Technical Report no. 2004-02

Lock-Free and Practical Deques using Single-Word Compare-And-Swap

Håkan Sundell **Philippas Tsigas**

CHALMERS | GÖTEBORG UNIVERSITY

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

Göteborg, 2004

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

Technical Report no. 2004-02
ISSN: 1650-3023

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

Göteborg, Sweden, 2004

Abstract

We present an efficient and practical lock-free implementation of a concurrent deque that is disjoint-parallel accessible 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 doubly linked list, and only requires single-word compare-and-swap atomic primitives, even for dynamic memory sizes. We have performed an empirical study using full implementations of the most efficient algorithms of lock-free deques known. 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.

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 [15] 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.

Researchers have addressed these problems by proposing non-blocking algorithms for shared data objects. Non-blocking methods 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 [8] 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, researchers also include obstruction-free [9] implementations to be non-blocking, although this kind of implementation is weaker than lock-free and thus does 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 with either a single lock or with 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[1] like Arora et al [2], 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, and there is also a publication series of a CAS2-based deque implementation [1],[4] with the latest version by Martin et al [12]. Independently of our work, Michael [13] has developed a deque implementation based on Compare-And-Swap (CAS)[3]. However, it is not disjoint-parallel accessible as all operations have to synchronize, even though they operate on different ends of the deque. Secondly, in order to use dynamic maximum deque sizes it requires an extended 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 of a concurrent deque that is disjoint-parallel accessible[5] (in the sense that operations on different ends of the deque do not necessarily interfere with each other) and implemented using common synchronization primitives that are available in modern systems. It can be extended to use dynamic maximum deque sizes (in the presence of a lock-free dynamic memory handler), still using normal CAS-operations. The algorithm is described in detail later in this paper, and the aspects concerning the underlying lock-free memory management are also presented. The precise semantics of the operations are defined and we give a proof that our implementation is lock-free and linearizable [10].

[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

[4] It is available on the Intel IA-32, but not on the Sparc or MIPS microprocessor architectures.

[5] There is a general and formal definition called disjoint-access-parallel by Israeli and Rappoport [11]

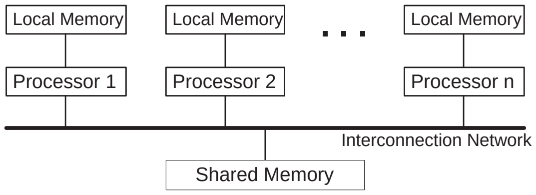

Figure 1. Shared Memory Multiprocessor System Structure

We have performed experiments that compare the performance of our algorithm with two of the most efficient algorithms of lock-free deques known; [13] and [12], the latter implemented using results from [3] and [6]. Experiments were performed on three different multiprocessor system equipped with either 2,4 or 29 processors. All of the systems are using different operating systems. Our results show that the CAS-based algorithms outperforms the CAS2-based implementations[6] for any number of threads, and for the system with full concurrency and non-uniform memory architecture our algorithm performs significantly better than the algorithm in [13].

The rest of the paper is organized as follows. In Section 2 we describe the type of systems that our implementation is aimed for. The actual algorithm is described in Section 3. In Section 4 we define the precise semantics for the operations on our implementation, and show their correctness by proving the lock-free and linearizability properties. The experimental evaluation is presented in Section 5. We conclude the paper with Section 6.

2 System Description

A typical abstraction of a shared memory multi-processor system configuration is depicted in Figure 1. Each node of the system contains a processor together with its local memory. All nodes are connected to the shared memory via an interconnection network. A set of co-operating tasks is running on the system performing their respective operations. Each task is sequentially executed on one of the processors, while each processor can serve (run) many tasks at a time. The co-operating tasks, possibly running on different processors, use shared data objects built in the shared memory to co-ordinate and communicate. Tasks synchronize their operations on the shared data objects through sub-operations on top of a cache-coherent shared memory. The shared memory may not though be uniformly accessible for all nodes in the system; processors can have different access times on different parts of the memory.

