J. Parallel Distrib. Comput. 68 (2008) 1008–1020

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Lock-free deques and doubly linked lists

Håkan Sundell [a,b,*], Philippas Tsigas [b]

[a] School of Business and Informatics, University College of Borås, 501 90 Borås, Sweden

[b] Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, 412 96 Göteborg, Sweden

ARTICLE INFO

Article history:
Received 23 January 2007
Received in revised form
11 January 2008
Accepted 7 March 2008
Available online 15 March 2008

Keywords:
Deque
Doubly linked list
Non-blocking
Lock-free
Shared data structure
Multi-thread
Concurrent

ABSTRACT

We present a practical lock-free shared data structure that efficiently implements the operations of a concurrent deque as well as a general doubly linked list. The implementation supports parallelism for disjoint accesses and uses atomic primitives which are available in modern computer systems. Previously known lock-free algorithms of doubly linked lists 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 only requires single-word compare-and-swap atomic primitives, supports fully dynamic list sizes, and allows traversal also through deleted nodes and thus avoids unnecessary operation retries. We have performed an empirical study of our new algorithm on two different multiprocessor platforms. Results of the experiments performed under high contention show that the performance of our implementation scales linearly with increasing number of processors. Considering deque implementations and 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 *doubly linked list* is a fundamental data structure and is used in kernel as well as user level applications. For example, doubly linked lists are often used for implementing the *deque* (i.e., doubly ended queue) or *dictionary* abstract data types. A general doubly linked list should support modifications and bidirectional traversal of a list of items. The items are not necessarily enumerable as they are only related through their relative position in the list. Thus, the data structure should at least support the following operations; *InsertBefore, InsertAfter, Delete, Read, Next, Prev, First*, and *Last*. A corresponding deque abstract data type should at least support the following operations; *PushLeft, PushRight, PopLeft*, and *PopRight*.

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*, i.e., other concurrent operations cannot 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. Some researchers also consider obstruction-free [3] implementations, that are weaker than the lock-free ones and do not guarantee progress of any concurrent operation.

The implementation of a lock-based concurrent doubly linked list is a trivial task, and can preferably be 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 exist no implementations of wait-free doubly linked lists. Greenwald [4] presented a CAS2-based doubly linked list implementation. However, it is only presented in the form of a dictionary abstract data type and thus requires enumerable items, and is not disjoint-access-parallel — a property identified by Israeli and Rappoport [5] as crucial for avoiding bottlenecks and permitting actual performed parallelism. Moreover, CAS2[1]

* Corresponding author at: School of Business and Informatics, University College of Borås, 501 90 Borås, Sweden.

E-mail addresses: phs@cs.chalmers.se (H. Sundell), tsigas@cs.chalmers.se (P. Tsigas).

[1] A CAS2 (DCAS) operation can atomically read-and-possibly-update the contents of two non-adjacent memory words.

0743-7315/$ – see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2008.03.001

is not available in modern computer systems. Recently, Attiya and Hellel [6] presented a CAS2-based doubly linked list allowing more parallelism. Valois [7] sketched a lock-free doubly linked list implementation that is based on common atomic hardware primitives. However, it is not general as it does not support any delete operations.

Several lock-free implementations [8–12] of the deque abstract data type have been proposed that are based on the doubly linked list. However, these implementations do not support all operations necessary for a *general* doubly linked list data structure, or are based on atomic hardware primitives which are not available in modern computer systems. The deque implementation by Michael [12] requires CAS of double-width for supporting fully dynamic sizes and is not disjoint-access-parallel.

Sundell and Tsigas [13,14] presented a practical lock-free deque implementation which can also support general doubly linked list operations.[2] The basic idea is to utilize the built-in redundancy of the doubly linked list and allow temporary inconsistency of the data structure while operations are performed on it. Afterwards, based on a very similar approach, Martin [15] designed an implementation of a lock-free doubly linked list specifically for the Java framework. This implementation has been adopted for use in a lock-free deque implementation in the Java concurrency package by Lea [16]. The basic technique of optimistic pointers has also been used consequently for the design of other data structures based on linked lists, e.g., a queue by Ladan-Mozes and Shavit [17]. However, the doubly linked list algorithm in [15] does not support traversals over deleted nodes[3] and do not guarantee that the underlying data structure is consistent[4] in both directions when it is idle. Recently Heller et al. [18] showed that it can be beneficial for linked list structures to allow concurrent threads to traverse over locked nodes (i.e., logically deleted nodes), a technique also exploited in a lock-free manner in [14,19].

In this paper we improve and extend on the algorithm by Sundell and Tsigas [14], by presenting the general operations for a lock-free doubly linked list as well as operations for a deque abstract data type. The algorithm is implemented using common synchronization primitives that are available in modern systems. The underlying data structure is fully consistent when it is idle, and the algorithm supports bidirectional traversals also through deleted nodes. The improved algorithm is compatible[5] with the lock-free memory reclamation scheme by Gidenstam et al. [20,21], and thus the data structure is fully dynamic in size and there is an upper bound of the amount of memory occupied at any time. The algorithm, which also allows disjoint accesses on the data structure to perform in parallel, is described in detail later in this paper together with the aspects concerning the underlying lock-free memory management.

We have performed an empirical study to measure the performance and scalability of our algorithm under high contention. Experiments were performed on two different multiprocessor systems equipped with 32 or 48 processors respectively, equipped with different operating systems and based on different architectures.

[2] Although the basic algorithm in the paper could support general doubly linked list operations, only operations related to the deque abstract data type were described, and only in the context of expensive reference counting methods for memory reclamation.

[3] When a traversal reaches a deleted node, the traversal must be restarted from one of the ends of the list.

[4] It is corrected when a traversal reaches the inconsistent part of the data structure, however with possibly severe time penalties.

[5] The algorithm in [14] contains recursive calls that made the use of hazard pointers infeasible as the maximum numbers of these would consequently be unbounded.

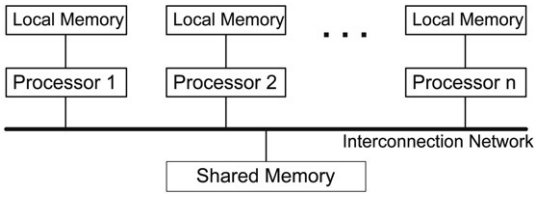

Fig. 1. Shared memory multiprocessor system structure.

The rest of the paper is organized as follows. In Section 2 we describe the type of systems that our implementation is aiming for. The actual algorithm is described in Section 3. In Section 4 we show the correctness of our algorithm 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 Fig. 1. Each node of the system contains a processor together with its local memory. All nodes are connected to the shared memory via an interconnection network. A set of co-operating tasks is running on the system performing their respective operations. Each task is sequentially executed on one of the processors, while each processor can serve (run) many tasks at a time. The co-operating tasks, possibly running on different processors, use shared data objects built in the shared memory to co-ordinate and communicate. Tasks synchronize their operations on the shared data objects through sub-operations on top of a cache-coherent shared memory. The shared memory may not though be uniformly accessible for all nodes in the system; processors can have different access times on different parts of the memory.

The shared memory system should support atomic read and write operations of single memory words, as well as stronger atomic primitives for synchronization. In this paper we use the Fetch-And-Add (FAA) and the Compare-And-Swap (CAS) atomic primitives; see Fig. 2 for a description. These read-modify-write style of operations are available on most common architectures or can be easily derived from other synchronization primitives [22, 23].

