Technical Report no. 2004-10

Wait-Free Reference Counting and Memory Management

Håkan Sundell

CHALMERS | GÖTEBORG UNIVERSITY

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

Göteborg, 2004

Technical Report in Computing Science at
Chalmers University of Technology and Göteborg University

Technical Report no. 2004-10
ISSN: 1650-3023

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

Göteborg, Sweden, 2004

Abstract

We present a practical wait-free implementation of a garbage collection scheme based on reference counting that uses atomic primitives which are available in modern computer systems. To the best of our knowledge, this is the first wait-free algorithm of a reference counting scheme that can support dynamic concurrent data structures. As all operations of wait-free algorithms are guaranteed to always finish in a finite number of their own steps independently of the other operations' actions, the new algorithm is especially suitable for real-time systems where execution time guarantees are of significant importance. We also present a wait-free algorithm of a free-list, for supporting concurrent allocation and freeing of memory blocks. The new algorithms are linearizable and are compatible to previous implementations of non-blocking dynamic data structures.

1 Introduction

Reliable methods for concurrent memory management are fundamental for the design of concurrent dynamic data structures. Thus, algorithms for concurrent garbage collection have been extensively researched. Most algorithms are though based on mutual exclusion, and as such the access window in time is often limited. This has been addressed as a significant issue, as for example done by Levanoni and Petrank in [10] where their goal was to minimize the access constraints. Moreover, mutual exclusion cause blocking and can consequently incur serious problems as deadlocks, priority inversion or starvation. These problems are especially important for real-time systems, and efficient solutions only exist for uni-processor systems [16]. Some 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. Thus, lock-free algorithms can potentially incur starvation, although the other problems with mutual exclusion are avoided. 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.

Valois [19] and Michael and Scott [14] presented a lock-free memory management scheme based on reference counting. The main drawback is that re-claimed memory can not be re-used for arbitrary purposes. Detlefs et al. [1] presented a lock-free reference counting scheme that allows arbitrary re-use of the memory but is based on the CAS2[1] atomic primitive that is not available in current architectures. Michael [11, 12] designed a lock-free[2] garbage collection scheme, and a similar scheme using CAS2 have been presented by Herlihy et al. [4]. These schemes can though not be used for designing arbitrary non-blocking dynamic data structures, as they only guarantee a fixed number of references from process owned variables to nodes to be safe for accessing. Using reference counting, an arbitrary number of pointers can be guaranteed to be safe, even from within the data structure itself. Michael has also designed a general lock-free memory allocation scheme [13] for memory blocks of arbitrary size, that has to be combined with a suitable garbage collection scheme for complete memory management support. Gidenstam et al. [2] have presented a general lock-free memory allocation scheme using another approach and with other properties. Hesselink and Groote [7, 8] have presented a wait-free implementation for memory management of a set of shared tokens. However, the solution is very limited[3] as it addresses a special access pattern, and can thus not be used for implementing shared data structures in general. Consequently, several implementations of lock-free dynamic data structures [18] exist in the literature, although very few wait-free dynamic data structures [6] are known to us.

In this paper we present a wait-free algorithm for implementing a concurrent garbage collection scheme based on reference counting. The algorithm is implemented using common synchronization primitives that are available in modern systems. We have also designed a wait-free algorithm for implementing a free-list supporting concurrent allocation and freeing of fixed-size memory blocks. The presented memory management scheme can be used in user applications in a straight-forward manner, and is compatible to previous implementations of non-blocking dynamic data structures. In the algorithm description, the precise semantics of the operations are defined and a proof that our implementation is wait-free and linearizable [5] is also given.

[1]This operation, also called DCAS in the literature, can atomically update two arbitrary memory words.

[2]The de-referencing operation of pointers is lock-free, although the actual garbage collecting step is wait-free.

[3]The algorithm does not support de-referencing or updating of links containing pointers to arbitrary objects.