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

```
function TAS(value:pointer to word):boolean
    atomic do
        if *value=0 then
            *value:=1;
            return true;
        else return false;

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

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

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

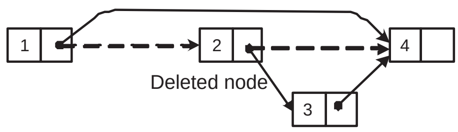

Figure 3. Concurrent insert and delete operation can delete both nodes.

3 Algorithm

The algorithm is based on a doubly-linked list data structure. To use the data structure as a deque, every node contains a value. The fields of each node item are described in Figure 4 as it is used in this implementation.

In order to make the deque construction concurrent and non-blocking, we are using three of the standard atomic synchronization primitives, Test-And-Set (TAS), 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 previous of 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 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 CAS, 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 [17] between every two normal nodes, and the latest method introduced by Harris [7] 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 be able to atomically update both the prev and the next pointer together with the deletion mark, all of these have to be put together in one memory word. For a 32-bit word this means a maximum of 32 768 (or 2 147 483 648 for a 64-bit word) possibly addressable nodes using the prev or next pointers. However, as will be shown later in Section 3.4, the algorithm can easily be extended to handle dynamic maximum sizes, thus making this limit obsolete.

3.1 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. This can be solved for example by careful reference counting. We have selected the lock-free memory management scheme invented by Valois [17] and corrected by Michael and Scott [14], 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 with this scheme 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.

Another memory management issue is how to de-reference pointers safely. If we simply de-reference the pointer, 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 following functions are defined for safe handling of the memory management:

function MALLOC_NODE() :**pointer to** Node
function READ_PREV(address:**pointer to** Link) :**pointer to** Node
function READ_NEXT(address:**pointer to** Link) :**pointer to** Node
function READ_PREV_DEL(address:**pointer to** Link) :**pointer to** Node
function READ_NEXT_DEL(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 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 count 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 function *COPY_NODE* increases the reference counter for the corresponding given node. *READ_PREV*, *READ_PREV_DEL*, *READ_NEXT* and *READ_NEXT_DEL* atomically de-references the given link in the corresponding direction and increases the reference counter for the corresponding node. In case the deletion mark of the link is set, the functions *READ_PREV* and *READ_NEXT* return NULL.

3.2 Pushing and Popping Nodes

The *PushLeft* operation, see Figure 4, first repeatingly tries in lines L4-L15 to insert the new node (*node*) between

[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.

```
union Link
    _: word
    ⟨prev, next, d⟩: ⟨pointer to Node, pointer to Node, boolean⟩

structure Node
    value: pointer to word
    link: union Link

// Global variables
head, tail: pointer to Node
// Local variables
node, prev, prev2, next, next2: pointer to Node
link1, link2, lastlink: union Link

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

procedure ReleaseReferences(node:pointer to Node)
    RELEASE_NODE(node.link.prev);
    RELEASE_NODE(node.link.next);

procedure PushLeft(value: pointer to word)
    node:=CreateNode(value);
    prev:=COPY_NODE(head);
    next:=READ_NEXT(&prev.link);
    while true do
        link1:=prev.link;
        if link1.next ≠ next then
            RELEASE_NODE(next);
            next:=READ_NEXT(&prev.link);
            continue;
        node.link:=⟨prev,link1.next,false⟩;
        link2:=⟨link1.prev,node,false⟩;
        if CAS(&prev.link,link1,link2) then
            COPY_NODE(node);
            break;
        Back-Off
    PushCommon(node,next);

procedure PushRight(value: pointer to word)
    node:=CreateNode(value);
    next:=COPY_NODE(tail);
    prev:=READ_PREV(&next.link);
    while true do
        link1:=prev.link;
        if link1.next ≠ next or prevlink.d = true then
            prev:=HelpInsert(prev,next);
            continue;
        node.link:=⟨prev,link1.next,false⟩;
        link2:=⟨link1.prev,node,false⟩;
        if CAS(&prev.link,link1,link2) then
            COPY_NODE(node);
            break;
        Back-Off
    PushCommon(node,next);

procedure PushCommon(node: pointer to Node, next: pointer to Node)
    while true do
        link1:=next.link;
        link2:=⟨node,link1.next,false⟩;
        if link1.d = true or node.link.d = true
         or node.link.next ≠ next then
            break;
        if CAS(&next.link,link1,link2) then
            COPY_NODE(node);
            RELEASE_NODE(link1.prev);
            if node.link.d = true then
                prev2:=COPY_NODE(node);
                prev2:=HelpInsert(prev2,next);
                RELEASE_NODE(prev2);
            break;
        Back-Off
    RELEASE_NODE(next);
    RELEASE_NODE(node);