3. The new lock-free algorithm

The doubly linked list data structure is depicted in Fig. 3. In the data structure every node contains a value. The fields of each node item are described in Fig. 4 as it is used in this implementation. Note that the doubly linked list data structure always contains the static head and tail dummy nodes. The data structure is given an orientation by denoting the head side as being *left* and the tail side as being *right*, and we can consequently use this orientation to relate nodes as being to the left or right of each other. New nodes are created dynamically with the *CreateNode* function which in turn makes use of the *NewNode* function for the actual memory allocation, see Section 3.2.

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 and thus define the nodes positional relations in the logical abstraction of the doubly linked list, but the prev pointers only give hints as to where to find the previous node. These hints are useful in a deterministic manner

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

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

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

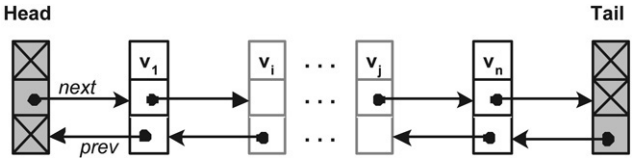

Fig. 3. The doubly linked list data structure.

thanks to the observation that a "late" non-updated prev pointer will always point to a node that is directly or some steps before the current node, and from that "hint" position it is always possible to traverse[6] through the next pointers to reach the logically 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 Fig. 5. There are several solutions to this problem, e.g., [24], and the latest method introduced by Harris [25] is to use a deletion mark. This deletion mark is updated atomically together with the next pointer, see the definition of the union *Link* in Fig. 4 where both the memory address and a boolean value are stored together within one single memory word. 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 our algorithm, the *SetMark* procedure is used for setting the deletion mark of an arbitrary pointer.

However, in our doubly linked list implementation, we never need to change both the prev and next pointers in one atomic update, and the pre-condition associated with each atomic pointer update only involves the pointer that is changed. Therefore it is possible to keep the prev and next pointers in separate words, duplicating the deletion mark in each of the words. In order to preserve the correctness of the algorithm, the deletion mark of the next pointer should always be set first, and the deletion mark of the

[6] As will be shown later, we have defined the doubly linked list 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
    ⟨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,lastlink: pointer to Node

function CreateNode(value: pointer to word):pointer to Node
CN1   node:=NewNode();
CN2   node.value:=value;
CN3   return node;

procedure SetMark(link: pointer to pointer to Node)
SM1   while true do
SM2       node:=*link;
SM3       if node.d = true or CAS(link
 ,node,⟨node.p,true⟩) then break;
```

Fig. 4. The node structure and auxiliary functions.

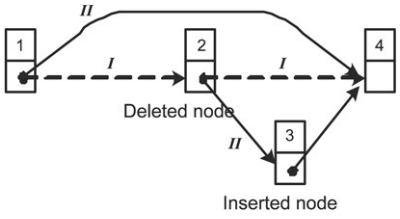

Fig. 5. Concurrent insert and delete operations can delete both nodes.

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 Fig. 6, for inserting a new node at an arbitrary position in our doubly linked list will thus be as follows: (I) After setting the new node's prev and next pointers, 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

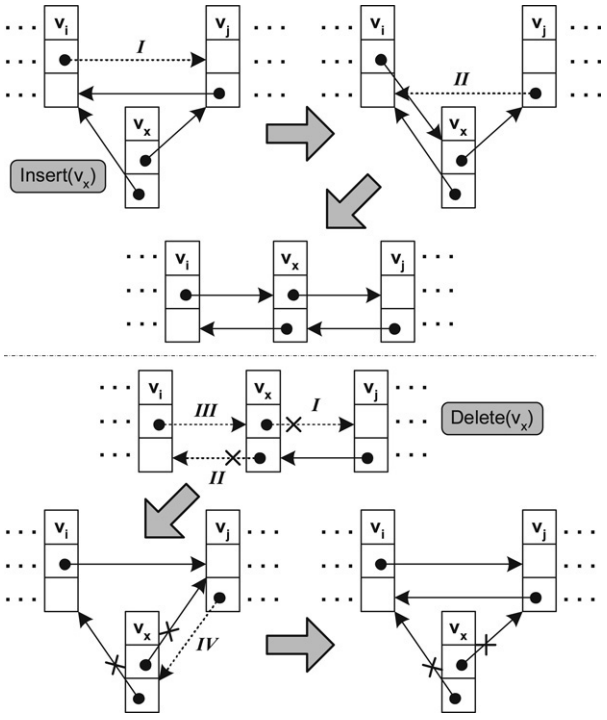

Fig. 6. Illustration of the basic steps of the algorithms for insertion and deletion of nodes at arbitrary positions in the doubly linked list, as described in Section 3.1.

prev pointer of the to-be-deleted node, (III) Atomically update the next pointer of the previous node of the to-be-deleted node, (IV) Atomically update the prev pointer of the next node of the to-be-deleted node. As will be shown later in the detailed description of the algorithm, helping techniques need to be applied in order to achieve the lock-free property, following the same steps as the main algorithm for inserting and deleting.

3.2. Memory management

As we are continuously dealing with pointers to memory areas which are concurrently allocated and recycled, we need a memory management which can guarantee the safety of dereferencing those pointers also in a concurrent environment. With *safe local reference* or *safe global reference* we denote the augmented pointers which are always possible to dereference for each individual thread or any thread respectively.

Our demands on the memory management consequently rules out the SMR or ROP methods by Michael [26,27] and Herlihy et al. [28] 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. [29], though based on the non-available CAS2 atomic primitive. Valois [7] and Michael and Scott [30] have presented a reference counting scheme based on the CAS atomic primitive. This scheme can only by used together with the corresponding scheme for memory allocation, and thus does not allow reuse of the reclaimed memory for arbitrary purposes. The SLFRC scheme by Herlihy et al. [31] improves the scheme by Detlefs et al. [29] to only require single-word atomic primitives by employing the corresponding ROP methods. An important drawback with all those reference counting schemes is their inability to reclaim cyclic garbage and that no upper bounds on the maximum memory usage can be established.

For our implementation, we have selected the lock-free memory management scheme proposed by Gidenstam et al. [21] which makes use of the CAS and FAA 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 and there also are no local references to it from pending concurrent operations. By supplying the scheme with appropriate callback functions, the scheme automatically breaks cyclic garbage as well as recursive chains of references among deleted nodes, and thus enables an upper bound on the maximum memory usage for the data structure.

The functionality defined by the memory management scheme is listed in Fig. 7 and are fully described in [21]. The callback procedures that need to be supplied to the memory management scheme are listed in Fig. 8. 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. Overall description of the detailed algorithm

The actual implementation of the needed operations of a Deque abstract data type or a general doubly linked list data structure, basically follows the steps of performing an insertion or deletion of a node at an arbitrary position as described in Section 3.1. For the *P*ushLeft vs. *P*ushRight or *P*opLeft vs. *P*opRight operations we need to first identify the target node(s) and then perform the respective steps of the overall insertion or deletion. However, at any time, due to concurrent modification (caused by overlapping operation invocations) of the underlying data structure, any of the steps might be impossible to perform as the pre-requisites have changed. In order to continue with the next steps, the cause of the problem first has to be determined and then the necessary of the remaining steps of the concurrent operation(s) are performed. A common problem of this sort appears when following a prev pointer, which has not been updated to match with the corresponding next pointer. The cause of this can be an overlapping operation in an intermediate step of performing either an insertion or deletion. For the purpose of resolving this problem, the function *CorrectPrev* has been defined, taking as arguments; (i) the node containing

```
function NewNode() :pointer to Node
procedure DeleteNode(node:pointer to Node)
function DeRefLink(address:pointer to Link) :pointer to Node
procedure ReleaseRef(node:pointer to Node)
function CASRef(address:pointer to Link, old:pointer to Node, new:pointer to Node) :boolean
procedure StoreRef(address:pointer to Link, node:pointer to Node)
```

Fig. 7. The functionality supported by the memory management scheme.

