Even Better DCAS-Based Concurrent Deques

David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite,
Paul A. Martin, Nir N. Shavit, and Guy L. Steele Jr.

Sun Microsystems Laboratories, 1 Network Drive, Burlington, MA 01803 USA

Abstract. The computer industry is examining the use of strong synchronization operations such as double compare-and-swap (DCAS) as a means of supporting non-blocking synchronization on tomorrow's multiprocessor machines. However, before such a primitive will be incorporated into hardware design, its utility needs to be proven by developing a body of effective non-blocking data structures using DCAS.
In a previous paper, we presented two linearizable non-blocking implementations of concurrent deques (double-ended queues) using the DCAS operation. These improved on previous algorithms by nearly always allowing unimpeded concurrent access to both ends of the deque while correctly handling the difficult boundary cases when the deque is empty or full. A remaining open question was whether, using DCAS, one can design a non-blocking implementation of concurrent deques that allows dynamic memory allocation but also uses only a single DCAS per push or pop in the best case.
This paper answers that question in the affirmative. We present a new non-blocking implementation of concurrent deques using the DCAS operation. This algorithm provides the benefits of our previous techniques while overcoming drawbacks. Like our previous approaches, this implementation relies on automatic storage reclamation to ensure that a storage node is not reclaimed and reused until it can be proved that the node is not reachable from any thread of control. This algorithm uses a linked-list representation with dynamic node allocation and therefore does not impose a fixed maximum capacity on the deque. It does not require the use of a "spare bit" in pointers. In the best case (no interference), it requires only one DCAS per push and one DCAS per pop. We also sketch a proof of correctness.

1 Introduction

In academic circles and in industry, it is becoming evident that non-blocking algorithms can deliver significant performance benefits [3, 20, 17] and resiliency benefits [9] to parallel systems. Unfortunately, there is a growing realization that existing synchronization operations on single memory locations, such as compare-and-swap (CAS), are not expressive enough to support design of efficient non-blocking algorithms [9, 10, 12], and software emulations of stronger primitives from weaker ones are still too complex to be considered practical [1, 4, 7, 8, 21]. In response, industry is currently examining the idea of supporting

	Array with centralized access (see [9])	Array used as circular buffer (see [2])	Linked list with tagged pointers (see [2])	Snark (with garbage collection) (this paper)
Left and right accesses interfere	yes	no	no	no
Fixed limit on size of deque	yes	yes	no	no
Tag bit needed in pointers	no	no	yes	no
DCAS ops per unimpeded pop	1	1	2	1
DCAS ops per unimpeded push	1	1	1	1
Number of reserved values	1	1	3	0
Storage allocator calls per push	0	0	1	1
Storage overhead per item	none	none	2 pointers	2 pointers

Table 1. Comparison of various DCAS-based deque algorithms

stronger synchronization operations in hardware. A leading candidate among such operations is double compare-and-swap (DCAS), a CAS performed atomically on *two* memory locations. However, before such a primitive can be incorporated into processor design, it is necessary to understand how much of an improvement it actually offers. One step in doing so is developing a body of efficient data structures and associated algorithms based on the DCAS operation.

There have recently been several proposed designs for non-blocking linearizable concurrent double-ended queues (*deques*) using the double compare-and-swap operation [9, 2]. Deques, as described in [15] and currently used in load balancing algorithms [3], are classic structures to examine, in that they involve all the intricacies of LIFO-stacks and FIFO-queues, with the added complexity of handling operations originating at both ends of the deque.

Massalin and Pu [16] were the first to present a collection of DCAS-based concurrent algorithms. They built a lock-free operating system kernel based on the DCAS operation (`CAS2`) offered by the Motorola 68040 processor, implementing structures such as stacks, FIFO-queues, and linked lists.

Greenwald, a strong advocate for using DCAS, built a collection of DCAS-based concurrent data structures improving on those of Massalin and Pu. In the best case (no interference from other threads), his array-based deque algorithms required one DCAS per push and one DCAS per pop. Unfortunately, these algorithms used DCAS in a restrictive way. The first ([9] pp. 196–197) used the two-word DCAS as if it were a three-word operation, keeping the two deque end pointers in the same memory word, and DCAS-ing on it and a second word containing a value; this prevents truly concurrent, noninterfering access to the two deque ends. The second algorithm ([9] pp. 219–220) assumed an array of unbounded size, and did not correctly detect when the deque is full in all cases.

Arora et al. [3] present an elegant CAS-based restricted deque with applications in job-stealing algorithms. This non-blocking implementation needs only a single CAS operation since it restricts one side of the deque to be accessed by only a single processor, and the other side to allow only pop operations.