function PopLeft(): pointer to word
    prev:=COPY_NODE(head);
    while true do
        node:=READ_NEXT(&prev.link);
        if node = tail then
            RELEASE_NODE(node);
            RELEASE_NODE(prev);
            return ⊥;
        link1:=node.link;
        if link1.d = true then
            DeleteNext(node);
            RELEASE_NODE(node);
            continue;
        next:=COPY_NODE(link1.next);
        link2:=⟨link1.prev,link1.next,true⟩
        if CAS(&node.link,link1,link2) then
            DeleteNext(node);
            prev:=HelpInsert(prev,next);
            RELEASE_NODE(prev);
            RELEASE_NODE(next);
            value:=node.value;
            break;
        RELEASE_NODE(node);
        RELEASE_NODE(next);
        Back-Off
    RemoveCrossReference(node);
    RELEASE_NODE(node);
    return value;

function PopRight(): pointer to word
    next:=COPY_NODE(tail);
    while true do
        node:=READ_PREV(&next.link);
        link1:=node.link;
        if link1.next ≠ next or link1.d = true then
            node:=HelpInsert(node,next);
            RELEASE_NODE(node);
            continue;
```

Figure 4. The algorithm, part 1(2).

```
    if node = head then
        RELEASE_NODE(next);
        RELEASE_NODE(node);
        return ⊥;
    prev:=COPY_NODE(link1.prev);
    link2:=⟨link1.prev,link1.next,true⟩
    if CAS(&node.link,link1,link2) then
        DeleteNext(node);
        prev:=HelpInsert(prev,next);
        RELEASE_NODE(prev);
        RELEASE_NODE(next);
        value:=node.value;
        break;
    RELEASE_NODE(prev);
    RELEASE_NODE(node);
    Back-Off
RemoveCrossReference(node);
RELEASE_NODE(node);
return value;

procedure DeleteNext(node: pointer to Node)
lastlink.d:=true;
prev:=READ_PREV_DEL(&node.link);
next:=READ_NEXT_DEL(&node.link);
while true do
    if prev = next then break;
    if next.link.d = true then
        next2:=READ_NEXT_DEL(&next.link);
        RELEASE_NODE(next);
        next:=next2;
        continue;
    prev2:=READ_NEXT(&prev.link);
    if prev2 = NULL then
        if lastlink.d = false then
            DeleteNext(prev);
            lastlink.d:=true;
        prev2:=READ_PREV_DEL(&prev.link);
        RELEASE_NODE(prev);
        prev:=prev2;
        continue;
    link1:=⟨prev.link.prev,prev2,false⟩;
    if prev2 ≠ node then
        lastlink.d:=false;
        RELEASE_NODE(prev);
        prev:=prev2;
        continue;
    RELEASE_NODE(prev2);
    link2:=⟨link1.prev,node.link.next,false⟩;
    if CAS(&prev.link,link1,link2) then
        COPY_NODE(link2.next);
        RELEASE_NODE(node);
        break;
    Back-Off
RELEASE_NODE(prev);
RELEASE_NODE(next);

function HelpInsert(prev: pointer to Node, node: pointer to Node)
  :pointer to Node
lastlink.d:=true;
while true do
    prev2:=READ_NEXT(&prev.link);
    if prev2 = NULL then
        if lastlink.d = false then
            DeleteNext(prev);
            lastlink.d:=true;
        prev2:=READ_PREV_DEL(&prev.link);
        RELEASE_NODE(prev);
        prev:=prev2;
        continue;
    link1:=node.link;
    if link1.d = true then
        RELEASE_NODE(prev2);
        break;
    if prev2 ≠ node then
        lastlink.d:=false;
        RELEASE_NODE(prev);
        prev:=prev2;
        continue;
    RELEASE_NODE(prev2);
    link2:=⟨prev,link1.next,false⟩;
    if CAS(&node.link,link1,link2) then
        COPY_NODE(prev);
        RELEASE_NODE(link1.prev);
        if prev.link.d = true then continue;
        break;
    Back-Off
return prev;

