Chapter 7

Lock-Free and Practical Deques and Doubly Linked Lists using Single-Word Compare-And-Swap[1]

Håkan Sundell, Philippas Tsigas
Department of Computing Science
Chalmers Univ. of Technol. and Göteborg Univ.
412 96 Göteborg, Sweden
E-mail: {phs, tsigas}@cs.chalmers.se

Abstract

We present an efficient and practical lock-free implementation of a concurrent deque that supports parallelism for disjoint accesses and uses atomic primitives which are available in modern computer systems. Previously known lock-free algorithms of deques are either based on non-available atomic synchronization primitives, only implement a subset of the functionality, or are not designed for disjoint accesses. Our algorithm is based on a general lock-free doubly linked list, and only requires single-word compare-and-swap

[1]This is a revided and extended version of the paper that appeared as a technical report [19]. A preliminary version of this paper was also submitted to PODC 2004.

atomic primitives. It also allows pointers with full precision, and thus supports dynamic deque sizes. We have performed an empirical study using full implementations of the most efficient known algorithms of lock-free deques. For systems with low concurrency, the algorithm by Michael shows the best performance. However, as our algorithm is designed for disjoint accesses, it performs significantly better on systems with high concurrency and non-uniform memory architecture. In addition, the proposed solution also implements a general doubly linked list, the first lock-free implementation that only needs the single-word compare-and-swap atomic primitive.

7.1 Introduction

A deque (i.e. double-ended queue) is a fundamental data structure. For example, deques are often used for implementing the ready queue used for scheduling of tasks in operating systems. A deque supports four operations, the *PushRight*, the *PopRight*, the *PushLeft*, and the *PopLeft* operation. The abstract definition of a deque is a list of values, where the *PushRight/PushLeft* operation adds a new value to the right/left edge of the list. The *PopRight/PopLeft* operation correspondingly removes and returns the value on the right/left edge of the list.

To ensure consistency of a shared data object in a concurrent environment, the most common method is mutual exclusion, i.e. some form of locking. Mutual exclusion degrades the system's overall performance [17] as it causes blocking, i.e. other concurrent operations can not make any progress while the access to the shared resource is blocked by the lock. Mutual exclusion can also cause deadlocks, priority inversion and even starvation.

In order to address these problems, researchers have proposed non-blocking algorithms for shared data objects. Non-blocking algorithms do not involve mutual exclusion, and therefore do not suffer from the problems that blocking could generate. Lock-free implementations are non-blocking and guarantee that regardless of the contention caused by concurrent operations and the interleaving of their sub-operations, always at least one operation will progress. However, there is a risk for starvation as the progress of some operations could cause some other operations to never finish. Wait-free [9] algorithms are lock-free and moreover they avoid starvation as well, as all operations are then guaranteed to finish in a limited number of their own steps. Recently, some researchers also include obstruction-free [11] implementations to the non-blocking set of implementations. These kinds of implementations are weaker than the lock-free ones and do not guarantee progress of any concurrent operation.

The implementation of a lock-based concurrent deque is a trivial task, and can preferably be constructed using either a doubly linked list or a cyclic array, protected by either a single lock or by multiple locks where each lock protects a part of the shared data structure. To the best of our knowledge, there exists no implementations of wait-free deques, but several lock-free implementations have been proposed. However, all previously lock-free deques lack in several important aspects, as they either only implement a subset of the operations that are normally associated with a deque and have concurrency restrictions[2] like Arora et al. [2], or are based on atomic hardware primitives like Double-Word Compare-And-Swap (CAS2)[3] which is not available in modern computer systems. Greenwald [5] presented a CAS2-based deque implementation as well as a general doubly linked list implementation [6], and there is also a publication series of a CAS2-based deque implementation [1],[4] with the latest version by Martin et al. [13]. Valois [20] sketched out an implementation of a lock-free doubly linked list structure using Compare-And-Swap (CAS)[4], though without any support for deletions and is therefore not suitable for implementing a deque. Michael [15] has developed a deque implementation based on CAS. However, it is not designed for allowing parallelism for disjoint accesses as all operations have to synchronize, even though they operate on different ends of the deque. Secondly, in order to support dynamic maximum deque sizes it requires an extended CAS operation that can atomically operate on two adjacent words, which is not available[5] on all modern platforms.

In this paper we present a lock-free algorithm for implementing a concurrent deque that supports parallelism for disjoint accesses (in the sense that operations on different ends of the deque do not necessarily interfere with each other). The algorithm is implemented using common synchronization primitives that are available in modern systems. It allows pointers with full precision, and thus supports dynamic maximum deque sizes (in the presence of a lock-free dynamic memory handler with sufficient garbage collection support), still using normal CAS-operations. The algorithm is

[2]The algorithm by Arora et al. does not support push operations on both ends, and does not allow concurrent invocations of the push operation and a pop operation on the opposite end.

[3]A CAS2 operations can atomically read-and-possibly-update the contents of two non-adjacent memory words. This operation is also sometimes called DCAS in the literature.

[4]The standard CAS operation can atomically read-and-possibly-update the contents of a single memory word

[5]It is available on the Intel IA-32, but not on the Sparc or MIPS microprocessor architectures. It is neither available on any currently known and common 64-bit architecture.

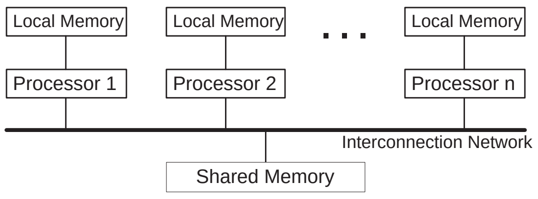


Figure 7.1: Shared Memory Multiprocessor System Structure

described in detail later in this paper, together with the aspects concerning the underlying lock-free memory management. In the algorithm description the precise semantics of the operations are defined and a proof that our implementation is lock-free and linearizable [12] is also given. We also give a detailed description of all the fundamental operations of a general doubly linked list data structure.

We have performed experiments that compare the performance of our algorithm with two of the most efficient algorithms of lock-free deques known; [15] and [13], the latter implemented using results from [3] and [7]. Experiments were performed on three different multiprocessor systems equipped with 2,4 or 29 processors respectively. All three systems used were running different operating systems and were based on different architectures. Our results show that the CAS-based algorithms outperforms the CAS2-based implementations[6] for any number of threads and any system. In non-uniform memory architectures with high contention our algorithm, because of its disjoint access property, performs significantly better than the algorithm in [15].

The rest of the paper is organized as follows. In Section 7.2 we describe the type of systems that our implementation is aiming for. The actual algorithm is described in Section 7.3. In Section 7.4 we define the precise semantics for the operations on our implementation, and show the correctness of our algorithm by proving the lock-free and linearizability properties. The experimental evaluation is presented in Section 7.5. In Section 7.6 we give the detailed description of the fundamental operations of a general doubly linked list. We conclude the paper with Section 7.7.

[6]The CAS2 operation was implemented in software, using either mutual exclusion or the results from [7], which presented an software CASn (CAS for n non-adjacent words) implementation.

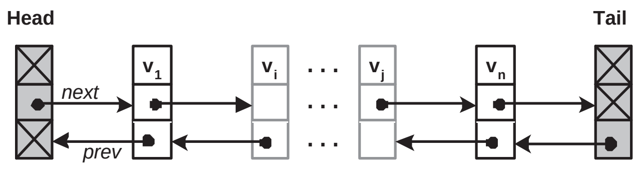


Figure 7.2: The doubly linked list data structure.

7.2 System Description

A typical abstraction of a shared memory multi-processor system configuration is depicted in Figure 7.1. 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.

7.3 The Algorithm

The algorithm is based on a doubly linked list data structure, see Figure 7.2. To use the data structure as a deque, every node contains a value. The fields of each node item are described in Figure 7.6 as it is used in this implementation. Note that the doubly linked list data structure always contains the static head and tail dummy nodes.

In order to make the doubly linked list construction concurrent and non-blocking, we are using two of the standard atomic synchronization primitives, Fetch-And-Add (FAA) and Compare-And-Swap (CAS). Figure 7.3 describes the specification of these primitives which are available in most modern platforms.