In a recent paper [2], we presented two new linearizable non-blocking implementations of concurrent deques using the DCAS operation. One used an array representation, and improved on previous algorithms by allowing uninterrupted concurrent access to both ends of the deque while correctly handling the difficult boundary cases when the deque is empty or full. In the best case, this array technique required one DCAS per push and one DCAS per pop. A drawback of the array representation was that it imposed a fixed maximum capacity on the queue. The second implementation corrected this by using a dynamic linked-list representation, and was the first non-blocking unbounded-memory deque implementation. Drawbacks of this list-based implementation were that it required a "spare bit" in certain pointers to serve as a boolean flag and that it required at least two (amortized) DCAS operations per pop.

A remaining open question was whether, using DCAS, one can design a non-blocking implementation of concurrent deques that allows dynamic memory allocation, as in the linked-list algorithms of [2], but also uses only a single DCAS per push or pop in the best case, as in array-based algorithms [2, 9]. This paper answers that question in the affirmative. Table 1 outlines the characteristics of the various algorithms. The first six rows indicate that the algorithm presented in this paper avoids drawbacks of previous work.

2 Modeling DCAS and Deques

Our computation model follows [5, 6, 14] as well as our own previous paper [2]. A *concurrent system* is a collection of n *processors*, which communicate through shared data structures called *objects*. Each object provides a set of primitive *operations* that are the only means of manipulating that object. Each processor is a thread of control [14] that sequentially invokes object operations by issuing an invocation and then receiving the associated response before issuing the next invocation. A *thread behavior* is the entire set of invocations and associated responses associated with a single thread; this set is totally ordered in time according to the order in which the thread issued and received the invocations and responses. A *system behavior* is the (disjoint) union of the thread behaviors of all the threads in a concurrent system.

A *history* is a system behavior upon which a total order has been imposed on invocations and responses that is consistent with the orderings of the thread behaviors. Each history may be regarded as a "real-time" order of operations where an operation A is said to *precede* another operation B if A's response occurs before B's invocation. Two operations are *concurrent* if they are unrelated by the real-time order. When we reason about the possible behaviors of a system or a thread within that system, we typically try to characterize the set of possible histories of the system.

A *sequential history* is a history in which each invocation is followed immediately by its associated response. The *sequential specification* of an object is a set of permitted sequential histories. The basic correctness requirement for a concurrent implementation is *linearizability* [14]: for every history H that may be realized by the system, there exists a sequential history that is in the intersection of the sequential specifications of all the objects in the system and whose total order of operations is consistent with the H's partial order of operations. In a linearizable implementation, each operation appears to take effect atomically at some point between its invocation and its associated response.

In our model, every shared memory location L of a multiprocessor machine's memory is a linearizable implementation of an object that provides every processor P_i with a set of sequentially specified machine operations (see [11, 13]):

$Read_i(\&L)$ reads location L and returns its value.
$Write_i(\&L, v)$ writes the value v to location L.
$DCAS_i(\&L1, \&L2, o1, o2, n1, n2)$ is a double-compare-and-swap operation with the semantics described below.

(The *address operator* `&` is used to pass the address of a location to an operation.) Because we assume a linearizable implementation, we can, in effect, assume that these operations are atomic when reasoning about programs that use them.

For the purposes of this paper, when we write code in a high-level language, we assume that each field of a high-level-language object and each global variable may be treated as a shared memory location. A simple reference to such a field or variable is a *Read* operation; a simple assignment to such a field or variable is a *Write* operation; and a method or subroutine called `DCAS` is used to perform the *DCAS* operation on two fields or variables.

The implementation we present is *non-blocking* (also called *lock-free*) [13]. Let us use the term *higher-level operations* to refer to operations of an object being implemented, and *lower-level operations* to refer to the (machine) operations in terms of which it is implemented. A *non-blocking* implementation is one for which any history that has invocations of some set O of higher-level operations but no associated responses may contain any number of responses for high-level operations concurrent with those in O. That is, even if some higher-level operations (each of which may be continuously taking steps, or not) never complete, other invoked operations may nevertheless continually complete. Thus the system as a whole can make progress; individual processors cannot be blocked, only delayed, by other processors continuously taking steps or failing to take steps. Using locks would violate the above condition, hence the alternate name *lock-free.*

Figure 1 contains code for the DCAS operation; for comparison, it also shows code for the simpler CAS operation (which is not used in the algorithms presented here). For either operation, the sequence of suboperations is assumed to be executed atomically, either through hardware support [12, 18, 19] or through a non-blocking software emulation [7, 21].

A CAS operation examines one memory location and compares its contents to an expected "old" value. If the contents match, then the contents are replaced with a specified "new" value and an indication of success is returned; otherwise the contents are unchanged and an indication of failure is returned.

