DCAS-Based Concurrent Deques

Ole Agesen*[1] David L. Detlefs† Christine H. Flood† Alexander T. Garthwaite†
Paul A. Martin† Nir N. Shavit† Guy L. Steele Jr.†

*VMware
†Sun Microsystems Laboratories

Abstract

The computer industry is currently 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 strong 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.

As part of this effort, we present two new linearizable non-blocking implementations of concurrent deques using the DCAS operation. The first uses an array representation, and improves on former 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. The second uses a linked-list representation, and is the first non-blocking unbounded-memory deque implementation. It too allows uninterrupted concurrent access to both ends of the deque.

1 Introduction

In academic circles and in industry, it is becoming evident that non-blocking algorithms can deliver significant performance [3, 23, 20] and resiliency benefits [11] 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 [11, 12, 16], and software emulations of stronger primitives from weaker ones are still too complex to be considered practical [1, 4, 7, 8, 24]. In response, industry is currently examining the idea of supporting 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 based on the DCAS operation.

This paper presents two novel designs of non-blocking linearizable concurrent double-ended queues (*deques*) using the double compare-and-swap operation. Deques, originally described in [18], 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. By being linearizable [17] and non-blocking [14], our concurrent deque implementations are guaranteed to behave as if operations on them were executed in a mutually exclusive manner, without actually using any form of mutual exclusion.

1.1 Related Work

Massalin and Pu [19] 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. He proposed various implementations of the DCAS operation in software and hardware, and presented two array-based concurrent deque algorithms using DCAS [11]. Unfortunately, his algorithms used DCAS in a restrictive way. The first (pages 196-197 of [11]) 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. Apart from the fact that this limits applicability by cutting the index range to half a memory word, it also prevents concurrent access to the two deque ends. The second algorithm (pages 219-220 of [11]) assumed an unbounded size array, and did not correctly detect when the deque is full in all cases.

Arora et al. [3] present an elegant CAS-based deque with applications in job-stealing algorithms. Their non-blocking implementation works with only a 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.

To the best of our knowledge, there is no linked-list-based deque implementation using DCAS in the literature. List-based implementations have the obvious advantage of avoiding fixed, static resource allocations.

[1] Work done while a member of Sun Microsystems Laboratories.

1.2 The new algorithms

This paper presents two novel deque implementations that are non-blocking and linearizable, and do not suffer from the above-mentioned drawbacks of former DCAS-based algorithms. The new array-based algorithm we present in Section 3 allows uninterrupted concurrent access to both ends of the deque, while returning appropriate exceptions in the tricky boundary cases when the deque is empty or full. The key to our algorithm is the rather interesting realization that a processor can detect these boundary cases, that is, determine whether the array is empty or full, without needing to check the relative locations of the two end pointers in one atomic operation.

In Section 4 we present a linked-list-based algorithm, which uses full-word pointers and does not restrict concurrency in accessing the deque's two ends. To the best of our knowledge this is the first linked-list based implementation of a deque using a DCAS operation. The algorithm is based on a new technique for splitting the pop operation into two steps, marking that a node is about to be deleted, and then deleting it. Once marked, the node is considered "deleted," and the actual deletion from the list can then be performed by the next push or next pop operation on that side of the deque. The key to making this algorithm work is the use of DCAS to correctly synchronize delete operations when processors detect that there are only marked nodes in the list, and attempt to delete one or more of these nodes concurrently from both ends. The cost of the splitting technique is an extra DCAS operation per pop. The benefit is that it allows non-blocking completion without needing to "lock" both of the deque's end pointers with a DCAS. The splitting also requires allocating a bit in the pointer word to indicate if it is pointing to a marked node that needs to be deleted. However, this extra bit can be easily avoided, as we explain later, by adding two dummy "delete-bit" records to the structure.

In summary, we believe that through the design of linearizable lock-free implementations of classical data structures such as deques, we will be able to understand better the power of the DCAS abstraction, and whether one should continue the effort to provide support for implementing it [1, 4, 7, 8, 11, 12, 24] on concurrent hardware and software processing platforms. The next section presents our computation model and a formal specification of the deque data structure.

2 Modeling DCAS and Deques

Our paper deals with implementing a deque on a shared memory multiprocessor machine, using the DCAS operation. This section describes the computation model we will use in our proofs and specifies the concurrent semantics of the deque data structure we will implement.

Our computation model follows [5, 6, 17]. A *concurrent system* consists of a collection of n *processors*. Processors communicate through shared data structures called *objects*. Each object has a set of primitive *operations* that provide the only means of manipulating that object. Each processor P is a sequential thread of control [17] which applies a sequence of operations to objects by issuing an invocation and receiving the associated response. A *history* is a sequence of invocations and responses of some system execution. Each history induces a "real-time" order of operations where an operation A *precedes* 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. A *sequential history* is a history in which each invocation is followed immediately by its corresponding response. The *sequential specification* of an object is the set of *legal* sequential histories associated with it. The basic correctness requirement for a concurrent implementation is *linearizability* [17]: every concurrent history is "equivalent" to some legal sequential history which is consistent with the real-time order induced by the concurrent history. In a linearizable implementation, operations appear to take effect atomically at some point between their invocation and response. In our model, every shared memory location L of a multiprocessor machine's memory is a linearizable implementation of an object which provides every processor P_i with the following set of sequentially specified machine operations (see [15, 14] for details):

$Read_i(L)$ 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 in the next section.

