\exists n_j . n_j \notin F_{t_1} \wedge \exists l_x \in Links(n_j) . l_x \mapsto_{t_1} n_i)\end{gathered} \tag{10}$$

TerminateNode (Implemented by the application programmer).

$$\begin{gathered}n_1 \in DL_{t_1}(p_{curr}) : \mathbf{TerminateNode(n_1, c)}, \\ \forall l_x \in Links(n_1).(l_x \mapsto_{t_2} \perp \wedge \\ \forall n_x \text{ s.t. } l_x \mapsto_{t_1} n_x \,.\, l_x \notin Ref_{t_2}(n_x))\end{gathered} \tag{11}$$

CleanUpNode (Implemented by the application programmer).

$$\begin{gathered}\exists p_i . n_1 \in DL_{t_1}(p_i) \wedge Del_{t_1}(n_1) : \mathbf{CleanUpNode(n_1)}, \\ \forall l_x \in Links(n_1).(l_x \mapsto_{t_2} \perp \vee \\ (\exists n_x . l_x \mapsto_{t_2} n_x \wedge \neg Del_{t_1}(n_x)))\end{gathered} \tag{12}$$

CleanUpLocal

$$\begin{gathered}: \quad \mathbf{CleanUpLocal}(), \\ \forall n_i \in DL_{t_1}(p_{curr}).(\forall l_x \in Links(n_i). \\ l_x \mapsto_{t_2} \perp \vee (\exists n_x . l_x \mapsto_{t_2} n_x \wedge \neg Del_{t_1}(n_x)))\end{gathered} \tag{13}$$

CleanUpAll

$$\begin{gathered}: \quad \mathbf{CleanUpAll}(), \\ \forall p_i .(\forall n_j \in DL_{t_1}(p_i).(\forall l_x \in Links(n_j). \\ l_x \mapsto_{t_2} \perp \vee (\exists n_x . l_x \mapsto_{t_2} n_x \wedge \neg Del_{t_1}(n_x))))\end{gathered} \tag{14}$$

StoreRef (Should only be used to update links in nodes that are inaccessible to all other threads.)

$$\begin{gathered}l_1 \mapsto_{t_1} \perp \wedge n_2 \in HP_{t_1}(p_{curr}) : \mathbf{SR(l_1, n_2)}, \\ l_1 \mapsto_{t_2} n_2 \wedge l_1 \in Ref_{t_2}(n_2) \wedge n_2 \in HP_{t_2}(p_{curr})\end{gathered} \tag{15}$$

$$\begin{gathered}\exists n_1 . l_1 \mapsto_{t_1} n_1 \wedge n_2 \in HP_{t_1}(p_{curr}) : \mathbf{SR(l_1, n_2)}, \\ l_1 \mapsto_{t_2} n_2 \wedge l_1 \notin Ref_{t_2}(n_1) \wedge \\ l_1 \in Ref_{t_2}(n_2) \wedge n_2 \in HP_{t_2}(p_{curr})\end{gathered} \tag{16}$$

A. Safety

In this section we prove that BEWARE&CLEANUP is safe, i.e., it only reclaims nodes that are reclaimable.

Lemma *1:* If a node n is *reclaimable* at time t_1 then $Ref_t(n) = \emptyset$ for all $t \geq t_1$.

Proof: Assume towards a contradiction that a node n was reclaimable at time t_1 and that later at time t_2 $Ref(n)_{t_2} \neq \emptyset$. The definition of reclaimable, Definition 9, implies that at time t_1 there were no shared links pointing to n and no thread had a hazard pointer to n.

Then, clearly, n has to have been stored to some shared link l after time t_1 and before t_2. There are only two operations that can update a shared link: *StoreRef*(SR) and *CompareAndSwapRef* (CASR). However both of these operations require that the thread issuing the operation has a hazard pointer to n at that time. There are two cases:
(i) The thread has had a hazard pointer set to n already before time t_1. This is impossible since there were no hazard pointers to n at time t_1.
(ii) The thread set the hazard pointer to n at a time later than t_1. This is impossible as the only way to set a hazard pointer to an existing node is the *DeRefLink* operation. This operation requires that there is a shared link pointing to n to dereference and at time t_1 there are no such links. (See also Lemma 7, the linearizability proof for *DeRefLink*.)