To insert or delete a node from the list we have to change the respective set of prev and next pointers. These have to be changed consistently, but

```
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 7.3: The Fetch-And-Add (FAA) and Compare-And-Swap (CAS) atomic primitives.

not necessarily all at once. Our solution is to treat the doubly linked list as being a singly linked list with auxiliary information in the prev pointers, with the next pointers being updated before the prev pointers. Thus, the next pointers always form a consistent singly linked list, but the prev pointers only give hints for where to find the previous node. This is possible because of the observation that a "late" non-updated prev pointer will always point to a node that is directly or some steps before the current node, and from that "hint" position it is always possible to traverse[7] through the next pointers to reach the directly previous node.

One problem, that is general for non-blocking implementations that are based on the singly linked list data structure, arises when inserting a new node into the list. Because of the linked list structure one has to make sure that the previous node is not about to be deleted. If we are changing the next pointer of this previous node atomically with the CAS operation, to point to the new node, and then immediately afterwards the previous node is deleted - then the new node will be deleted as well, as illustrated in Figure 7.4. There are several solutions to this problem. One solution is to use the CAS2 operation as it can change two pointers atomically, but this operation is not available in any modern multiprocessor system. A second solution is to insert auxiliary nodes [20] between every two normal nodes, and the latest method introduced by Harris [8] is to use a deletion mark. This deletion mark is updated atomically together with the next pointer. Any concurrent

[7]As will be shown later, we have defined the deque data structure in a way that makes it possible to traverse even through deleted nodes, as long as they are referenced in some way.

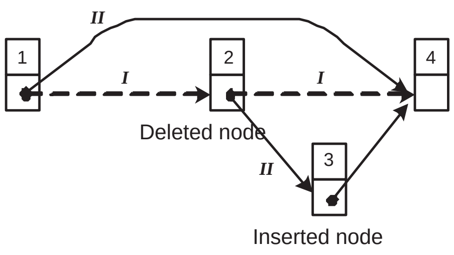


Figure 7.4: Concurrent insert and delete operation can delete both nodes.

insert operation will then be notified about the possibly set deletion mark, when its CAS operation will fail on updating the next pointer of the to-be-previous node. For our doubly linked list we need to be informed also when inserting using the prev pointer.

In order to allow usage of a system-wide dynamic memory handler (which should be lock-free and have garbage collection capabilities), all significant bits of an arbitrary pointer value must be possible to be represented in both the next and prev pointers. In order to atomically update both the next and prev pointer together with the deletion mark as done by Michael [15], the CAS-operation would need the capability of atomically updating at least $30+30+1 = 61$ bits on a 32-bit system (and $62+62+1 = 125$ bits on a 64-bit system as the pointers are then 64 bit). In practice though, most current 32 and 64-bit systems only support CAS operations of single word-size.

However, in our doubly linked list implementation, we never need to change both the prev and next pointers in one atomic update, and the pre-condition associated with each atomic pointer update only involves the pointer that is changed. Therefore it is possible to keep the prev and next pointers in separate words, duplicating the deletion mark in each of the words. In order to preserve the correctness of the algorithm, the deletion mark of the next pointer should always be set first, and the deletion mark of the prev pointer should be assured to be set by any operation that have observed the deletion mark on the next pointer, before any other updating steps are performed. Thus, full pointer values can be used, still by only using standard CAS operations.

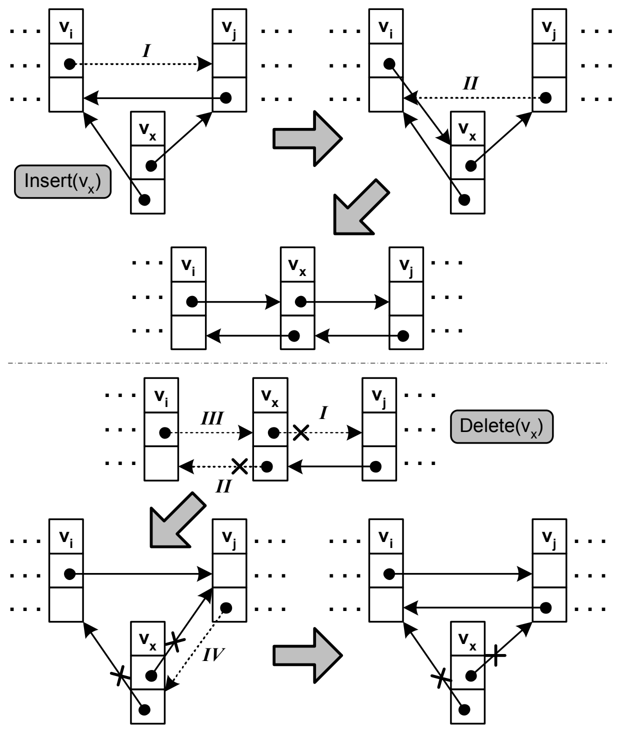


Figure 7.5: Illustration of the basic steps of the algorithms for insertion and deletion of nodes at arbitrary positions in the doubly linked list.

7.3.1 The Basic Steps of the Algorithm

The main algorithm steps, see Figure 7.5, for inserting a new node at an arbitrary position in our doubly linked list will thus be like follows: *I*) Atomically update the next pointer of the to-be-previous node, *II*) Atomically update the prev pointer of the to-be-next node. The main steps of the algorithm for deleting a node at an arbitrary position are the following: *I*) Set the deletion mark on the next pointer of the to-be-deleted node, *II*) Set the deletion mark on the prev pointer of the to-be-deleted node, *III*) Atomically update the next pointer of the previous node of the to-be-deleted node, *IV*) Atomically update the prev pointer of the next node of the to-be-deleted node. As will be shown later in the detailed description of the algorithm, helping techniques need to be applied in order to achieve the lock-free property, following the same steps as the main algorithm for inserting and deleting.

7.3.2 Memory Management

As we are concurrently (with possible preemptions) traversing nodes that will be continuously allocated and reclaimed, we have to consider several aspects of memory management. No node should be reclaimed and then later re-allocated while some other process is (or will be) traversing that node. For efficiency reasons we also need to be able to trust the prev and next pointers of deleted nodes, as we would otherwise be forced to re-start the traversing from the head or tail dummy nodes whenever reaching a deleted node while traversing and possibly incur severe performance penalties. This need is especially important for operations that try to help other delete operations in progress. Our demands on the memory management therefore rules out the SMR or ROP methods by Michael [14] and Herlihy et al. [10] respectively, as they can only guarantee a limited number of nodes to be safe via the hazard pointers, and these guarantees are also related to individual threads and never to an individual node structure. However, stronger memory management schemes as for example reference counting would be sufficient for our needs. There exists a general lock-free reference counting scheme by Detlefs et al. [3], though based on the non-available CAS2 atomic primitive.

For our implementation, we selected the lock-free memory management scheme invented by Valois [20] and corrected by Michael and Scott [16], which makes use of the FAA and CAS atomic synchronization primitives. Using this scheme we can assure that a node can only be reclaimed when there is no prev or next pointer in the list that points to it. One problem though with this scheme, a general problem with reference counting, is that it can not handle cyclic garbage (i.e. 2 or more nodes that should be recycled but reference each other, and therefore each node keeps a positive reference count, although they are not referenced by the main structure). Our solution is to make sure to break potential cyclic references directly before a node is possibly recycled. This is done by changing the next and prev pointers of a deleted node to point to active nodes, in a way that is consistent with the semantics of other operations.

The memory management scheme should also support means to de-reference pointers safely. If we simply de-reference a next or prev pointer using the means of the programming language, it might be that the corresponding node has been reclaimed before we could access it. It can also be that the deletion mark that is connected to the prev or next pointer was set, thus marking that the node is deleted. The scheme by Valois et al. supports lock-free pointer de-referencing and can easily be adopted to handle deletion marks.

The following functions are defined for safe handling of the memory management:

```
function MALLOC_NODE() :pointer to Node
function READ_NODE(address:pointer to Link) :pointer to Node
function READ_DEL_NODE(address:pointer to Link) :pointer to Node
function COPY_NODE(node:pointer to Node) :pointer to Node
procedure RELEASE_NODE(node:pointer to Node)
```

The functions *READ_NODE* and *READ_DEL_NODE* atomically de-references the given link and increases the reference counter for the corresponding node. In case the deletion mark of the link is set, the *READ_NODE* function then returns NULL. The function *MALLOC_NODE* allocates a new node from the memory pool of pre-allocated nodes. The function *RELEASE_NODE* decrements the reference counter on the corresponding given node. If the reference counter reaches zero, the function then calls the *ReleaseReferences* function that will recursively call *RELEASE_NODE* on the nodes that this node has owned pointers to, and then it reclaims the node. The *COPY_NODE* function increases the reference counter for the corresponding given node.

As the details of how to efficiently apply the memory management scheme to our basic algorithm are not always trivial, we will provide a detailed description of them together with the detailed algorithm description in this section.

7.3.3 Pushing and Popping Nodes

The *PushLeft* operation, see Figure 7.7, inserts a new node at the leftmost position in the deque. The algorithm first repeatedly tries in lines L4-L14 to insert the new node (*node*) between the head node (*prev*) and the leftmost node (*next*), by atomically changing the next pointer of the head node. Before trying to update the next pointer, it assures in line L5 that the *next* node is still the very next node of head, otherwise *next* is updated in L6-L7. After the new node has been successfully inserted, it tries in lines P1-P13 to update the prev pointer of the next node. It retries until either i) it succeeds with the update, ii) it detects that either the next or new node is deleted, or iii) the next node is no longer directly next of the new node. In any of the two latter, the changes are due to concurrent Pop or Push operations, and the responsibility to update the prev pointer is then left to those. If the update succeeds, there is though the possibility that the new node was

