Lock-Free and Practical Doubly Linked List-Based Deques Using Single-Word Compare-and-Swap

Håkan Sundell and Philippas Tsigas

Department of Computing Science,
Chalmers University of Technology and Göteborg University,
412 96 Göteborg, Sweden
{phs, tsigas}@cs.chalmers.se
http://www.cs.chalmers.se/~{phs, tsigas}

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 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.

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 [1] as it causes blocking,

T. Higashino (Ed.): OPODIS 2004, LNCS 3544, pp. 240–255, 2005.

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 [2] 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 [3] 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 previous 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[1] like Arora et al. [4], or are based on atomic hardware primitives like Double-Word Compare-And-Swap (CAS2)[2] 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 [7],[8] with the latest version by Martin et al. [9]. Valois [10] sketched out an implementation of a lock-free doubly linked list structure using Compare-And-Swap (CAS)[3], though without any support for deletions and is therefore not suitable for implementing a deque. Michael [11] has developed a deque implementation based on CAS. However, it is not designed to allow 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

[1] 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.

[2] 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.

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

CAS operation that can atomically operate on two adjacent words, which is not available[4] 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). An earlier description of this algorithm appeared as a technical report [12] in March 2004. 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 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 [13] is also given.

We have performed experiments that compare the performance of our algorithm with two of the most efficient algorithms of lock-free deques known; [11] and [9], the latter implemented using results from [14] and [15]. 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[5] 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 [11].

The rest of the paper is organized as follows. In Section 2 we describe the type of targeted systems. The actual algorithm is described in Section 3. The experimental evaluation is presented in Section 4. We conclude the paper with Section 5.

2 System Description

Each node of the shared memory multi-processor system contains a processor together with its local memory. All nodes are connected to the shared memory via an interconnection network. A set of co-operating tasks is running on the system performing their respective operations. Each task is sequentially executed on one of the processors, while each processor can serve (run) many tasks at a time. The co-operating tasks, possibly running on different processors, use shared

[4] 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.

[5] The CAS2 operation was implemented in software, using either mutual exclusion or the results from [15], which presented an software CASn (CAS for n non-adjacent words) implementation.

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.

3 The New Lock-Free Algorithm

The algorithm is based on a doubly linked list data structure, see Figure 1. To use the data structure as a deque, every node contains a value. The fields of each node item are described in Figure 5 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 2 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 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

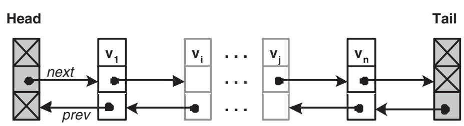

Fig. 1. The doubly linked list data structure

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

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

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

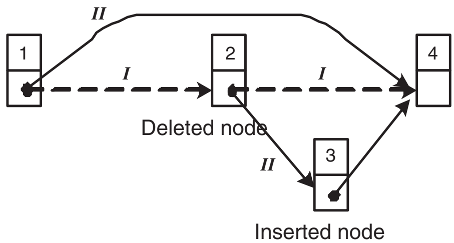

Fig. 3. Concurrent insert and delete operations can delete both nodes

possible to traverse[6] 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 a 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 3. 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 [10] between every two normal nodes, and the latest method introduced by Harris [16] is to use a deletion mark. This deletion mark is updated atomically together with the next pointer. Any concurrent 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 [11], 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

[6] 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.

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 has 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.

3.1 The Basic Steps of the Algorithm

The main algorithm steps, see Figure 4, for inserting a new node at an arbitrary position in our doubly linked list will thus be as 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