A DCAS operation may be viewed as two yoked CAS operations: mismatch in either causes both to fail. (Note: the algorithms in this paper do not require the overloaded versions of DCAS that we used in our previous paper [2].)

```
boolean CAS(val *addr,
            val old,
            val new1) {
  atomically {
    if (*addr == old) {
      *addr = new;
      return true;
    } else return false;
  }
}
```

```
boolean DCAS(val *addr1, val *addr2,
             val old1, val old2,
             val new1, val new2) {
  atomically {
    if ((*addr1 == old1) &&
        (*addr2 == old2)) {
      *addr1 = new1;
      *addr2 = new2;
      return true;
    } else return false;
  }
}
```

Fig. 1. Single and Double Compare-and-Swap Operations

We assume that a CAS operation is substantially more expensive than a simple read or write of a shared variable, and that a DCAS is rather more expensive than a CAS. We also assume that memory operations (Read, Write, DCAS) that operate on distinct locations can be carried out concurrently, but those that operate on the same location are carried out sequentially, so there is a potential performance advantage in, for example, avoiding having operations on one end of a deque touch variables associated with the other end of the deque.

A *deque* S is a concurrent shared object created by a `makeDeque(length)` operation that allows each processor to perform one of four types of operations on S: `pushRight`, `popRight`, `pushLeft`, and `popLeft`.

We require that a concurrent implementation of a deque object be one that is linearizable to a standard sequential deque of the type described in [15].

The state of a deque is a sequence of items $S = \langle v_0, \ldots, v_k \rangle$ having cardinality $|S|$ where $0 \leq |S| \leq$ `length`. A deque is initially empty, that is, has cardinality 0. A deque is said to be full when its cardinality is `length`. (For the purposes of this paper, the `length` of the deque is essentially the total amount of storage available for allocation as deque node objects.)

The four possible push and pop operations induce the following state transitions of the sequence $S = \langle v_0, \ldots, v_k \rangle$, with appropriate returned values:

– `pushRight`(v_{new}), if S is not full, changes S to be $\langle v_0, \ldots, v_k, v_{new} \rangle$ and returns "okay"; if S is full, it returns "full" and S is unchanged.
– `pushLeft`(v_{new}), if S is not full, changes S to be $\langle v_{new}, v_0, \ldots, v_k \rangle$ and returns "okay"; if S is full, it returns "full" and S is unchanged.
– `popRight`$()$, if S is not empty, changes S to be $\langle v_0, \ldots, v_{k-1} \rangle$ and returns v_k; if S is empty, it returns "empty" and S is unchanged.
– `popLeft`$()$, if S is not empty, changes S to be $\langle v_1, \ldots, v_k \rangle$ and returns v_0; if S is empty, it returns "empty" and S is unchanged.

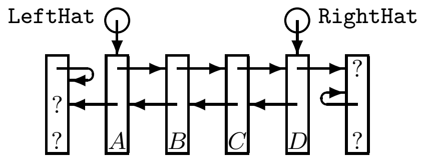
For example, starting with an empty deque $S = \langle \rangle$, `pushRight`(1) changes the state to $S = \langle 1 \rangle$; `pushLeft`(2) transitions to $S = \langle 2, 1 \rangle$; then `pushRight`(3) transitions to $S = \langle 2, 1, 3 \rangle$. A subsequent `popLeft`$()$ transitions to $S = \langle 1, 3 \rangle$ and returns 2; then `popLeft`$()$ transitions to $S = \langle 3 \rangle$ and returns 1 (which had been pushed from the right).

3 The “Snark” Linked-list Deque

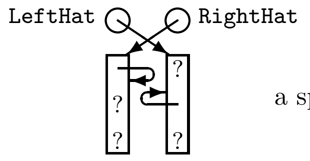
Our implementation (we have arbitrarily nicknamed it *Snark*) represents a deque as a doubly-linked list of nodes. Each node in the list contains two link pointers `R` and `L` and a value `V` (see Figure 2 below). There are two global “anchor” variables, arbitrarily called `LeftHat` and `RightHat` (lines 7–8), which generally point to the leftmost node and the rightmost node in the chain.

A node whose `L` field points to that same node is said to be *left-dead*; a node whose `R` field points to that same node is said to be *right-dead*. If `LeftHat` points to a node that is not left-dead, then the `L` field of that node points to a right-dead node; if `RightHat` points to a node that is not right-dead, then the `R` field of that node points to a left-dead node. As we will see, `LeftHat` points to a left-dead node if and only if `RightHat` points to a right-dead node; such a situation represents a deque with no items in it. The special node `Dummy` is both left-dead and right-dead (lines 6–7); as we will see, no other node is ever both left-dead and right-dead. In all cases, once a node becomes left-dead, it remains left-dead (until the node is determined to be inaccessible and therefore eligible to be reclaimed); once a node becomes right-dead, it remains right-dead. These rules may seem somewhat complicated, but they lead to a uniform implementation of pop operations.

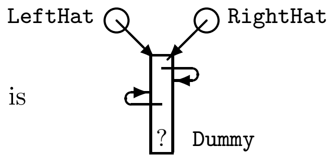
A typical deque, with values A, B, C, and D in it, looks like this:



where ? indicates a “don’t care” pointer or value. An empty deque looks like:



a special case of which is



Figures 3 and 4 show non-blocking implementations of push and pop operations on the right-hand end of the deque. We describe these operations in detail. The left-hand-side operations shown in Figures 5 and 6 are symmetric.