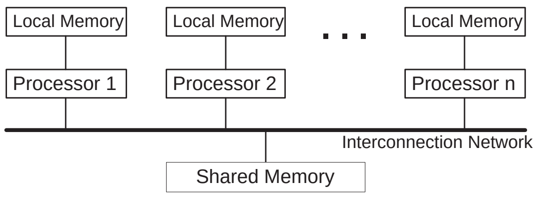

Figure 1. Shared Memory Multiprocessor System Structure

The rest of the paper is organized as follows. In Section 2 we describe the type of systems that our implementation is aiming for. The actual algorithms are described in Section 3. In Section 4 we define the precise semantics of the operations on our implementation, and show the correctness of our algorithms by proving the wait-free and linearizability properties. We conclude the paper with Section 5.

2 System Description

A typical abstraction of a shared memory multi-processor 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 might not 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), Compare-And-Swap (CAS) and the Swap (SWAP) 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 [15] [9].

3 New Algorithm

The new algorithm for garbage collection is based on reference counting. For this purpose, we assume that each

```
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;

function SWAP(address:pointer to word, newvalue:word):word
    atomic do
        oldvalue := *address;
        *address := newvalue;
        return oldvalue;
```

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

memory block (that possibly represents a node in a dynamic data structure) has a reference count field, see Figure 3. Moreover, we assume that this field (i.e. mm_ref) will be present at each memory block indefinitely, and will thus also be possible to access on nodes that have been reclaimed by the memory management scheme.

```
structure Node
    mm_ref: integer /* Initially 1 */
    mm_next: pointer to Node
    ...
```

Figure 3. The Node structure

The reference counting scheme is using the same interpretation of the reference count variable as the lock-free algorithm by Valois [19] and Michael and Scott [14]. Thus, the real reference count is the mm_ref value divided by two, and the even or odd value of mm_ref is used for consistently agreeing on when to reclaim the node.

A basic problem with reference counting arises when de-referencing pointers. Without mutual exclusion or exotic imaginary multi-word atomic primitives we can not increase the reference count of the de-referenced node at the same time as de-referencing the pointer. Moreover, without the assumption of indefinitely present mm_ref fields, it is not possible to verify that the mm_ref fields still exists while attempting to increment it. Between the de-reference and the reference count increment statements, it is thus possible that a concurrent process has removed the last reference to the node and then reclaimed it. The solution by Valois [19] is to allow increments of the mm_ref field of possibly reclaimed nodes and to afterwards verify that the pointer still points to the same node, otherwise the reference count is decremented and the de-reference scheme is repeated. However, the number of repeats is unbounded.

The key idea of our solution is that operations announce the location of the shared link before attempting to de-reference the link. Concurrent operations that have changed the link are obliged to check the possible announcement, before possibly decrementing the reference count of the node that was previously pointed to. If the announcement matches the link, the concurrent operation should then provide the de-referencing operation with the address of a node that has a positive reference count and was recently pointed to from the link. The possible answer from helping is given back in the same shared variable that was used for announcing the pending de-reference operation.

However, if the same shared variable is used for announcing several subsequent de-reference operations, it might be that slow concurrent helping operations have pending answers that refer to old announcements. As CAS is used for answering the announcement, it can not verify if the announcement is old or new if the announcement concerns the very same link, i.e. the CAS can not solve the ABA problem. In order to avoid this problem, we need to use a pool of shared variables for announcing, continuously keeping track of which shared variables that have a pending CAS from a concurrent helping operation, and only use shared variables for new announcements that have no pending CAS attempts for answering.

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

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

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

procedure HelpDeRef(link:**pointer to pointer to** Node)

The *DeRefLink* operation, see Figure 4, de-references a given link and returns the corresponding address of a node with an incremented reference count. The algorithm first announces the link in a free shared variable in lines D1-D3. It de-references the link in line D4 and increases the reference count of the corresponding node in line D5. In line D6 it removes the announcement and simultaneously detects a possible answer from helping. If an answer was received it will decrement the reference count of the possibly reclaimed node in line D8 by calling the *ReleaseRef* operation, and instead uses the answer for returning.