```
union Link
        _: word
        ⟨$p, d$⟩: ⟨pointer to Node, boolean⟩

structure Node
        value: pointer to word
        prev: union Link
        next: union Link

// Global variables
head, tail: pointer to Node
// Local variables
node,prev,prev2,next,next2: pointer to Node
link1,lastlink: union Link

function CreateNode(value: pointer to word):pointer to Node
C1      node:=MALLOC_NODE();
C2      node.value:=value;
C3      return node;

procedure ReleaseReferences(node: pointer to Node)
RR1       RELEASE_NODE(node.prev.p);
RR2       RELEASE_NODE(node.next.p);
```

Figure 7.6: The basic algorithm details.

deleted (and thus the prev pointer of the *next* node was possibly already updated by the concurrent Pop operation) directly before the CAS in line P5, and then the prev pointer is updated by calling the *HelpInsert* function in line P10.

The *PushRight* operation, see Figure 7.8, inserts a new node at the rightmost position in the deque. The algorithm first repeatedly tries in lines R4-R13 to insert the new node (*node*) between the rightmost node (*prev*) and the tail node (*next*), by atomically changing the next pointer of the *prev* node. Before trying to update the next pointer, it assures in line R5 that the *next* node is still the very next node of *prev*, otherwise *prev* is updated by calling the *HelpInsert* function in R6, which updates the the prev pointer of the *next* node. After the new node has been successfully inserted, it tries in lines P1-P13 to update the prev pointer of the next node, following the same scheme as for the *PushLeft* operation.

The *PopLeft* operation, see Figure 7.9, tries to delete and return the

```
procedure PushLeft(value: pointer to word)
L1      node:=CreateNode(value);
L2      prev:=COPY_NODE(head);
L3      next:=READ_NODE(&prev.next);
L4      while true do
L5          if prev.next $\neq$ $\langle$next,false$\rangle$ then
L6              RELEASE_NODE(next);
L7              next:=READ_NODE(&prev.next);
L8              continue;
L9          node.prev:=$\langle$prev,false$\rangle$;
L10         node.next:=$\langle$next,false$\rangle$;
L11         if CAS(&prev.next,$\langle$next,false$\rangle$,$\langle$node,false$\rangle$) then
L12             COPY_NODE(node);
L13             break;
L14         Back-Off
L15     PushCommon(node,next);
```

Figure 7.7: The algorithm for the PushLeft operation.

value of the leftmost node in the deque. The algorithm first repeatedly tries in lines PL2-PL22 to mark the leftmost node (*node*) as deleted. Before trying to update the next pointer, it first assures in line PL4 that the deque is not empty, and secondly in line PL9 that the node is not already marked for deletion. If the deque was detected to be empty, the function returns. If *node* was marked for deletion, it tries to update the next pointer of the *prev* node by calling the *HelpDelete* function, and then *node* is updated to be the leftmost node. If the prev pointer of *node* was incorrect, it tries to update it by calling the *HelpInsert* function. After the node has been successfully marked by the successful CAS operation in line PL13, it tries in line PL14 to update the next pointer of the *prev* node by calling the *HelpDelete* function, and in line PL16 to update the prev pointer of the *next* node by calling the *HelpInsert* function. After this, it tries in line PL23 to break possible cyclic references that includes *node* by calling the *RemoveCrossReference* function.

The *PopRight* operation, see Figure 7.10, tries to delete and return the value of the rightmost node in the deque. The algorithm first repeatedly tries in lines PR2-PR19 to mark the rightmost node (*node*) as deleted. Before trying to update the next pointer, it assures i) in line PR4 that the node is not already marked for deletion, ii) in the same line that the prev pointer of the tail (*next*) node is correct, and iii) in line PR7 that the deque is not empty. If the deque was detected to be empty, the function returns. If *node* was marked for deletion or the prev pointer of the *next* node was

```
procedure PushRight(value: pointer to word)
R1    node:=CreateNode(value);
R2    next:=COPY_NODE(tail);
R3    prev:=READ_NODE(&next.prev);
R4    while true do
R5        if prev.next $\neq \langle$next,false$\rangle$ then
R6            prev:=HelpInsert(prev,next);
R7            continue;
R8        node.prev:=$\langle$prev,false$\rangle$;
R9        node.next:=$\langle$next,false$\rangle$;
R10       if CAS(&prev.next,$\langle$next,false$\rangle$,$\langle$node,false$\rangle$) then
R11           COPY_NODE(node);
R12           break;
R13       Back-Off
R14   PushCommon(node,next);

procedure PushCommon(node, next: pointer to Node)
P1    while true do
P2        link1:=next.prev;
P3        if link1.d = true or node.next $\neq \langle$next,false$\rangle$ then
P4            break;
P5        if CAS(&next.prev,link1,$\langle$node,false$\rangle$) then
P6            COPY_NODE(node);
P7            RELEASE_NODE(link1.p);
P8            if node.prev.d = true then
P9                prev2:=COPY_NODE(node);
P10               prev2:=HelpInsert(prev2,next);
P11               RELEASE_NODE(prev2);
P12           break;
P13       Back-Off
P14   RELEASE_NODE(next);
P15   RELEASE_NODE(node);
```

Figure 7.8: The algorithm for the PushRight operation.

```
function PopLeft(): pointer to word
PL1     prev:=COPY_NODE(head);
PL2     while true do
PL3         node:=READ_NODE(&prev.next);
PL4         if node = tail then
PL5             RELEASE_NODE(node);
PL6             RELEASE_NODE(prev);
PL7             return ⊥;
PL8         link1:=node.next;
PL9         if link1.d = true then
PL10            HelpDelete(node);
PL11            RELEASE_NODE(node);
PL12            continue;
PL13        if CAS(&node.next,link1,⟨link1.p,true⟩) then
PL14            HelpDelete(node);
PL15            next:=READ_DEL_NODE(&node.next);
PL16            prev:=HelpInsert(prev,next);
PL17            RELEASE_NODE(prev);
PL18            RELEASE_NODE(next);
PL19            value:=node.value;
PL20            break;
PL21        RELEASE_NODE(node);
PL22        Back-Off
PL23    RemoveCrossReference(node);
PL24    RELEASE_NODE(node);
PL25    return value;
```

Figure 7.9: The algorithm for the PopLeft function.

```
function PopRight(): pointer to word
PR1   next:=COPY_NODE(tail);
PR2   node:=READ_NODE(&next.prev);
PR3   while true do
PR4       if node.next $\neq \langle$next,false$\rangle$ then
PR5           node:=HelpInsert(node,next);
PR6           continue;
PR7       if node = head then
PR8           RELEASE_NODE(node);
PR9           RELEASE_NODE(next);
PR10          return $\perp$;
PR11      if CAS(&node.next,$\langle$next,false$\rangle$,$\langle$next,true$\rangle$) then
PR12          HelpDelete(node);
PR13          prev:=READ_DEL_NODE(&node.prev);
PR14          prev:=HelpInsert(prev,next);
PR15          RELEASE_NODE(prev);
PR16          RELEASE_NODE(next);
PR17          value:=node.value;
PR18          break;
PR19      Back-Off
PR20  RemoveCrossReference(node);
PR21  RELEASE_NODE(node);
PR22  return value;
```

Figure 7.10: The algorithm for the PopRight function.

incorrect, it tries to update the prev pointer of the *next* node by calling the *HelpInsert* function, and then *node* is updated to be the rightmost node. After the node has been successfully marked it follows the same scheme as the *PopLeft* operation.

7.3.4 Helping and Back-Off

The *HelpDelete* sub-procedure, see Figure 7.11, tries to set the deletion mark of the prev pointer and then atomically update the next pointer of the previous node of the to-be-deleted node, thus fulfilling step 2 and 3 of the overall node deletion scheme. The algorithm first ensures in line HD1-HD4 that the deletion mark on the prev pointer of the given node is set. It then repeatedly tries in lines HD8-HD34 to delete (in the sense of a chain of next pointers starting from the head node) the given marked node (*node*) by changing the next pointer from the previous non-marked node. First, we can safely assume that the next pointer of the marked node is always referring to a node (*next*) to the right and the prev pointer is always referring to a node (*prev*) to the left (not necessarily the first). Before trying to update the next pointer with the CAS operation in line HD30, it assures in line HD9 that *node* is not already deleted, in line HD10 that the *next* node is not marked, in line HD16 that the *prev* node is not marked, and in HD24 that *prev* is the previous node of *node*. If *next* is marked, it is updated to be the next node. If *prev* is marked we might need to delete it before we can update *prev* to one of its previous nodes and proceed with the current deletion, but in order to avoid unnecessary and even possibly infinite recursion, *HelpDelete* is only called if a next pointer from a non-marked node to *prev* has been observed (i.e. *lastlink.d* is false). Otherwise if *prev* is not the previous node of *node* it is updated to be the next node.

The *HelpInsert* sub-function, see Figure 7.12, tries to update the prev pointer of a node and then return a reference to a possibly direct previous node, thus fulfilling step 2 of the overall insertion scheme or step 4 of the overall deletion scheme. The algorithm repeatedly tries in lines HI2-HI27 to correct the prev pointer of the given node (*node*), given a suggestion of a previous (not necessarily the directly previous) node (*prev*). Before trying to update the prev pointer with the CAS operation in line HI22, it assures in line HI4 that the *prev* node is not marked, in line HI13 that *node* is not marked, and in line HI16 that *prev* is the previous node of *node*. If *prev* is marked we might need to delete it before we can update *prev* to one of its previous nodes and proceed with the current insertion, but in order to avoid unnecessary recursion, *HelpDelete* is only called if a next pointer