The right-side push operation first obtains a fresh `Node` structure from the storage allocator (Figure 3, line 2). (Note that the problem of implementing a

```
1 structure Node {
2   Node *R;
3   Node *L;
4   val V;       }
5 Node Dummy = new Node();
6 Dummy.L = Dummy.R = Dummy;
7 Node *LeftHat = Dummy;
8 Node *RightHat = Dummy;
```

Fig. 2. The array-based deque—data structure and hats (anchors).

```
 1 val pushRight(val v) {
 2   nd = new Node();  /* Allocate new Node structure */
 3   if (nd == null) return "full";
 4   nd->R = Dummy;
 5   nd->V = v;
 6   while (true) {
 7     rh = RightHat;                                /* Labels A, B,   */
 8     rhR = rh->R;                                  /* etc., are used */
 9     if (rhR == rh) {                              /* in the proof   */
10       nd->L = Dummy;                              /* of correctness */
11       lh = LeftHat;
12       if (DCAS(&RightHat, &LeftHat, rh, lh, nd, nd))          /* A */
13         return "okay";
14     } else {
15       nd->L = rh;
16       if (DCAS(&RightHat, &rh->R, rh, rhR, nd, nd))           /* B */
17         return "okay";
18 } } }                                  // Please forgive this brace style
```

Fig. 3. Simple linked-list deque—right-hand-side push.

non-blocking storage allocator is not addressed in this paper, but would need to be solved to produce a completely non-blocking deque implementation.) We assume that if allocatable storage has been completely exhausted (even after automatic reclamation has occurred), the `new` operation will yield a null pointer; the push operation treats this as sufficient cause to report that the deque is full (line 3). Otherwise, the `R` field of the new node is made to point to `Dummy` (line 4) and the value to be pushed is stored into the `V` field (line 5); all that remains is to splice this new node into the doubly-linked chain. But an attempt to splice might fail (because of an action by some other concurrent push or pop), so a "while true" loop (line 6) is used to iterate until a splice succeeds.

The `RightHat` is copied into local variable `rh` (line 7)—this is important. If `rh` points to a right-dead node (line 9), then the deque is empty. In this case, the new node should become the only node in the deque. Its `L` field is made to point to `Dummy` (line 10) and then a DCAS is used (line 12) to atomically make both `RightHat` and `LeftHat` point to the new node—but only if neither hat has changed. If this DCAS succeeds, then the push has succeeded (line 13); if the DCAS fails, then control will go around the "while true" loop to retry.

If the deque is not empty, then the new node must be added to the right-hand end of the doubly-linked chain. The copied content of the `RightHat` is stored into the `L` field of the new node (line 15) and then a DCAS is used (line 16) to make both the `RightHat` and the former right-end node point to the new node, which thus becomes the new right-end node. If this DCAS operation succeeds, then the push has succeeded (line 17); if the DCAS fails, then control will go around the "while true" loop to retry.

The right-side pop operation also uses a "while true" loop (line 2) to iterate until an attempt to pop succeeds. The `RightHat` is copied into local variable `rh`

```
 1 val popRight() {
 2   while (true) {
 3     rh = RightHat;                     // Delicate order of operations
 4     lh = LeftHat;                      // here (see proof of Theorem 4
 5     if (rh->R == rh) return "empty";   // and the Conclusions section)
 6     if (rh == lh) {
 7       if (DCAS(&RightHat, &LeftHat, rh, lh, Dummy, Dummy))     /* C */
 8         return rh->V;
 9     } else {
10       rhL = rh->L;
11       if (DCAS(&RightHat, &rh->L, rh, rhL, rhL, rh)) {         /* D */
12         result = rh->V;
13         rh->R = Dummy;                                         /* E */
14         rh->V = null;  /* optional (see text) */
15         return result;
16 } } } }                        // Stacking braces this way saves space
```

Fig. 4. Simple linked-list deque—right-hand-side pop.

(line 7)—this is important. If `rh` points to a right-dead node, then the deque is empty and the pop operation reports that fact (line 4).

Otherwise, there are two cases, depending on whether there is exactly one item or more than one item in the deque. There is exactly one item in the deque if and only if the `LeftHat` and `RightHat` point to the same node (line 6). In that case, a DCAS operation is used to reset both hats to point to `Dummy` (line 7); if it succeeds, then the pop succeeds and the value to be returned is in the `V` field of the popped node (line 8). (It is assumed that, after exit from the `popRight` routine, the node just popped will be reclaimed by the automatic storage allocator, through garbage collection or some such technique.)