The *ReleaseRef* operation, see Figure 4, decrements the reference count of a given node. If all references to the node has been removed, the node will eventually be

```
union LinkOrPointer
    _: pointer to pointer to Node
    _: pointer to Node


/* Global variables */
annReadAddr[NR_THREADS][NR_THREADS]: union LinkOrPointer
annIndex[NR_THREADS]: integer
annBusy[NR_THREADS][NR_THREADS]: integer

/* Local variables */
threadId: integer /* Unique and fixed number for each thread between
  0 and NR_THREADS-1 */
n1,n2,node: pointer to Node
index,id: integer

function DeRefLink(link:pointer to pointer to Node): pointer to Node
D1   Choose index such that annBusy[threadId][index]=0
D2   annIndex[threadId]:=index;
D3   annReadAddr[threadId][index]:=link;
D4   node:=*link;
D5   if node≠⊥ then FAA(&node.mm_ref,2);
D6   n1:=SWAP(&annReadAddr[threadId][index],⊥);
D7   if n1 ≠ link then
D8       if node≠⊥ then ReleaseRef(node);
D9       node:=n1;
D10  return node;

procedure ReleaseRef(node:pointer to Node)
R1   FAA(&node.mm_ref,-2);
R2   if node.mm_ref=0 and CAS(&node.mm_ref,0,1) then
R3       Recursively call ReleaseRef for all held references by node
R4       FreeNode(node);

procedure HelpDeRef(link:pointer to pointer to Node)
H1   for id:=0 to NR_THREADS-1 do
H2       index:=annIndex[id];
H3       if annReadAddr[id][index]=link then
H4           FAA(&annBusy[id][index],1);
H5           node:=DeRefLink(link);
H6           if not CAS(&annReadAddr[id][index],link,node) then
H7               if node≠⊥ then ReleaseRef(node);
H8           FAA(&annBusy[id][index],-1);
```

Figure 4. Reference counting functions

reclaimed (by either this operation or by pending concurrent *ReleaseRef* operations). The algorithm decrements the reference count in line R1. In line R2 it detects if the reference count is zero and tries to agree on whether this operation should reclaim the node. If the node should be reclaimed, it releases all references from links that is contained in the structure of this node in line R3 and then reclaims the node by calling the *FreeNode* operation.

The *HelpDeRef* procedure, see Figure 4, tries to help concurrent *DeRefLink* operations by giving them recent results of de-referencing the corresponding given link. The algorithm checks the announcement variables for each process in lines H2-H3 in order detect if the given link matches the announcement. If the announcement matches, it first makes sure that this announcement variable can not be reused for subsequent announcements, by increasing the busy count in line H4. In line H5 it gets the recent dereference result of the link by calling the *DeRefLink* operation. If it fails to answer the announcement by using the CAS in line H6, the corresponding node's reference count is decreased by calling the *ReleaseRef* operation. After possibly helping the announcement, it releases its claim on the announcing variable by decrementing the corresponding busy count in line H8.

3.1 Allocating and freeing nodes

In order to be able to dynamically allocate and free memory blocks for representing the nodes in a dynamic data structure, we need to store information about the free nodes in a shared data structure. A solution that has been used by Valois [19] and others, is to keep the free nodes in a linked list structure (i.e. free-list) and update the head pointer using CAS. One problem with this approach is that during an attempt to remove the first item in the list, the CAS operation can not verify if the mm_next field of the node has been updated (as could have happened if the node has been removed and then later re-inserted) while possibly updating the head pointer. As we are using reference counting for garbage collection, we can increase the reference count directly before reading the mm_next field and thus make sure that the node can not be reclaimed.

As all concurrent alloc and free operation will operate on the same head pointer using CAS, a successful CAS for one operation will mean that all the other concurrent CAS attempts will possibly fail. Thus, some operations might need to retry the CAS with an potentially unbounded number of attempts.

The key ideas of our solution are to force the operations to work on different parts of the global free-list and use helping mechanisms. Each process has a shared variable for announcing the need of a free node. For the first successful CAS attempt to remove a node from the free-list, each alloc operation has to possibly help another process. The specific process to help is incremented in a round-robin manner, so that eventually every process will have got potential help. Before possibly conflicting with CAS attempts on the free-list, each free operation also has to possibly help another process with allocation. In order to avoid conflicts with concurrent free operations, each process operates on two separate free-lists. Moreover, in order to avoid conflicts with concurrent alloc operations, all alloc operations operate on the same free-list, thus always leaving one free-list free of conflicts for the corresponding free operation.