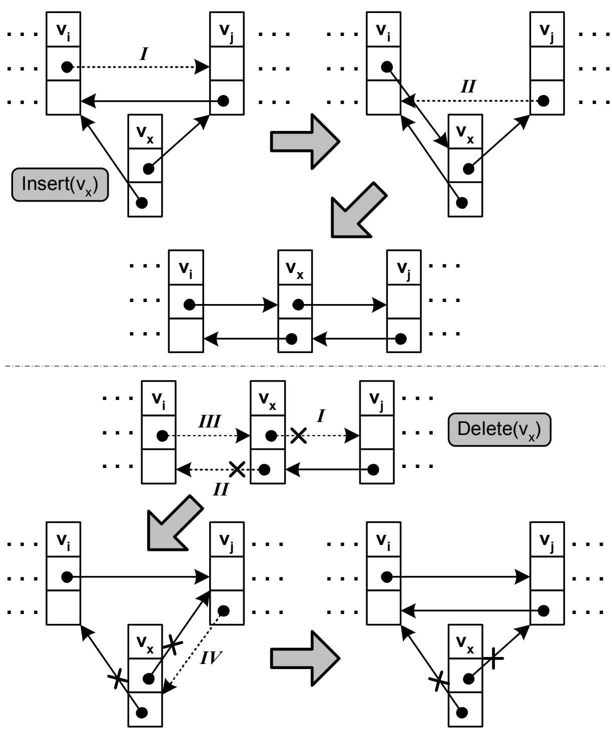

Fig. 4. Illustration of the basic steps of the algorithms for insertion and deletion of nodes at arbitrary positions in the doubly linked list, as described in Section 3.1

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.

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 the 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 rule out the SMR or ROP methods by Michael [17] and Herlihy et al. [18] respectively, as they can only guarantee a limited number of nodes to be safe, 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. [14], though based on the non-available CAS2 atomic primitive.

For our implementation, we selected the lock-free memory management scheme invented by Valois [10] and corrected by Michael and Scott [19], 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 DEREF(address:**pointer to** Link) :**pointer to** Node
function DEREF_D(address:**pointer to** Link) :**pointer to** Node

function COPY(node:**pointer to** Node) :**pointer to** Node
procedure REL(node:**pointer to** Node)

The functions *DEREF* and *DEREF_D* 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 *DEREF* function then returns NULL. The function *MALLOC_NODE* allocates a new node from the memory pool. The function *REL* decrements the reference counter on the corresponding given node. If the reference counter reaches zero, the function then calls the *TerminateNode* function that will recursively call *REL* on the nodes that this node has owned pointers to, and then it reclaims the node. The *COPY* 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.

3.3 Pushing and Popping Nodes

The *PushLeft* operation, see Figure 5, 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 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 linearizability point of the *PushLeft* operation is the successful CAS operation in line L11.

The *PushRight* operation, see Figure 5, 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 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 in the *PushLeft* operation. The linearizability point of the *PushRight* operation is the successful CAS operation in line R10.

```
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
last,link1: union Link

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

procedure TerminateNode(node: pointer to
 Node)
RR1   REL(node.prev.p);
RR2   REL(node.next.p);

procedure PushLeft(value: pointer to word)
L1    node:=CreateNode(value);
L2    prev:=COPY(head);
L3    next:=DEREF(&prev.next);
L4    while T do
L5        if prev.next ≠ ⟨next,F⟩ then
L6            REL(next);
L7            next:=DEREF(&prev.next);
L8            continue;
L9        node.prev:=⟨prev,F⟩;
L10       node.next:=⟨next,F⟩;
L11       if CAS(&prev.next,⟨next,F⟩
 ,⟨node,F⟩) then
L12           COPY(node);
L13           break;
L14       Back-Off
L15   PushCommon(node,next);

procedure PushRight(value: pointer to word)
R1    node:=CreateNode(value);
R2    next:=COPY(tail);
R3    prev:=DEREF(&next.prev);
R4    while T do
R5        if prev.next ≠ ⟨next,F⟩ then
R6            prev:=HelpInsert(prev,next);
R7            continue;
R8        node.prev:=⟨prev,F⟩;
R9        node.next:=⟨next,F⟩;
R10       if CAS(&prev.next,⟨next,F⟩
 ,⟨node,F⟩) then
R11           COPY(node);
R12           break;
R13       Back-Off
R14   PushCommon(node,next);

procedure MarkPrev(node: pointer to Node)
MP1   while T do
MP2       link1:=node.prev;
MP3       if link1.d = T or CAS(&node.prev
 ,link1,⟨link1.p,T⟩) then break;

procedure PushCommon(node, next: pointer
 to Node)
P1    while T do
P2        link1:=next.prev;
P3        if link1.d = T or node.next ≠
 ⟨next,F⟩ then
P4            break;
P5        if CAS(&next.prev,link1
 ,⟨node,F⟩) then
P6            COPY(node);
P7            REL(link1.p);
P8            if node.prev.d = T then
P9                prev2:=COPY(node);
P10               prev2:=HelpInsert(prev2,next);
P11               REL(prev2);
P12           break;
P13       Back-Off
P14   REL(next);
P15   REL(node);

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

function PopRight(): pointer to word
PR1   next:=COPY(tail);
PR2   node:=DEREF(&next.prev);
PR3   while T do
PR4       if node.next ≠ ⟨next,F⟩ then
PR5           node:=HelpInsert(node,next);
PR6           continue;
PR7       if node = head then
PR8           REL(node);
PR9           REL(next);
PR10          return ⊥;
PR11      if CAS(&node.next,⟨next,F⟩
 ,⟨next,T⟩) then
PR12          HelpDelete(node);
PR13          prev:=DEREF_D(&node.prev);
PR14          prev:=HelpInsert(prev,next);
PR15          REL(prev);
PR16          REL(next);
```