procedure RemoveCrossReference(node: pointer to Node)
while true do
    link1:=node.link;
    prev:=link1.prev;
    if prev.link.d = true then
        prev2:=READ_PREV_DEL(&prev.link);
        node.link:=⟨prev2,link1.next,true⟩;
        RELEASE_NODE(prev);
        continue;
    next:=link1.next;
    if next.link.d = true then
        next2:=READ_NEXT_DEL(&next.link);
        node.link:=⟨link1.prev,next2,true⟩;
        RELEASE_NODE(next);
        continue;
    break;
```

Figure 5. The algorithm, part 2(2).

the head node (*prev*) and the leftmost node (*next*), by atomically changing the next pointer of the head node. Before trying to update the link field, it assures in line L6 that the *next* node is still the very next node of head, otherwise *next* is updated in L7-L8. After the new node has been successfully inserted, it tries in lines P2-P15 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 P7, and then the prev pointer is updated by calling the *HelpInsert* function in line P12.

The *PushRight* operation, see Figure 4, first repeatedly tries in lines R4-R14 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 link field, it assures in line R6 that the *next* node is still the very next node of *prev*, otherwise *prev* is updated by calling the *HelpInsert* function in R7-R8, which updates the prev pointer of the *next* node. After the new node has been successfully inserted, it tries in lines P2-P15 to update the prev pointer of the next node, following the same scheme as for the *PushLeft* operation.

The *PopLeft* operation, see Figure 4, first repeatedly tries in lines PL2-PL24 to mark the leftmost node (*node*) as deleted. Before trying to update the link field, 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 *DeleteNext* 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 PL15, it tries in line PL16 to update the next pointer of the *prev* node by calling the *DeleteNext* function, and in line PL17 to update the prev pointer of the *next* node by calling the *HelpInsert* function. After this, it tries in line PL25 to break possible cyclic references that includes *node* by calling the *RemoveCrossReference* function.

The *PopRight* operation, see Figure 4, first repeatedly tries in lines PR2-PR24 to mark the rightmost node (*node*) as deleted. Before trying to update the link field, it assures i) in line PR5 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 PR9 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.

3.3 Helping and Back-Off

The *DeleteNext* procedure, see Figure 5, repeatedly tries in lines DN4-DN32 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 link field with the CAS operation in line DN28, it assures in line DN5 that *node* is not already deleted, in line DN6 that the *next* node is not marked, in line DN12 that the *prev* node is not marked, and in DN21 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 infinite recursion, *DeleteNext* 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* procedure, see Figure 5, repeatedly tries in lines HI2-HI28 to correct the prev pointer of the given node (*node*), given a suggestion of a previous (not necessarily the first) node (*prev*). Before trying to update the link field with the CAS operation in line HI23, 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, *DeleteNext* is only called if a next pointer 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 HI23 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 HI26 and then the update is possibly retried with a new *prev* node.

The *RemoveCrossReference* procedure, see Figure 5, tries to break cross-references between the given node (*node*) and any of the nodes that it references, by repeatedly updating the prev or next pointer as long as it references a marked node. First, we can safely assume that the link field of *node* is not concurrently updated by any other operation. Before the procedure is finished, it assures in line RC4 that the previous node (*prev*) is not marked, and in line RC10 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 link field of *node* in lines RC6 or RC12.

Because the *DeleteNext* 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 proportional to the number of threads, and for each consecutive entering of the back-off mode during one operation invocation, the duration of the back-off is increased exponentially.

3.4 Extending to dynamic maximum sizes

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, 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). However, most current 32 and 64-bit systems only support CAS-operations of single word-size.

An interesting observation of the current algorithm is that it never changes both the prev and next pointer in the atomic updates, and the pre-condition associated with the atomic CAS-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. Thus, full pointer values can be used, still by only using standard CAS-operations. In order to preserve the correctness of the algorithm, the deletion mark of the next pointer should always be set first, in the *PopLeft/PopRight* functions, and the deletion mark of the prev pointer should be possibly set in the very beginning of the *DeleteNext* procedure. The remaining changes are trivial and the full extended algorithm is presented in Appendix A.