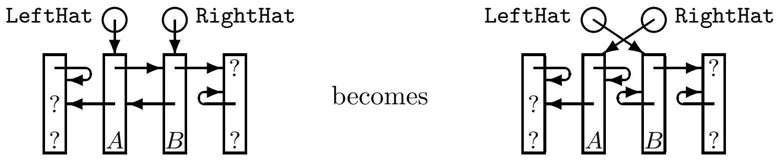
If there is more than one item in the deque, then the rightmost node must be removed from the doubly-linked chain. A DCAS is used (line 11) to move the `RightHat` to the node to the immediate left of the rightmost node; at the same time, the `L` field of that rightmost node is changed to contain a self-pointer, thus making the rightmost node left-dead. If this DCAS operation fails, then control will go around the "while true" loop to retry; but if the DCAS succeeds, then the pop succeeds and the value to be returned is in the `V` field of the popped node. Before this value is returned, the `R` field is cleared (line 13) so that *previously* popped nodes may be reclaimed. It may also be desirable to clear the `V` field immediately (line 14) so that the popped value will not be retained indefinitely by the queue structure. If the `V` field does not contain references to other data structures, then line 14 may be omitted.

The push and pop operations work together in a completely straightforward manner except in one odd case. If a `popRight` operation and a `popLeft` operation occur concurrently when there are exactly two nodes in the deque, then each operation may (correctly) discover that `LeftHat` and `RightHat` point to different nodes (line 6 in each of Figures 4 and 6) and therefore proceed to perform a DCAS for the multinode case (line 11 in each of Figures 4 and 6). Both of these

```
val pushLeft(val v) {
  nd = new Node();  /* Allocate new Node structure */
  if (nd == null) return "full";
  nd->L = Dummy;
  nd->V = v;
  while (true) {
    lh = LeftHat;
    lhL = lh->L;
    if (lhL == lh) {
      nd->R = Dummy;
      rh = RightHat;
      if (DCAS(&LeftHat, &RightHat, lh, rh, nd, nd))          /* A' */
        return "okay";
    } else {
      nd->R = lh;
      if (DCAS(&LeftHat, &lh->L, lh, lhL, nd, nd))            /* B' */
        return "okay";
} } }                        // We were given a firm limit of 15 pages
```

Fig. 5. Simple linked-list deque—left-hand-side push.

DCAS operations may succeed, because they operate on disjoint pairs of memory locations. The result is that the hats pass each other:



But this works out just fine: there had been two nodes in the deque and both have been popped, but as they are popped they are made right-dead and left-dead, so that the deque is now correctly empty.

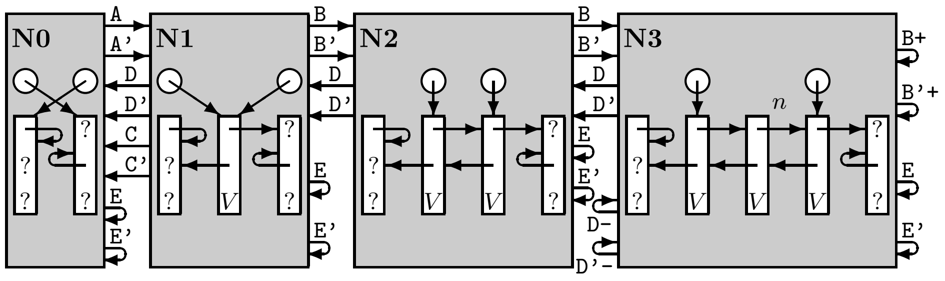
4 Sketch of Correctness Proof for the "Snark" Algorithm

We reason on a state transition diagram in which each node represents a class of possible states for the deque data structure and each transition arc corresponds to an operation in the code that can modify the data structure. For every node and every distinct operation in the code, there must be an arc from that node for that operation unless it can be proved that, when the deque is in the state represented by that node, either the operation must fail or the operation cannot be executed because flow of control cannot reach that operation with the deque in the prescribed state.

The possible states of a Snark deque are shown in the following state transition diagram:

```
val popLeft() {
  while (true) {
    lh = LeftHat;                      // Delicate order of operations
    rh = RightHat;                     // here (see proof of Theorem 4
    if (lh->L == lh) return "empty";   // and the Conclusions section)
    if (lh == rh) {
      if (DCAS(&LeftHat, &RightHat, lh, rh, Dummy, Dummy))    /* C' */
        return lh->V;
    } else {
      lhR = lh->R;
      if (DCAS(&LeftHat, &lh->R, lh, lhR, lhR, lh)) {         /* D' */
        result = lh->V;
        lh->L = Dummy;                                        /* E' */
        lh->V = null;  /* optional (see text) */
        return result;
} } } }                  // Better to stack braces than to omit a lemma
```

Fig. 6. Simple linked-list deque—left-hand-side pop.



The rightmost node shown actually represents an infinite set of nodes, one for each integer n for $n \geq 1$, where there are $n+2$ items in the deque. The labels on the transition arcs correspond to the labels on operations that modify the linked-list data structure in Figures 3, 4, 5, and 6. The labels `B+` and `B'+` indicate a transition that increases n by 1; the labels `D-` and `D'-` indicate a transition that decreases n by 1. We will also use labels such as `A` and `A'` in the text that follows to refer to DCAS and assignment operations in those figures.