Fig. 5. The algorithm, part 1(2)

The *PopLeft* operation, see Figure 5, tries to delete and return the 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 linearizability point of a *PopLeft* operation that fails, is the read operation of the next pointer in line PL3. The linearizability point of a *PopLeft* operation that succeeds, is the read operation of the next pointer in line PL3.

The *PopRight* operation, see Figure 5, 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 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. The linearizability point of a *PopRight* operation that fails, is the read operation of the next pointer in line PR4. The linearizability point of a *PopRight* operation that succeeds, is the CAS sub-operation in line PR11.

3.4 Helping and Back-Off

The *HelpDelete* sub-procedure, see Figure 6, 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 that the deletion mark on the prev pointer of the given node is set. It then repeatedly tries in lines HD6-HD38 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 HD34, it assures in line HD6 that *node* is not already deleted, in line HD7 that the *next* node is not marked, in line HD14 that the *prev* node is not marked, and in HD28 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

```
PR17        value:=node.value;
PR18        break;
PR19     Back-Off
PR20  RemoveCrossReference(node);
PR21  REL(node);
PR22  return value;

procedure HelpDelete(node: pointer to Node)
HD1   MarkPrev(node);
HD2   last:=⊥;
HD3   prev:=DEREF_D(&node.prev);
HD4   next:=DEREF_D(&node.next);
HD5   while T do
HD6      if prev = next then break;
HD7      if next.next.d = T then
HD8         MarkPrev(next);
HD9         next2:=DEREF_D(&next.next);
HD10        REL(next);
HD11        next:=next2;
HD12        continue;
HD13     prev2:=DEREF(&prev.next);
HD14     if prev2 = ⊥ then
HD15        if last ≠ ⊥ then
HD16           MarkPrev(prev);
HD17           next2:=DEREF_D(&prev.next);
HD18           if CAS(&last.next,⟨prev,F⟩
 ,⟨next2,F⟩) then REL(prev);
HD19           else REL(next2);
HD20           REL(prev);
HD21           prev:=last;
HD22           last:=⊥;
HD23        else
HD24           prev2:=DEREF_D(&prev.prev);
HD25           REL(prev);
HD26           prev:=prev2;
HD27        continue;
HD28     if prev2 ≠ node then
HD29        if last ≠ ⊥ then REL(last);
HD30        last:=prev;
HD31        prev:=prev2;
HD32        continue;
HD33     REL(prev2);
HD34     if CAS(&prev.next,⟨node,F⟩
 ,⟨next,F⟩) then
HD35        COPY(next);
HD36        REL(node);
HD37        break;
HD38     Back-Off
HD39  if last ≠ ⊥ then REL(last);
HD40  REL(prev);
HD41  REL(next);

function HelpInsert(prev, node: pointer to
 Node): pointer to Node
HI1   last:=⊥;
HI2   while T do
HI3      prev2:=DEREF(&prev.next);
HI4      if prev2 = ⊥ then
HI5         if last ≠ ⊥ then
HI6            MarkPrev(prev);
HI7            next2:=DEREF_D(&prev.next);
HI8            if CAS(&last.next,⟨prev,F⟩
 ,⟨next2,F⟩) then REL(prev);
HI9            else REL(next2);
HI10           REL(prev);
HI11           prev:=last;
HI12           last:=⊥;
HI13        else
HI14           prev2:=DEREF_D(&prev.prev);
HI15           REL(prev);
HI16           prev:=prev2;
HI17        continue;
HI18     link1:=node.prev;
HI19     if link1.d = T then
HI20        REL(prev2);