We assume, based on current trends in computer architecture design [13], that DCAS is a relatively expensive operation, that is, has longer latency than traditional CAS, which in turn has longer latency than both a read or a write. We assume this is true even when operations are executed sequentially. We also assume the availability of a storage allocation/collection mechanism as in Lisp [25] and the Java™ programming language [10]. The details of the allocator are not exposed to the user[1], yet it will be assumed that it includes an operation:

New_i(v) that allocates a new structure v in memory and returns a pointer to it.

The implementations we present will be *non-blocking* (also called *lock-free*) [14]. Let us use the term *higher-level operations* in referring to operations of the data type being implemented, and *lower-level operations* in referring to the (machine) operations in terms of which it is implemented. A *non-blocking* implementation is one in which any infinite history containing a higher-level operation that has an invocation but no response must also contain infinitely many responses concurrent with that operation. In other words, if some higher level operation continuously takes steps and does not complete, it must be because some other invoked operations are continuously completing their responses. This definition guarantees that the system as a whole makes progress and that individual processors cannot be blocked, only delayed by other processors continuously taking steps. Using locks would violate the above condition, hence the alternate name: *lock-free*.

2.1 The DCAS operation

Figure 1 contains the code of the DCAS operation. The sequence of operations is assumed to be executed atomically, either through hardware support [16, 21, 22], through a non-blocking software emulation [7, 24], or via a blocking software emulation [2]. Note that the DCAS operation is overloaded; if the middle two arguments are pointers,

[1]Note that the problem of implementing a non-blocking storage allocator is not addressed in this paper but would need to be solved to produce a completely non-blocking deque implementation.

```
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;
    }
  }
}

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 {
      *old1 = *addr1;
      *old2 = *addr2;
      return false;
    }
  }
}
```

Figure 1: The Double Compare-and-Swap Operation

then the original contents of the tested locations are stored into the indicated locations, thus providing a way for the DCAS operation to return more information than just a success/failure flag. This overloaded definition merely represents two interfaces from our higher level language to the same single machine instruction, in order to direct the compiler whether to store or to discard the contents of the result registers after execution of the DCAS instruction.

2.2 The Deque data structure

A *deque* object S is a concurrent shared object created by a `make_deque(length_S)` operation and allowing each processor P_i, $0 \leq i \leq n-1$, to perform one of four types of operations on S: `pushRight`$_i(v)$, `pushLeft`$_i(v)$, `popRight`$_i()$, and `popLeft`$_i()$. Each push operation has input v of type `val`. Each pop operation returns an output in the set `val`.

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 [18]. We specify this sequential deque using a state-machine representation that captures all of its allowable sequential histories. These sequential histories include all sequences of push and pop operations induced by the state machine representation, but do not include the actual states of the machine. In the following description, we abuse notation slightly for the sake of brevity.

The state of a deque is a sequence [9] of items $S = \langle v_0, \ldots, v_k \rangle$ from the range of `values`, having cardinality $0 \leq |S| \leq$ `length_S`. The deque is initially in the empty state (following `make_deque(`*length_S*`)`), that is, has cardinality 0, and is said to have reached a full state if its cardinality is `length_S`.

The four possible push and pop operations, executed sequentially, 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 the sequence $S = \langle v_0, \ldots, v_k, v_{new} \rangle$ 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 the sequence $S = \langle v_{new}, v_0, \ldots, v_k \rangle$ and returns "okay"; if S is full, it returns "full" and S is unchanged.
- `popRight`() if S is not empty, changes S to be the sequence $S = \langle v_0, \ldots, v_{k-1} \rangle$ 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 the sequence $S = \langle v_1, \ldots, v_k \rangle$ and returns v_0; if S is empty, it returns "empty" and S is unchanged.

For example, starting with an empty deque $S = \langle\rangle$, the following sequence of operations and corresponding transitions can occur: `pushRight`(1) changes the state to $S = \langle 1 \rangle$; `pushLeft`(2) transitions to $S = \langle 2, 1 \rangle$; a subsequent `pushRight`(3) transitions to $S = \langle 2, 1, 3 \rangle$. A subsequent `popLeft`() transitions to $S = \langle 1, 3 \rangle$ and returns 2. Another subsequent `popLeft`() transitions to $S = \langle 3 \rangle$ and returns 1 (which had been pushed from the right).

It is desirable for performance reasons that operations on opposite ends of the deque interfere with each other as little as possible; the ideal is that no contention occurs if the deque contains more than one item.

3 The array-based algorithm

```
  Initially L == 0, R == 1; S[0..length_S-1] of "null";

 0 val popRight {
 1   newS = "null";
 2   while (true) {
 3     oldR = R;
 4     newR = (oldR - 1) mod length_S;
 5     oldS = S[newR];
 6     if (oldS == "null") {
 7       if (oldR == R)
 8         if (DCAS(&R, &S[newR],
 9                  oldR, oldS, oldR, oldS))
10           return "empty";
11     }
12     else {
13       saveR = oldR;
14       if (DCAS(&R, &S[newR],
15                &oldR, &oldS, newR, newS))
16         return oldS;
17       else if (oldR == saveR) {
18         if (oldS == "null") return "empty";
19       }
20     }
21   }
22 }
```

Figure 2: The array-based deque - right-hand-side pop.