We say that a node is “in the deque from the left” if it is not left-dead and it is reachable from the node referred to by the `LeftHat` by zero or more steps of following pointers in the `L` field. We say that a node is “in the deque from the right” if it is not right-dead and it is reachable from the node referred to by the `RightHat` by zero or more steps of following pointers in the `R` field.

The Snark algorithm is proved correct largely by demonstrating that, for every DCAS operation and every possible state of the deque data structure, if the DCAS operation succeeds then a correct transition occurs as shown in the state diagram. In cases where there is no corresponding arc on the state diagram, it is necessary to prove either that the DCAS cannot succeed if the deque is in that state or that control cannot reach the DCAS with the deque in that state. Here we provide proofs only for these latter cases.

Lemma 1. *A node is in the deque from the left if and only if it is in the deque from the right (therefore from now on we may say simply "in the deque").*

Lemma 2. *If a node is in the deque and then is removed, thereafter that node is never in the deque again.*

Lemma 3. *No node except the* `Dummy` *node is ever both left-dead and right-dead.*

Proof. Initially, only the `Dummy` node exists. Inspection of the code for `pushRight` and `pushLeft` shows that newly created nodes are never made left-dead or right-dead. Only operation `D` ever makes an existing node right-dead, and only operation `D'` ever makes an existing node left-dead. But `D` and `D'` each operate on a node that is in the deque, and as it makes a node left-dead or right-dead, it removes it from the deque. By Lemma 2, a node cannot be removed twice. So the same node is never made right-dead by `D` and also made left-dead by `D'`. ∎

Lemma 4. *No node is ever made left-dead or right-dead after the node is removed from the deque.*

Proof. By Lemma 2, after a node is removed from the deque it is never in the deque again. Only operation `D` ever makes an existing node right-dead, and only operation `D'` ever makes a node left-dead. But each of these operations succeeds only on a node that is in the deque. ∎

Lemma 5. *Once a node is right-dead, it stays right-dead as long as it is reachable from any thread.*

Proof. Only operations `B`, `D'`, and `E` change the `R` field of a node. But `B` succeeds only if the node referred to by `rh` is not right-dead, and `D` always makes the node referred to by `rh` right-dead. Operation `E` always stores into the `R` field of a node that has been made left-dead as it was removed from the deque. By Lemma 3, the node was not right-dead when it was removed from the deque; by Lemma 4, the node cannot become right-dead after it was removed from the deque. Therefore when operation `E` changes the `R` field of a node, that node is not right-dead. ∎

Lemma 6. *Once a node is left-dead, it stays left-dead as long as it is reachable from any thread.*

Lemma 7. *The* `RightHat` *points to a right-dead node if and only if the deque is empty, and the* `LeftHat` *points to a left-dead node if and only if the deque is empty.*

Proof. Initially both `RightHat` and `LeftHat` point to the `Dummy` node, so this invariant is initially true. Operations `A` and `A'` make both `RightHat` and `LeftHat` point to a new node that is not left-dead or right-dead, so the deque is not empty. Operation `B` can succeed only if the `RightHat` points to a node that is not right-dead, and it changes `RightHat` to point to a new node that is not right-dead. A symmetric remark applies to `B'`. Operations `C` and `C'` make both `RightHat` and

`LeftHat` point to the `Dummy` node, which is both left-dead and right-dead, so the deque is empty. If operation `D` moves `RightHat` from a node that it not right-dead to a node that is right-dead, then the deque had only one item in it; then the `LeftHat` also points to the node just removed from the deque by operation `D`, and that operation, as it moved the `RightHat` and emptied the deque, also made the node left-dead. A symmetric remark applies to `D'`. Operations `E` and `E'` do not change whether a node is left-dead or right-dead (see proof of Lemma 5). ∎

Theorem 1. *Operation* `A` *fails if the deque is not empty.*

Proof. Operation `A` is executed only after the node referred to by `rh` has been found to be right-dead. By Lemma 5, once a node is right-dead, it remains right-dead. Therefore, if the deque is non-empty when `A` is executed, then `RightHat` must point to some other node than the one referred to by `rh`; therefore `RightHat` does not match `rh` and the DCAS must fail. ∎

Theorem 2. *Operation* `B` *fails if the deque is empty.*

Proof. Operation `B` is executed only after `rhR` has been found unequal to `rh`. If the deque is empty when `B` is executed, and `RightHat` equals `rh`, then the node referred to by `rh` must have become right-dead; but that means that `rh->R` equals `rh`, and therefore cannot match `rhR`, and so the DCAS must fail. ∎

Theorem 3. *Operation* `C` *fails unless there is exactly one item in the deque.*

Proof. When `C` is executed, `rh` equals `lh`, so `C` can succeed only when `RightHat` and `LeftHat` point to the same node. If the deque has two or more items in it, then `RightHat` and `LeftHat` contain different values, so the DCAS must fail.