Thus, since there cannot be any threads capable of using *StoreRef* or *CompareAndSwapRef* to update a link to point to n at time t_1 or later, we have that $Ref_t(n) = \emptyset$ for all $t \geq t_1$. ■

Lemma *2:* A node n returned by a *DeRefLink*(l_1) operation performed by a thread p is not *reclaimable* and cannot become *reclaimable* before a corresponding *ReleaseRef*(n) operation is performed by the same thread.

Proof: The node n is not reclaimable when *DeRefLink* returns because p has set a hazard pointer to point to n at line D4. Furthermore, line D5 verifies that n is referenced by l_1 also after the hazard pointer was set. This guarantees that n cannot have become reclaimable between line D3 and D4 since n is still referenced by a shared link.[3] ■

Lemma *3:* The mm_ref field together with the hazard pointers provide an approximation of the $Ref_t(n)$ set in a way that does not affect the safety of BEWARE&CLEANUP.

Proof: The reference count field, mm_ref, in each node approximates the set of links referencing n, $Ref_t(n)$, at any time t. As such the mm_ref field of a node n is only guaranteed to be accurate when there are no ongoing[4] operations concerning n. The only operations that may change the mm_ref field of a node n are *CompareAndSwapRef* and *StoreRef*.

For the memory reclamation method the critical aspect of the $Ref_t(n)$ set is to know whether it is empty or non-empty to determine if the node is reclaimable or not. In particular, the important case is when the mm_ref field is to be increased, since delaying a decrease of the reference count will not compromise the safety of the memory reclamation method.

However, although the mm_ref field of a node n to be stored in a shared link by a *CompareAndSwapRef* or *StoreRef* operation

[3]Note: Between D3 and D5 n might have been moved away from l_1 and then moved back again. Further, between D3 and D4 the "original" n could actually have been removed and reclaimed and then the same memory could be reused for a new node n which is stored in l_1 before D5. This is also not a problem as the "new" n is what the DeRefLink really returns.

[4]Consider any crashed operations as ongoing.

is not increased in the same atomic time instant as the operation takes effect, it does not matter for the safety of the memory reclamation method. That is so since the thread performing the operation is required to have set a hazard pointer to the node n before the operation was started and keep it until after the operation has finished. Thus n is clearly not reclaimable during the duration of the operation. ■

Lemma *4:* The operation *Scan* will never reclaim a node n that is not reclaimable.

Proof: *Scan* is said to reclaim a node n when it is returned to the pool of free memory, which takes place at line SC24.

Assume that *Scan* reclaimed a node n at time t_3 and let time t_1 and t_2 denote the time *Scan* executed line SC5 and line SC20 for the node n, respectively. It is possible to make two useful observations:
(i) First, note that there exists no thread p such that $n \in HP(p)$ during the whole interval between t_1 and t_2 since such a hazard pointer would have been detected by *Scan* (lines SC10 - SC14). Consequently, any thread that is able to access n after time t_3 must have dereferenced (with *DeRefLink*) a shared link pointing to n after time t_1.
(ii) Second, since *Scan* is reclaiming the node, we know that the mm_trace field of n, which were set to **true** at line SC5, and the mm_ref field, which was verified to be zero at line SC6, still had those values when line SC20 was reached. This implies that:
(i) There were no *StoreRef* or *CompareAndSwapRef* operations to store n in a shared link that started before t_1 and had not finished before t_2. This is so since the hazard pointers to n, which are required by these operations, would have been detected when *Scan* searched the hazard pointers at lines SC10 - SC14.
(ii) There were no *StoreRef* or *CompareAndSwapRef* operations to store n in a shared link that finished between t_1 and t_2, as such an operation would have cleared the mm_trace field and thereby would have caused the comparison at SC20 to fail.
Therefore, there were no ongoing *StoreRef* or *CompareAndSwapRef* operations to store n in a shared link at the time *Scan* executed line SC5 and, consequently, as these operations are the only ones that can increase the mm_ref field, we have $Ref_{t_1}(n) = \emptyset$. Further, because of (ii) there cannot have been any *StoreRef* or *CompareAndSwapRef* operation to store n in a shared link that started and finished between t_1 and t_2.