```
procedure HelpDelete(node: pointer to Node)
HD1  while true do
HD2     link1:=node.prev;
HD3     if link1.d = true or
HD4       CAS(&node.prev,link1,⟨link1.p,true⟩) then break;
HD5  lastlink.d:=true;
HD6  prev:=READ_DEL_NODE(&node.prev);
HD7  next:=READ_DEL_NODE(&node.next);
HD8  while true do
HD9     if prev = next then break;
HD10    if next.next.d = true then
HD11       next2:=READ_DEL_NODE(&next.next);
HD12       RELEASE_NODE(next);
HD13       next:=next2;
HD14       continue;
HD15    prev2:=READ_NODE(&prev.next);
HD16    if prev2 = NULL then
HD17       if lastlink.d = false then
HD18          HelpDelete(prev);
HD19          lastlink.d:=true;
HD20       prev2:=READ_DEL_NODE(&prev.prev);
HD21       RELEASE_NODE(prev);
HD22       prev:=prev2;
HD23       continue;
HD24    if prev2 ≠ node then
HD25       lastlink.d:=false;
HD26       RELEASE_NODE(prev);
HD27       prev:=prev2;
HD28       continue;
HD29    RELEASE_NODE(prev2);
HD30    if CAS(&prev.next,⟨node,false⟩,⟨next,false⟩) then
HD31       COPY_NODE(next);
HD32       RELEASE_NODE(node);
HD33       break;
HD34    Back-Off
HD35 RELEASE_NODE(prev);
HD36 RELEASE_NODE(next);
```

Figure 7.11: The algorithm for the HelpDelete sub-operation.

```
function HelpInsert(prev, node: pointer to Node)
  :pointer to Node
HI1    lastlink.d:=true;
HI2    while true do
HI3        prev2:=READ_NODE(&prev.next);
HI4        if prev2 = NULL then
HI5            if lastlink.d = false then
HI6                HelpDelete(prev);
HI7                lastlink.d:=true;
HI8            prev2:=READ_DEL_NODE(&prev.prev);
HI9            RELEASE_NODE(prev);
HI10           prev:=prev2;
HI11           continue;
HI12       link1:=node.prev;
HI13       if link1.d = true then
HI14           RELEASE_NODE(prev2);
HI15           break;
HI16       if prev2 ≠ node then
HI17           lastlink.d:=false;
HI18           RELEASE_NODE(prev);
HI19           prev:=prev2;
HI20           continue;
HI21       RELEASE_NODE(prev2);
HI22       if CAS(&node.prev,link1,⟨prev,false⟩) then
HI23           COPY_NODE(prev);
HI24           RELEASE_NODE(link1.p);
HI25           if prev.prev.d = true then continue;
HI26           break;
HI27       Back-Off
HI28   return prev;
```

Figure 7.12: The algorithm for the HelpInsert sub-function.

from a non-marked node to *prev* has been observed (i.e. *lastlink.d* is false). If *node* is marked, the procedure is aborted. Otherwise if *prev* is not the previous node of *node* it is updated to be the next node. If the update in line HI22 succeeds, there is though the possibility that the *prev* node was deleted (and thus the prev pointer of *node* was possibly already updated by the concurrent Pop operation) directly before the CAS operation. This is detected in line HI25 and then the update is possibly retried with a new *prev* node.

Because the *HelpDelete* and *HelpInsert* are often used in the algorithm for "helping" late operations that might otherwise stop progress of other concurrent operations, the algorithm is suitable for pre-emptive as well as fully concurrent systems. In fully concurrent systems though, the helping strategy as well as heavy contention on atomic primitives, can downgrade the performance significantly. Therefore the algorithm, after a number of consecutive failed CAS operations (i.e. failed attempts to help concurrent operations) puts the current operation into back-off mode. When in back-off mode, the thread does nothing for a while, and in this way avoids disturbing the concurrent operations that might otherwise progress slower. The duration of the back-off is initialized to some value (e.g. proportional to the number of threads) at the start of an operation, and for each consecutive entering of the back-off mode during one operation invocation, the duration of the back-off is changed using some scheme, e.g. increased exponentially.

7.3.5 Avoiding Cyclic Garbage

The *RemoveCrossReference* sub-procedure, see Figure 7.13, tries to break cross-references between the given node (*node*) and any of the nodes that it references, by repeatedly updating the prev and next pointer as long as they reference a marked node. First, we can safely assume that the prev or next field of *node* is not concurrently updated by any other operation, as this procedure is only called by the main operation that deleted the node and both the next and prev pointers are marked and thus any concurrent update using CAS will fail. Before the procedure is finished, it assures in line RC3 that the previous node (*prev*) is not marked, and in line RC9 that the next node (*next*) is not marked. As long as *prev* is marked it is traversed to the left, and as long as *next* is marked it is traversed to the right, while continuously updating the prev or next field of *node* in lines RC5 or RC11.