4 Correctness

In this section we present the proof of our algorithm. We first prove that our algorithm is a linearizable one [10] 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 explained in this section. We start by defining the sequential semantics of our operations and then introduce two definitions concerning concurrency aspects in general.

Definition 1 *We denote with 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:*

$$Q_{t_1} : \mathbf{L}(\mathbf{v_1}), Q_{t_2} = [v_1] + Q_{t_1} \quad (1)$$

$$Q_{t_1} : \mathbf{R}(\mathbf{v_1}), Q_{t_2} = Q_{t_1} + [v_1] \quad (2)$$

$$Q_{t_1} = \emptyset : \mathbf{PL}() = \bot, Q_{t_2} = \emptyset \quad (3)$$

$$Q_{t_1} = [v_1] + Q_1 : \mathbf{PL}() = \mathbf{v_1}, Q_{t_2} = Q_1 \quad (4)$$

$$Q_{t_1} = \emptyset : \mathbf{PR}() = \bot, Q_{t_2} = \emptyset \quad (5)$$

$$Q_{t_1} = Q_1 + [v_1] : \mathbf{PR}() = \mathbf{v_1}, Q_{t_2} = Q_1 \quad (6)$$

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 [10], 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.link.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.link.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 R11 (see Figure 4) 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 R6 and the CAS in R11) 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 R11. □

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 L12 (see Figure 4) 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 L12. □

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

Proof: The decision point for a *PopRight* operation which fails ($PR() = \perp$) is the check in line PR9. Passing of the decision point together with the verification in line PR5 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 link field in line PR4, i.e. the state-read point. Consequently, the linearizability point will be the read sub-operation in line PR4. □

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 PR15 succeeds. Passing of the decision point together with the verification in line PR5 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 PR15, 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 PR15. □

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

Proof: The decision point for a *PopLeft* operation which fails ($PL() = \perp$) 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 link field in line PL3, i.e. the state-read point. Consequently, the linearizability point will be the read sub-operation 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 PL15 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 in line PL3, i.e. the state-read point. Directly after passing the CAS sub-operation in line PL15 (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 PL15 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 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 L12 (R11) operation, the corresponding prev pointer is tried to be changed in line P7 repeatedly until i) it either succeeds, ii) either the next or this node is deleted as detected in line P4, iii) or a new node is inserted as detected in line P5. 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 P10, then the prev pointer of the next node is corrected by the *HelpInsert* function.

After successfully marking the to-be-deleted nodes in line PL15 (PR15), 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 *DeleteNext* procedure in line PL16 (PR16). It will also make sure that the corresponding prev pointer of the next code will be corrected by calling the *HelpInsert* function in line PL17 (PR17).

The *DeleteNext* 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 DN28 or some concurrent *DeleteNext* already succeeded as detected in line DN5.

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 HI23 or the node is deleted as detected in line HI13. If it succeeded with changing the prev pointer, the prev node might have been deleted directly before changing the prev pointer, and therefore it is detected if the prev node is marked in line HI26 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 PL16 and PL17 (PR16 and PR17), it is assured by the *PopLeft* (*PopRight*) operation by calling the *RemoveCrossReference* procedure in line PL25 (PR25) 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 , 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 RC4 and possibly changed in line RC6. The next pointer is correspondingly detected in line RC10 and possibly changed in line RC12. □

