Technical Report no. 2005-04

Practical and Efficient Lock-Free Garbage Collection Based on Reference Counting

Anders Gidenstam **Marina Papatriantafilou** **Håkan Sundell**
Philippas Tsigas

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, 2005

Technical Report in Computer Science and Engineering at
Chalmers University of Technology and Göteborg University

Technical Report no. 2005-04
ISSN: 1652-926X

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, March 2005

Abstract

We present an efficient and practical lock-free implementation of a garbage collection scheme based on reference counting aimed for the 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 data structures that require strong garbage collection.

Keywords: reference counting, garbage collection, lock-free, shared memory.

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. It is important in non-blocking algorithms that the effects of the concurrent operations can be observed by the involved processes in a consistent manner. The common consistency requirement 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. Valois as well as Michael and Scott [22, 13] presented a 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 [12] and Gidenstam et al. [2].

Various lock-free garbage collection schemes have been presented in the literature. Michael [10, 11] proposed the hazard pointer algorithm that focuses on local references. A similar scheme has been proposed by Herlihy et al. [5]; this 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, which is available in some 32-bit architectures, but only in very few of the current 64-bit architectures. 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 be able to trust global references (i.e. pointers from within the data structure) to objects. This constraint can be strong and restrictive, and causes 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.

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. [22, 13] 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, which is a double-word compare-and-swap operation that can atomically update two arbitrary memory words, which is not available on any modern architecture. Herlihy et al. [4, 15] presented 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[1]. It has been identified in [13] 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 to be reclaimed.

In the context of wait-free memory management, a wait-free extension of Valois' scheme has been presented by Sundell [18, 17]. Hesselink and Groote [7, 8] has 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, into a general lock-free reference counting scheme, with the aim of keeping only the advantages of the involved techniques while avoiding the respective drawbacks. Our new lock-free garbage collection 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 refer-

[1] A compare-and-swap operation that can atomically update two adjacent memory words is available in some 32-bit architectures, but only in very few of the current 64-bit architectures.

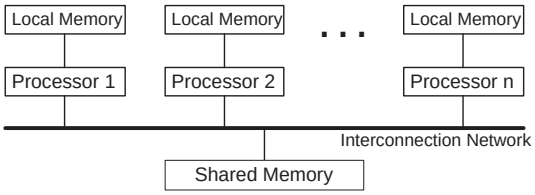

Figure 1. Shared Memory Multiprocessor System Structure

```
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;
```

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

ences. We also show how to bound the amount of 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. In Section 5 we define the precise semantics of the operations on our implementation, and show the correctness of our algorithm by proving the lock-free and linearizability properties as well as proving an upper bound of the memory that can be temporarily held for reclaiming by our scheme. Section 6 presents an experimental evaluation of the new scheme in context of a lock-free data structure. We conclude the paper with Section 7.

2 System Description

A typical abstraction of a shared memory multiprocessor system configuration is depicted in Figure 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 at a time. 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 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; see Figure 2 for a description. These read-modify-write style of operations are available on most common architectures or can be easily derived from other synchronization primitives [14] [9].

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 the possible 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 possible garbage that is not part of the data structure and for*

	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, 15]	Yes	No [e]	Yes	No [c]
Michael [10, 11]	No	Yes	Yes	Yes [d]
Valois et al. [22, 13]	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.

which future access by any thread is not possible.

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 exist an upper bound on the number of nodes that is not part of the data structure, but not yet reclaimed to the system.*

In real 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 accessible 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 3. 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 are 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 methods which can also employ the *hazard pointer* (HP) scheme of Michael [10, 11]. Roughly speaking, hazard pointers are used for guaranteeing the safety of local references and reference counts for guaranteeing 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 3 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 of 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 of 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 Appendix 5.2. The threshold THRESHOLD_2 is set according to the HP scheme or less or equal than THRESHOLD_1.

4.1 Application Programming Interface

The following functions are defined for safe handling of the reference counted nodes:

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

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

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

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

function NewNode:**pointer to** Node

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

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 to 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.