```
procedure RemoveCrossReference(node: pointer to Node)
RC1    while true do
RC2        prev:=node.prev.p;
RC3        if prev.next.d = true then
RC4            prev2:=READ_DEL_NODE(&prev.prev);
RC5            node.prev:=⟨prev2,true⟩;
RC6            RELEASE_NODE(prev);
RC7            continue;
RC8        next:=node.next.p;
RC9        if next.next.d = true then
RC10           next2:=READ_DEL_NODE(&next.next);
RC11           node.next:=⟨next2,true⟩;
RC12           RELEASE_NODE(next);
RC13           continue;
RC14       break;
```

Figure 7.13: The algorithm for the RemoveCrossReference sub-operation.

7.4 Correctness Proof

In this section we present the correctness proof of our algorithm. We first prove that our algorithm is a linearizable one [12] and then we prove that it is lock-free. 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 and then introduce two definitions concerning concurrency aspects in general.

Definition 1 *We denote with Q_t the abstract internal state of a deque at the time t. $Q_t = [v_1, \ldots, v_n]$ is viewed as an list of values v, where $|Q_t| \geq 0$. The operations that can be performed on the deque are* PushLeft*(L),* PushRight*(R),* PopLeft*(PL) and* PopRight*(PR). 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 define 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:*

$$Q_{t_1} : \mathbf{L}(\mathbf{v_1}), Q_{t_2} = [v_1] + Q_{t_1} \tag{7.1}$$

$$Q_{t_1} : \mathbf{R}(\mathbf{v_1}), Q_{t_2} = Q_{t_1} + [v_1] \tag{7.2}$$

$$Q_{t_1} = \emptyset : \mathbf{PL}() = \bot, Q_{t_2} = \emptyset \tag{7.3}$$

$$Q_{t_1} = [v_1] + Q_1 : \mathbf{PL}() = \mathbf{v_1}, Q_{t_2} = Q_1 \tag{7.4}$$

$$Q_{t_1} = \emptyset : \mathbf{PR}() = \bot, Q_{t_2} = \emptyset \tag{7.5}$$

$$Q_{t_1} = Q_1 + [v_1] : \mathbf{PR}() = \mathbf{v_1}, Q_{t_2} = Q_1 \tag{7.6}$$

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 $\|$ and is commutative, i.e. $Op_1 \| Op_2$ and $Op_2 \| Op_1$. The precedence relation is extended to relate sub-operations of operations. Consequently, if $Op_1 \rightarrow Op_2$, then for any sub-operations op_1 and op_2 of Op_1 and Op_2, respectively, it holds that $op_1 \rightarrow op_2$. We also define the direct precedence relation $\rightarrow_d$, such that if $Op_1 \rightarrow_d Op_2$, then $Op_1 \rightarrow Op_2$ and moreover there exists no operation Op_3 such that $Op_1 \rightarrow Op_3 \rightarrow Op_2$.*

Definition 3 *In order for an implementation of a shared concurrent data object to be* linearizable *[12], 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.*

Next we are going to study the possible concurrent executions of our implementation. First we need to define the interpretation of the abstract internal state of our implementation.

Definition 4 *The value v is* present *($\exists i. Q[i] = v$) in the abstract internal state Q of our implementation, when there is a connected chain of next pointers (i.e. prev.next) from a* present *node (or the head node) in the doubly linked list that connects to a node that contains the value v, and this node is not marked as deleted (i.e. node.next.d=false).*

Definition 5 *The* decision point *of an operation is defined as the atomic statement where the result of the operation is finitely decided, i.e. independent of the result of any sub-operations after the decision point, the operation will have the same result. We define the* state-read point *of an operation to be the atomic statement where a sub-state of the priority queue is read, and this sub-state is the state on which the decision point depends. We also define the* state-change point *as the atomic statement where the operation changes the abstract internal state of the priority queue after it has passed the corresponding decision point.*

We will now use these points in order to 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 1 *A* PushRight *operation ($R(v)$), takes effect atomically at one statement.*

Proof: The decision, state-read and state-change point for a *PushRight* operation which succeeds ($R(v)$), is when the CAS sub-operation in line R10 (see Figure 7.8) succeeds. The state of the deque was ($Q_{t_1} = Q_1$) directly before the passing of the decision point. The prev node was the very last present node as it pointed (verified by R5 and the CAS in R10) to the tail node directly before the passing of the decision point. The state of the deque directly after passing the decision point will be $Q_{t_2} = Q_1 + [v]$ as the next pointer of the prev node was changed to point to the new node which contains the value v. Consequently, the linearizability point will be the CAS sub-operation in line R10. □

Lemma 2 *A* PushLeft *operation ($L(v)$), takes effect atomically at one statement.*

Proof: The decision, state-read and state-change point for a *PushLeft* operation which succeeds ($L(v)$), is when the CAS sub-operation in line L11 (see Figure 7.7) succeeds. The state of the deque was ($Q_{t_1} = Q_1$) directly before the passing of the decision point. The state of the deque directly after passing the decision point will be $Q_{t_2} = [v] + Q_1$ as the next pointer of the head node was changed to point to the new node which contains the value v. Consequently, the linearizability point will be the CAS sub-operation in line L11. □

Lemma 3 *A* PopRight *operation which fails ($PR() = \bot$), takes effect atomically at one statement.*

Proof: The decision point for a *PopRight* operation which fails ($PR() = \bot$) is the check in line PR7. Passing of the decision point together with the verification in line PR4 gives that the next pointer of the head node must have been pointing to the tail node ($Q_{t_1} = \emptyset$) directly before the read sub-operation of the prev field in line PR2 or the next field in line HI3, i.e. the state-read point. Consequently, the linearizability point will be the read sub-operation in line PR2 or line HI3. □

Lemma 4 *A* PopRight *operation which succeeds ($PR() = v$), takes effect atomically at one statement.*

Proof: The decision point for a *PopRight* operation which succeeds ($PR() = v$) is when the CAS sub-operation in line PR11 succeeds. Passing of the decision point together with the verification in line PR4 gives that the next pointer of the to-be-deleted node must have been pointing to the tail node ($Q_{t_1} = Q_1 + [v]$) directly before the CAS sub-operation in line PR11, i.e. the state-read point. Directly after passing the CAS sub-operation (i.e. the state-change point) the to-be-deleted node will be marked as deleted and therefore not present in the deque ($Q_{t_2} = Q_1$). Consequently, the linearizability point will be the CAS sub-operation in line PR11. □

Lemma 5 *A* PopLeft *operation which fails ($PL() = \bot$), takes effect atomically at one statement.*

Proof: The decision point for a *PopLeft* operation which fails ($PL() = \bot$) is the check in line PL4. Passing of the decision point gives that the next pointer of the head node must have been pointing to the tail node ($Q_{t_1} = \emptyset$) directly before the read sub-operation of the next pointer in line PL3, i.e. the state-read point. Consequently, the linearizability point will be the read sub-operation of the next pointer in line PL3. □

Lemma 6 *A* PopLeft *operation which succeeds ($PL() = v$), takes effect atomically at one statement.*

Proof: The decision point for a *PopLeft* operation which succeeds ($PL() = v$) is when the CAS sub-operation in line PL13 succeeds. Passing of the decision point together with the verification in line PL9 gives that the next pointer of the head node must have been pointing to the present to-be-deleted node ($Q_{t_1} = [v] + Q_1$) directly before the read sub-operation of the next pointer in line PL3, i.e. the state-read point. Directly after passing the CAS sub-operation in line PL13 (i.e. the state-change point) the to-be-deleted node will be marked as deleted and therefore not present in the deque ($\neg\exists i.Q_{t_2}[i] = v$). Unfortunately this does not match the semantic definition of the operation.

However, none of the other concurrent operations linearizability points is dependent on the to-be-deleted node's state as marked or not marked during the time interval from the state-read to the state-change point. Clearly, the linearizability points of Lemmas 1 and 2 are independent as the to-be-deleted node would be part (or not part if not present) of the corresponding Q_1 terms. The linearizability points of Lemmas 3 and 5 are independent, as those linearizability points depend on the head node's next pointer pointing to the tail node or not. Finally, the linearizability points of Lemma 4 as well as this lemma are independent, as the to-be-deleted node would be part (or not part if not present) of the corresponding Q_1 terms, otherwise the CAS sub-operation in line PL13 of this operation would have failed.

Therefore all together, we could safely interpret the to-be-deleted node to be not present already directly after passing the state-read point (($Q_{t_2} = Q_1$). Consequently, the linearizability point will be the read sub-operation of the next pointer in line PL3. □

Lemma 7 *When the deque is idle (i.e. no operations are being performed), all next pointers of present nodes are matched with a correct prev pointer from the corresponding present node (i.e. all linked nodes from the head or tail node are present in the deque).*

Proof: We have to show that each operation takes responsibility for that the affected prev pointer will finally be correct after changing the corresponding next pointer. After successfully changing the next pointer in the *PushLeft* (*PushRight*) in line L11 (R10) operation, the corresponding prev pointer is tried to be changed in line P5 repeatedly until i) it either succeeds, ii) either the next or this node is deleted as detected in line P3, iii) or a new node is inserted as detected in line P3. If a new node is inserted the corresponding *PushLeft* (*PushRight*) operation will make sure that the prev pointer is corrected. If either the next or this node is deleted, the corresponding *PopLeft* (*PopRight*) operation will make sure that the prev pointer is corrected. If the prev pointer was successfully changed it is possible that this node was deleted before we changed the prev pointer of the next node. If this is detected in line P8, then the prev pointer of the next node is corrected by the *HelpInsert* function.

After successfully marking the to-be-deleted nodes in line PL13 (PR11), the *PopLeft* (*PopRight*) functions will make sure that the connecting next pointer of the prev node will be changed to point to the closest present node to the right, by calling the *HelpDelete* procedure in line PL14 (PR12). It will also make sure that the corresponding prev pointer of the next code will be corrected by calling the *HelpInsert* function in line PL16 (PR14).

The *HelpDelete* procedure will repeatedly try to change the next pointer of the prev node that points to the deleted node, until it either succeeds changing the next pointer in line HD30 or some concurrent *HelpDelete* already succeeded as detected in line HD9.

The *HelpInsert* procedure will repeatedly try to change the prev pointer of the node to match with the next pointer of the prev node, until it either succeeds changing the prev pointer in line HI22 or the node is deleted as detected in line HI13. If it succeeded with changing the prev pointer, the prev node has possibly been deleted directly before changing the prev pointer, and therefore it is detected if the prev node is marked in line HI25 and then the procedure will continue trying to correctly change the prev pointer. □

Lemma 8 *When the deque is idle, all previously deleted nodes are garbage collected.*

Proof: We have to show that each *PopRight* or *PopLeft* operation takes responsibility for that the deleted node will finally have no references to it. The possible references are caused by other nodes pointing to it. Following Lemma 7 we know that no present nodes will reference the deleted node. It remains to show that all paths of references from a deleted node will finally reference a present node, i.e. there are no cyclic referencing. After the node is deleted in lines PL14 and PL16 (PR12 and PR14), it is assured by the *PopLeft* (*PopRight*) operation by calling the *RemoveCrossReference* procedure in line PL23 (PR20) that both the next and prev pointers are pointing to a present node. If any of those present nodes are deleted before the referencing deleted node is garbage collected in line PL24 (PR21), the *RemoveCrossReference* procedures called by the corresponding *PopLeft* or *PopRight* operation will assure that the next and prev pointers of the previously present node will point to present nodes, and so on recursively. The *RemoveCrossReference* procedure repeatedly tries to change prev pointers to point to the previous node of the referenced node until the referenced node is present, detected in line RC3 and possibly changed in line RC5. The next pointer is correspondingly detected in line RC9 and possibly changed in line RC11. □

Lemma 9 *The path of prev pointers from a node is always pointing a present node that is left of the current node.*

Proof: We will look at all possibilities where the prev pointer is set or changed. The setting in line L9 (R8) is clearly to the left as it is verified by L5 and L11 (R5 and R10). The change of the prev pointer in line P5 is to the left as verified by P3 and that nodes are never moved relatively to each other. The change of the prev pointer in line HI22 is to the left as verified by line HI3 and HI16. Finally, the change of the prev pointer in line RC5 is to the left as it is changed to the prev pointer of the previous node. □

Lemma 10 *All operations will terminate if exposed to a limited number of concurrent changes to the deque.*

Proof: The amount of changes an operation could experience is limited. Because of the reference counting, none of the nodes which are referenced to by local variables can be garbage collected. When traversing through prev or next pointers, the memory management guarantees atomicity of the operations, thus no newly inserted or deleted nodes will be missed. We also know that the relative positions of nodes that are referenced to by local variables will not change as nodes are never moved in the deque. Most loops in the operations retry because a change in the state of some node(s) was detected in the ending CAS sub-operation, and then retry by re-reading the local variables (and possibly correcting the state of the nodes) until no concurrent changes was detected by the CAS sub-operation and therefore the CAS succeeded and the loop terminated. Those loops will clearly terminate after a limited number of concurrent changes. Included in that type of loops are L4-L14, R4-R13, P1-P13, PL2-PL22 and PR3-PR19.

The loop HD8-HD34 will terminate if either the prev node is equal to the next node in line HD9 or the CAS sub-operation in line HD30 succeeds. From the start of the execution of the loop, we know that the prev node is left of the to-be-deleted node which in turn is left of the next node. Following from Lemma 9 this order will hold by traversing the prev node through its prev pointer and traversing the next node through its next pointer. Consequently, traversing the prev node through the next pointer will finally cause the prev node to be directly left of the to-be-deleted node if this is not already deleted (and the CAS sub-operation in line HD30 will finally succeed), otherwise the prev node will finally be directly left of the next node (and in the next step the equality in line HD9 will hold). As long as the prev node is marked it will be traversed to the left in line HD20, and if it is the left-most marked node the prev node will be deleted by recursively calling *HelpDelete* in line HD18. If the prev node is not marked it will be traversed to the right. As there is a limited number of changes and thus a limited number of marked nodes left of the to-be-deleted node, the prev node will finally traverse to the right and either of the termination criteria will be fulfilled.

The loop HI2-HI27 will terminate if either the to-be-corrected node is marked in line HI13 or if the CAS sub-operation in line HI22 succeeds and prev node is not marked. From the start of the execution of the loop, we know that the prev node is left of the to-be-corrected node. Following from Lemma 9 this order will hold by traversing the prev node through its prev pointer. Consequently, traversing the prev node through the next pointer will finally cause the prev node to be directly left of the to-be-corrected node if this is not deleted (and the CAS sub-operation in line HI22 will finally succeed), otherwise line HI13 will succeed. As long as the prev node is marked it will be traversed to the left in line HI8, and if it is the left-most marked node the prev node will be deleted by calling *HelpDelete* in line HI6. If the prev node is not marked it will be traversed to the right. As there is a limited number of changes and thus a limited number of marked nodes left of the to-be-corrected node, the prev node will finally traverse to the right and either of the termination criteria will be fulfilled.

The loop RC1-RC14 will terminate if both the prev node and the next node of the to-be-deleted node is not marked in line RC3 respectively line RC9. We know that from the start of the execution of the loop, the prev node is left of the to-be-deleted node and the next node is right of the to-be-deleted node. Following from Lemma 9, traversing the prev node through the next pointer will finally reach a not marked node or the head node (which is not marked), and traversing the next node through the next pointer will finally reach a not marked node or the tail node (which is not marked), and both of the termination criteria will be fulfilled. □

Lemma 11 *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. 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. 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 *PushRight*, *PushLeft*, *PopRight* or *PopLeft* operation will progress. Consequently, independent of any number of concurrent operations, one operation will always progress. □

Theorem 1 *The algorithm implements a correct, memory stable, lock-free and linearizable deque.*

Proof: Following from Lemmas 1, 2, 3, 4, 5 and 6 and by using the respective linearizability points, we can create an identical (with the same semantics) sequential execution that preserves the partial order of the operations in a concurrent execution. Following from Definition 3, the implementation is therefore linearizable.

Lemmas 10 and 11 give that our implementation is lock-free.

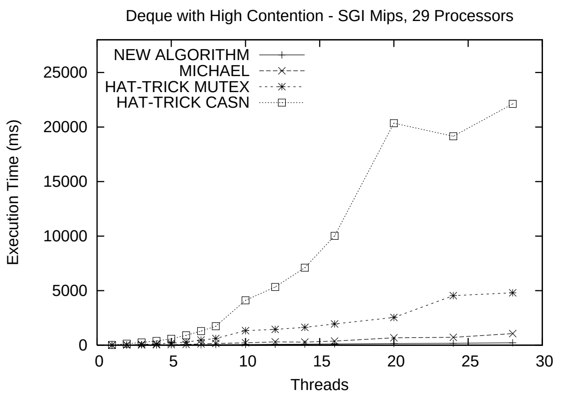
Following from Lemmas 10, 1, 2, 3, 4, 5 and 6 we can conclude that all operations will terminate with the correct result.

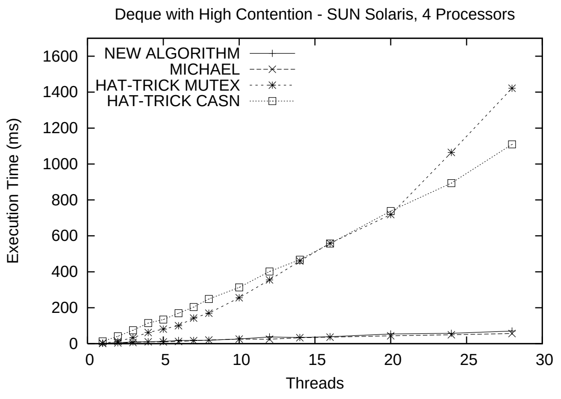
Following from Lemma 8 we know that the maximum memory usage will be proportional to the number of present values in the deque.

□

7.5 Experimental Evaluation

In our experiments, each concurrent thread performed 1000 randomly chosen sequential operations on a shared deque, with a distribution of 1/4 *PushRight*, 1/4 *PushLeft*, 1/4 *PopRight* and 1/4 *PopLeft* operations. Each experiment was repeated 50 times, and an average execution time for each experiment was estimated. Exactly the same sequence of operations was





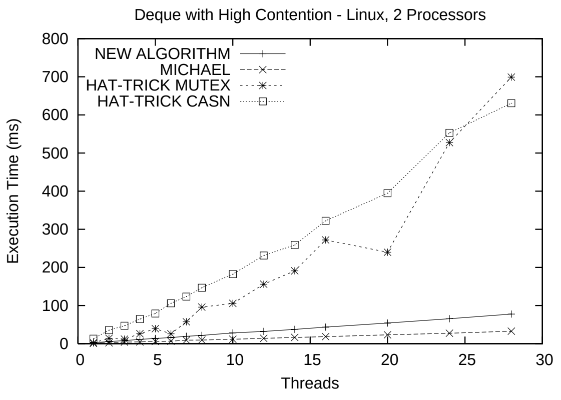
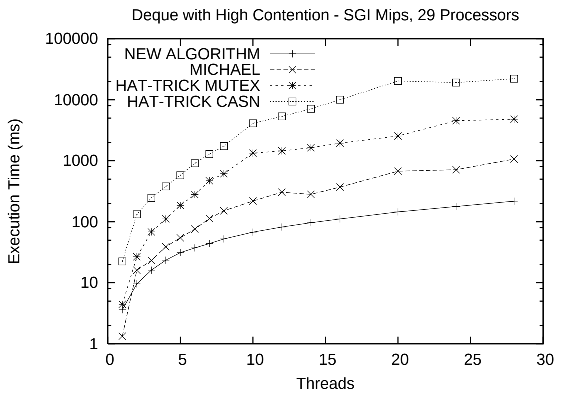
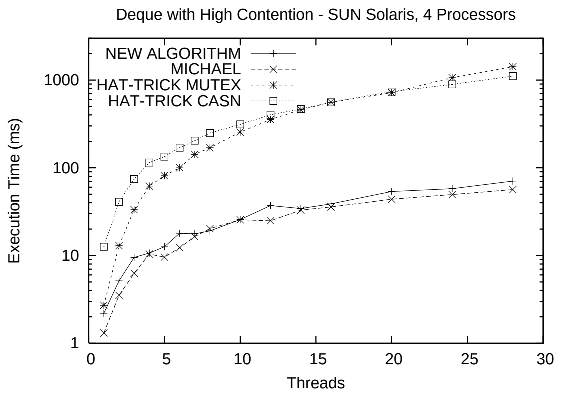


Figure 7.14: Experiment with deques and high contention.





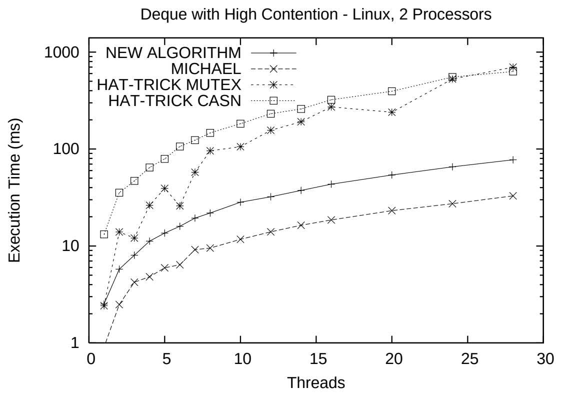


Figure 7.15: Experiment with deques and high contention, logarithmic scales.

performed for all different implementations compared. Besides our implementation, we also performed the same experiment with the lock-free implementation by Michael [15] and the implementation by Martin et al. [13], two of the most efficient lock-free deques that have been proposed. The algorithm by Martin et al. [13] was implemented together with the corresponding memory management scheme by Detlefs et al. [3]. However, as both [13] and [3] use the atomic operation CAS2 which is not available in any modern system, the CAS2 operation was implemented in software using two different approaches. The first approach was to implement CAS2 using mutual exclusion (as proposed in [13]), which should match the optimistic performance of an imaginary CAS2 implementation in hardware. The other approach was to implement CAS2 using one of the most efficient software implementations of CASN known that could meet the needs of [13] and [3], i.e. the implementation by Harris et al. [7].

A clean-cache operation was performed just before each sub-experiment using a different implementation. All implementations are written in C and compiled with the highest optimization level. The atomic primitives are written in assembly language.

The experiments were performed using different number of threads, varying from 1 to 28 with increasing steps. Three different platforms were used, with varying number of processors and level of shared memory distribution. To get a highly pre-emptive environment, we performed our experiments on a Compaq dual-processor Pentium II PC running Linux, and a Sun Ultra 80 system running Solaris 2.7 with 4 processors. In order to evaluate our algorithm with full concurrency we also used a SGI Origin 2000 system running Irix 6.5 with 29 250 MHz MIPS R10000 processors. The results from the experiments are shown in Figure 7.14. The average execution time is drawn as a function of the number of threads.

Our results show that both the CAS-based algorithms outperform the CAS2-based implementations for any number of threads. For the systems with low or medium concurrency and uniform memory architecture, [15] has the best performance. However, for the system with full concurrency and non-uniform memory architecture our algorithm performs significantly better than [15] from 2 threads and more, as a direct consequence of the nature of our algorithm to support parallelism for disjoint accesses.

7.6 General Operations for a Lock-Free Doubly Linked List

In this section we provide the details for the general operations of a lock-free doubly linked list, i.e. traversing the data structure in any direction and inserting and deleting nodes at arbitrary positions. Note that the linearizability points for these operations are defined without respect to the deque operations[8]. For maintaining the current position we adopt the *cursor* concept by Valois [20], that is basically just a reference to a node in the list.

In order to be able to traverse through deleted nodes, we also have to define the position of deleted nodes that is consistent with the normal definition of position of active nodes for sequential linked lists.

Definition 6 *The* position *of a cursor that references a node that is present in the list is the referenced node. The position of a cursor that references a deleted node, is represented by the node that was directly to the next of the deleted node at the very moment of the deletion (i.e. the setting of the deletion mark). If that node is deleted as well, the position is equal to the position of a cursor referencing that node, and so on recursively. The actual position is then interpreted to be at an imaginary node directly previous of the representing node.*

The *Next* function, see Figure 7.16, tries to change the cursor to the next position relative to the current position, and returns the status of success. The algorithm repeatedly in line NT2-NT11 checks the next node for possible traversal until the found node is present and is not the tail dummy node. If the current node is the tail dummy node, false is returned in line NT2. In line NT3 the next pointer of the current node is de-referenced and in line NT4 the deletion state of the found node is read. If the found node is deleted and the current node was deleted when directly next of the found node, this is detected in line NT5 and then the position is updated according to Definition 6 in line NT10. If the found node was detected as present in line NT5, the cursor is set to the found node in line NT10 and true is returned (unless the found node is the tail dummy node when instead false is returned) in line NT11. Otherwise it is checked if the found

[8]The general doubly linked list operation and the deque operations are compatible in the respect that the underlying data structure will be consistent. However, the linearizability point of the *PopLeft* operation is only defined with respect to the other deque operations and not with respect to the genaral doubly linked list operations.