The following is a non-blocking implementation of a deque in an array using DCAS. We describe in detail the code in Figures 2 and 3 which deal with the right-hand-side push and pop operations, with the understanding that the left-hand-side operations in Figures 18 and 19 are symmetric. As depicted in Figure 4, the deque consists of an array `S[0..length_S-1]` indexed by two counters, `R` and `L`. We assume that `mod` is the modulus operation over the integers (-1 `mod` 6 = 5, -2 `mod` 6 = 4, and so on). Henceforth, we assume that all indexes into the array `S` are modulo `length_S`,

```
 1 val pushRight(val v) {
 2   while (true) {
 3     oldR = R;
 4     newR = (oldR + 1) mod length_S;
 5     oldS = S[oldR];
 6     if (oldS != "null") {
 7       if (oldR == R)
 8         if (DCAS(&R, &S[oldR],
 9                  oldR, oldS, oldR, oldS))
10           return "full";
11     }
12     else {
13       saveR = oldR;
14       if DCAS(&R, &S[oldR],
15               &oldR, &oldS, newR, v)
16         return "okay";
17       else if (oldR == saveR)
18         return "full";
19     }
20   }
21 }
```

Figure 3: The array-based deque - right-hand-side push.

which implies that `S` is viewed as being circular. We also assume a distinguished value "null" (denoted as "0" in some of the figures) not occuring in the real data values (denoted in the code as `val`) range in `S`.

Think of the array `S[0..length_S-1]` as if it were laid out with indexes increasing from left to right. The code works as follows. Initially, `L` points immediately to the left of `R`. The pointers `L` and `R` always point to the next location into which a value can be inserted. It will always be true that if there is no value immediately to the right of `L` (respectively, to the left of `R`), then the deque is in the empty state. Similarly, it will always be true that if there is a non-null value in location `L` (respectively, location `R`), then the deque is in the full state. Figure 4 shows such empty and full states. During the algorithm's execution, the use of a DCAS guarantees that at most one processor can succeed in modifying the entry at any array location from a null to a non-null value or vice versa.

To perform a `popRight`, a processor first reads `R` and the location in `S` corresponding to `R-1` (Lines 3-5). It then checks whether `S[R-1]` is null. (As we noted above, `S[R-1]` is shorthand for `S[R-1 mod length_S]`). If `S[R-1]` is null, then the processor reads `R` again to see if it has changed (Lines 6-7). This additional read is a performance enhancement added under the assumption that the common case is that a null value is read because another processor "stole" the item, and not because the deque is really empty. The test is thus that if `R` has not changed and `S[R-1]` is null, then the deque may be empty since the location to the left of `R` always contains a value unless there are no items in the deque. However, the conclusion that the deque is empty can only be made based on an instantaneous view of `R` and `S[R-1]`, and so the processor performs a DCAS to check if this is in fact the case (Lines 8-10). If so, the processor returns an indication that the deque is empty. If not, then either the value read in `S[R-1]` is no longer null or the index `R` has changed. In either case, the processor must loop around and start again, since there might now be an item to pop.

If `S[R-1]` is not null, the processor attempts to pop that item (Lines 12-20). It uses a DCAS to try to decrement the counter `R` and to place a null value in `S[R-1]`, while returning (via `&newR` and `&newS`) the old value in `S[R-1]` and the old value of the counter `R` (Lines 13-15). A successful DCAS (and hence a successful `popRight` operation) is depicted in Figure 5. If the DCAS succeeds, the processor returns `S[R-1]`. If it fails, then it needs to check what the reason for the failure was. If the reason for the DCAS failing was that `R` changed, then the processor must retry again (repeat the loop) since there may be items still left in the deque. If `R` has not changed (Line 17), then the DCAS must have failed because `S[R-1]` changed. If it changed to null (Line 18), then it must be that the deque is empty. This empty deque would have been the result of a competing `popLeft` that "stole" the last item from the `popRight`, as in Figure 6. If, on the other hand, `S[R-1]` was not null, since the DCAS failed it must mean that it is different than the value of `S[R-1]` that the processor first read (and tried to DCAS with), and so some other processor(s) must have performed a pop and a push between the read and the DCAS operation. The processor must thus loop back and try again, since there may still be items in the deque. We note that Lines 17-18 are an optimization, and one can instead loop directly back if the DCAS fails. We add this optimization to allow detecting a possible empty state without going through the loop, which in case the queue was indeed empty, would require another DCAS operation (Lines 6-10).