In the following description of our algorithm, we assume for simplicity reasons that there will always be enough nodes in the free-list, and thus the allocation can never fail[4]. The following functions are defined for safe handling of allocating and freeing nodes:

```
function AllocNode():pointer to Node
procedure FreeNode(node: pointer to Node)
```

The *AllocNode* operation, see Figure 5, returns a newly allocated node with a reference count indicating one active reference. The algorithm first reads in line A1-A2 which specific process to eventually help. It then repeatedly checks if that process was already helped in line A4 or instead tries to remove the first node in the active free-list. If the specified process has been helped, it resets the announcement variable in line A4 and returns the node with an adjustment of the reference count that enables future reclaiming. When trying to remove the first node in the active free-list, it first checks if the current free-list is empty in line A7 and then consequently changes the active free-list to the next free-list in order. If the active free-list is not empty, it first increases the reference count of the first node in the free-list in line A9, and tries to remove the node from the free-list using CAS in line A10. If the CAS in line A10 succeeds, the algorithm has to decide whether to use the removed node or to instead possibly give it to the process targeted for helping. If the process to be helped has not already been helped (by this or another process) and the CAS in line A12 succeeded, then the targeted process has got the removed node and the algorithm makes another try to allocate a new node. Thus, in line A14 and A16 we know that the targeted process has been helped in some way, and therefore increase the helpCurrent shared variable so that other processes will eventually get helped by subsequent *AllocNode* or *FreeNode* operation invocations. If the removed node was not given to the targeted process, we return

[4]The modification to the algorithm to also detect the out of memory condition is very simple. By counting the number of retries in the A3-A18 loop, we know that there is no memory left if the number of retries is more than a certain threshold. This threshold is given by the maximum number of retries taken such that the algorithm is wait-free (in the case of available memory) and thus depends on the number of participating threads.

```
/* Global variables */
currentFreeList: integer; /* Initially 0 */
freeList[NR_THREADS*2]: pointer to Node;
/* Initially freeList[0] points to the linked list of all available Nodes */
/* and freeList[1...NR_THREADS-1] is initialized to ⊥ */
helpCurrent; integer; /* Initially 0 */
annAlloc[NR_THREADS]: pointer to Node; /* Initially ⊥ */

function FixRef(node:pointer to Node, fix:integer):pointer to Node
        FAA(&node.mm_ref,fix);
        return node;

function AllocNode():pointer to Node
A1      helped:=false;
A2      helpId:=helpCurrent;
A3      while true do
A4          if annAlloc[threadId]≠⊥ then return
              FixRef(SWAP(&annAlloc[threadId],⊥),-1);
A5          current:=currentFreeList;
A6          node:=freeList[current];
A7          if node=⊥ then CAS(&currentFreeList,current,
              (current+1)%(NR_THREADS*2));
A8          else
A9              FAA(&node.mm_ref,2);
A10             if CAS(&freeList[current],node,node.mm_next) then
A11                 if not helped and annAlloc[helpId]=⊥ then
A12                     if CAS(&annAlloc[helpId],⊥,node) then
A13                         helped:=true;
A14                         CAS(&helpCurrent,helpId,
                              (helpId+1)%NR_THREADS);
A15                         continue;
A16                 CAS(&helpCurrent,helpId,
                      (helpId+1)%NR_THREADS);
A17                 return FixRef(node,-1);
A18             else ReleaseRef(node);

procedure FreeNode(node: pointer to Node)
F1      helpId:=helpCurrent;
F2      CAS(&helpCurrent,helpId,(helpId+1)%NR_THREADS);
F3      if not CAS(&annAlloc[helpId],⊥,node) then
F4          current:=currentFreeList;
F5          if current≤threadId or current>(NR_THREADS+threadId)
              then index:=NR_THREADS+threadId;
F6          else index:=threadId;
F7          while true do
F8              node.mm_next:=freeList[index];
F9              if CAS(&freeList[index],node.mm_next,node) then
                  break;
F10             index:=(index+NR_THREADS)%(NR_THREADS*2);
```

Figure 5. Management of the node free-list.

the node in line A17 and adjust its reference count to enable future reclaiming. If the CAS in line A10 failed, the algorithm calls *ReleaseRef* in line A18 on the node that failed to be removed (because it was no longer the first node or maybe removed) and then makes another try to allocate a new node.