Since $Ref_{t_1}(n) = \emptyset$, no *DeRefLink* operation can finish by successfully dereferencing n after time t_1, unless n is stored to a shared link after time t_1. However, as we have seen above, such a store operation must begin after time t_1 and finish after time t_2 and the thread performing it must therefore, by our first observation that no single thread could have held a hazard pointer to n during the whole interval between t_1 and t_2, have dereferenced n after t_1. This is a clearly a contradiction and therefore it is impossible for any thread to successfully dereference n after time t_1.

From the above we have that $Ref_t(n) = \emptyset$ for $t \geq t_1$ and $\forall p\ .\ n \notin HP_t(p)$ for $t \geq t_2$ and therefore, since $t_3 > t_2 > t_1$, n is reclaimable at time t_3 . ■

Lemma *5:* The operation *Scan* never reclaims a node n that is accessed by a concurrent *CleanUpAll* operation.

Proof: Assume, w.l.o.g., that n is stored at position i in the DL_Nodes array of *Scan*. Before reclaiming the node, *Scan* writes NULL into DL_Nodes[i] (line SC21) and then checks that DL_Claims[i] is zero (line SC22).

Before accessing node n, the *CleanUpAll* operation reads n from DL_Nodes[i] (line CA3), then increases DL_Claims[i] (line CA5) and then verifies that DL_Nodes[i] still contains n (line CA6).

Now, for a concurrent *CleanUpAll* operation to also access n, it has to perform both reads of DL_Nodes[i] (line CA3 and line CA5) before *Scan* performs line SC21. But then DL_Claims[i] has been increased (line CA3) and *Scan* will detect this at line SC22 and will not reclaim the node. If, on the other hand, *Scan* reads a claim-count of 0 at line SC22, then the concurrent *CleanUpAll* operation will read NULL from DL_Nodes[i] at line CA6 and will not access the node. ■

Theorem *3:* BEWARE&CLEANUP is a safe memory reclamation method that guarantees the safety of both local and global references.

Proof: The theorem follows from Lemmas 2, 3, 4 and 5. ■

B. Bounding the number of deleted but unreclaimed nodes

Lemma *6:* For each thread p the maximum number of deleted but not reclaimed nodes in $DL(p)$ 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.[5]

Proof: The only operation that increases the size of $DL(p)$ is *DeleteNode* and when $|DL(p)|$ reaches THRESHOLD_C it runs *CleanUpAll* before attempting to reclaim nodes.

First consider the case where there are no concurrent *DeleteNode* operations by other threads. In this case there cannot be any deleted nodes that point to nodes in $DL(p)$ left in the system after p's *CleanUpAll* is done. So, what may prevent p from reclaiming one particular node in $DL(p)$? The node might have: (i) a hazard pointer pointing to it, or (ii) there might be some live nodes still pointing to it. The number of links in live nodes, α, that might point to a deleted node depends on the application data-structure using the memory reclamation method. Recall that we require that any application operation that deletes a node must also remove all references to that node from all live nodes in the data-structure before the delete is completed. This ensures that there are at all times at most $N \cdot \alpha$ links in live nodes that point to deleted nodes. So, in the absence of concurrent *DeleteNode* operations the maximum number of nodes in $DL(p)$ that a *Scan* is unable to reclaim is $N \cdot (k + \alpha)$.

In cases where there are concurrent *DeleteNode* operations, there are three cases to consider: (i) Additional nodes that hold pointers to the nodes in $DL(p)$ might be deleted after the start of p's *CleanUpAll* and prevent p's *Scan* from reclaiming any node. In that case p has to repeat the loop in *DeleteNode* again and call *CleanUpAll* and *Scan* again. This is fine in the lock-free setting, since some concurrent operation has made progress (i.e., by deleting other nodes). (ii) Some concurrent *DeleteNode* operation may get delayed or crash. If this happens when the operation executes between line DN2 and DN5 or between SC22 and SC23 in its call to *Scan*, it will "hide" one deleted node that might contain links that point to nodes in $DL(p)$ from p's *CleanUpAll*. In this way any other thread can prevent p from reclaiming up to l_{max} nodes in $DL(p)$. (iii) Finally, concurrent *CleanUpAll* operations might prevent p from reclaiming reclaimable nodes

[5]Note that the numbers l_{max} and α depend only on the application.

by claiming them for performing *CleanUpNode* operations on them. However, if such a node is encountered p's *Scan* will use *TerminateNode* to set the links of the node to NULL and set the DL_Done flag for the node, which prevents future *CleanUpAll* operations from preventing the node from being reclaimed. If p needs to repeat the *Scan*, it can only be prevented from reclaiming at most N of the reclaimable nodes it failed to reclaim due to concurrent *CleanUpAll*s during the previous *Scan*.