To perform a `pushRight`, a sequence similar to `popRight` is performed. AS depicted in Figure 7, for a given `R`, the location into which an item will be pushed is the one to which `R` points (Line 5 in both procedures). A successful `pushRight` operation into an empty deque is depicted in Figure 7, and a successful `pushRight` operation into an almost-full deque appears in the bottom of Figure 8. All tests to see if a location is null are now replaced with tests to see if it is non-null (such as in Lines 6-10). Furthermore, in the final stage of

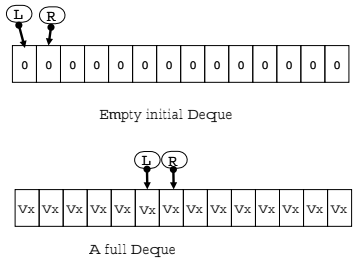


Figure 4: Empty and full array-based deques.

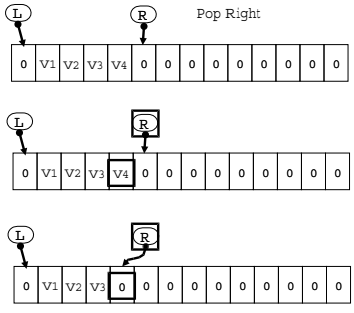


Figure 5: A successful array-based `popRight`.

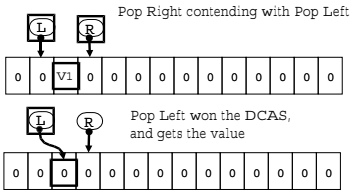


Figure 6: A `popRight` contending with a `popLeft`.

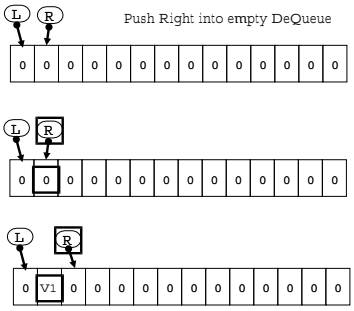


Figure 7: A successful array-based `pushRight`.

the code, in case the DCAS failed, there is a check to see if the `R` index has changed. If it has not, then the failure must be due to a non-null value in that location, which means that the deque is full. Unlike the case of a `popRight`, the DCAS (Lines 14-15) is only checking to see if there was a null value, and if none is found it does not matter what the non-null value was: in all cases it means that the deque is full (Line 18).

Figure 8 shows how an almost full deque becomes full following push operations from the left and the right. Notice how `L` has wrapped around and is "to-the-right" of `R`, until the deque becomes full, in which case `L` and `R` cross again. This switching around of the relative location of the `L` and `R` pointers is rather confusing. The key to our algorithm is the observation that we can determine the state of the deque, not based on these relative locations of `L` and `R`, but rather by examining the combination of where a given pointer variable is pointing and the value in the location associated with that pointer.

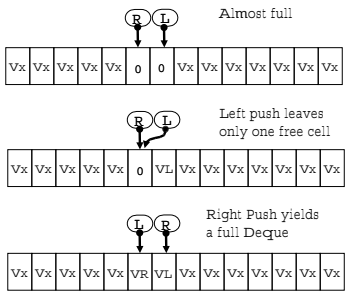


Figure 8: Filling the array.

Theorem 3.1 *The array based deque algorithm of Section 3 is a non-blocking linearizable implementation of a deque data structure.*

4 The linked-list-based algorithm

The previous sections presented an array-based deque implementation appropriate in situations where the maximum size of the deque can be predicted in advance. In this section we present an implementation that avoids the need to limit size, by allowing dynamic memory allocation. We represent a deque as a doubly-linked list. Each node in the list contains two link pointers and a value.

```
typedef node {
 pointer *L;
 pointer *R;
 val_or_null_or_SentL_or_SentR value;
}
```

It is assumed that there are three distinguished values (called `null`, `sentL`, and `sentR`) that can be stored in the value field of a node but are never requested to be pushed onto the deque. The doubly-linked list has two distinguished nodes called "sentinels." The left sentinel is at a known fixed address `SL`; its `L` pointer is never used, and its value field always contains `sentL`. Similarly, the right sentinel is at a known fixed address `SR` with value `sentR` and an unused `R` pointer.

The basic intuition behind this implementation is that a node is always removed (as part of a `popRight` or `popLeft` operation) in two separate, atomic steps: first the node is "logically" deleted, by replacing its value with null and setting a special `deleted` bit to 1 in the sentinel to indicate the presence of a logically deleted node; second, the node is "physically" deleted by modifying pointers so that the node is no longer in the doubly-linked chain of nodes, as well as resetting the bit in the sentinel to 0. If a processor that is removing a node is suspended between these two steps, then any other process can perform the second step or otherwise work around the fact that the second step has not yet been performed.

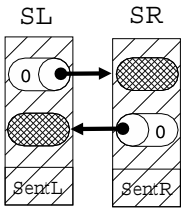
More concretely, the `deleted` bit in each sentinel is represented as part of its pointer into the list. The following structure is thus maintained in a single word, by assuming sufficient pointer alignment to free one low-order bit. [2]

```
typedef pointer {
 node *ptr;
 boolean deleted;
}
```

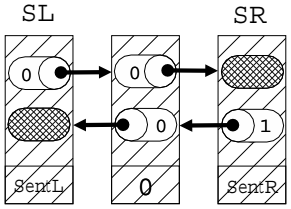
Initially `SR->L == SL` and `SL->R == SR`, as in the top part of Figure 9. All push and pop procedures use an auxiliary delete procedure, which we describe last. We describe only the `popRight` and `pushRight` procedures. The `popLeft` and `pushLeft` procedures are symmetric and can be found in Figures 20 and 21.