```
function Next(cursor: pointer to pointer to Node): boolean
NT1    while true do
NT2        if *cursor = tail then return false;
NT3        next:=READ_DEL_NODE(&(*cursor).next);
NT4        d := next.next.d;
NT5        if d = true and (*cursor).next ≠ ⟨next,true⟩ then
NT6            if (*cursor).next.p = next then HelpDelete(next);
NT7            RELEASE_NODE(next);
NT8            continue;
NT9        RELEASE_NODE(*cursor);
NT10       *cursor:=next;
NT11       if d = false and next ≠ tail then return true;
```

Figure 7.16: The algorithm for the Next operation.

node is not already fully deleted in line NT6 and then fulfils the deletion by calling the *HelpDelete* procedure after which the algorithm retries at line NT2. The linearizability point of a *Next* function that succeeds is the read sub-operation of the next pointer in line NT3. The linearizability point of a *Next* function that fails is line NT2 if the node positioned by the original cursor was the tail dummy node, and the read sub-operation of the next pointer in line NT3 otherwise.

The *Prev* function, see Figure 7.17, tries to change the cursor to the previous position relative to the current position, and returns the status of success. The algorithm repeatedly in line PV2-PV11 checks the next node for possible traversal until the found node is present and is not the head dummy node. If the current node is the head dummy node, false is returned in line PV2. In line PV3 the prev pointer of the current node is de-referenced. If the found node is directly previous of the current node and the current node is present, this is detected in line PV4 and then the cursor is set to the found node in line PV6 and true is returned (unless the found node is the head dummy node when instead false is returned) in line PV7. If the current node is deleted then the cursor position is updated according to Definition 6 by calling the *Next* function in line PV8. Otherwise the prev pointer of the current node is updated by calling the *HelpInsert* function in line PV10 after which the algorithm retries at line PV2. The linearizability point of a *Prev* function that succeeds is the read sub-operation of the prev pointer in line PV3. The linearizability point of a *Prev* function that fails is line PV2 if the node positioned by the original cursor was the head dummy