```
procedure TerminateNode(node: pointer to Node)
TN1   StoreRef(&node.prev,⊥);
TN2   StoreRef(&node.next,⊥);

procedure CleanUpNode(node: pointer to Node)
CU1   while true do
CU2     prev:=DeRefLink(&node.prev);
CU3     if prev.prev.d = false then break;
CU4     prev2:=DeRefLink(&prev.prev);
CU5     CASRef(&node.prev,⟨prev.p,true⟩,⟨prev2.p,true⟩);
CU6     ReleaseRef(prev2,prev);
CU7   while true do
CU8     next:=DeRefLink(&node.next);
CU9     if next.next.d = false then break;
CU10    next2:=DeRefLink(&next.next);
CU11    CASRef(&node.next,⟨next.p,true⟩,⟨next2.p,true⟩);
CU12    ReleaseRef(next2,next);
CU13  ReleaseRef(prev,next);
```

Fig. 8. Callback procedures for the memory management.

the prev pointer to be fixed, and (ii) an arbitrary "hint" node that is left of the first argument. The duties of this function are to fulfill the corresponding steps II or III–IV of the overlapping operation(s), until the prev pointer of the node in question is correct (or if the node is deleted when there no longer is any purpose of correcting) and then return this pointer as the function's result. Other operations can call this function whenever detecting a prev pointer that is not correct; typically after following a prev pointer and later detecting the mismatch of the corresponding next pointer, e.g., via a failed CAS sub-operation. The function can also be used for fulfilling the normal steps II or III–IV of the current operation, thus being a normal part of the execution also without interference from concurrent operation invocations. The actual implementation of the function is somewhat involved and is described in detail in Section 3.6.

3.4. Operations for the deque abstract data type

The *PushLeft* and the *PushRight* operations, are listed in Fig. 9. The *PopLeft* and *PopRight* operations, are listed in Fig. 10.

3.5. General operations for a lock-free doubly linked list

In this section we provide the details for the general operations of a lock-free doubly linked list, i.e., traversing the data structure in any direction and inserting and deleting nodes at arbitrary positions. For maintaining the current position we adopt the *cursor* concept by Valois [7], that is basically just a reference to a node in the list.

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

Definition 1. The *position* of a cursor that references a node that is present in the list is the referenced node. The position of a cursor that references a deleted node, is represented by the node that was logically next of the deleted node at the very moment of the deletion (i.e., the setting of the deletion mark). If that node is deleted as well, the position is equal to the position of a cursor referencing that node, and so on recursively. The actual position is then interpreted to be at an imaginary node logically previous of the representing node.

The *Next* function, see Fig. 11, tries to change the cursor to the next position relative to the current position, and returns the status of success. The algorithm repeatedly in line NT2–NT12 checks the next node for possible traversal until the found node is present and is not the tail dummy node. If the current node is the tail dummy node, false is returned in line NT2. In line NT3 the next pointer of the current node is de-referenced and in line NT4 the deletion state of the found node is read. If the found node is deleted and the current node was deleted when directly next of the found node, this is detected in line NT5 and then the position is updated according to Definition 1 in line NT11. If the found node was detected as present in line NT5, the cursor is set to the found node in line NT11 and true is returned (unless the found node is the tail dummy node when instead false is returned) in line NT12. Otherwise it fulfils the deletion in lines NT6–NT7 after which the algorithm retries at line NT2.

The *Prev* function, see Fig. 11, tries to change the cursor to the previous position relative to the current position, and returns the status of success. The algorithm repeatedly in line PV2–PV13 checks the next node for possible traversal until the found node is present and is not the head dummy node. If the current node is the head dummy node, false is returned in line PV2. In line PV3 the prev pointer of the current node is de-referenced. If the found node is logically previous of the current node and the current node is present, this is detected in line PV4 and then the cursor is set to the found node in line PV6 and true is returned (unless the found node is the head dummy node when instead false is returned) in line PV7. If the current node is deleted then the cursor position is updated according to Definition 1 by calling the *Next* function in line PV10. Otherwise the prev pointer of the current node is updated by calling the *CorrectPrev* function in line PV12 after which the algorithm retries at line PV2.

The *Read* function, see Fig. 12, returns the current value of the node referenced by the cursor, unless this node is deleted or the node is equal to any of the dummy nodes when the function instead returns a non-value. In line RD1 the algorithm checks if the node referenced by the cursor is either the head or tail dummy node, and then returns a non-value. The value of the node is read in line RD2, and in line RD3 it is checked if the node is deleted and then returns a non-value, otherwise the value is returned in line RD4.

The *InsertBefore* operation, see Fig. 13, inserts a new node directly before the node positioned by the given cursor and later changes the cursor to position the inserted node. If the node positioned by the cursor is the head dummy node, the new node will be inserted directly after the head dummy node. The algorithm checks in line IB1 if the cursor position is equal to the head dummy node, and consequently then calls the *InsertAfter* operation to insert the new node directly after the head dummy node. The algorithm repeatedly tries in lines IB5–IB13 to insert the new node (*node*) between the previous node (*prev*) of the cursor and the cursor positioned node, by atomically changing the next pointer of the prev node to instead point to the new node. If the node positioned by the cursor is deleted this is detected in line IB5 and the cursor is updated by calling the *Next* function. If the update of the next pointer of the prev node by using the CAS operation in line IB11 fails, this is because either the prev node is no longer the logically previous node of the cursor positioned node, or that the cursor positioned node is deleted. If the prev node is no longer the logically previous node this is detected in line IB12 and then the

```
procedure PushLeft(value: pointer to word)
L1    node:=CreateNode(value);
L2    prev:=DeRefLink(&head);
L3    next:=DeRefLink(&prev.next);
L4    while true do
L5      StoreRef(&node.prev,⟨prev,false⟩);
L6      StoreRef(&node.next,⟨next,false⟩);
L7      if CASRef(&prev.next,⟨next,false⟩
      ,⟨node,false⟩) then break;
L8      ReleaseRef(next);
L9      next:=DeRefLink(&prev.next);
L10     Back-Off
L11   PushEnd(node,next);
procedure PushEnd(node, next: pointer to Node)
P1    while true do
P2      link1:=next.prev;
P3      if link1.d = true or node.next ≠
      ⟨next,false⟩ then break;
P4      if CASRef(&next.prev,link1,⟨node,false⟩) then
```