The code for the `popRight` procedure appears in Figure 11. The processor begins by checking if the right sentinel is in one of the following states:

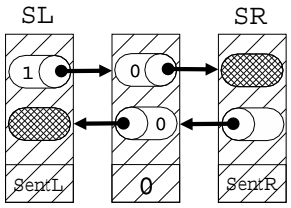
[2] One can altogether eliminate the need for a "deleted" bit by introducing a special dummy type "delete-bit" node, distinguishable from regular nodes, in place of the bit. Each processor would have a dummy node for the left and one for the right, and pointing to a node indirectly via its dummy node, as in Figure 10, represents a bit value of true, and pointing directly represents false.



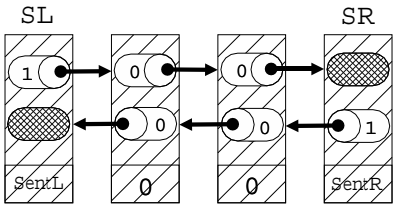
Empty Deque



Empty Deque with
a right−deleted cell



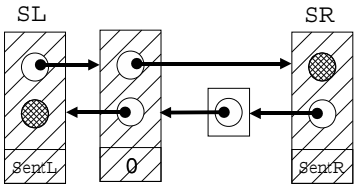
Empty Deque with
a left−deleted cell



Empty Deque with
two deleted cells

Figure 9: Four versions of an empty linked-list-based deque.

- it points to the left sentinel (Line 5 of the code and the top of Figure 9),
- it points to a node that has a null value (and is thus logically deleted) (Lines 6-12 of the code and the upper part of Figure 15 and the three lower parts of 9), or,
- it points to a node that has a non-null value (Lines 13-19 of the code and Figure 12).



Empty Deque with one
deleted cell marked by
a right dummy node

Figure 10: Replacing the deleted bit with a dummy node.

If the next node is a sentinel (Line 5), then the deque is empty, as in the top of Figure 9, and the processor can return "empty." If the left node is not a sentinel, it must check if the `deleted` bit in the right sentinel is true (Line 6). If so, the processor must call the `deleteRight` procedure, which will remove the null node on the right-hand side, and then retry the pop (Line 7). If the `deleted` bit (in the right sentinel) is false, then the processor must make sure that the node to be popped does not have a null value (Line 8), which indicates that the deque could be empty. This null node would be the result of a `deleteLeft`, with the `deleted` bit true in the left sentinel, as shown in the third diagram from the top of Figure 9. The way to test for this case is to atomically check, using a DCAS operation, if there is both a null value in the node and a false `deleted` bit in the pointer to that node from the right sentinel (Lines 9-11). Otherwise, the deque must have been modified concurrently between the original reads and the DCAS test, so the processor should loop and try again.

```
 1 val popRight() {
 2   while (true) {
 3     oldL = SR->L;
 4     v = oldL.ptr->value;
 5     if (v == "SentL") return "empty";
 6     if (oldL.deleted == true)
 7       deleteRight();
 8     else if (v == "null") {
 9       if (DCAS(&SR->L, &oldL.ptr->value,
10                oldL, v, oldL, v))
11         return "empty";
12     }
13     else {
14       newL.ptr = oldL.ptr;
15       newL.deleted = true;
16       if (DCAS(&SR->L, &oldL.ptr->value,
17                oldL, v, newL, "null"))
18         return v;
19     }
20   }
21 }
```

Figure 11: The linked-list-based deque - right side pop.

Finally, there is the case in which the `deleted` bit is false and `v` is not null, as depicted in Figure 12. Using a DCAS,

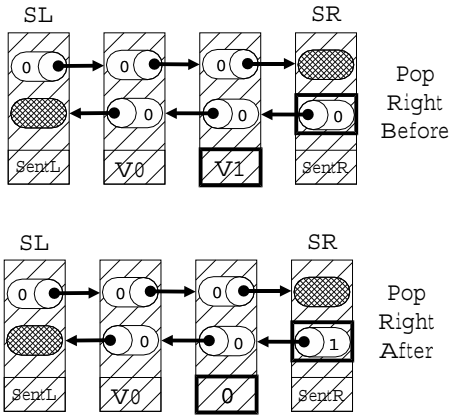


Figure 12: Non-null node before and after being popped by a `popRight`.

```
 1 val pushRight(val v) {
 2   newL.ptr = new Node();
 3   if (newL.ptr == "null") return "full";
 4   newL.deleted = false;
 5   while (true) {
 6     oldL = SR->L;
 7     if (oldL.deleted == true)
 8       deleteRight();
 9     else {
10       newL.ptr->R.ptr = SR;
11       newL.ptr->R.deleted = false;
12       newL.ptr->L = oldL;
13       newL->value = v;
14       oldLR.ptr = SR;
15       oldLR.deleted = false;
16       if (DCAS(&SR->L, &SR->L.ptr->R,
17                oldL, oldLR, newL, newL))
18          return "okay";
19       }
20   }
21 }
```

Figure 13: The linked-list-based deque - right side push.

the processor atomically swaps `v` out from the node, changing the value to null, and at the same time changing the `deleted` bit in the pointer to it in `SR` to true (Lines 14-17). If the DCAS is successful (Line 18), the processor returns `v` as the result of the pop, leaving the deque in a state where the right sentinel's `deleted` bit is true, indicating that the node is logically deleted. The next `popRight` or `pushRight` will call the `deleteRight` procedure,[3] to complete the physical deletion. If the DCAS fails, then concurrent pushes or pops must have modified the sentinel pointer, so that it no longer points to the node whose delete was attempted. The processor should loop back to retry the pop.