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 L10 (R9) is clearly to the left as it is verified by L6 and L12 (R5 and R11). The change of the prev pointer in line P7 is to the left as verified by P5 and that nodes are never moved relatively to each other. The change of the prev pointer in line HI23 is to the left as verified by line HI3 and HI16. Finally, the change of the prev pointer in line RC6 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 is 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-L15, R4-R14, P1-P15, PL2-PL24 and PR2-PR24.

The loop DN4-DN32 will terminate if either the prev node is equal to the next node in line DN5 or the CAS sub-operation in line DN28 succeeds. We know from the start of the execution of the loop, 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 DN28 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 DN5 will hold). As long as the prev node is marked it will be traversed to the left in line DN16, and if it is the left-most marked node the prev node will be deleted by recursively calling *DeleteNext* in line DN14. 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-HI28 will terminate if either the to-be-corrected node is marked in line HI13 or if the CAS sub-operation in line HI23 succeeds and prev node is not marked. We know that from the start of the execution of the loop, 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 HI23 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 *DeleteNext* 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-RC15 will terminate if both the prev node and the next node of the to-be-deleted node is not marked in line RC4 respectively line RC10. 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.

□

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 sequential operations were performed for all different implementations compared. Besides our implementation, we also performed the same experiment with the lock-free implementation by Michael [13] and the implementation by Martin et al [12], two of the most efficient lock-free deques that have been proposed. The algorithm by Martin et al [12] was implemented together with the corresponding memory management scheme by Detlefs et al [3]. However, as both [12] 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 [12]), 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 [12] and [3], i.e. the implementation by Harris et al [6].

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 assembler.

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 6. The average execution time is drawn as a function of the number of threads.

Our results show that both the CAS-based algorithms outperforms the CAS2-based implementations for any number of threads. For the systems with low or medium concurrency and uniform memory architecture, [13] has the best performance. However, for the system with full concurrency and non-uniform memory architecture our algorithm performs significantly better than [13] from 2 threads and more, as a direct consequence of the disjoint-parallel accessible nature of our algorithm.

6 Conclusions

We have presented the first lock-free algorithmic implementation of a concurrent deque that has all the following features: i) it is disjoint-parallel accessible with retained parallelism, ii) uses a fully described lock-free memory management scheme, and iii) uses atomic primitives which are available in modern computer systems, even when extended for dynamic maximum sizes.

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 [16] library.

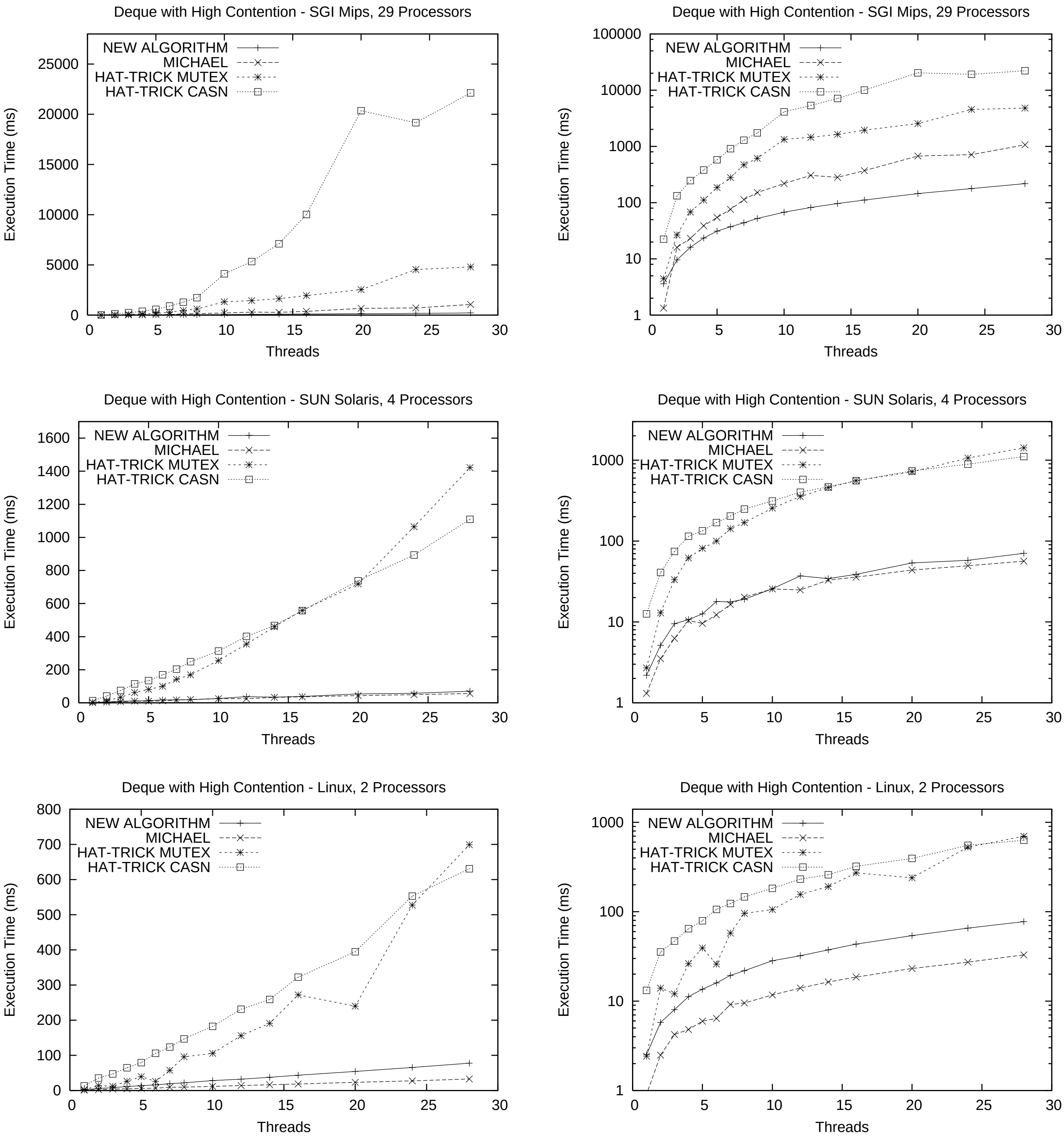

Figure 6. Experiment with deques and high contention. Logarithmic scales to the right.

References

[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] 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.

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

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

[9] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[10] 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.

[11] A. Israeli and L. Rappoport, "Disjoint-access-parallel implementations of strong shared memory primitives," in *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*. ACM, Aug. 1994, pp. 151–160.

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

[13] M. M. Michael, "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.

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

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

[16] 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.

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

A The algorithm for dynamic maximum sizes

This section shows the the full details of how to extend the algorithm for dynamic maximum sizes, following the guidelines earlier presented in Section 3.4.

As the link structure now can contain full pointer values, see Figure 7 , the following functions are added for safe handling of the memory management:

function READ_NODE(address:**pointer to** Link) :**pointer to** Node
function READ_DEL_NODE(address:**pointer to** Link) :**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 function *READ_NODE* returns NULL.

The remaining details of the extended algorithm are showed in Figures 7 and 8.