In Section 4.4 we give an example of how these functions can be used in the context of a lock-free queue algorithm based on linked lists.

```
/* 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;
```

Figure 4. Reference counting functions, part I

```
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 5. Reference counting functions, part II

```
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 6. Callback functions

Callbacks

The following functions are callbacks that have to be defined by the designer of each specific data structure:

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

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

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. [2]

The procedure *CleanUpNode* will make sure that all claimed references from the links of the given node will

[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.

```
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 7. Internal functions.

only point to active nodes, thus removing redundant passages through an arbitrary number of deleted nodes.

4.2 Auxiliary Procedures

Auxiliary functions that are defined for internal use by the reference counting scheme:

procedure Scan()
procedure CleanUpLocal()
procedure CleanUpAll()

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. 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 has been deleted by any thread.

4.3 Detailed Algorithm Description

DeRefLink (Fig. 4), 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 still not yet reclaimed and that it cannot not be reclaimed until the hazard pointer now pointing to it is released. If `*link` has changed since the last read, it retries.

ReleaseRef (Fig. 4), removes this thread's hazard pointer pointing to `node`. Note that if the node `node` is deleted, has a reference count of zero and no other hazard pointers are pointing to it, the node can now be reclaimed by *Scan* (invoked by the thread that called *DeleteNode* on the node).

CompareAndSwapRef (Fig. 4), 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 `node` has been increased. Notice that the node `node` 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 (Fig. 4), 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, which is safe since the thread calling *StoreRef* is required to have a hazard pointer to the node `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 (Fig. 4), allocates memory for the new node from the underlying memory allocator and initializes the header fields each node should have. It also sets a hazard pointer to the node.

DeleteNode (Fig. 5), 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 set of deleted but not

yet reclaimed nodes. 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 set of deleted but not yet reclaimed nodes is larger than or equal to THRESHOLD_2 a *Scan* is performed which will reclaim all nodes in the set that are not referenced by other nodes or threads.

If the thread's set of deleted but not yet reclaimed nodes is now full, that is, it contains THRESHOLD_1 nodes, the thread will first run *CleanUpLocal* at line DN9 to make sure that all of its deleted nodes only points 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 sets of deleted nodes of all threads.

Callbacks

TerminateNode (Fig. 6), should clear all links in the node node by writing NULL to the links in node. This is done by using either *CompareAndSwapRef* or *StoreRef* depending on whether there might be concurrent updates of the these links or not.

CleanUpNode (Fig. 6), should make sure that none of the links of the node node points to nodes that were deleted before this invocation of *CleanUpNode* started.

Auxiliary procedures

Scan (Fig. 7), 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 that can safely be reclaimed *Scan* first sets the mm_trace bit of all deleted nodes that have reference count zero (lines SC1 to SC7). 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 SC8 to SC14). In the lines SC15 to SC30 *Scan* traverses all not yet reclaimed nodes deleted by this thread. For each of these nodes the tests at line SC19 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 SC21. If there are no such *CleanUpAll* operations *Scan* uses *TerminateNode* to release all references the node might contain and then reclaim the node (lines SC22 and SC23). In case there might be concurrent *CleanUpAll* operations accessing the node *Scan* uses the concurrent version of *TerminateNode* to set all of the node's links to NULL. By setting the DL_Done flag at line SC26 before the node is reinserted into the set of unreclaimed nodes at line SC27 later *CleanUpAll* operations cannot prevent this node from being reclaimed by a subsequent *Scan*.

CleanUpLocal (Fig. 7), traverses the thread's list of deleted but unreclaimed nodes and calls *CleanUpNode* on each of them to make sure that their links do not reference any nodes that were already deleted when *CleanUpLocal* started.

CleanUpAll (Fig. 7), traverses the DL_Nodes arrays of all threads and try to make sure that none of the nodes it finds contains links to nodes that were already deleted when *CleanUpAll* started. The tests at line CA4 prevent *CleanUpAll* from needlessly interfere 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 SC20 or be after line SC27 without having reclaimed the node.

4.4 Example Application

The application of the new algorithm for memory management to lock-free algorithms for dynamic data structures can be done straight forward in a similar manner to previously presented memory management schemes. Figure 8 shows the lock-free queue algorithm by Valois et al. [21, 13] as it would be integrated with the new algorithm for memory management.

4.5 Algorithm Extensions

For simplicity reasons, the algorithm in this paper is described with a fixed number of threads. However, the scheme can easily be extended for a dynamic number of threads in a similar way as described for the HP scheme in [11]. 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 management.

4.6 Algorithm Correctness and Bounds on Unreclaimed Memory

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

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

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