The code for the `pushRight` procedure appears in Figure 13. The processor starts, as depicted in Figure 14, by allocating a new node pointed to by a variable `newL` (Lines 2-4). If the allocation fails, a "full" indicator is returned. Otherwise, the processor checks if the `deleted` bit in the right sentinel is true (Lines 6-7). If so, as depicted in Figure 15, it calls (in Line 8) the `deleteRight` procedure, which removes the null node to which the sentinel points, prior to continuing with the push. If the `deleted` bit is false, then the processor fills in the pointers in the `newL` node to point

[3]Note that the `popRight` operation could also call the `deleteRight` procedure before returning `v`.

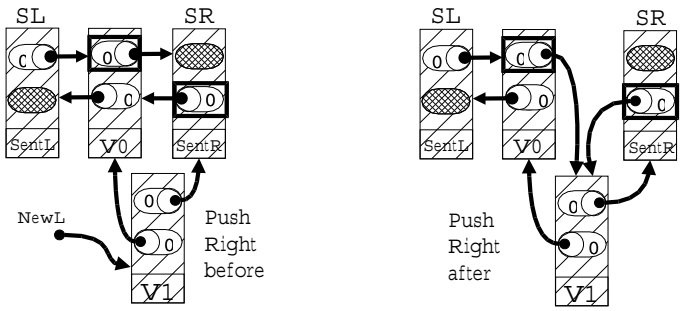


Figure 14: Before `pushRight` (on the *left*) and after a successful `pushRight` (on the *right*).

to the right sentinel and left neighbor of that sentinel (Lines 10-13).

The processor then tries to insert the new node into the list by using a DCAS to redirect the pointers in the sentinel and its left neighbor to the new node, as depicted in Figure 14 (Lines 14-17). If the DCAS is successful, the node is added (Line 18), and otherwise it must be the case that the deque was changed concurrently, and so the processor must loop and try to push the node again.

The code of the `deleteRight` procedure appears in Figure 17. The procedure begins by checking that `oldL`, the left pointer in the right sentinel, has its `deleted` bit set to true (Line 4). If not, the deletion has been completed by another processor and the current processor can return. If the bit is true, the next step is to determine the configuration of the deque. To this end, the processor reads `oldLL`, the node immediately to the left of the one to be deleted (Line 6). This node could be in one of three states: it may contain a null value, a non-null value, or the value `sentL`. The actions to be taken for the non-null value, as shown in the upper part of Figure 15, and for the `sentL` value, as shown in the upper middle part of Figure 9, are the same (Lines 6-13). The processor checks whether `oldL.ptr` (the pointer in `SR`) and `oldLLR.ptr` (the pointer in the neighbor of the node to be deleted) point to the same node (Line 8), the one to be deleted. If this is not the case, one must loop and try again since the deque has been modified. If they do point to the same node (Lines 9-13), then the processor uses a DCAS to attempt to redirect the pointers so that `SR` and `oldLL` point to each other, thereby deleting the null node pointed to by `oldL`. The case of the null value is somewhat different. A null value means that there are two null items in the deque, as shown in the bottom part of Figure 9. To delete both, the processor checks `oldR.deleted`, the `deleted` bit of the left sentinel, to see if the bits in both sentinels are true (Line 22). If this is the case, it attempts to point the sentinels to each other using a DCAS (Lines 23-24). In case the DCAS fails, the processor must go through the loop again, until the deletion is completed, i.e. the `deleted` bit is read as false.

The most interesting case occurs when the deque contains only two nodes, *left* and *right*, both logically deleted and thus containing the null value, and a `deleteLeft` is about to be executed concurrently with a `deleteRight`, as depicted in the top part of Figure 16. Assume that the processors reached this state by the following sequence of events. The `deleteLeft` operation started first, while the *right* node was still non-null. The `deleteLeft` then at-

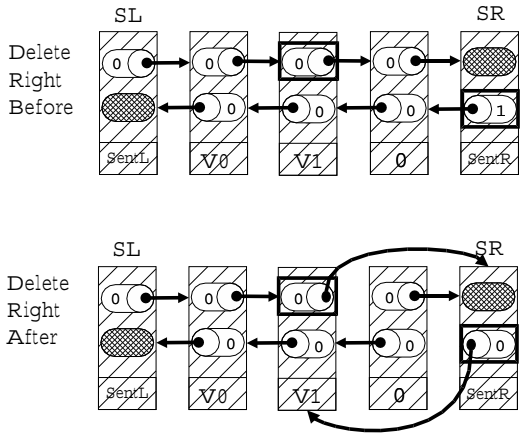


Figure 15: Null node before and after being deleted by a `deleteRight`.