So, the maximum number of nodes in $DL(p)$ that p cannot reclaim is less than $N \cdot (k + l_{max} + \alpha + 1)$. ■

Lemma 6 directly implies theorem 2 and the following corollary.

Corollary *1:* The cleanup threshold, THRESHOLD_C, used by BEWARE&CLEANUP should be set to $N \cdot (k + l_{max} + \alpha + 1)$.

C. Linearizability

Lemma *7:* The *DeRefLink* ($DRL(l_1) = n_1$) operation is atomic.

Proof: A *DeRefLink*(DRL) operation has direct interactions with the *CompareAndSwapRef* (CASR) operations that target the same link and the memory reclamation in *Scan*. A CASR operation takes effect before the DRL operation iff the CAS instruction at line C1 is executed before `*link` is read at line D5 in DRL.

A *Scan* that reads the hazard pointer set by DRL at line D4, after the latter was set, will not free the node dereferenced by DRL (the test at line SC20 in *Scan* prevents this). If a concurrent *Scan* read the hazard pointer in question, after it was set by DRL, then it will not free the node. If *Scan* read the hazard pointer in question, before it was set by DRL, then *Scan* will detect that the reference count of the node is non-zero or has been non-zero during the execution of the *Scan* operation. ■

Lemma *8:* The *ReleaseRef* ($RR(n_1)$) operation is atomic.

Proof: The operation *ReleaseRef* takes effect at line R2 when the hazard pointer to the node is set to NULL. ■

Lemma *9:* The *NewNode* ($NN() = n_1$) operation is atomic if the memory allocator that is used to manage the pool of free memory itself is linearizable.

Proof: The operation *NewNode* takes effect when the memory for the new node is removed from the pool of free memory. For a linearizable memory allocator this will take place at a well-defined time instant. ■

Lemma *10:* The *DeleteNode* ($DN(n_1)$) operation is atomic.

Proof: The operation *DeleteNode* takes effect when the node is marked as deleted at line DN2. ■

Lemma *11:* The *CompareAndSwapRef* ($CASR(l_1, n_1, n_2)$) operation is atomic.

Proof: The $CASR$ operation has direct interactions with other $CASR$ and *DeRefLink* operations that target the same link and with the memory reclamation in *Scan*. A CASR operation takes effect when the CAS instruction at line C1 is executed. ■

Lemma *12:* The *StoreRef* ($SR(l_1, n_1)$) operation is atomic.

Proof: The *StoreRef* ($SR(l_1, n_1)$) operation has direct interactions with the memory reclamation in *Scan*.

A SR operation takes effect when the FAA instruction at line S6 is executed, or at line S2 otherwise. ■

Lemma *13:* The reclamation of a deleted node n by *Scan* is atomic.

Proof: *Scan* is said to reclaim a node n when that node is returned to the pool of free memory, which takes place at line SC24. This action is atomic, since the memory allocator's free operation is linearizable and Lemmas 4 and 5 guarantee that no other operation/thread can access the node concurrently. ■

Theorem *4:* BEWARE&CLEANUP correctly implements a linearizable memory reclamation method.

Proof: This follows from Lemmas 7, 8, 9, 10, 11 and 12. ■

D. Proof of the Lock-free Property

Lemma *14:* With respect to the retries caused by synchronization, among a set of concurrent operations of the BEWARE&CLEANUP method, there will always be one operation that will make progress, regardless of the actions of the other concurrent operations.

Proof: We now examine the possible execution paths of our implementation. The operations *ReleaseRef*, *NewNode*, *StoreRef* and *CompareAndSwapRef* do not contain any loops and will thus always do progress regardless of the actions by the other concurrent operations. In the remaining concurrent operations there are several potentially unbounded loops that can delay the termination of the operations. We call these loops retry-loops. If we omit the conditions that are because of the operations semantics (i.e., searching for the correct criteria etc.), then these loops retries when sub-operations detect that a shared variable has changed value from what was expected. This change is detected either by a subsequent read sub-operation or by a failed CAS. These shared variables are only changed concurrently by other CAS sub-operations. The read operation in line D5 will possibly fail because of a successful CAS operation in lines C1, TN7 or CN5. Likewise, the CAS operations in lines C1, TN7 or CN5 will possibly fail if one of the other CAS operations has been successful. According to the definition of CAS, for any number of concurrent CAS sub-operations on a word, exactly one will succeed. This means that for any subsequent retry, there must be one CAS that succeeded. As this succeeding CAS will cause its retry loop to exit, and our implementation does not contain any cyclic dependencies between retry-loops that exit with CAS, this means that the corresponding *DeRefLink* operation or *TerminateNode* sub-operation will progress.