```
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();
IQ2   node.next := NULL;
IQ3   head := NULL; tail:= NULL;
IQ3   StoreRef(&head,node);
IQ4   StoreRef(&tail,node);

function Dequeue():pointer to Value
DQ1  while true do
DQ2       node1 := DeRefLink(&head);
DQ3       next := DeRefLink(&node2.next);
DQ4       if next = NULL return NULL;
DQ5       if CompareAndSwapRef(&head,node1,next) then break;
DQ6       ReleaseRef(node1); ReleaseRef(next);
DQ7  DeleteNode(node1);
DQ8  value := next.value; ReleaseRef(next);
DQ9  return value;

procedure Enqueue(value:pointer to Value)
EQ1   node := NewNode();
EQ2   node.next := NULL; node.value := value;
EQ3   old := DeRefLink(&tail); 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);
```

Figure 8. Example of a Queue algorithm using the new memory management scheme.

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 corresponding proofs of the above theorems are left to Section 5, where they are presented using a series of lemmas.

5 Correctness Proof

In this section we present the correctness proof of our algorithm. We first prove that our algorithm 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 and last we prove that the algorithm is a linearizable and lock-free one [6]. A set of definitions that will help us to structure and shorten the proof is first described in this section. We start by defining the sequential semantics of our operations.

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 SC23 in* Scan*. Any (preferably lock-free) memory allocator can be used to manage the free pool.*

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 successful DeRefLink *operation. The array of hazard pointers in the implementation, the array* HP*, may also contain pointers temporarily set by unsuccessful* DeRefLink *operations, but these are not considered as part of the $HP_t(p)$ sets.*

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.*

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

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

Let $l_x \mapsto_t n_x$ denote that the shared link l_x points to node n_x at time t.

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

The operations that are of interest for linearizability are DeRefLink*(DRL),* ReleaseRef*(RR),* NewNode*(NN),* DeleteNode*(DN) and* CompareAndSwapRef*(CASR).*

For the safety and correctness of the memory management the following additional internal operations are also of interest: TerminateNode*(TN),* Scan*(SCAN),* CleanUpNode*(CUN),* CleanUpLocal*(CUL),* CleanUpAll*(CUA).*

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.

DeRefLink

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

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

ReleaseRef

$$n \in HP_{t_1}(p_{curr}) : \mathbf{RR}(\mathbf{n_1}), n \notin HP_{t_2}(p_{curr}) \tag{3}$$

NewNode

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

DeleteNode

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

CompareAndSwapRef

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

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

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

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

Scan

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

TerminateNode (Implemented by the application programmer).

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

CleanUpNode (Implemented by the application programmer).

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

CleanUpLocal

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

CleanUpAll

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

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

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

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

Definition 2 *A node n is said to be* reclaimable *at time t iff $Ref_t(n) = \emptyset$ and $\forall p.n \notin HP_t(p)$ and $Del_t(n)$.*

5.1 Safety

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

Proof: (sketch) Assume towards a contradiction that a node n was reclaimable at time t_1 and that later at time t_2 $Ref(n)_{t_2} \neq \emptyset$. From the definition of reclaimable follows that at time t_1 there were no shared links pointing to n and no process 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 process issuing the operation has a hazard pointer to n at that time. There are two cases:
i) The process has had a hazard pointer set to n since already before time t_1. This is impossible since there were no

hazard pointers to n at time t_1.
ii) The process 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 and it requires that there is a shared link pointing to n to dereference and at time t_1 there are no such links. (See also the linearizability proof for *DeRefLink*)

So since there cannot be any processes 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 process p is not* reclaimable *and cannot become* reclaimable *before a corresponding ReleaseRef(n) operation is performed by the same process.*