tempted to remove the *left* node by performing a DCAS on the left sentinel's pointer and the *right* node's `L` pointer. The `deleteRight`, which started later, detected the two null nodes and attempted to remove both, by using a DCAS to redirect the pointers of the sentinels towards each other (Lines 17-25). As can be seen at the top of Figure 16, the DCAS operations performed by `deleteLeft` and `deleteRight` overlap on the pointer in the left sentinel. If the `deleteLeft` executes the DCAS first, the result of the success is a deque with one null node and the `deleted` bit of the right sentinel still true, as shown in the bottom left of Figure 16. If, on the other hand, the `deleteRight` executes the DCAS first, the result is an empty deque consisting only of the two sentinels, with their `deleted` bits false, as in the the bottom right of Figure 16.

Theorem 4.1 *The linked list based deque algorithm of Section 4 is a non-blocking linearizable implementation of a deque data structure.*

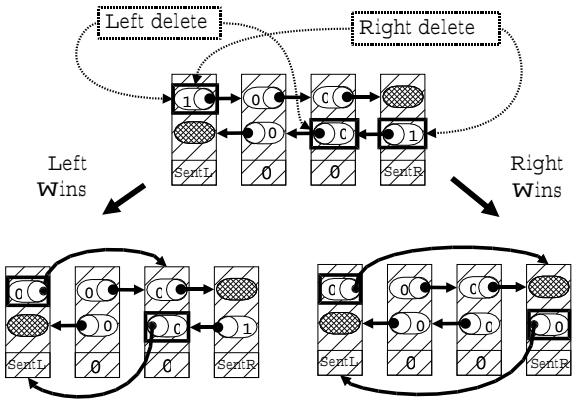


Figure 16: Contending `deleteLeft` and `deleteRight`.

```
1 deleteRight() {
2   while (true) {
3     oldL = SR->L;
4     if (oldL.deleted == false) return;
5     oldLL = oldL.ptr->L.ptr;
6     if (oldLL->value != "null") {
7       oldLLR = oldLL->R;
8       if (oldL.ptr == oldLLR.ptr) {
9         newR.ptr = SR;
10        newR.deleted = false;
11        if (DCAS(&SR->L, &oldLL->R,
12                 oldL, oldLLR, oldLL, newR))
13          return;
14        }
15    }
16    else {  /* there are two null items */
17      oldR = SL->R;
18      newL.ptr = SL;
19      newL.deleted = false;
20      newR.ptr = SR;
21      newR.deleted = false;
22      if (oldR.deleted)
23        if (DCAS(&SR->L, &SL->R,
24                 oldL, oldR, newL, newR))
25          return;
26    }
27  }
28 }
```

Figure 17: The linked-list-based deque - right side delete.

5 Acknowledgements

We would like to thank Steve Heller, Maurice Herlihy, Doug Lea, and the anonymous referees for their many suggestions and comments.

References

[1] Afek, Y., Merritt, M., Taubenfeld, G., and Touitou, D. Disentangling multi-object operations. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing* (August 1997), pp. 111–120. Santa Barbara, CA.

[2] Agesen, O., and Cartwright Jr., R. S. Platform independent double compare and swap operation, Dec. 1998. U.S. Patent Application Express Mail Number EL092132504US Ref P3368.

[3] Arora, N. S., Blumofe, B., and Plaxton, C. G. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures* (1998).

[4] Attiya, H., and Dagan, E. Universal operations: Unary versus binary. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing* (May 23-26 1996). Phila. PA.

[5] Attiya, H., Lynch, N., and Shavit, N. Are wait-free algorithms fast? *Journal of the ACM 41*, 4 (July 1994), 725–763.

[6] Attiya, H., and Rachman, O. Atomic snapshots in $O(n \log n)$ operations. *SIAM Journal on Computing 27*, 2 (Mar. 1998), 319–340.

[7] Barnes, G. A method for implementing lock-free shared data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures* (June 1993), pp. 261–270.

[8] Bershad, B. N. Practical considerations for non-blocking concurrent objects. In *Proceedings 13th IEEE International Conference on Distributed Computing Systems* (May 25–28 1993), IEEE Computer Society Press, pp. 264–273. Los Alamitos CA.

[9] Corman, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms*, 1st ed. McGraw Hill, 1989.

[10] Gosling, J., Joy, B., and Steele Jr., G. L. *The Java™ Language Specification*. Addison-Wesley, 2550 Garcia Avenue, Mountain View, CA 94043-1100, 1996.

[11] Greenwald, M. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, 8 1999.

[12] Greenwald, M. B., and Cheriton, D. R. The synergy between non-blocking synchronization and operating system structure. In *2nd Symposium on Operating Systems Design and Implementation* (October 28–31 1996), pp. 123–136. Seattle, WA.

[13] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufmann Publishers, 1995.

[14] Herlihy, M. Wait-free synchronization. *ACM Transactions On Programming Languages and Systems 13*, 1 (Jan. 1991), 123–149.

[15] Herlihy, M. A methodology for implementing highly concurrent data structures. *ACM Transactions on Programming Languages and Systems 15*, 5 (Nov. 1993), 745–770.