If the deque is empty, and `RightHat` and `LeftHat` point to same node, then by Lemma 7 that node must be both left-dead and right-dead, and by Lemma 3 that node must be the `Dummy` node, which is created right-dead and (by Lemma 5) always remains right-dead. But then the test in line 5 of `popRight` would have prevented control from reaching operation `C`. Therefore, if `C` is executed with the deque empty, `RightHat` and `LeftHat` necessarily contain different values, so the DCAS must fail. ∎

Theorem 4. *Operation* `D` *fails if the deque is empty.*

Proof. This is the most difficult and delicate of our proofs. Suppose that some thread of control T is about to execute operation `D`. Then T, at line 3 of `popRight`, read a value from `RightHat` (now in T's local variable `rh`) that pointed to a node that was not right-dead when T executed line 5; therefore the deque was not empty at that time. Also, T must have read a value from `LeftHat` in line 4 that turned out not to be equal to `rh` when T executed line 6.

Now suppose, as T executes `D` in line 12, that the deque is empty. How might the deque have become empty since T executed line 5? Only through the execution of `C` or `C'` or `D` or `D'` by some other thread U. If U executed `C` or `D`, then it changed the value of `RightHat`; in this case T's execution of DCAS `D` must fail, because `RightHat` will not match `rh`.

So consider the case that U executed `C'` or `D'`. (Note that, for the execution of `D` by T to succeed, there cannot have been another thread U' that performed a `C'` or `D'` after U but before T's execution of DCAS `D'`, because that would require a preceding execution of `A` or of `A'`, either of which would have changed `RightHat`, causing T's execution of DCAS `D` to fail.)

Now, if U executed `C'`, then U changed the value of `RightHat` (to point to `Dummy`); therefore T's execution of DCAS `D` must fail.

If, on the other hand, U executed `D'` to make the deque empty, then the deque must have had one item in it when U executed DCAS `D'`. But thread U read values for `LeftHat` (in line 3 of `popLeft`) and `RightHat` (in line 4) that were found in line 6 not to be equal. Therefore, when U read `RightHat` in line 4, either the deque did not have exactly one item in it or the value of `LeftHat` had been changed since U read `LeftHat` in line 3. If `LeftHat` had been changed, then execution of `D'` by U would have to fail, contrary to our assumption. Therefore, if there is any hope left for execution of `D'` by U to succeed, the deque must *not* have had exactly one item in it when U read `RightHat` in line 4.

How, then, might the deque have come to hold exactly one item after U executed line 4? Only through some operation by a third thread. If that operation was `A'` or `B'` or `C'` or `D'`, that operation must have changed `LeftHat`; but that would cause the execution of DCAS `D'` by U to fail, contrary to our assumption. Therefore the operation by a third thread must have been `A` or `B` or `C` or `D`. Consider, then, the most recent execution (relative to the execution of `D` by T) of DCAS `A` or `B` or `C` or `D` that caused the deque to contain exactly one item, and let V be the thread that executed it. (It is well-defined which of these DCAS executions is most recent because DCAS operations `A`, `B`, `C`, and `D` all synchronize on a common variable, namely `RightHat`.)

If this DCAS operation by thread V occurred after thread T read `RightHat` in line 3, then it changed `RightHat` after T read `RightHat`, and the execution of DCAS `D` by T must fail. Therefore, if there is any hope left for execution of `D` by T to succeed, then execution of the most recent DCAS `A` or `B` or `C` or `D` (by V) must have occurred before T read `RightHat` in line 3.

To summarize the necessary order of events: (a) U reads `LeftHat` in line 3 of `popLeft`; (b) V executes `A` or `B` or `C` or `D`, resulting in the deque containing one item; (c) T executes lines 3, 4, 5, and 6 of `popRight`; (d) U executes `D'`; (e) T executes `D`. Moreover, there was no execution of `A` or `B` or `C` or `D` by any other thread after event (b) but before event (e), and there cannot have been any execution of `A'` or `B'` or `C'` or `D'` after event (a) but before event (d).

Therefore the deque contained exactly one item during the entire time that T executed lines 3 though 6 of `popRight`. But if so, the test in line 6 would have prevented control from reaching `D`.

Whew! We have exhausted all possible cases; therefore, if DCAS `D` is executed when the deque is empty, it must fail. ∎

Theorem 5. *Operation* `E` *always succeeds and does not change the number of items in the deque.*

Symmetric theorems apply to operations `A'`, `B'`, `C'`, and `D'`.