```
procedure PushRight(value: pointer to word)
R1    node:=CreateNode(value);
R2    next:= DeRefLink(&tail);
R3    prev:=DeRefLink(&next.prev);
R4    while true do
R5      StoreRef(&node.prev,⟨prev,false⟩);
R6      StoreRef(&node.next,⟨next,false⟩);
R7      if CASRef(&prev.next,⟨next,false⟩
      ,⟨node,false⟩) then break;
R8      prev:=CorrectPrev(prev,next);
R9      Back-Off
R10   PushEnd(node,next);

P5      if node.prev.d = true then
P6        node:=CorrectPrev(node,next);
P7      break;
P8    Back-Off
P9    ReleaseRef(next, node);
```

Fig. 9. The algorithm for the PushLeft and PushRight operations.

CorrectPrev function is called in order to update the prev pointer of the cursor positioned node. If the update using CAS in line IB11 succeeds, the cursor position is set to the new node in line IB14 and the prev pointer of the previous cursor positioned node is updated by calling the *CorrectPrev* function in line IB15.

The *InsertAfter* operation, see Fig. 13, inserts a new node directly after the node positioned by the given cursor and later changes the cursor to position the inserted node. If the node positioned by the cursor is the tail dummy node, the new node will be inserted directly before the tail dummy node. The algorithm checks in line IA1 if the cursor position is equal to the tail dummy node, and consequently then calls the *InsertBefore* operation to insert the new node directly after the head dummy node. The algorithm repeatedly tries in lines IA5–IA13 to insert the new node (*node*) between the cursor positioned node and the next node (*next*) of the cursor, by atomically changing the next pointer of the cursor positioned node to instead point to the new node. If the update of the next pointer of the cursor positioned node by using the CAS operation in line IA8 fails, this is because either the next node is no longer the logically next node of the cursor positioned node, or that the cursor positioned node is deleted. If the cursor positioned node is deleted, the operation to insert directly after the cursor position now becomes the problem of inserting directly before the node that represents the cursor position according to Definition 1. It is detected in line IA10 if the cursor positioned node is deleted and then it calls the InsertBefore operation in line IA12. If the update using CAS in line IA8 succeeds, the cursor position is set to the new node in line IA14 and the prev pointer of the previous cursor positioned node is updated by calling the *CorrectPrev* function in line IA15.

The *Delete* operation, see Fig. 14, tries to delete the non-dummy node referenced by the given cursor and returns the value if successful, otherwise a non-value is returned. If the cursor positioned node is equal to any of the dummy nodes this is detected in line D2 and a non-value is returned. The algorithm repeatedly tries in line D4–D8 to set the deletion mark of the next pointer of the cursor positioned node. If the deletion mark is already set, this is detected in line D5 and a non-value is returned. If the CAS operation in line D8 succeeds, the deletion process is completed by setting the mark on the prev pointer in line D9–D11 and calling the *CorrectPrev* function in line D12. The value of the deleted node is read in line D14 and the value returned in line D16.

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

3.6. Helping and back-off

The *CorrectPrev* sub-function, see Fig. 15, tries to update the prev pointer of a node and then return a reference to a possibly logically previous node, thus fulfilling step II of the overall insertion scheme or steps III–IV of the overall deletion scheme as described in Section 3.1. The algorithm repeatedly tries in lines CP3–CP25 to correct the prev pointer of the given node (*node*), given a suggestion of a previous (not necessarily the logically previous) node (*prev*). Before trying to update the prev pointer with the CAS operation in line CP22, it assures in line CP5 that the *prev* node is not marked, in line CP3 that *node* is not marked, and in line CP16 that *prev* is the previous node of *node*. If *prev* is marked we need to delete it before we can update *prev* to one of its previous nodes and proceed with the current operation. If *lastlink* is valid and thus references the node previous of the current *prev* node, the next pointer of the *lastlink* node is updated with CAS in line CP8. Otherwise *prev* is updated to be the prev pointer of the *prev* node

```
function PopLeft(): pointer to word
PL1    prev:=DeRefLink(&head);
PL2    while true do
PL3      node:=DeRefLink(&prev.next);
PL4      if node = tail then
PL5        ReleaseRef(node, prev);
PL6        return ⊥;
PL7      next:= DeRefLink(&node.next);
PL8      if next.d = true then
PL9        SetMark(&node.prev);
PL10       CASRef(&prev.next,node,⟨next.p,false⟩);
PL11       ReleaseRef(next.p, node);
PL12       continue;
PL13     if CASRef(&node.next,next
 ,⟨next.p,true⟩) then
PL14       prev:=CorrectPrev(prev,next);
PL15       ReleaseRef(prev);
PL16       value:=node.value;
PL17       break;
PL18     ReleaseRef(next, node);
PL19     Back-Off
PL20   ReleaseRef(next, node);
PL21   return value;
```

```
function PopRight(): pointer to word
PR1    next:=DeRefLink(&tail);
PR2    node:=DeRefLink(&next.prev);
PR3    while true do
PR4      if node.next ≠ ⟨next,false⟩ then
PR5        node:=CorrectPrev(node,next);
PR6        continue;
PR7      if node = head then
PR8        ReleaseRef(node, next);
PR10       return ⊥;
PR11     if CASRef(&node.next,⟨next, false⟩
 ,⟨next,true⟩) then
PR13       prev:=DeRefLink(&node.prev);
PR14       prev:=CorrectPrev(prev,next);
PR15       ReleaseRef(prev, next);
PR17       value:=node.value;
PR18       break;
PR19     Back-Off
PR21   ReleaseRef(node);
PR22   return value;
```

Fig. 10. The algorithm for the PopLeft and PopRight operations.

```
function Next(cursor: pointer to pointer to
 Node): boolean
NT1    while true do
NT2      if *cursor = tail then return false;
NT3      next:=DeRefLink(&(*cursor).next);
NT4      d := next.next.d;
NT5      if d = true and (*cursor).next
 ≠ ⟨next,true⟩ then
NT6        SetMark(&next.prev);
NT7        CASRef(&(*cursor).next
 ,next,next.next.p);
NT8        ReleaseRef(next);
NT9        continue;
NT10     ReleaseRef(*cursor);
NT11     *cursor:=next;
NT12     if d = false and next ≠ tail then return true;
```

```
function Prev(cursor: pointer to pointer to
 Node): boolean
PV1    while true do
PV2      if *cursor = head then return false;
PV3      prev:=DeRefLink(&(*cursor).prev);
PV4      if prev.next = ⟨*cursor,false⟩ and
 (*cursor).next.d = false then
PV5        ReleaseRef(*cursor);
PV6        *cursor:=prev;
PV7        if prev ≠ head then return true;
PV8      else if (*cursor).next.d = true then
PV9        ReleaseRef(prev);
PV10       Next(cursor);
PV11     else
PV12       prev:=CorrectPrev(prev,*cursor);
PV13       ReleaseRef(prev);
```

Fig. 11. The algorithms for the Next and Prev operations.

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

Fig. 12. The algorithm for the Read function.

in lines CP12–CP14. 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 in lines CP17–CP19. If the update in line CP22 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 *Delete* operation) directly before the CAS operation. This is detected in line CP23 and then the update is possibly retried with a new *prev* node.

Because *CorrectPrev* 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.

4. Correctness proof

In this section we sketch the correctness proof of our algorithm. We prove that our algorithm is a linearizable one [32] and that it is lock-free. A set of definitions that will help us to structure and shorten the proof is first described in this section.

Definition 2. 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:

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

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

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

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

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

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

Definition 3. We denote with L_t the abstract internal state of a doubly linked list at the time t. L_t is viewed as a set of nodes n, where each node n_i has a value v_i and the relations *Del*(n_i) and *Next*(n_i, n_j), and where $n_{head}, n_{tail} \in L_t$. For positioning there

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

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

Fig. 13. The algorithms for the InsertBefore and InsertAfter operations.