```
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
    node:=MALLOC_NODE();
    node.value:=value;
    return node;

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

procedure PushLeft(value: pointer to word)
    node:=CreateNode(value);
    prev:=COPY_NODE(head);
    next:=READ_NODE(&prev.next);
    while true do
        if prev.next ≠ ⟨next,false⟩ then
            RELEASE_NODE(next);
            next:=READ_NODE(&prev.next);
            continue;
        node.prev:=⟨prev,false⟩;
        node.next:=⟨next,false⟩;
        if CAS(&prev.next,⟨next,false⟩,⟨node,false⟩) then
            COPY_NODE(node);
            break;
        Back-Off
    PushCommon(node,next);

procedure PushRight(value: pointer to word)
    node:=CreateNode(value);
    next:=COPY_NODE(tail);
    prev:=READ_NODE(&next.prev);
    while true do
        if prev.next ≠ ⟨next,false⟩ then
            prev:=HelpInsert(prev,next);
            continue;
        node.prev:=⟨prev,false⟩;
        node.next:=⟨next,false⟩;
        if CAS(&prev.next,⟨next,false⟩,⟨node,false⟩) then
            COPY_NODE(node);
            break;
        Back-Off
    PushCommon(node,next);

procedure PushCommon(node, next: pointer to Node)
    while true do
        link1:=next.prev;
        if link1.d = true or node.next ≠ ⟨next,false⟩ then
            break;
        if CAS(&next.prev,link1,⟨node,false⟩) then
            COPY_NODE(node);
            RELEASE_NODE(link1.p);
            if node.prev.d = true then
                prev2:=COPY_NODE(node);
                prev2:=HelpInsert(prev2,next);
                RELEASE_NODE(prev2);
            break;
        Back-Off
    RELEASE_NODE(next);
    RELEASE_NODE(node);