The *FreeNode* operation, see Figure 5, frees the given node. The algorithm first reads in line F1 which specific process to eventually help, and then increase the helpCurrent shared variable in line F2 so that other processes will eventually get helped by subsequent *AllocNode* or *FreeNode* invocations. If the CAS succeeds in line F3, the targeted process has been helped and the algorithm has finished. Otherwise, the algorithm decides in lines F4-F6 which of the two possible (for this process) free-lists that is likely to not conflict with concurrent *AllocNode* invocations. The algorithm then repeatedly tries to insert the node in lines F8-F9 by updating the head pointer of the current free-list using CAS. If the CAS in line F9 succeeds, the algorithm is done. Otherwise it chooses the free-list with the other index and retries to insert the node in that free-list.

3.2 Usage for dynamic data structures

In order to be used safely, the presented memory management operations have to be used with care according to well-defined rules. In order for the *DeRefLink* operation to work correctly, every operation that changes a shared link must always call the *HelpDeRef* operation on that link, before eventually calling *ReleaseRef* on the node that was previously pointed to by the link. In order to avoid memory leakage, for each *AllocNode* or *DeRefLink* call there should be a matching *ReleaseRef* call. Thus, the *FreeNode* operation should never be called directly by the user application. For increasing the reference count when copying shared pointers, the FixRef(node,2) operation should be used. Direct write operations to links with an address of a reference counted node, may only be done if the previous value of the link is known to be NULL and there is no concurrent updates pending on that link. Otherwise the update must be done using CAS, and if successful the *HelpDeRef* must be called before calling *ReleaseRef* on the old value of the link. See Figure 6 for a possible substitute for the CAS calls in non-blocking algorithms of dynamic data structures. Consequently, the algorithm follows common user models for concurrent reference counting as for example the one by Detlefs et al. [1], and is therefore compatible to previous implementations of non-blocking data structures.

```
function CompareAndSwapLink(link:pointer to pointer to Node,
  old:pointer to Node, new:pointer to Node):Boolean
    if CAS(link,old,new) then
        HelpDeRef(link);
        return true;
    return false;
```

Figure 6. Operations for link modifications.

4 Correctness Proof

In this section we present the proof of our algorithm. We first prove that our algorithm is a linearizable one [5] and then we prove that it is wait-free. A set of definitions that will help us to structure and shorten the proof is first explained in this section. We start by defining the sequential semantics of our operations and then introduce two definitions concerning concurrency aspects in general.

Definition 1 *We denote with F_t the abstract internal state of the free-list containing free nodes available for allocation at time t. We interpret $n \in F$ to be true when $\exists x.annAlloc[x] = n \vee freeList[x] \rightsquigarrow n$, where $\rightsquigarrow$ means that there is a connected path through eventual mm_next pointers that leads to the node. We denote with $Ref(p, n)$ the number of references registered for process (i.e. thread) p and node n. We interpret $Ref(p_{cur}, n_1) = x$ as that the current process (p_{cur}) have contributed with an increment of $2x$ to the n_1.mm_ref field. We denote with $Del(n)$ that node n will eventually be put in the free-list if all concurrent operations will run to completion. We denote with $l \mapsto n$ that link l points to node n. The operations that are of interest for linearizability are $AllocNode(AN)$, $FreeNode(FN)$, $DeRefLink(DL)$ and $ReleaseRef(RR)$. The time t_1 is defined as the time just before the atomic execution of the operation that we are looking at, and the time t_2 is defined as the time just after the atomic execution of the same operation. In the following expressions that defines the sequential semantics of our operations, the syntax is $S_1 : O_1, S_2$, where S_1 is the conditional state before the operation O_1, and S_2 is the resulting state after performing the corresponding operation:*

$$\exists n_1.n_1 \in F_{t_1} : \mathbf{AN}() = \mathbf{n_1}, n_1 \notin F_{t_2} \wedge Ref(p_{cur}, n_1) = 1 \quad (1)$$

$$n_1 \notin F_{t_1} : \mathbf{FN}(\mathbf{n_1}) = \bot, n_1 \in F_{t_2} \quad (2)$$