[16] Herlihy, M. P., and Moss, J. Transactional memory: Architectural support for lock-free data structures. Tech. Rep. CRL 92/07, Digital Equipment Corporation, Cambridge Research Lab, 1992.

[17] Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions On Programming Languages and Systems 12*, 3 (July 1990), 463–492.

[18] Knuth, D. E. *The Art of Computer Programming: Fundamental Algorithms*, 2nd ed. Addison-Wesley, 1968.

[19] Massalin, H., and Pu, C. A lock-free multiprocessor OS kernel. Tech. Rep. TR CUCS–005–9, Columbia University, New York, NY, 1991.

[20] Michael, M. M., and Scott, M. L. Correction of a memory management method for lock-free data structures. Tech. Rep. TR 599, Computer Science Department, University of Rochester, 1995.

[21] Motorola. *MC68020 32-Bit Microprocessor User's Manual*, 2nd ed. Prentice-Hall, 1986.

[22] Motorola. *MC68030 User's Manual*. Prentice-Hall, 1989.

[23] Rinard, M. C. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Transactions on Computer Systems 17*, 4 (Nov. 1999), 337–371.

[24] Shavit, N., and Touitou, D. Software transactional memory. *Distributed Computing 10*, 2 (February 1997), 99–116.

[25] Steele Jr., G. L. *Common Lisp*. Digital Press, Digital Equipment Corporation, 1990.

A Appendix: The matching left hand side code

```
 0 val popLeft {
 1   newS = "null";
 2   while (true) {
 3     oldL = L;
 4     newL = (oldL + 1) mod length_S;
 5     oldS = S[newL];
 6     if (oldS == "null") {
 7       if (oldL == L)
 8         if (DCAS(&L, &S[newL],
 9                  oldL, oldS, oldL, oldS))
10            return "empty";
11     }
12     else {
13       saveL = oldL;
14       if (DCAS(&L, &S[newL],
15                &oldL, &oldS, newL, newS))
16         return oldS;
17       else if (oldL == saveL) {
18         if (oldS == "null") return "empty";
19       }
20     }
21   }
22 }
```

Figure 18: The array based deque left-hand-side pop.

```
 1 val pushLeft(val v) {
 2   while (true) {
 3     oldL = L;
 4     newL = (oldL - 1) mod length_S;
 5     oldS = S[oldL];
 6     if (oldS != "null") {
 7       if (oldL == L)
 8         if (DCAS(&L, &S[oldL],
 9                  oldL, oldS, oldL, oldS))
10           return "full" ;
11     }
12     else {
13       saveL = oldL;
14       if (DCAS(&L, &S[oldL],
15                &oldL, &oldS, newL, v))
16         return "okay";
17       else if (oldL == saveL)
18         return "full";
19     }
20   }
21 }
```

Figure 19: The array based deque left-hand-side push.

```
1 val popLeft() {
2   while (true) {
3     oldR = SL->R;
4     v = oldL.ptr->value;
5     if (v == "SentR") return "empty";
6     if (oldR.deleted == true)
7       deleteLeft();
8     else if (v == "null") {
9       if (DCAS(&SL->R, &oldR.ptr->value,
10                oldR, v, oldR, v))
11        return "empty";
12    }
13    else {
14      newR.ptr = oldR.ptr;
15      newR.deleted = true;
16      if (DCAS(&SL->R, &oldR.ptr->value,
17                oldR, v, newR, "null"))
18        return v;
19    }
20  }
21 }
```

Figure 20: The linked-list based deque - left side pop.

```
1 val pushLeft(val v) {
2   newR.ptr = new Node();
3   if (newR.ptr == "null") return "full";
4   newR.deleted = false;
5   while (true) {
6     oldR = SL->R;
7     if (oldR.deleted == true)
8       deleteLeft();
9     else {
10      newR.ptr->L.ptr = SR;
11      newR.ptr->L.deleted = false;
12      newR.ptr->R = oldR;
13      newR->value = v;
14      oldRL.ptr = SL;
15      oldRL.deleted = false;
16      if (DCAS(&SL->R, &SL->R.ptr->L,
17                oldR, oldRL, newR, newR))
18        return "okay";
19      }
20  }
21 }
```

Figure 21: The linked-list based deque - left side push.

```
1 deleteLeft() {
2   while (true) {
3     oldR = SL->R;
4     if (oldR.deleted == false) return;
5     oldRR = oldR.ptr->R.ptr;
6     if (oldRR->value != "null") {
7       oldRRL = oldRR->L;
8       if (oldR.ptr == oldRRL.ptr) {
9         newL.ptr = SL;
10        newL.deleted = false;
11        if (DCAS(&SL->R, &oldRR->L,
12                  oldR, oldRRL, oldRR, newL))
13          return;
14      }
15    }
16    else {  /* there are two null items */
17      oldL = SR->L;
18      newR.ptr = SR;
19      newR.deleted = false;
20      newL.ptr = SL;
21      newL.deleted = false;
22      if (oldL.deleted)
23        if (DCAS(&SL->R, &SR->L,
24                  oldR, oldL, newR, newL))
25          return;
26    }
27  }
28 }
```

Figure 22: The linked-list based deque - left side delete.