```
function Delete(cursor: pointer to
 pointer to Node): pointer to word
D1    node:=*cursor;
D2    if node = head or node = tail then return ⊥;
D3    while true do
D4      next:=DeRefLink(&(*cursor).next);
D5      if next.d = true then
D6        ReleaseRef(next);
D7        return ⊥;
D8    if CAS(&node.next,next,⟨next.p,true⟩) then
D9      while true do
D10       prev:=DeRefLink(&node.prev);
D11       if prev.d = true or CAS(
 &node.prev,prev,⟨prev.p,true⟩) then break;
D12     prev:=CorrectPrev(prev.p,next);
D13     ReleaseRef(prev); ReleaseRef(next);
D14     value:=node.value;
D15     ReleaseRef(node); DeleteNode(node);
D16     return value;
```

Fig. 14. The algorithm for the Delete function.

```
function CorrectPrev(prev, node: pointer
 to Node):pointer to Node
CP1   lastlink:=⊥;
CP2   while true do
CP3     link1:=node.prev; if link1.d = true
 then break;
CP4     prev2:=DeRefLink(&prev.next);
CP5     if prev2.d = true then
CP6       if lastlink ≠ ⊥ then
CP7         SetMark(&prev.prev);
CP8         CASRef(&lastlink.next,prev,⟨prev2.p,false⟩);
CP9         ReleaseRef(prev2.p, prev);
CP10        prev:=lastlink; lastlink:=⊥;
CP11        continue;
CP12      ReleaseRef(prev2.p);
CP13      prev2:=DeRefLink(&prev.prev);
CP14      ReleaseRef(prev); prev:=prev2;
CP15      continue;
CP16    if prev2 ≠ node then
CP17      if lastlink ≠ ⊥ then ReleaseRef(lastlink);
CP18      lastlink:=prev;
CP19      prev:=prev2;
CP20      continue;
CP21    ReleaseRef(prev2);
CP22    if CASRef(&node.prev,link1,⟨prev,false⟩) then
CP23      if prev.prev.d = true then continue;
CP24      break;
CP25    Back-Off
CP26  if lastlink ≠ ⊥ then ReleaseRef(lastlink);
CP27  return prev;
```

Fig. 15. The algorithm for the CorrectPrev function.

is a function $Pos(c) = n$ which maps a cursor c with the corresponding node n according to Definition 1. The operations that can be performed on the doubly linked list are *Next*(*NT*), *Prev*(*PV*), *Read*(*RD*), *InsertBefore*(*IB*),*InsertAfter*(*IA*) and *Delete*(*DE*). The following expressions define the sequential semantics of our operations, with same syntax as in Definition 2:

$$Pos(c_{t_1}) = n_{tail} : \mathbf{NT}(\mathbf{c_{t_1}}) = \mathbf{F}, \qquad c_{t_2} = n_{tail} \tag{7}$$

$$Next(n_i, n_{tail}) \wedge Pos(c_{t_1}) = n_i : \mathbf{NT}(\mathbf{c_{t_1}}) = \mathbf{F}, \quad c_{t_2} = n_{tail} \tag{8}$$

$$\exists n_j \in L_{t_1}.Next(n_i, n_j) \wedge c_{t_1} = n_i : \mathbf{NT}(\mathbf{c_{t_1}}) = \mathbf{T}, \\ c_{t_2} = Pos(n_j) \wedge \neg Del(n_j) \tag{9}$$

$$Pos(c_{t_1}) = n_{head} : \mathbf{PV}(\mathbf{c_{t_1}}) = \mathbf{F}, \qquad c_{t_2} = n_{head} \tag{10}$$

$$Next(n_{head}, n_i) \wedge Pos(c_{t_1}) = n_i : \mathbf{PV}(\mathbf{c_{t_1}}) = \mathbf{F}, \quad c_{t_2} = n_{head} \tag{11}$$

$$\exists n_j \in L_{t_1}.Next(n_j, n_i) \wedge \neg Del(n_j) \wedge Pos(c_{t_1}) \\ = n_i : \mathbf{PV}(\mathbf{c_{t_1}}) = \mathbf{T}, \quad c_{t_2} = n_j \tag{12}$$

$$c_{t_1} = n_{head} \vee c_{t_1} = n_{tail} : \mathbf{RD}(\mathbf{c_{t_1}}) = \bot, \qquad c_{t_2} = c_{t_1} \tag{13}$$

$$c_{t_1} = n_i \wedge Del(n_i) : \mathbf{RD}(\mathbf{c_{t_1}}) = \bot, \qquad c_{t_2} = c_{t_1} \tag{14}$$

$$c_{t_1} = n_i \wedge n_i \mapsto v_i : \mathbf{RD}(\mathbf{c_{t_1}}) = \mathbf{v_i}, \qquad c_{t_2} = c_{t_1} \tag{15}$$

$$\exists n_j \in L_{t_1}.Next(n_{head}, n_j) \wedge Pos(c_{t_1}) = n_{head} : \mathbf{IB}(\mathbf{c_{t_1,v_k}}),$$

$$c_{t_2} = n_k \wedge n_k \in L_{t_2} \wedge Next(n_{head}, n_k) \wedge Next(n_k, n_j) \wedge n_k \mapsto v_k \tag{16}$$

$$\exists n_j \in L_{t_1}.Next(n_j, n_i) \wedge Pos(c_{t_1}) = n_i : \mathbf{IB}(\mathbf{c_{t_1,v_k}}),$$

$$c_{t_2} = n_k \wedge n_k \in L_{t_2} \wedge Next(n_j, n_k) \wedge Next(n_k, n_i) \wedge n_k \mapsto v_k \tag{17}$$

$$\exists n_j \in L_{t_1}.Next(n_j, n_{tail}) \wedge Pos(c_{t_1}) = n_{tail} : \mathbf{IA}(\mathbf{c_{t_1,v_k}}),$$

$$c_{t_2} = n_k \wedge n_k \in L_{t_2} \wedge Next(n_j, n_k) \wedge Next(n_k, n_{tail}) \wedge n_k \mapsto v_k \tag{18}$$

$$\exists n_j \in L_{t_1}.Next(n_i, n_j) \wedge Pos(c_{t_1}) = n_i \wedge \neg Del(n_i) : \mathbf{IA}(\mathbf{c_{t_1,v_k}}),$$

$$c_{t_2} = n_k \wedge n_k \in L_{t_2} \wedge Next(n_i, n_k) \wedge Next(n_k, n_j) \wedge n_k \mapsto v_k \tag{19}$$

$$c_{t_1} = n_{head} \vee c_{t_1} = n_{tail} : \mathbf{DE}(\mathbf{c_{t_1}}) = \bot, \qquad c_{t_2} = c_{t_1} \tag{20}$$

$$c_{t_1} = n_i \wedge Del(n_i) : \mathbf{DE}(\mathbf{c_{t_1}}) = \bot, \qquad c_{t_2} = c_{t_1} \tag{21}$$

$$c_{t_1} = n_i \wedge n_i \mapsto v_i : \mathbf{DE}(\mathbf{c_{t_1}}) = \mathbf{v_i}, \qquad c_{t_2} = c_{t_1} \wedge Del(n_i). \tag{22}$$

Definition 4. In order for an implementation of a shared concurrent data object to be *linearizable* [32], 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.

Note that the following linearizability points are described in the context of either the operations for the deque abstract data type or the general doubly linked data structure, not both.

Definition 5. The linearizability points for the deque abstract data type are defined as follows, with the corresponding equation number within parenthesis:

- The *PushLeft* operation (1) – the successful CAS operation in line L7.
- The *PushRight* operation (2) – the successful CAS operation in line R7.
- A *PopLeft* operation that fails (3) – the read operation of the next pointer in line PL3.
- A *PopLeft* operation that succeeds (4) – the read operation of the next pointer in line PL3.
- A *PopRight* operation that fails (5) – the read operation of the next pointer in line PR4.
- A *PopRight* operation that succeeds (6) – the CAS sub-operation in line PR11.

In the proofs of the following lemmas we will use the fact that if a deletion mark of a node is detected not to be set, it can be safely interpreted that the state also holds in earlier statements in the execution history. We will also use the fact that the cursors are local, and thus can the state changes of these be interpreted to have taken effect at an arbitrary step within the invocation of the operation.