$$Ref(p_{cur}, n_1) = x \wedge \exists n_1.l_1 \mapsto n_1 : \mathbf{DL}(\mathbf{l_1}) = \mathbf{n_1}, Ref(p_{cur}, n_1) = x + 1 \quad (3)$$

$$Ref(p_{cur}, n_1) = x \wedge (x \neq 1 \vee \exists p \neq p_{cur}.Ref(p, n_1) \neq 0) : \mathbf{RR}(\mathbf{n_1}), Ref(p_{cur}, n_1) = x - 1 \quad (4)$$

$$Ref(p_{cur}, n_1) = 1 \wedge \forall p \neq p_{cur}.Ref(p, n_1) = 0 : \mathbf{RR}(\mathbf{n_1}), Ref(p_{cur}, n_1) = 0 \wedge Del(n_1) \quad (5)$$

Definition 2 *In a global time model each concurrent operation Op "occupies" a time interval $[b_{Op}, f_{Op}]$ on the linear time axis ($b_{Op} < f_{Op}$). The precedence relation (denoted by '$\rightarrow$') is a relation that relates operations of a possible execution, $Op_1 \rightarrow Op_2$ means that Op_1 ends before Op_2 starts. The precedence relation is a strict partial order. Operations incomparable under $\rightarrow$ are called* overlapping. *The overlapping relation is denoted by $\parallel$ and is commutative, i.e. $Op_1 \parallel Op_2$ and $Op_2 \parallel Op_1$.*

Definition 3 *In order for an implementation of a shared concurrent data object to be linearizable [5], for every concurrent execution there should exist an equal (in the sense of the effect) and valid (i.e. it should respect the semantics of the shared data object) sequential execution that respects the partial order of the operations in the concurrent execution.*

Lemma 1 *The address of a pointer to a link (i.e. pointer to a node) and a pointer to a node can never be equal.*

Proof: The only possibility for equivalence is that the link should be the first field of the node structure. However, this is impossible as a node always starts with the mm_ref field. □

We will now show the existence and location in execution history of a point where the concurrent operation can be viewed as it occurred atomically, i.e. the linearizability point.

Lemma 2 *The DeRefLink operation ($DL(l_1) = n_1$), takes effect atomically at one statement that is executed during its invocation.*

Proof: Following from Lemma 1 we know that if the operation has been helped by a concurrent *HelpDeRef* operation (i.e. the CAS in line H6 succeeds with setting the annReadAddr element to a pointer to node type), then $n1 \neq link$ in line D6.

If the operation did not get helped, then surely $l_1 \mapsto n_1$ in line D4. As *HelpDeRef* is only called before the last reference count is possible removed (and cause $Del(n_1)$), we know that $\neg Del(n_1)$ in line D6 and thus also in line D4. Surely $Ref(p_{cur}, n_1) = Ref(p_{cur}, n_1) + 1$ holds after executing line D5. As we know that no concurrent statement that cares about the absolute reference count has been executed between line D4 and D5 (the only statement that cares about absolute reference counts is line R2, which we know have not executed as it should have executed after a possible *HelpDeRef*), we can safely interpret the reference count increase to have occurred in line D4. Thus, the linearizability point is the read sub-operation of the link in line D4.

If the operation got helped, we know that line H4 must have been executed after line D1 and that line H6 must have been executed before line D6. Thus we can assume that the operation takes effect ($DL(l_1) = n_1$) during the call to *DeRefLink* in line D5. If that operation call is also helped, we can rely on this assumption recursively. As there is a limited number of processes, and all calls to *DeRefLink* are contained in each others invocations (i.e. between line D1 and line D6 of the helped operations), there must be a last *DeRefLink* call that does not get helped. Thus, the linearizability point is the read sub-operation of the link in line D4, executed by some concurrent process. □

Lemma 3 *The ReleaseRef operation ($RR(n_1)$), takes effect atomically at one statement that is executed during its invocation.*

Proof: Surely $Ref(p_{cur}, n_1) = Ref(p_{cur}, n_1) - 1$ holds after executing line R1. If $\forall p.Ref(p, n_1) = 0$ after line R1, we have to show that the node n_1 eventually will be freed ($Del(n_1)$) if all concurrent processes finish their current invocations.