HI21        break;
HI22     if prev2 ≠ node then
HI23        if last ≠ ⊥ then REL(last);
HI24        last:=prev;
HI25        prev:=prev2;
HI26        continue;
HI27     REL(prev2);
HI28     if link1.p = prev then break;
HI29     if prev.next = node and CAS(
 &node.prev,link1,⟨prev,F⟩) then
HI30        COPY(prev);
HI31        REL(link1.p);
HI32        if prev.prev.d ≠ T then break;
HI33     Back-Off
HI34  if last ≠ ⊥ then REL(last);
HI35  return prev;

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

Fig. 6. The algorithm, part 2(2)

prev to one of its previous nodes and proceed with the current deletion. This extra deletion is only attempted if a next pointer from a non-marked node to *prev* has been observed (i.e. *last* is valid). Otherwise if *prev* is not the previous node of *node* it is updated to be the next node.

The *HelpInsert* sub-function, see Figure 6, 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-HI33 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 HI29, it assures in line HI4 that the *prev* node is not marked, in line HI19 that *node* is not marked, and in line HI22 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 deletion. This extra deletion is only attempted if a next pointer from a non-marked node to *prev* has been observed (i.e. *last* is valid). 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 HI29 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 HI32 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.

3.5 Avoiding Cyclic Garbage

The *RemoveCrossReference* sub-procedure, see Figure 6, 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 fully 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 fully marked, and in line RC9 that the next node (*next*) is not fully 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.

3.6 General Operations of Doubly Linked Lists and Correctness Proofs

Due to page restrictions, the detailed description of the general operations of a doubly linked list (i.e. traversals and arbitrary inserts and deletes) as well as detailed proofs of correctness of the lock-free and linearizability criteria are described in an extended version of this paper [20].

4 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 were performed for all different implementations compared. Besides our implementation, we also performed the same experiment with the lock-free implementation by Michael [11] and the implementation by Martin et al. [9], two of the most efficient lock-free deques that have been proposed. The algorithm by Martin et al. was implemented together with the corresponding memory management scheme by Detlefs et al. [14]. However, as both [9] and [14] 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 [9]). The other approach was to implement CAS2 using one of the most efficient software implementations of CASN known that could meet the needs of [9] and [14], i.e. the implementation by Harris et al. [15].

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. 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, [11] has the best performance. However, for the system with full concurrency and non-uniform memory architecture our algorithm performs significantly better than [11] from 2 threads and more, as a direct consequence of the nature of our algorithm to support parallelism for disjoint accesses.