function PopLeft(): pointer to word
    prev:=COPY_NODE(head);
    while true do
        node:=READ_NODE(&prev.next);
        if node = tail then
            RELEASE_NODE(node);
            RELEASE_NODE(prev);
            return ⊥;
        link1:=node.next;
        if link1.d = true then
            DeleteNext(node);
            RELEASE_NODE(node);
            continue;
        if CAS(&node.next,link1,⟨link1.p,true⟩) then
            DeleteNext(node);
            next:=READ_DEL_NODE(&node.next);
            prev:=HelpInsert(prev,next);
            RELEASE_NODE(prev);
            RELEASE_NODE(next);
            value:=node.value;
            break;
        RELEASE_NODE(node);
        Back-Off
    RemoveCrossReference(node);
    RELEASE_NODE(node);
    return value;

function PopRight(): pointer to word
    next:=COPY_NODE(tail);
    node:=READ_NODE(&next.prev);
    while true do
        if node.next ≠ ⟨next,false⟩ then
            node:=HelpInsert(node,next);
            continue;
        if node = head then
            RELEASE_NODE(node);
            RELEASE_NODE(next);
            return ⊥;
```

Figure 7. The algorithm for dynamic maximum sizes, part 1(2).

```
    if CAS(&node.next,⟨next,false⟩,⟨next,true⟩) then
        DeleteNext(node);
        prev:=READ_DEL_NODE(&node.prev);
        prev:=HelpInsert(prev,next);
        RELEASE_NODE(prev);
        RELEASE_NODE(next);
        value:=node.value;
        break;
    Back-Off
RemoveCrossReference(node);
RELEASE_NODE(node);
return value;

procedure DeleteNext(node: pointer to Node)
while true do
    link1:=node.prev;
    if link1.d = true or
       CAS(&node.prev,link1,⟨link1.p,true⟩) then break;
lastlink.d:=true;
prev:=READ_DEL_NODE(&node.prev);
next:=READ_DEL_NODE(&node.next);
while true do
    if prev = next then break;
    if next.next.d = true then
        next2:=READ_DEL_NODE(&next.next);
        RELEASE_NODE(next);
        next:=next2;
        continue;
    prev2:=READ_NODE(&prev.next);
    if prev2 = NULL then
        if lastlink.d = false then
            DeleteNext(prev);
            lastlink.d:=true;
        prev2:=READ_DEL_NODE(&prev.prev);
        RELEASE_NODE(prev);
        prev:=prev2;
        continue;
    if prev2 ≠ node then
        lastlink.d:=false;
        RELEASE_NODE(prev);
        prev:=prev2;
        continue;
    RELEASE_NODE(prev2);
    if CAS(&prev.next,⟨node,false⟩,⟨next,false⟩) then
        COPY_NODE(next);
        RELEASE_NODE(node);
        break;
    Back-Off
RELEASE_NODE(prev);
RELEASE_NODE(next);

function HelpInsert(prev, node: pointer to Node)
 :pointer to Node
lastlink.d:=true;
while true do
    prev2:=READ_NODE(&prev.next);
    if prev2 = NULL then
        if lastlink.d = false then
            DeleteNext(prev);
            lastlink.d:=true;
        prev2:=READ_DEL_NODE(&prev.prev);
        RELEASE_NODE(prev);
        prev:=prev2;
        continue;
    link1:=node.prev;
    if link1.d = true then
        RELEASE_NODE(prev2);
        break;
    if prev2 ≠ node then
        lastlink.d:=false;
        RELEASE_NODE(prev);
        prev:=prev2;
        continue;
    RELEASE_NODE(prev2);
    if CAS(&node.prev,link1,⟨prev,false⟩) then
        COPY_NODE(prev);
        RELEASE_NODE(link1.p);
        if prev.prev.d = true then continue;
        break;
    Back-Off
return prev;

procedure RemoveCrossReference(node: pointer to Node)
while true do
    prev:=node.prev.p;
    if prev.next.d = true then
        prev2:=READ_DEL_NODE(&prev.prev);
        node.prev:=⟨prev2,true⟩;
        RELEASE_NODE(prev);
        continue;
    next:=node.next.p;
    if next.next.d = true then
        next2:=READ_DEL_NODE(&next.next);
        node.next:=⟨next2,true⟩;
        RELEASE_NODE(next);
        continue;
    break;
```

Figure 8. The algorithm for dynamic maximum sizes, part 2(2).