Proof: (sketch) 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 which guarantees that n cannot have become reclaimable between line D3 and D4 since n is still referenced by a shared link.[3][4] □

Lemma 3 *The mm_ref field together with the hazard pointers provide a safe approximation of the $Ref_t(n)$ set.*

Proof: (sketch) The reference count field, mm_ref, in each node is an approximation of the set of links referencing n $Ref(n)$. As such the mm_ref field of a node n is only guaranteed to be accurate when there are no ongoing[5] operations concerning n. The only operations that may change the mm_ref field of a node n is *CompareAndSwapRef* and *StoreRef*.

For the memory management scheme the critical aspect of the $Ref(n)$ 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 management scheme.

Thus, although the mm_ref field of a node n to be stored in a shared link by a *CompareAndSwapRef* or *StoreRef* operation is not increased in the same atomic time instant as the operation takes effect, that *does not matter* for the safety of the memory management scheme since the node is clearly not reclaimable during the duration of the operation anyway, since the process performing the operation is required to have a hazard pointer set to n. □

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

Proof: (sketch) *Scan* is said to reclaim a node n when it is returned to the pool of free memory, which takes place at line SC23.

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 SC19 for the node n, respectively.

First, note that there exists no process p such that $n \in HP(p)$ during the whole interval between t_1 and t_2 since such a hazard pointer would be detected by *Scan* (lines SC9 - SC13). Consequently, any process that is able to access n after time t_3 must have dereferenced (with *DeRefLink*) a shared link pointing n after time t_1.

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 SC19 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, since the hazard pointers to n these operations require would have been detected when *Scan* searched the hazard pointers at lines SC9 - SC13.
ii) There were no *StoreRef* or *CompareAndSwapRef* operations to store n in a shared link that finished between t_1 and t_2, as a such operation would have cleared the mm_trace field and thereby caused the comparison at SC19 to fail.
Therefore, there were no ongoing *StoreRef* or *CompareAndSwapRef* operation 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 dereference 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 begins after time t_1 and finish after time t_2 and the process performing it must therefor, by our first observation that no single process 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 contradicition and therefor it is impossible for any process to successfully dereference n after time t_1.

[3] Note 1: Between D3 and D5 n might have been moved away from l_1 and then moved back again.

[4] Note 2: 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 no problem as the "new" n is what the DeRefLink really returns.

[5] Consider any crashed operations as ongoing.

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 therefor, since $t_3 > t_2 > t_1$, n is reclaimable at time t_3 . □

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

Proof: (sketch) Before reclaiming the node n stored at position i in its DL_Nodes array *Scan* writes NULL into DL_Nodes[i] (line SC20) and then checks that DL_Claims[i] is zero (line SC21).

Before accessing the 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 do both reads of DL_Nodes[i] (line CA3 and line CA5) before *Scan* performs line SC20, but then DL_Claims[i] has been increased (line CA3) and *Scan* will detect this at line SC21 and will not reclaime the node.

If, on the other hand, *Scan* reads a claim count of 0 at line SC21, then the concurrent *CleanUpAll* operation will read NULL from DL_Nodes[i] at line CA6 and will not access the node. □

5.2 Proof of the bound on the number of deleted but unreclaimed nodes

Theorem 3 *For each thread p_i the maximum number of deleted but not reclaimed nodes in $DL(p_i)$ 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.)*

Proof: (sketch) The only operation that increases the size of $DL(p_i)$ is *DeleteNode* and when $|DL(p_i)|$ reaches THRESHOLD_1 it runs *CleanUpAll* before attempting to reclaim nodes.

First consider the case where there are no concurrent *DeleteNode* operations by other threads. Then, after *CleanUpAll*, there cannot be any deleted nodes that point to nodes in $DL(p_i)$ left. So, what may prevent p_i from reclaiming one particular node in $DL(p_i)$? 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 manager. We require that each application operation that deletes a node must also remove all references to that node from the live nodes of the data-structure before it is completed, which ensures that there at all times are 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 $DL(p_i)$ that a *Scan* is unable to reclaim is $N \cdot (k + \alpha)$.