Definition 6. The linearizability points for the general doubly linked list data structure are defined as follows, with the corresponding equation number within parenthesis:

- A *Next* function that succeeds (9) – the read sub-operation of the next pointer in line NT3.
- A *Next* function that fails (7) and (8) – line NT2 if the node positioned by the original cursor was the tail dummy node, and the read sub-operation of the next pointer in line NT3 otherwise.
- A *Prev* function that succeeds (12) – the read sub-operation of the prev pointer in line PV3.
- A *Prev* function that fails (10) and (11) – line PV2 if the node positioned by the original cursor was the head dummy node, and the read sub-operation of the prev pointer in line PV3 otherwise.
- A *Read* function that returns a value (14) – the read sub-operation of the next pointer in line RD3.
- A *Read* function that returns a non-value (13) – the read sub-operation of the next pointer in line RD3, unless the node positioned by the cursor was the head or tail dummy node when the linearizability point is line RD1.
- The *InsertBefore* operation – the successful CAS operation in line IB11, or equal to the linearizability point of the *InsertAfter* operation if that operation was called in line IB1.
- The *InsertAfter* operation – the successful CAS operation in line IA8, or equal to the linearizability point of the *InsertBefore* operation if that operation was called in line IA1 or IA12.
- A *Delete* function that returns a value (21) – the successful CAS operation in line D8.
- A *Delete* function that returns a non-value (20) – the read sub-operation of the next pointer in line D4, unless the node positioned by the cursor was the head or tail dummy node when the linearizability point instead is line D2.

We will now try to prove the lock-free property by first showing that any operation invocation will terminate if ignoring the interference caused by concurrent operation invocations, and then showing that for every possible concurrent change at least one operation will make progress towards termination.

Lemma 1. *The path of prev pointers from a node is always pointing a present node that is to the left of the current node.*

Proof. We will look at all possibilities where the prev pointer is set or changed. Firstly, nodes are never moved relatively to each other. The setting in line IB9 is clearly to the left as it is verified by IB11. The setting in line IA6 is clearly to the left as it is verified by IA8. The change of the prev pointer in line CP22 is to the left as verified by line CP4 and CP16. Finally, the change of the prev pointer in line CU5 is to the left as it is changed to the prev pointer of the previous node. □

Lemma 2. *Any operation invocation will terminate if exposed to a limited number of concurrent changes to the underlying data structure.*

Proof. We assume that the number of changes to the underlying data structure an operation could experience is limited. Because of the reference counting, none of the nodes which are referenced to by local variables can be garbage collected. When traversing through prev or next pointers, the memory management guarantees atomicity of the operations, thus no newly inserted or deleted nodes will be missed. We also know that the relative positions of nodes that are referenced to by local variables will not change as nodes are never moved in the doubly linked list. 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 IB5–IB13, IA5–IA13 and D4–D8.

The loop CP3–CP25 will terminate if either the to-be-corrected node is marked in line CP3 or if the CAS sub-operation in line CP22 succeeds and prev node is not marked. We know that from the start of the execution of the loop, the prev node is left of the to-be-corrected node. Following from Lemma 1 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 logically previous of the to-be-corrected node if this is not deleted (and the CAS sub-operation in line CP22 will finally succeed), otherwise line CP3 will succeed. As long as the prev node is marked it will be traversed to the left in line CP6–CP15. 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 CU1–CU14 will terminate if both the prev node and the next node of the to-be-deleted node is not marked in line CU3, respectively line CU9. 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 1, 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 3. *With respect to the retries caused by synchronization, one operation will always 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 *InsertBefore*, *InsertAfter* or *Delete* operation will progress. Consequently, independent of any number of concurrent operations, one operation will always progress. □

Theorem 1. *The algorithm implements a lock-free and linearizable deque abstract data type and a lock-free and linearizable general doubly linked list data structure.*

Proof. By using the respective linearizability points as defined in Definitions 5 and 6 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 4, the implementation is therefore linearizable. Lemmas 2 and 3 give that our implementation is lock-free. □

5. Experimental evaluation

In this section we evaluate the performance of our implementations by the means of some custom micro-benchmarks. The purpose is to estimate how well the implementations scale with increasing number of threads under high contention.