```
function Prev(cursor: pointer to pointer to Node): boolean
PV1    while true do
PV2        if *cursor = head then return false;
PV3        prev:=READ_DEL_NODE(&(*cursor).prev);
PV4        if prev.next = $\langle$*cursor,false$\rangle$ and (*cursor).next.d = false then
PV5            RELEASE_NODE(*cursor);
PV6            *cursor:=prev;
PV7            if prev $\neq$ head then return true;
PV8        else if (*cursor).next.d = true then Next(cursor);
PV9        else
PV10           prev:=HelpInsert(prev,*cursor);
PV11           RELEASE_NODE(prev);
```

Figure 7.17: The algorithm for the Prev operation.

```
function Read(cursor: pointer to pointer to Node): pointer to word
RD1    if *cursor = head or *cursor = tail then return $\perp$;
RD2    value:=(*cursor).value;
RD3    if (*cursor).next.d = true then return $\perp$;
RD4    return value;
```

Figure 7.18: The algorithm for the Read function.

node, and the read sub-operation of the prev pointer in line PV3 otherwise.

The *Read* function, see Figure 7.18, returns the current value of the node referenced by the cursor, unless this node is deleted or the node is equal to any of the dummy nodes when the function instead returns a non-value. In line RD1 the algorithm checks if the node referenced by the cursor is either the head or tail dummy node, and then returns a non-value. The value of the node is read in line RD2, and in line RD3 it is checked if the node is deleted and then returns a non-value, otherwise the value is returned in line RD4. The linearizability point of a *Read* function that returns a value is the read sub-operation of the next pointer in line RD3. The linearizability point of a *Read* function that returns a non-value is the read sub-operation of the next pointer in line RD3, unless the node positioned by the cursor was the head or tail dummy node when the linearizability point is line RD1.

The *InsertBefore* operation, see Figure 7.19, inserts a new node directly before the node positioned by the given cursor and later changes the cursor to position the inserted node. If the node positioned by the cursor is the head dummy node, the new node will be inserted directly after the head dummy