In the case where there are concurrent *DeleteNode* operations three more things of interest may occur: i) Additional nodes that might hold pointers to the nodes in $DL(p_i)$ might be deleted after the start of *CleanUpAll* and prevent *Scan* from reclaiming any node. However, in that case p_i is free to retry *CleanUpAll* and *Scan* again since some concurrent operation has made progress. ii) Some concurrent *DeleteNode* operation may get delayed or crash, either between line DN2 and DN5 or between SC21 and SC22 in its call to *Scan* which will "hide" one deleted node that might contain links that point to nodes in $DL(p_i)$ from p_i's *CleanUpAll*. In this way each other thread can prevent p_i from reclaiming up to l_{max} nodes in $DL(p_i)$. iii) Finally, concurrent *CleanUpAll* operations might prevent p_i from reclaiming reclaimable nodes by claiming them for performing *CleanUpNode* operations on them. However, if such a node is encountered p_i'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_i needs to retry 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 $DL(p_i)$ that p_i cannot reclaim is less than $N(k + l_{max} + \alpha + 1)$. □

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

Corollary 2 *The number of deleted but not yet reclaimed nodes in the system is bounded from above by*

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

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.

5.3 Linearizability

Lemma 6 *The DeRefLink ($DRL(l_1) = n_1$) operation is atomic.*

Proof: (sketch) A *DeRefLink*(DRL) operation has direct interactions with *CompareAndSwapRef*(CASR) that targets 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 it was set will not free the node dereferenced by DRL (the test at line SC19 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 7 *The ReleaseRef ($RR(n_1)$) operation is atomic.*

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

Lemma 8 *The NewNode ($NN() = n_1$) operation is atomic if the memory allocator used to manage the pool of free memory itself is linearizable.*

Proof: (sketch) 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 9 *The DeleteNode ($DN(n_1)$) operation is atomic.*

Proof: (sketch) The operation *DeleteNode* takes effect when the node is marked as deleted at at line DN2. □

Lemma 10 *The CompareAndSwapRef ($CASR(l_1, n_1, n_2)$) operation is atomic.*

Proof: (sketch) The *CompareAndSwapRef* ($CASR(n_1)$) operation has direct interactions with other *CompareAndSwapRef*(CASR) and *DeRefLink* operations that targets the same link and with the memory reclaimation in *Scan*.

A CASR operation takes effect when the CAS instruction at line C1 is executed. □

Lemma 11 *The reclamation of a deleted node n by Scan is atomic.*

Proof: (sketch) *Scan* is said to reclaim a node n when it is returned to the pool of free memory, which takes place at line SC23. The node is safe to reclaim because the tests at line SC19 guarantees that i) *Scan* found no hazard pointers pointing to the node and, ii) the reference count of the node has been consistently 0 since before the hazard pointers were scanned. That ii) holds is ensured by the n.mm_trace bit, which detects if a node, n, had $Ref(n) = 0$ when *Scan* executed line SC4, but a pointer to it was later stored in a shared link variable so that there were no hazard pointer to it when *Scan* read them at line SC9 to SC13. Then, this invocation of *Scan* will not attempt to reclaim the node even if the shared link to the node is removed again, since the n.mm_trace was cleared when the pointer to the node was stored in a shared variable (either by a *CompareAndSwapRef* or a *StoreRef*).

The test at line SC21 guarantees that there are no other threads that might access the node as part of the *CleanUpAll* operation. A *CleanUpAll* operation verifies that the node is still there at line CA6 after increasing the claim counter at line CA5 before attempting to access the node. If *Scan* reads a claim count of 0 at line SC21 then there are no concurrent *CleanUpAll* operations that might access the node, because they will read NULL at line CA6. □

Theorem 4 *The algorithm implements a linearizable garbage collector.*

Proof: This follows from Lemmas 6, 7, 8, 9 and 10. □

5.4 Proof of the Lock-free Property

Lemma 12 *With respect to the retries caused by synchronization, one operation will always do progress regardless of the actions by the other concurrent operations.*

Proof: We now examine the possible execution paths of our implementation. The operations *ReleaseRef*, *NewNode* 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.), the loop retries when sub-operations detect that a shared variable has changed value. This is detected either by a subsequent read sub-operation or 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, 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* suboperation will progress.

In the operation *DeleteNode* there are calls to three suboperations, *CleanUpLocal*, *Scan* and *CleanUpAll* which contains loops, inside an unbounded loop. The loop in the suboperation *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* only their progress, i.e. that they set the mm_del bit on additional nodes, 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 3. The loop in *DeleteNode* is also bounded by the bound in Theorem 3.