In our experiments, each concurrent thread performed a given number of randomly chosen sequential operations on the particular data structure. The experiments were performed using a different number of threads, varying from 1 to a chosen limit with increasing steps. Each experiment was repeated several times, and an average execution time for each experiment was estimated. Exactly the same sequence of operations was performed for all different implementations compared. A clean-cache operation was performed just before each sub-experiment. All implementations are written in C and compiled with the highest optimization level. The atomic primitives are written in assembly.

In an ideal parallel system the experiment's execution time should scale with a time complexity of a constant with respect to the number of threads. However, under high contention the shared memory acts like a bottleneck, which relaxes down to the memory's bandwidth with increasing number of disjoint accesses. Thus, the expected time complexity under full contention without disjoint accesses is linear at best. As a reference in this context are lock-free implementations in the literature using non-greedy helping techniques known to scale worse, as helping can cause each thread to perform all the other thread's operation steps as well as its own.

5.1. General doubly linked list experiment

In this experiment, each thread performed 50 000 operations from a distribution of 30% traversal, 10% *InsertBefore*, 10% *InsertAfter* and 50% *Delete* operations. The traversal operations are alternately changing between the *Next* versus *Prev* operation whenever any of the ends of the list are reached. The purpose of the chosen distribution was to simulate a worst-case scenario in the context of contention (i.e., the list size is constantly kept as short as possible).

Two different platforms were used, with varying number of processors and level of shared memory distribution. We performed our experiments on a Sun Fire 15 K system running Solaris 9 with 48 Ultrasparc III processors of 900 MHz each, and also on an IBM p690+ Regatta system running AIX with 32 processors of 1.7 GHz each. The results from the experiments with up to 32 threads are shown in Fig. 16. The average execution time is drawn as a function of the number of threads.

The results show that our new algorithm scales linearly with increasing number of threads, consequently from the nature of our algorithm to use greedy helping techniques with a successful back-off strategy. We expect our implementation to scale better with decreasing contention (i.e., longer lists) thanks to the nature of our algorithm to support parallelism for disjoint accesses.

5.2. Deque experiment

In this experiment, each thread performed 1000[7] operations from a distribution of 1/4 *PushRight*, 1/4 *PushLeft*, 1/4 *PopRight* and 1/4 *PopLeft* operations. Besides our implementation, we also performed the same experiment with the lock-free implementation by Michael [12] and the implementation by Martin et al. [11], 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. [29]. However, as both [11,29] 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 [11]). The other approach was to implement CAS2 using one of the most efficient software implementations of CASN known that could meet the needs of [11,29], i.e., the implementation by Harris et al. [33].

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 with up to 32 threads are shown in Fig. 17. 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, [12] has the best performance. However, for the system with full concurrency and non-uniform memory architecture our algorithm performs significantly better than [12] from 2 threads and more, as a direct consequence of the nature of our algorithm to support parallelism for disjoint accesses.

[7] Even with as few as 1000 operations per thread some implementations took over 20 s per experiment. To compensate for accuracy, each experiment was repeated 50 times.

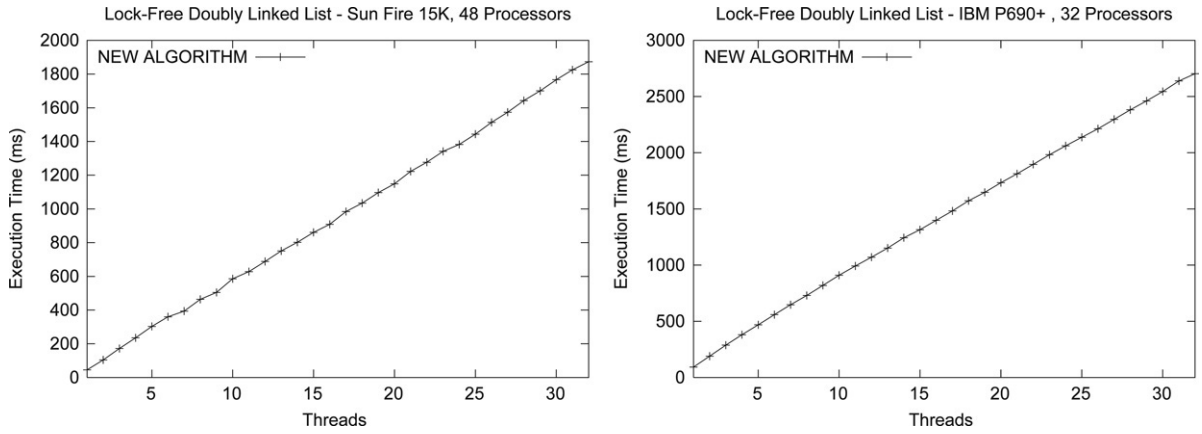

Fig. 16. Experiment with doubly linked lists and high contention.

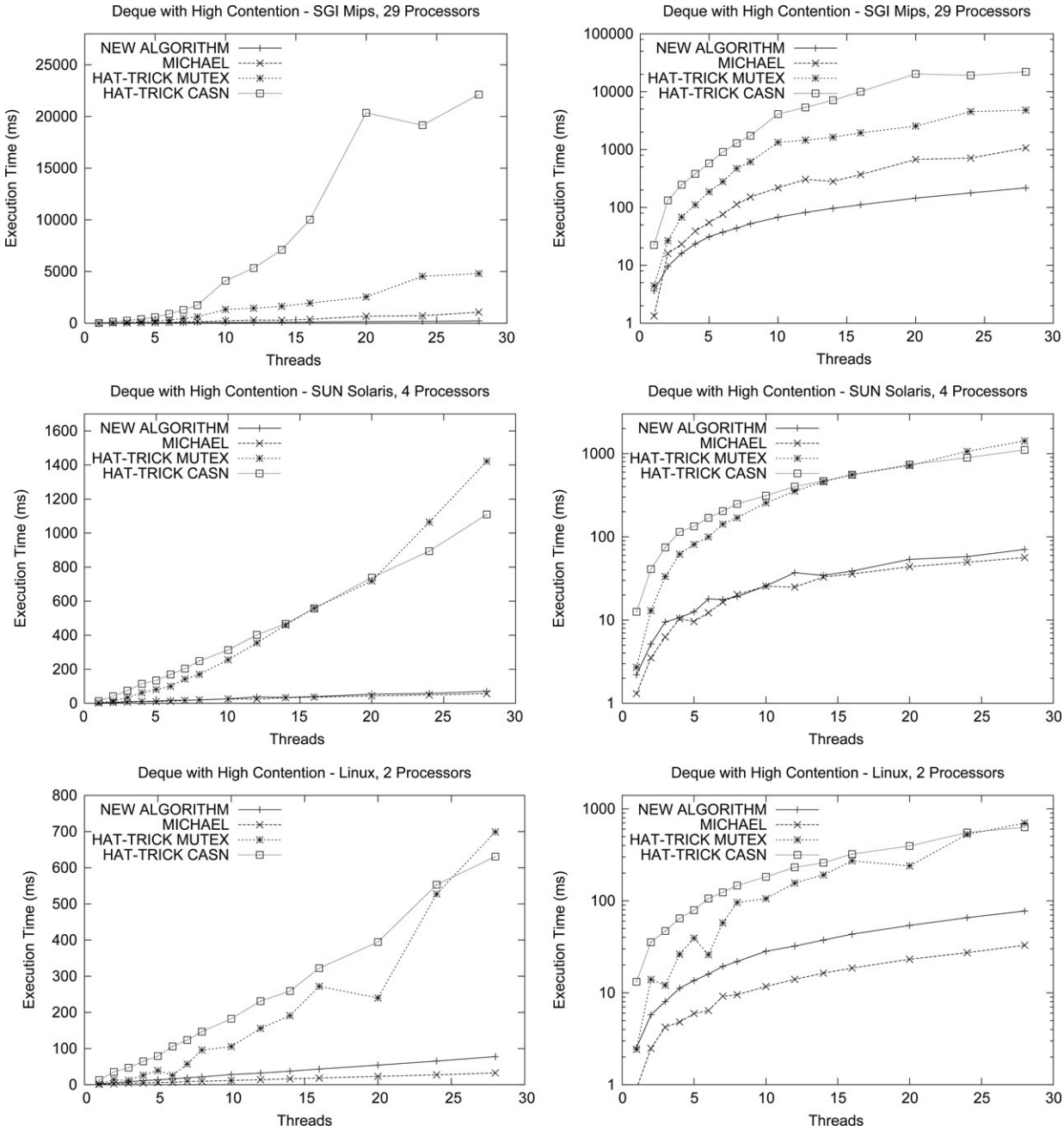

Fig. 17. Experiment with deques and high contention. Logarithmic scales in the right column.

6. Conclusions

We have presented the first lock-free algorithmic implementation of a concurrent doubly linked list 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, (iv) allows pointers with full precision to be used and thus supports fully dynamic list sizes, and (v) supports deterministic and well defined traversals also through deleted nodes.

We have performed an empirical study of our new algorithm on two different multiprocessor platforms. Results of the experiments performed under high contention show that the performance of our implementation scales linearly with increasing number of processors. We have also 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 [34] library.

References

[1] A. Silberschatz, P. Galvin, Operating System Concepts, Addison Wesley, 1994.
[2] M. Herlihy, Wait-free synchronization, ACM Transactions on Programming Languages and Systems 11 (1) (1991) 124–149.
[3] M. Herlihy, V. Luchangco, M. Moir, Obstruction-free synchronization: Double-ended queues as an example, in: Proceedings of the 23rd International Conference on Distributed Computing Systems, 2003.
[4] M. Greenwald, Two-handed emulation: How to build non-blocking implementations of complex data-structures using DCAS, in: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing, ACM Press, 2002, pp. 260–269.
[5] A. Israeli, L. Rappoport, Disjoint-access-parallel implementations of strong shared memory primitives, in: Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, 1994.
[6] H. Attiya, E. Hillel, Built-in coloring for highly-concurrent doubly-linked lists, in: Proceedings of the 20th International Symposium on Distributed Computing, 2006, pp. 31–45.
[7] J.D. Valois, Lock-free data structures, Ph.D. Thesis, Rensselaer Polytechnic Institute, Troy, New York, 1995.
[8] M. Greenwald, Non-blocking synchronization and system design, Ph.D. Thesis, Stanford University, Palo Alto, CA, 1999.