In the operation *DeleteNode* there are calls to three sub-operations, *CleanUpLocal*, *Scan* and *CleanUpAll* which contains loops, inside an unbounded loop. The loop in the sub-operation *CleanUpNode*, used by *CleanUpLocal* and *CleanUpAll*, is bounded in the absence of concurrent *DeleteNode* operations because of Assumption 1. If there are concurrent *DeleteNode* operations, *CleanUpNode* may have to loop only due to their progress, i.e., if they set the mm_del bit on additional nodes, they might force the loop in *CleanUpNode* to continue. So *CleanUpNode* is lock-free. The loops in *CleanUpAll* are all bounded and the loop in *CleanUpLocal* and in *Scan* are bounded since the size of the DL_Nodes list is bounded by Theorem 2. The loop in *DeleteNode* is also bounded by the bound in Theorem 2.

Consequently, independent of any number of concurrent operations, one operation will always progress. ■

Theorem *5:* BEWARE&CLEANUP is a lock-free memory reclamation method.

Proof: The theorem follows from Lemma 14. ■

E. On the time complexity of the operations

In the following we count memory reads, writes and atomic synchronization primitives as single steps.

Lemma *15:* The operations *NewNode*, *ReleaseRef*, *StoreRef* and *CompareAndSwapRef* have worst-case time complexity $O(k)$, $O(k)$, $O(1)$ and $O(1)$, respectively, regardless of the number of concurrent operations. k is the number of hazard pointers per thread.

Proof: The lemma follows from Lemma 14 and the definitions of the operations in Program 3. ■

Since the other operations of the memory reclamation method are lock-free there are no upper bounds on the worst-case number of steps needed to complete an operation in the presence of concurrently progressing operations. However, under the assumption of no concurrent operations, we have the following worst-case bounds.

Lemma *16:* The operation *DeRefLink* has worst-case time complexity $O(k)$, where k is the number of hazard pointers per thread, in the absence of concurrent operations.

Proof: It is easy to see that the only way *DeRefLink* can remain in the unbounded loop starting at line D2 is if the contents of `*link` changes between line D3 and D5. This contradicts the assumption of the existence of no concurrent operations. Finding a free hazard pointer at line D1 takes $O(k)$ reads. ■

A worst-case execution of *DeleteNode* involves calls to *CleanUpLocal*, *Scan* and *CleanUpAll* (lines DN9-10) and it is easy to see that the worst-case time complexity of *CleanUpAll* overshadows the other two sub-operations. However, *CleanUpAll* uses the *CleanUpNode* operation, which is to be provided by the data structure or application using BEWARE&CLEANUP.

Let T_{CUN} denote the worst-case time complexity of *CleanUpNode*.

Lemma *17:* The operations *CleanUpAll* and *DeleteNode* have a worst-case time complexity of

$$O(N^2(k + l_{max} + \alpha + 1)T_{CUN}))$$

in the absence of concurrent operations, 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.

Proof: The operation *CleanUpAll* applies *CleanUpNode* to all nodes in the deletion lists of all threads. By the assumption above on the worst-case complexity of *CleanUpNode* and by the maximum number of deleted but unreclaimed nodes we get the worst case time complexity of the lemma.

A worst-case execution of *DeleteNode* involves calls to *CleanUpLocal*, *Scan* and *CleanUpAll* (lines DN9-10), of which *CleanUpAll* dominates the other. Hence, the worst-case time complexity is given by the same asymptotic formula. ■

Note that a reasonable upper bound of T_{CUN} is $O(l_{max}N^2 \cdot (k + l_{max} + \alpha + 1))$, since *CleanUpNode* may in the worst case have to traverse a maximal length chain of all deleted nodes for each link in the node being cleaned. Since the total size of the deletion lists is $N^2 \cdot (k + l_{max} + \alpha + 1)$ it is reasonable to consider that a maximal length chain of deleted nodes is no longer than this.