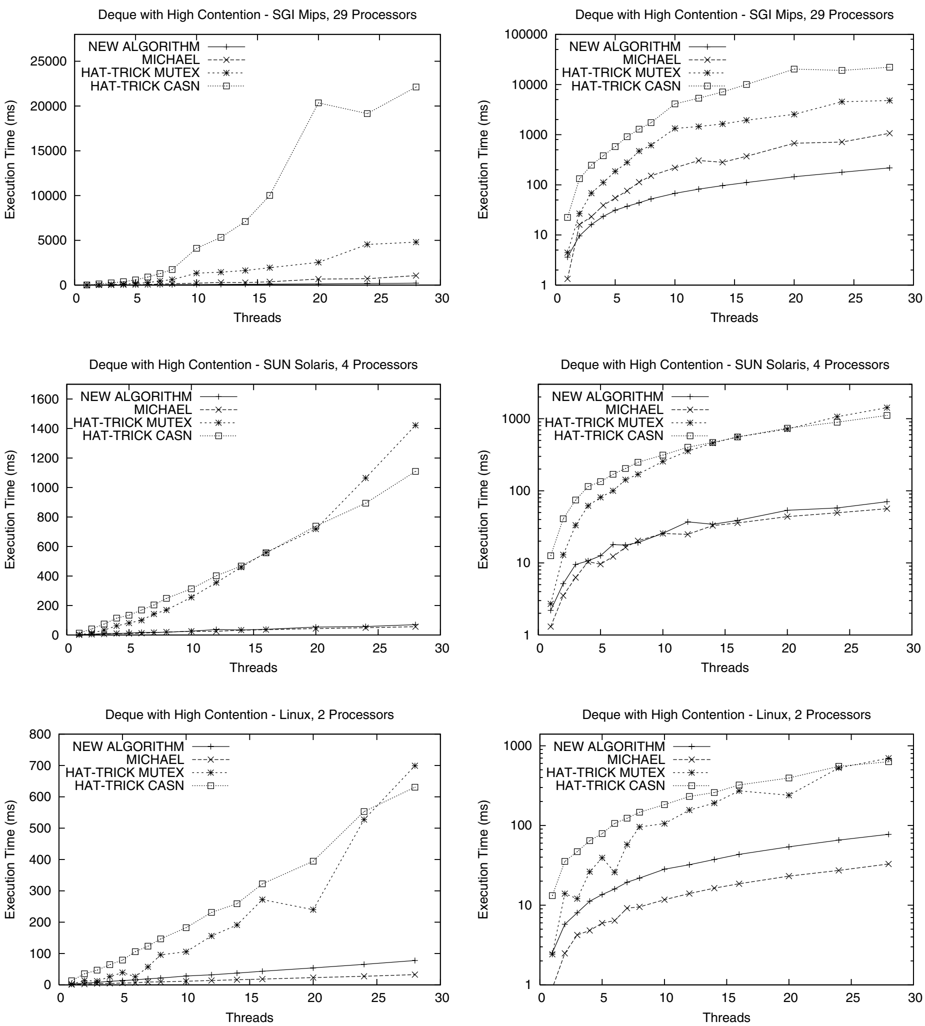

Fig. 7. Experiment with deques and high contention. Logarithmic scales in the right column

5 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 [21] library.

References

1. Silberschatz, A., Galvin, P.: Operating System Concepts. Addison Wesley (1994)
2. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems **11** (1991) 124–149
3. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: Proceedings of the 23rd International Conference on Distributed Computing Systems. (2003)
4. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. In: ACM Symposium on Parallel Algorithms and Architectures. (1998) 119–129
5. Greenwald, M.: Non-Blocking Synchronization and System Design. PhD thesis, Stanford University, Palo Alto, CA (1999)
6. Greenwald, M.: 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) 260–269
7. Agesen, O., Detlefs, D., Flood, C.H., Garthwaite, A., Martin, P., Shavit, N., Steele Jr., G.L.: DCAS-based concurrent deques. In: ACM Symposium on Parallel Algorithms and Architectures. (2000) 137–146
8. Detlefs, D., Flood, C.H., Garthwaite, A., Martin, P., Shavit, N., Steele Jr., G.L.: Even better DCAS-based concurrent deques. In: International Symposium on Distributed Computing. (2000) 59–73
9. Martin, P., Moir, M., Steele, G.: DCAS-based concurrent deques supporting bulk allocation. Technical Report TR-2002-111, Sun Microsystems (2002)
10. Valois, J.D.: Lock-Free Data Structures. PhD thesis, Rensselaer Polytechnic Institute, Troy, New York (1995)

11. Michael, M.M.: CAS-based lock-free algorithm for shared deques. In: Proceedings of the 9th International Euro-Par Conference. Lecture Notes in Computer Science, Springer Verlag (2003)
12. Sundell, H., Tsigas, P.: Lock-free and practical deques using single-word compare-and-swap. Technical Report 2004-02, Computing Science, Chalmers University of Technology (2004)
13. Herlihy, M., Wing, J.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems **12** (1990) 463–492
14. Detlefs, D., Martin, P., Moir, M., Steele Jr, G.: Lock-free reference counting. In: Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing. (2001)
15. Harris, T., Fraser, K., Pratt, I.: A practical multi-word compare-and-swap operation. In: Proceedings of the 16th International Symposium on Distributed Computing. (2002)
16. Harris, T.L.: A pragmatic implementation of non-blocking linked lists. In: Proceedings of the 15th International Symposium of Distributed Computing. (2001) 300–314
17. Michael, M.M.: 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) 21–30
18. Herlihy, M., Luchangco, V., Moir, M.: The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structure. In: Proceedings of 16th International Symposium on Distributed Computing. (2002)
19. Michael, M.M., Scott, M.L.: Correction of a memory management method for lock-free data structures. Technical report, Computer Science Department, University of Rochester (1995)
20. Sundell, H.: Efficient and Practical Non-Blocking Data Structures. PhD thesis, Department of Computing Science, Chalmers University of Technology (2004)
21. Sundell, H., Tsigas, P.: NOBLE: A non-blocking inter-process communication library. In: Proceedings of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers. (2002)