If $\forall p.Ref(p, n_1) = 0$ and the CAS fails in line R2, this means that n_1.mm_ref > 0 because of a concurrent FAA operation that have executed before the CAS operation in line R2. These FAA operations must be either from executing line D5 or line A9. If mm_ref was increased due to line D5, then line D8 will call *ReleaseRef* as *HelpDeRef* must have been called (*HelpDeRef* must have been called before the reference count was decreased). If mm_ref was increased due to line A9 then the CAS in line A10 will fail as n_1 is not in the free-list and will call *ReleaseRef* in line A18. Thus, for any of the pending or done superfluous FAA operations the corresponding call to *ReleaseRef* will eventually be done, and thus eventually the CAS of the corresponding concurrent process in line R2 will succeed. So we fulfill the condition $Del(n_1)$ directly after line R1.

Consequently, the linearizability point will be the FAA operation in line R1. □

Lemma 4 *The AllocNode operation ($AN() = n_1$), takes effect atomically at one statement that is executed during its invocation.*

Proof: If the SWAP operation in line A4 is executed, we know that annAlloc[threadId]=n_1 and thus $n_1 \in F_{t_1}$ at the time t_1 just before the SWAP operation, and that annAlloc[threadId]=$\perp$ and thus $n_1 \notin F_{t_2}$ at the time t_2 just after the SWAP operation. As we know that the helping process must have executed line A12 and thus also line A9, it must be that $Ref(p_{cur}, n_1) = 1$ also in line A4 of the current process. Consequently, the linearizability point will be the SWAP operation in line A4.

If the FixRef function is called in line A17, we know that the CAS operation in line A10 has succeeded. Thus, we know that freeList[current]=n_1 and thus $n_1 \in F_{t_1}$ at the time t_1 just before the CAS operation, and that freeList[current]$\not\rightsquigarrow n_1$ and thus $n_1 \notin F_{t_2}$ at the time t_2 just after the CAS operation. As we know that line A9 have been executed, it must be that $Ref(p_{cur}, n_1) = 1$ also in line A10. Consequently, the linearizability point will be the CAS operation in line A10. □

Lemma 5 *The FreeNode operation ($FN(n_1)$), takes effect atomically at one statement that is executed during its invocation.*

Proof: If the CAS operation in line F3 has succeeded, we know that annAlloc[helpId]=$\perp$ and thus $n_1 \notin F_{t_1}$ at the time t_1 just before the CAS operation, and that annAlloc[helpId]=n_1 and thus $n_1 \in F_{t_2}$ at the time t_2 just after the CAS operation. Consequently, the linearizability point will be the CAS operation in line F3.

If the CAS operation in line F9 has succeeded, we know that freeList[current]$\not\rightsquigarrow n_1$ and thus $n_1 \notin F_{t_1}$ at the time t_1 just before the CAS operation, and that freeList[current]=n_1 and thus $n_1 \in F_{t_2}$ at the time t_2 just after the CAS operation. Consequently, the linearizability point will be the CAS operation in line F9. □

Lemma 6 *The DeRefLink operation will always terminate in a finite number of its own steps.*

Proof: This is obvious from the algorithm description as it contains no unbounded loops. □

Lemma 7 *The ReleaseRef operation will always terminate in a finite number of its own steps.*

Proof: The CAS operation in line R2 guarantees that only one invocation for a node will execute line R3. As there are a finite number of nodes, it follows that the number of recursive calls to *ReleaseRef* caused by line R3 is also finite. □

Lemma 8 *The HelpDeRef operation will always terminate in a finite number of its own steps.*

Proof: This is obvious from the algorithm description as it contains no unbounded loops. □

Lemma 9 *The AllocNode operation will always terminate in a finite number of its own steps.*

Proof: The operation will terminate when it either gets helped (annAlloc[threadId]$\neq \perp$) or the CAS in line A10 succeeds and this operation has already helped the process with number helpId. If the CAS in line A10 fails, this is because of a concurrent succeeding CAS at either line A10 or line F9.