[9] O. Agesen, D. Detlefs, C.H. Flood, A. Garthwaite, P. Martin, N. Shavit, G.L. Steele Jr., DCAS-based concurrent deques, in: ACM Symposium on Parallel Algorithms and Architectures, 2000, pp. 137–146.
[10] D. Detlefs, C.H. Flood, A. Garthwaite, P. Martin, N. Shavit, G.L. Steele Jr., Even better DCAS-based concurrent deques, in: International Symposium on Distributed Computing, 2000, pp. 59–73.
[11] P. Martin, M. Moir, G. Steele, DCAS-based concurrent deques supporting bulk allocation, Tech. Rep. TR-2002-111, Sun Microsystems, 2002.
[12] M.M. Michael, CAS-based lock-free algorithm for shared deques, in: Proceedings of the 9th International Euro-Par Conference, in: Lecture Notes in Computer Science, Springer Verlag, 2003.
[13] H. Sundell, P. Tsigas, Lock-free and practical deques using single-word compare-and-swap, Tech. Rep. 2004-02, Computing Science, Chalmers University of Technology, March 2004.
[14] H. Sundell, P. Tsigas, Lock-free and practical doubly linked list-based deques using single-word compare-and-swap, in: Proceedings of the 8th International Conference on Principles of Distributed Systems, in: LNCS, vol. 3544, Springer Verlag, 2004, pp. 240–255.
[15] P. Martin, A practical lock-free doubly-linked list, December 2004, personal communication.
[16] D. Lea, The java.util.concurrent synchronizer framework, Science of Computer Programming 58 (3) (2005) 293–309.
[17] E. Ladan-Mozes, N. Shavit, An optimistic approach to lock-free fifo queues, in: Proc. of the 18th Annual Conference on Distributed Computing, DISC 2004, 2004, pp. 117–131.
[18] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W.N. Scherer, N. Shavit, A lazy concurrent list-based set algorithm, in: Proceedings of the 9th International Conference on Principles of Distributed Systems, 2005.
[19] H. Sundell, P. Tsigas, Scalable and lock-free concurrent dictionaries, in: Proceedings of the 19th ACM Symposium on Applied Computing, ACM press, 2004, pp. 1438–1445.
[20] A. Gidenstam, M. Papatriantafilou, H. Sundell, P. Tsigas, Practical and efficient lock-free garbage collection based on reference counting, Tech. Rep. 2005-04, Computing Science, Chalmers Univ. of Tech., 2005.
[21] A. Gidenstam, M. Papatriantafilou, H. Sundell, P. Tsigas, Efficient and reliable lock-free memory reclamation based on reference counting, in: Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks, IEEE, 2005.
[22] M. Moir, Practical implementations of non-blocking synchronization primitives, in: Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing, 1997.
[23] P. Jayanti, A complete and constant time wait-free implementation of cas from ll/sc and vice versa, in: DISC, vol. 1998, 1998, pp. 216–230.
[24] J.D. Valois, Lock-free linked lists using compare-and-swap, in: Proceedings of the 14th Annual Principles of Distributed Computing, 1995, pp. 214–222.
[25] T.L. Harris, A pragmatic implementation of non-blocking linked lists, in: Proceedings of the 15th International Symposium of Distributed Computing, 2001, pp. 300–314.
[26] M.M. Michael, Safe memory reclamation for dynamic lock-free objects using atomic reads and writes, in: Proceedings of the 21st ACM Symposium on Principles of Distributed Computing, 2002, pp. 21–30.
[27] M.M. Michael, Hazard pointers: Safe memory reclamation for lock-free objects, IEEE Transactions on Parallel and Distributed Systems 15 (8).
[28] M. Herlihy, V. Luchangco, M. Moir, The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structure, in: Proceedings of 16th International Symposium on Distributed Computing, 2002.
[29] D. Detlefs, P. Martin, M. Moir, G. Steele Jr., Lock-free reference counting, in: Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing, 2001.
[30] M.M. Michael, M.L. Scott, Correction of a memory management method for lock-free data structures, Tech. rep., Computer Science Department, University of Rochester, 1995.
[31] M. Herlihy, V. Luchangco, P. Martin, M. Moir, Dynamic-sized lock-free data structures, in: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing, ACM, 2002, 131–131.
[32] M. Herlihy, J. Wing, Linearizability: A correctness condition for concurrent objects, ACM Transactions on Programming Languages and Systems 12 (3) (1990) 463–492.
[33] T. Harris, K. Fraser, I. Pratt, A practical multi-word compare-and-swap operation, in: Proceedings of the 16th International Symposium on Distributed Computing, 2002.
[34] H. Sundell, 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, 2002.

Håkan Sundell main research interests are in efficient shared data structures for practical multi-thread programming on multiprocessor and multi-core computers. He is known as a pioneer in the Swedish IT-history with roots in the early 1980's emerging programming community. He received a M.Sc. degree in computer science in 1996 at Göteborg University. Between the years 1995–1999 he worked as a senior consultant and systems programmer within the telecommunication and multimedia industry. In 2004 he received a Ph.D. degree in computer science at Chalmers University of Technology. At present he is a senior lecturer at the School of Business and Informatics, University College of Borås, Sweden.

Philippas Tsigas research interests include concurrent data structures for multiprocessor systems, communication and coordination in parallel systems, fault-tolerant computing, mobile computing. He received a B.Sc. in Mathematics from the University of Patras, Greece and a Ph.D. in Computer Engineering and Informatics from the same University. Philippas was at the National Research Institute for Mathematics and Computer Science, Amsterdam, the Netherlands (CWI), and at the Max-Planck Institute for Computer Science, Saarbrücken, Germany, before. At present he is an associate professor at the Department of Computing Science at Chalmers University of Technology, Sweden (www.cs.chalmers.se/~tsigas).