```
procedure InsertBefore(cursor: pointer to pointer to Node,
 value: pointer to word)
IB1    if *cursor = head then return InsertAfter(cursor,value);
IB2    node:=CreateNode(value);
IB3    while true do
IB4        if (*cursor).next.d = true then Next(cursor);
IB5        prev:=READ_DEL_NODE(&(*cursor).prev);
IB6        node.prev:=⟨prev,false⟩;
IB7        node.next:=⟨(*cursor),false⟩;
IB8        if CAS(&prev.next,⟨(*cursor),false⟩,⟨node,false⟩) then
IB9            COPY_NODE(node);
IB10           break;
IB11       if prev.next ≠ ⟨(*cursor),false⟩ then prev:=HelpInsert(prev,*cursor);
IB12       RELEASE_NODE(prev);
IB13       Back-Off
IB14   next:=(*cursor);
IB15   *cursor:=COPY_NODE(node);
IB16   node:=HelpInsert(node,next);
IB17   RELEASE_NODE(node);
IB18   RELEASE_NODE(next);
```

Figure 7.19: The algorithm for the InsertBefore operation.

node. The algorithm checks in line IB1 if the cursor position is equal to the head dummy node, and consequently then calls the *InsertAfter* operation to insert the new node directly after the head dummy node. The algorithm repeatedly tries in lines IB4-IB13 to insert the new node (*node*) between the previous node (*prev*) of the cursor and the cursor positioned node, by atomically changing the next pointer of the prev node to instead point to the new node. If the node positioned by the cursor is deleted this is detected in line IB4 and the cursor is updated by calling the *Next* function. If the update of the next pointer of the prev node by using the CAS operation in line IB8 fails, this is because either the prev node is no longer the directly previous node of the cursor positioned node, or that the cursor positioned node is deleted. If the prev node is no longer the directly previous node this is detected in line IB11 and then the *HelpInsert* function is called in order to update the prev pointer of the cursor positioned node. If the update using CAS in line IB8 succeeds, the cursor position is set to the new node in line IB15 and the prev pointer of the previous cursor positioned node is updated by calling the *HelpInsert* function in line IB16. The linearizability

```
procedure InsertAfter(cursor: pointer to pointer to Node,
 value: pointer to word)
IA1   if *cursor = tail then return InsertBefore(cursor,value);
IA2   node:=CreateNode(value);
IA3   while true do
IA4       next:=READ_DEL_NODE(&(*cursor).next);
IA5       node.prev:=⟨(*cursor),false⟩;
IA6       node.next:=⟨next,false⟩;
IA7       if CAS(&(*cursor).next,⟨next,false⟩,⟨node,false⟩) then
IA8           COPY_NODE(node);
IA9           break;
IA10      RELEASE_NODE(next);
IA11      if (*cursor).next.d = true then
IA12          RELEASE_NODE(node);
IA13          return InsertBefore(cursor,value);
IA14      Back-Off
IA15  *cursor:=COPY_NODE(node);
IA16  node:=HelpInsert(node,next);
IA17  RELEASE_NODE(node);
IA18  RELEASE_NODE(next);
```

Figure 7.20: The algorithm for the InsertAfter operation.

point of the *InsertBefore* operation is the successful CAS operation in line IB8, or equal to the linearizability point of the *InsertBefore* operation if that operation was called in line IB1.

The *InsertAfter* operation, see Figure 7.20, inserts a new node directly after the node positioned by the given cursor and later changes the cursor to position the inserted node. If the node positioned by the cursor is the tail dummy node, the new node will be inserted directly before the tail dummy node. The algorithm checks in line IA1 if the cursor position is equal to the tail dummy node, and consequently then calls the *InsertBefore* operation to insert the new node directly after the head dummy node. The algorithm repeatedly tries in lines IA4-IA14 to insert the new node (*node*) between the cursor positioned node and the next node (*next*) of the cursor, by atomically changing the next pointer of the cursor positioned node to instead point to the new node. If the update of the next pointer of the cursor positioned node by using the CAS operation in line IA7 fails, this is because either the next node is no longer the directly next node of the cursor positioned node, or that the cursor positioned node is deleted. If the cursor positioned

```
function Delete(cursor: pointer to pointer to Node): pointer to word
D1      if *cursor = head or *cursor = tail then return $\perp$;
D2      while true do
D3         link1:=(*cursor).next;
D4         if link1.d = true then return $\perp$;
D5         if CAS(&(*cursor).next,link1,$\langle$link1.p,true$\rangle$) then
D6            HelpDelete(*cursor);
D7            prev:=COPY_NODE((*cursor).prev.p);
D8            prev:=HelpInsert(prev,link1.p);
D9            RELEASE_NODE(prev);
D10           value:=(*cursor).value;
D11           RemoveCrossReference(*cursor);
D12           return value;
```

Figure 7.21: The algorithm for the Delete function.

node is deleted, the operation to insert directly after the cursor position now becomes the problem of inserting directly before the node that represents the cursor position according to Definition 6. It is detected in line IA11 if the cursor positioned node is deleted and then it calls the InsertBefore operation in line IA13. If the update using CAS in line IA7 succeeds, the cursor position is set to the new node in line IA15 and the prev pointer of the previous cursor positioned node is updated by calling the *HelpInsert* function in line IA16. The linearizability point of the *InsertAfter* operation is the successful CAS operation in line IA7, or equal to the linearizability point of the *InsertAfter* operation if that operation was called in line IA1 or IA13.

The *Delete* operation, see Figure 7.21, tries to delete the non-dummy node referenced by the given cursor and returns the value if successful, otherwise a non-value is returned. If the cursor positioned node is equal to any of the dummy nodes this is detected in line D1 and a non-value is returned. The algorithm repeatedly tries in line D3-D5 to set the deletion mark of the next pointer of the cursor positioned node. If the deletion mark is already set, this is detected in line D4 and a non-value is returned. If the CAS operation in line D5 succeeds, the deletion process is completed by calling the *HelpDelete* procedure in line D6 and the *HelpInsert* function in line D8. In order to avoid possible problems with cyclic garbage the *RemoveCrossReference* procedure is called in line D11. The value of the deleted node is read in line D10 and the value returned in line D12. The linearizability point of a *Delete* function that returns a value is the successful CAS operation in line D5. The linearizability point of a *Delete* function that returns a non-value is the the read sub-operation of the next pointer in line D3, unless the node positioned by the cursor was the head or tail dummy node when the linearizability point instead is line D1.

The remaining necessary functionality for initializing the cursor positions like *First()* and *Last()* can be trivially derived by using the dummy nodes. If an *Update()* functionality is necessary, this could easily be achieved by extending the value field of the node data structure with a deletion mark, and throughout the whole algorithm interpret the deletion state of the whole node using this mark when semantically necessary, in combination with the deletion marks on the next and prev pointers.

7.7 Conclusions

We have presented the first lock-free algorithmic implementation of a concurrent deque that has all the following features: i) it supports parallelism for disjoint accesses, ii) uses a fully described lock-free memory management scheme, iii) uses atomic primitives which are available in modern computer systems, and iv) allows pointers with full precision to be used, and thus supports dynamic deque sizes. In addition, the proposed solution also implements all the fundamental operations of a general doubly linked list data structure in a lock-free manner. The doubly linked list operations also support deterministic and well defined traversals through even deleted nodes, and are therefore suitable for concurrent applications of linked lists in practice.

We have performed experiments that compare the performance of our algorithm with two of the most efficient algorithms of lock-free deques known, using full implementations of those algorithms. The experiments show that our implementation performs significantly better on systems with high concurrency and non-uniform memory architecture.

We believe that our implementation is of highly practical interest for multi-processor applications. We are currently incorporating it into the NOBLE [18] library.

Bibliography

[1] O. Agesen, D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. L. Steele Jr., "DCAS-based concurrent deques," in *ACM Symposium on Parallel Algorithms and Architectures*, 2000, pp. 137–146.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *ACM Symposium on Parallel Algorithms and Architectures*, 1998, pp. 119–129.

[3] 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.

[4] D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. L. Steele Jr., "Even better DCAS-based concurrent deques," in *International Symposium on Distributed Computing*, 2000, pp. 59–73.

[5] M. Greenwald, "Non-blocking synchronization and system design," Ph.D. dissertation, Stanford University, Palo Alto, CA, 1999.

[6] ——, "Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS," in *Proceedings of the twenty-first annual symposium on Principles of distributed computing.* ACM Press, 2002, pp. 260–269.

[7] T. Harris, K. Fraser, and I. Pratt, "A practical multi-word compare-and-swap operation," in *Proceedings of the 16th International Symposium on Distributed Computing*, 2002.

[8] T. L. Harris, "A pragmatic implementation of non-blocking linked lists," in *Proceedings of the 15th International Symposium of Distributed Computing*, Oct. 2001, pp. 300–314.

[9] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 1, pp. 124–149, Jan. 1991.

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

[11] ——, “Obstruction-free synchronization: Double-ended queues as an example,” in *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[12] 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.

[13] P. Martin, M. Moir, and G. Steele, “DCAS-based concurrent deques supporting bulk allocation,” Sun Microsystems, Tech. Rep. TR-2002-111, 2002.

[14] 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.

[15] ——, “CAS-based lock-free algorithm for shared deques,” in *Proceedings of the 9th International Euro-Par Conference*, ser. Lecture Notes in Computer Science. Springer Verlag, Aug. 2003.

[16] 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.

[17] A. Silberschatz and P. Galvin, *Operating System Concepts*. Addison Wesley, 1994.

[18] 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.

[19] ——, “Lock-free and practical deques using single-word compare-and-swap,” Computing Science, Chalmers University of Technology, Tech. Rep. 2004-02, Mar. 2004.

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