Space limitations prevent us from presenting a proof of linearizability and a proof that the algorithms are non-blocking—that is, if any subset of the processors invoke `push` or `pop` operations but fail to complete them (whether the thread be suspended, or simply unlucky enough never to execute a DCAS successfully), the other processors are in no way impeded in their use of the deque and can continue to make progress. However, we observe informally that a thread has not made any change to the deque data structure (and therefore has not made any progress visible to other threads) until it performs a successful DCAS, and once a thread has performed a single successful DCAS then, as observed by other threads, a push or pop operation on the deque has been completed. Moreover, each DCAS used to implement a `push` or `pop` operation has no reason to fail unless some other `push` or `pop` operation has succeeded since it was invoked.

5 Conclusions

We have presented non-blocking algorithms for concurrent access to a double-ended queue that supports the four operations `pushRight`, `popRight`, `pushLeft`, and `popLeft`. They depend on a multithreaded execution environment that supports automatic storage reclamation in such a way that a node is reclaimed only when no thread can possibly access it. Our technique improves on previous methods in requiring only one DCAS per push or pop (in the absence of interference) while allowing the use of dynamically allocated storage to hold queue items.

We have two remaining concerns about this algorithm and the style of programming that it represents. First, the implementation of the `pop` operations is not entirely satisfactory because a `popRight` operation, for example, necessarily reads `LeftHat` as well as `RightHat`, causing potential interference with `pushLeft` and `popLeft` operations even when there are many items in the queue, which in hardware implementations of interest could degrade performance.

Second, the proof of correctness is complex and delicate. While DCAS operations are certainly more expressive than CAS operations, and can serve as a useful building block for concurrent algorithms such as the one presented here that can be encapsulated as a library, after our experience we are not sure that we can wholeheartedly recommend DCAS as the synchronization primitive of choice for everyday concurrent applications programming. In an early draft of this paper, we had transposed lines 4 and 5 of Figure 4 (and similarly lines 4 and 5 of Figure 6); we thought there was no need for `popRight` to look at the `LeftHat` until the case of an empty deque had been disposed of. We were wrong. As we discovered when the proof of Theorem 4 would not go through, that version of the code was faulty, and it was not too difficult to construct a scenario in which the same node (and therefore the same value) could be popped twice from the queue. As so many (including ourselves) have discovered in the past, when it comes to concurrent programming, intuition can be extremely unreliable and is no substitute for careful proof. While we believe that non-blocking algorithms are an important strategy for building robust concurrent systems, we also believe it is desirable to build them upon concurrency primitives that keep the necessary proofs of correctness as simple as possible.

References

1. Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling multi-object operations. In *Proc. 16th ACM Symp. Principles of Dist. Computing*, pages 111–120, August 1997. Santa Barbara, CA.
2. O. Agesen, D. Detlefs, C. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. Steele. DCAS-based concurrent deques. In *Proc. 12th ACM Symp. Parallel Algorithms and Architectures (to appear)*, July 2000.
3. N. S. Arora, R. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proc. 10th ACM Symp. Parallel Algorithms and Architectures*, 1998.
4. H. Attiya and E. Dagan. Universal operations: Unary versus binary. In *Proc. 15th ACM Symp. Principles of Dist. Computing*, May 23–26 1996. Philadelphia, PA.
5. Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? *Journal of the ACM*, 41(4):725–763, July 1994.
6. Hagit Attiya and Ophir Rachman. Atomic snapshots in $O(n \log n)$ operations. *SIAM Journal on Computing*, 27(2):319–340, March 1998.
7. G. Barnes. A method for implementing lock-free shared data structures. In *Proc. 5th ACM Symp. Parallel Algorithms and Architectures*, pages 261–270, June 1993.
8. B. N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proc. 13th IEEE International Conf. on Distributed Computing Systems*, pages 264–273. IEEE Computer Society Press, May 25–28 1993. Los Alamitos, CA.
9. M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, 1999.
10. M. B. Greenwald and D. R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *2nd Symp. Operating Systems Design and Implementation*, pages 123–136, October 28–31 1996. Seattle, WA.
11. M. Herlihy. A methodology for implementing highly concurrent data structures. *ACM Trans. Programming Languages and Systems*, 15(5):745–770, November 1993.
12. M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. Technical Report CRL 92/07, Digital Equipment Corporation, Cambridge Research Lab, 1992.
13. M. P. Herlihy. Wait-free synchronization. *ACM Trans. Programming Languages and Systems*, 13(1):123–149, January 1991.
14. M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Programming Languages and Systems*, 12(3):463–492, July 1990.
15. D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison-Wesley, 2nd edition, 1968.
16. H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report TR CUCS–005–9, Columbia University, New York, NY, 1991.1¡
17. M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR 599, Computer Science Department, University of Rochester, 1995.
18. Motorola. *MC68020 32-Bit Microprocessor User's Manual*. Prentice-Hall, 1986.
19. Motorola. *MC68030 User's Manual*. Prentice-Hall, 1989.
20. Martin C. Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Trans. Computer Systems*, 17(4):337–371, November 1999.
21. N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997.