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

Theorem 5 *The algorithm implements a lock-free garbage collector.*

Proof: Lemma 12 give that our implementation is lock-free. □

6 Experimental Evaluation

We have performed experiments in our effort to estimate the average overhead of using the new lock-free memory management algorithm in comparison to previous lock-free memory management schemes 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 [20, 16]. As presented, the implementation of this algorithm uses the lock-free memory management with reference counting by Valois et al. [22, 13]. In order to fit better with the new memory management scheme, the recursion 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 for each experiment was estimated. Exactly the same sequence of operations were performed for all different implementations compared.

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 management by Valois et al., and ii) using the new lock-free memory management (including support for dynamic number of threads) with 6 hazard pointers per thread. These are the only memory management schemes 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 fig. 8) and (ii) work with available atomic primitives. Both implementations use a shared fixed-size memory pool (i.e. freelist) 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 our algorithm with higher concurrency we also used a 8-processor SGI Origin 2000 system running Irix 6.5. A clean-cache operation was performed 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. The results from the experiments are shown in Figure 9. The average execution time is drawn as a function of the number of threads.

Our results show that the new lock-free memory management algorithm outperforms the corresponding algorithm by Valois et al. for any number of threads. The advantage of using the new algorithm shows to be even more significant for systems with non-uniform memory architecture.

7 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 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.

Experimental results indicate that our new lock-free garbage collection 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 of highly practical interest for multi-processor applications. We are currently incorporating it into the NOBLE [19] library.

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.

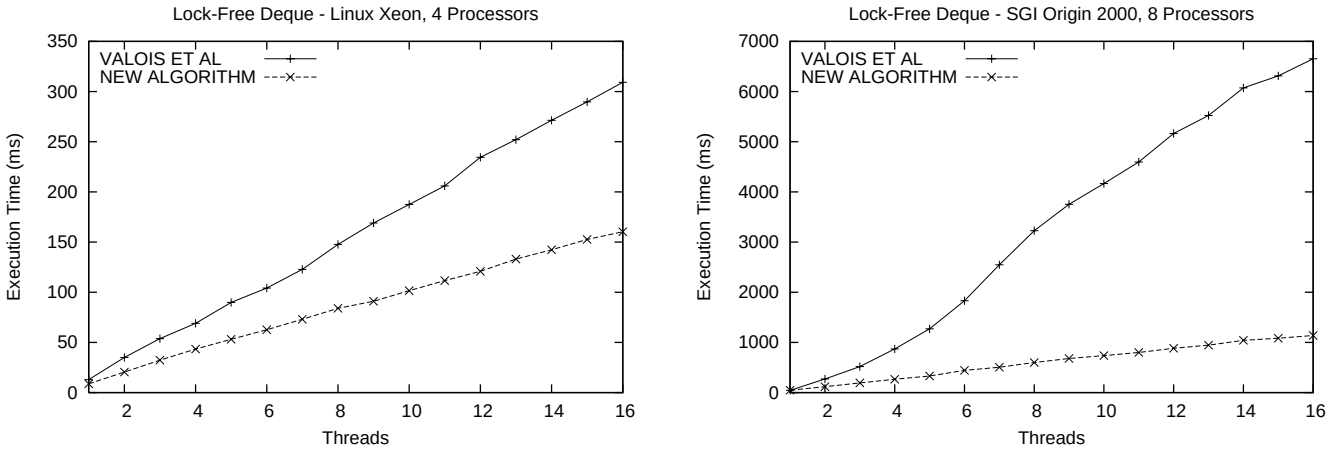

Figure 9. Experiment with lock-free deques and various memory management schemes.

[2] A. Gidenstam, M. Papatriantafilou, and P. Tsigas, "Allocating memory in a lock-free manner," Computing Science, Chalmers University of Technology, Tech. Rep. 2004-04, 2004.

[3] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 1, pp. 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*. ACM Press, 2002, pp. 131–131.

[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*, vol. 12, no. 3, pp. 463–492, 1990.

[7] W. H. Hesselink and J. F. Groote, "Waitfree distributed memory management by create and read until deletion (CRUD)," CWI, Amsterdam, Tech. Rep. SEN-R9811, 1998.

[8] ——, "Wait-free concurrent memory management by create and read until deletion (CaRuD)," *Distributed Computing*, vol. 14, no. 1, pp. 31–39, Jan. 2001.

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

[10] 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*, 2002, pp. 21–30.

[11] ——, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 8, Aug. 2004.

[12] ——, "Scalable lock-free dynamic memory allocation," in *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2004, pp. 35–46.

[13] M. M. Michael and M. L. Scott, "Correction of a memory management method for lock-free data structures," Computer Science Department, University of Rochester, Tech. Rep., 1995.

[14] 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.

[15] M. Moir, V. Luchangco, and M. Herlihy, "Lock-free implementation of dynamic-sized shared data structure," International Patent WO 2003/060 705 A3, Jan., 2003.

[16] H. Sundell, "Efficient and practical non-blocking data structures," Ph.D. dissertation, Chalmers University of Technology, Nov. 2004.

[17] ——, "Wait-free reference counting and memory management," Computing Science, Chalmers University of Technology, Tech. Rep. 2004-10, Nov. 2004.

[18] ——, "Wait-free reference counting and memory management," in *Proceedings of the 19th International Parallel & Distributed Processing Symposium*. IEEE press, Apr. 2005.

[19] H. Sundell and P. Tsigas, "NOBLE: A non-blocking interprocess communication library," in *Proceedings of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*, ser. Lecture Notes in Computer Science. Springer Verlag, 2002.

[20] ——, "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*. Lecture Notes in Computer Science, Springer Verlag, Dec. 2004.

[21] J. D. Valois, “Implementing lock-free queues,” in *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, 1994, pp. 64–69.

[22] ——, “Lock-free data structures,” Ph.D. dissertation, Rensselaer Polytechnic Institute, Troy, New York, 1995.