All concurrent *FreeNode* operations and *AllocNode* operations with a successful CAS in line A10, have or will make an attempt to help the process with number helpId. The concurrent *FreeNode* operation has made an attempt to help in line F3. Surely, a concurrent *AllocNode* invocation has made an attempt before its second succeeding CAS in line A10. This means that, after a certain number of failed CAS in line A10 of an operation, for every subsequent failed CAS in line A10, there has been an attempt from a concurrent operation to help the process with number helpId. As the helpCurrent variable is increased using CAS for every attempt, all possible values of helpCurrent will eventually be read as helpId by the concurrent processes in lines A2 or F1. Thus, an *AllocNode* invocation that continuously fails the CAS in line A10, will eventually be helped. □

Lemma 10 *The FreeNode operation will always terminate in a finite number of its own steps.*

Proof: We have to show that if the CAS in line F3 fails, there is a limited number of subsequent failing CAS attempts in line F9.

Assume for the worst case scenario that both freeList[threadId] and freeList[threadId+NR_THREADS] points to a non-empty list of free nodes. We know for sure that other concurrent *FreeNode* invocation can not cause any conflicts as they operate on separate freeList indices. The CAS in line F9 can not fail if the freeList with the corresponding used index is empty. If the CAS in line F9 fails, it must be due to a successful CAS in line A10 by a concurrent *AllocNode* invocation. As the number of processes is limited, there is a limited number of pending CAS calls in line A10 by concurrent *AllocNode* invocations that can address any of the two possibly conflicting freeList indices (threadId or threadId+NR_THREADS). All newly invocated concurrent *AllocNode* operations will operate on only one possible index (currentFreeList). If this index changes to be equal to the other possible conflicting index, the freeList with the previous index must be empty and thus the CAS in line F9 will succeed. Thus, the number of subsequent failed CAS attempts in line F9 is limited. □

Theorem 1 *The presented algorithm for concurrent garbage collection using reference counting and the algorithm for concurrent allocating and freeing of nodes, are both linearizable and wait-free.*

Proof: According to Lemmas 2, 3, 4 and 5 the *DeRefLink*, *ReleaseRef*, *AllocNode* and *FreeNode* operations take effect atomically at one statement that is executed within their invocations. Consequently, the algorithms for these operations are linearizable according to the given specification.

According to Lemmas 6, 7, 8, 9 and 10 the *DeRefLink*, *ReleaseRef*, *HelpDeRef*, *AllocNode* and *FreeNode* operations will always terminate within a finite number of their own steps. Consequently, the algorithms for these operations are wait-free. □

5 Conclusions

We have presented the first wait-free algorithm for implementing a garbage collection scheme using reference counting, that is designed for general usage. We have also designed an accompanying wait-free algorithm for implementing a free-list supporting concurrent allocation and freeing of fixed-size memory blocks. The resulting memory management scheme is implemented using atomic primitives that are available on common platforms. The user interface to the memory management operations is straightforward and is compatible to previous non-blocking dynamic data structures.

We have made successful attempts to incorporate the new wait-free memory management scheme in the lock-free implementation of a priority queue presented in [18]. Preliminary experimental results show asymptotically similar performance behavior in average compared to the default lock-free memory management scheme. However, the main strength of wait-free algorithms is not in high average performance, but rather in reliable execution guarantees that could be exploited in for example real-time systems.

We believe that our results will trigger and enable future developments of new algorithms of wait-free dynamic data structures. We are currently incorporating our results in the NOBLE [17] software 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.

[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, 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.

[5] 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.

[6] T. Herman and V. Damian-Iordache, “Space-optimal wait-free queues,” in *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*. ACM Press, 1997, p. 280.

[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] Y. Levanoni and E. Petrank, “A scalable reference counting garbage collector,” Department of Computer Science, Technion, Haifa, Israel, Tech. Rep. CS–0967, 1999.

[11] 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.

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

[13] ——, “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.

[14] 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.

[15] 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.

[16] R. Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[17] 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*, ser. Lecture Notes in Computer Science. Springer Verlag, 2002.

[18] ——, “Fast and lock-free concurrent priority queues for multi-thread systems,” in *Proceedings of the 17th International Parallel and Distributed Processing Symposium*. IEEE press, Apr. 2003, p. 11.

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