Discussion on the expected case: The worst case time complexity for a *DeleteNode* operation seems rather high. However, one must bear in mind that the worst case only occurs in a very particular situation which is extremely rare in practice. For the worst case to occur, first, each thread has to fill it's deletion list with nodes. If THRESHOLD_S < THRESHOLD_C, all nodes in the thread's deletion list must be unreclaimable each time *Scan* is called. This implies that $O(N(k + l_{max} + \alpha + 1))$ *DeleteNode* operations must be performed by each thread before this situation can occur. Furthermore, the nodes in the deletion lists must be chained together in such a way that the maximal chains of deleted but unreclaimed nodes contain all deleted nodes and are not broken up in small pieces early on during the *CleanUpAll* operation.

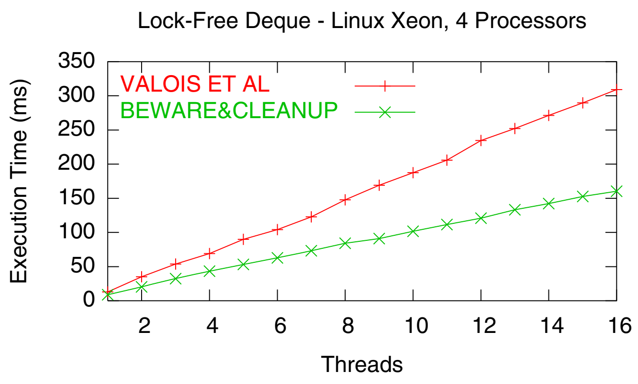
One interesting observation is that in BEWARE&CLEANUP the worst-case time complexity of the operations (in absence of concurrent operations) is low and predictable for all operations, except for *DeleteNode*. For a pure reference counting approach any operation that decreases a reference count, such as *ReleaseRef*, *CompareAndSwapRef* and *StoreRef*, can trigger the reclamation of an arbitrary number of nodes.

VI. EXPERIMENTAL STUDY

We have performed a set of experiments for observing the average overhead of using the BEWARE&CLEANUP lock-free memory reclamation in comparison to previous lock-free memory reclamation methods supporting reference counting. For this purpose we have chosen the lock-free algorithm of a deque (double-ended queue) data structure by Sundell and Tsigas [29], [26]. As presented, the implementation of this algorithm uses the lock-free memory reclamation with reference counting by Valois et al. [32], [21]. In order to fit better with the BEWARE&CLEANUP method, the recursive calls in the deque algorithm were unrolled.

In our experiments, each concurrent thread performed 10000 randomly chosen sequential operations on a shared deque, with an equal distribution among the *PushRight*, *PushLeft*, *PopRight* and *PopLeft* operations. Each experiment was repeated 50 times, and an average execution time among them was computed. Each thread executed the same sequence of operations for all different implementations compared; the sequences are random and were computed off-line, prior to the actual execution.

The experiments were performed using different number of threads, varying from 1 to 16 with increasing steps. In our experiments we compare two implementations of the lock-free deque; (i) using the lock-free memory reclamation by Valois et al., and (ii) using the new lock-free memory reclamation (including support for dynamic number of threads) with $k = 6$ hazard pointers per thread (this is set as in the HP method, i.e., depending on the number of local references used per thread). To the best of our knowledge, these are the only memory reclamation methods which: (i) satisfy the demands of the lock-free deque algorithm (as well as other common lock-free algorithms that need to traverse through nodes which may concurrently be deleted, such as the Queue algorithm used for the example in Program 6) and (ii) are based on available hardware atomic primitives. Both deque implementations use an individual implementation of a shared fixed-size memory pool (i.e., free-list) for memory allocation and freeing. Two different platforms were used, with varying number of processors and level of shared memory distribution. Firstly, we performed our experiments on a 4-processor Xeon PC running Linux. In order to evaluate the BEWARE&CLEANUP method with higher concurrency we also used a 8-processor SGI Origin 2000 system running Irix 6.5. A clean-cache operation was performed



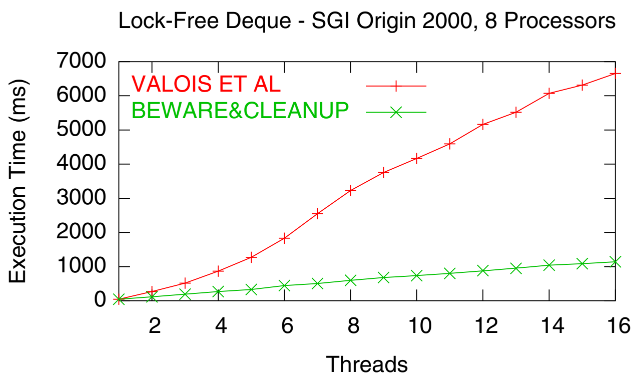


Fig. 3. Experimental results on two multi-processor platforms of performing a random set of operations on a lock-free doubly-ended queue (deque); implemented with either the memory reclamation by Valois et al. or with the new method.

just before each sub-experiment. All implementations are written in C and compiled with the highest optimization level. The atomic primitives are written in assembly, where systems (e.g., SGI) lacking native CAS support are using the corresponding LL/SC instructions instead, see [22]. The results from the experiments are shown in Fig. 3. The average execution time is drawn as a function of the number of threads.

From the experimental results, we see that the BEWARE&CLEANUP lock-free memory reclamation method outperforms the corresponding method by Valois et al. for any number of threads. The advantage of using the BEWARE&CLEANUP method appears to be even more significant for systems with non-uniform memory architecture. A possible reason for this increased advantage could be that the increased memory contention caused by several threads concurrently traversing potentially long (compared to the total data structure size) chains of deleted nodes has a larger impact on the system performance in non-uniform memory architectures than in uniform memory architectures.

VII. CONCLUSIONS

We have presented BEWARE&CLEANUP, which is, to the best of our knowledge, the first lock-free method for memory reclamation based on reference counting that has all the following features: (i) guarantees the safety of local as 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. BEWARE&CLEANUP is application guided and is aimed for use with arbitrary lock-free dynamic data structures.

Besides the analytical study, we have also presented an experimental study which gives evidence that the BEWARE&CLEANUP lock-free memory reclamation scheme can, in the respects of performance and reliability, significantly improve implementations of lock-free dynamic data structures that require the safety of global references in dynamic nodes. We believe that our implementation is of highly practical interest for multi-processor applications. We are incorporating it into the NOBLE [28] library.

REFERENCES

[1] J. M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518, July 1977.

[2] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, 1960.

[3] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele. Lock-free reference counting. In *Proceedings of the 20th annual ACM symposium on Principles of distributed computing*, pages 190–199. ACM Press, Aug. 2001.

[4] L. D. Deutsch and D. G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, Sept. 1976.

[5] A. Gidenstam, M. Papatriantafilou, and P. Tsigas. Allocating memory in a lock-free manner. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA '05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 329–242. Springer Verlag, Oct. 2005.

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

[7] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[8] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Dynamic-sized lock-free data structures. In *Proceedings of the 21st annual symposium on Principles of distributed computing*, pages 131–131. ACM Press, 2002.

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

[10] 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 (DISC 2002)*, volume 2508 of *Lecture Notes in Computer Science*, pages 339–353. Springer Verlag, Oct. 2002.

[11] M. P. Herlihy and J. E. B. Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):304–311, May 1992.

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

[13] 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.

[14] M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. In *Proceedings of the 11th International Conference On the Principles Of Distributed Systems (OPODIS '07)*, volume 4878 of *Lecture Notes in Computer Science*, pages 401–414. Springer-Verlag, 2007.

[15] P. Jayanti. A complete and constant time wait-free implementation of CAS from LL/SC and vice versa. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC '98)*, volume 1499 of *Lecture Notes in Computer Science*, pages 216–230. Springer Verlag, Sept. 1998.

[16] P. Joisha. Overlooking roots: A framework for making nondeferred reference-counting garbage collection fast. In M. Sagiv, editor, *ISMM'07 Proceedings of the Fifth International Symposium on Memory Management*, pages 141–158, Montréal, Canada, Oct. 2007. ACM Press.

[17] L. Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11):806–811, 1977.

[18] M. M. Michael. Safe memory reclamation for dynamic lock-free objects

using atomic reads and writes. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing (PODC '02)*, pages 21–30. ACM Press, 2002.

[19] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(8):491–504, Aug. 2004.

[20] M. M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of SIGPLAN 2004 Conference on Programming Languages Design and Implementation (PLDI '04)*, pages 35–46. ACM Press, June 2004.

[21] M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, Computer Science Department, Dec. 1995.

[22] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th annual ACM Symposium on Principles of Distributed Computing (PODC '97)*, pages 219–228, 1997.

[23] M. Moir, V. Luchangco, and M. Herlihy. Lock-free implementation of dynamic-sized shared data structure. US Patent WO 2003/060705 A3, Jan. 2003.

[24] Y. G. Park and B. Goldberg. Reference escape analysis: Optimizing reference counting based on the lifetime of references. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26, 9 of *ACM SIGPLAN NOTICES*, pages 178–189, New York, June 1991. ACM Press.

[25] S. Schneider, C. Antonopoulos, and D. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 2006 International Symposium on Memory Management (ISMM '06)*, pages 84–94. ACM, June 2006.

[26] H. Sundell. *Efficient and Practical Non-Blocking Data Structures*. PhD thesis, Chalmers University of Technology, Nov. 2004.

[27] H. Sundell. Wait-free reference counting and memory management. In *Proceedings of the 19th International Parallel & Distributed Processing Symposium (IPDPS '05)*, page 24b. IEEE, Apr. 2005.

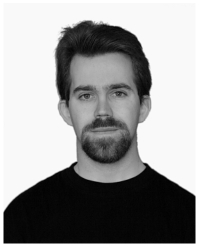
[28] H. Sundell and P. Tsigas. NOBLE: A non-blocking inter-process communication library. In *Proceedings of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*, 2002.

[29] H. Sundell and P. Tsigas. Lock-free and practical deques using single-word compare-and-swap. In *Proceedings of the 8th International Conference on Principles of Distributed Systems (OPODIS '04)*, volume 3544 of *Lecture Notes in Computer Science*, pages 240–255. Springer Verlag, Dec. 2004.

[30] H. Sundell and P. Tsigas. Practical and lock-free doubly linked lists. In *Proceedings of the 2007 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '07)*, pages 264–270. CSREA Press, June 2007.

[31] J. D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 64–69, Oct. 1994.

[32] J. D. Valois. *Lock-Free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, 1995.



Anders Gidenstam's research interests include non-blocking operating system services, concurrent data structures and algorithms for multiprocessor and multicore computers. He received a M.Sc. degree in computer science and engineering in 2001 at Chalmers University of Technology and in 2007 he received a Ph.D. degree in computer science at the same university. He is currently a PostDoc researcher at Max Planck Institute for Computer Science, Saarbrücken, Germany.

Marina Papatriantafilou is an associate professor at the Department of Computer Science and Engineering, Chalmers University of Technology, Sweden. She received the Diploma and PhD degrees from the Department of Computer Engineering and Informatics, University of Patras, Greece. She has also worked at the National Research Institute for Mathematics and Computer Science in the Netherlands (CWI), Amsterdam and at the Max-Planck Institute for Computer Science (MPII) Saarbruecken, Germany. She is interested in research on distributed and multiprocessor computing, including synchronization, communication/coordination, and networking issues (http://www.cse.chalmers.se/˜ptrianta/).

Håkan Sundell's main research interests are in efficient shared data structures for practical multi-thread programming on multiprocessor and multi-core computers. He is known as a pioneer in the Swedish IT-history with roots in the early 80's emerging programming community. He received a M.Sc. degree in computer science in 1996 at Göteborg University. Between the years 1995-1999 he worked as a senior consultant and systems programmer within the telecommunication and multimedia industry. In 2004 he received a Ph.D. degree in computer science at Chalmers University of Technology. At present he is a senior lecturer at the School of Business and Informatics, University College of Borås, Sweden.

Philippas Tsigas' research interests include data sharing for multiprocessor/multicore systems, interprocess communication and coordination in parallel systems, fault-tolerant computing, mobile computing and information visualization. He received a BSc in Mathematics from the University of Patras, Greece and a PhD in Computer Engineering and Informatics from the same University. Philippas was at the National Research Institute for Mathematics and Computer Science, Amsterdam, the Netherlands (CWI), and at the Max-Planck Institute for Computer Science, Saarbrücken, Germany, before. At present he is an associate professor at the Department of Computing Science at Chalmers University of Technology, Sweden (www.cse.chalmers.se/˜tsigas).