DCAS-based Concurrent Deques Supporting Bulk Allocation

Paul Martin, Mark Moir, and Guy Steele

DCAS-based Concurrent Deques Supporting Bulk Allocation

Paul Martin, Mark Moir, and Guy Steele

SMLI TR-2002-111 October 2002

Abstract:

We present a lock-free implementation of a dynamically sized double-ended queue (deque) that is based on the double compare-and-swap (DCAS) instruction. This implementation improves over the best previous one by allowing storage to be allocated and freed in bulk when the size of the deque changes significantly, and to avoid invocation of the storage allocator at all while the size remains relatively stable. We achieved this implementation in two steps by first solving the easier problem of implementing the deque for a garbage-collected environment, and then applying the Lock-Free Reference Counting methodology we recently proposed in order to achieve a version independent of garbage collection.

M/S MTV29-01
2600 Casey Avenue
Mountain View, CA 94043

email addresses:
paul.martin@sun.com
mark.moir@sun.com
guy.steele@sun.com

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@sun.com>. All technical reports are available online on our Website, http://research.sun.com/techrep/.

DCAS-based Concurrent Deques Supporting Bulk Allocation

Paul Martin Mark Moir Guy Steele
Sun Microsystems Laboratories
1 Network Drive
Burlington, MA 01803

1 Introduction

A major goal of our group's research is to investigate scalable synchronization mechanisms for shared-memory multiprocessors. A particular focus of this effort is scalable mechanisms for sharing objects between concurrent threads. (By *object*, we simply mean a data structure and the operations by which it can be accessed.) Clearly some form of synchronization is needed between concurrent operations to avoid corrupting the object. Currently, this synchronization is almost always in the form of mutual exclusion.

The use of mutual exclusion for this purpose is the source of a variety of problems including susceptibility to faults and delays, performance bottlenecks, design complications, and, in real-time systems, priority inversion. In recent years, significant research attention has been given to the problem of implementing shared objects without resorting to the use of mutual exclusion. This research has focused on designing lock-free and wait-free implementations of shared objects. A *lock-free* implementation guarantees that after a finite number of steps of an operation on the object, some operation completes. A *wait-free* one makes the stronger guarantee that after a finite number of steps of an operation, *that* operation will complete. Both requirements preclude the traditional use of mutual exclusion because if a thread acquires a lock and then takes no more steps, no other thread can ever complete an operation. In this paper, we focus on lock-free implementations of shared objects, as we believe that the weaker requirement will usually be sufficient in practice.

The task of designing correct lock-free and wait-free implementations is notoriously difficult. Verifying the correctness of such implementations is also a considerable challenge. Herlihy was therefore motivated to propose the use of *universal constructions* [7], which take sequential code for the operations required, and automatically produce a lock-free or wait-free implementation of the desired object. Herlihy showed that so-called "universal" synchronization primitives, such as compare-and-swap (CAS) and the load-linked/store-conditional (LL/SC) instruction pair, are both necessary and sufficient for building universal constructions. His first universal construction was quite impractical, but significant progress has been made since towards more practical universal constructions (e.g., [2, 10]). Even after all of this work, the best universal constructions are still very complicated, and as a result, they are unlikely to perform well and are not widely deployed in practice.

There are two major reasons that state-of-the-art universal constructions are impractical. The first is that their generality comes at a price: it is not possible to apply object-specific optimizations because they are required to implement *any* object. However, as mentioned above, the difficulty of designing and verifying lock-free and wait-free implementations of shared objects makes it im-

practical for programmers to design a new, optimized lock-free implementation for each new object required. The second reason is that existing universal primitives—such as CAS and LL/SC—can access only a single variable atomically. This provides a significant challenge because the potentially complicated modifications made to a shared object by an operation must "appear" to happen atomically, and this must somehow be achieved using simple, single-variable atomic instructions.

It is natural to ask, therefore, whether synchronization primitives that can access multiple words atomically can improve the situation. One such primitive is double compare-and-swap (DCAS), which generalizes CAS in the natural way to two independently chosen locations. (DCAS is defined precisely in Section 2.) We believe that such primitives can help both by supporting the implementation of simpler, more practical universal constructions, and by making it easier to design optimized lock-free implementations of specific objects. Greenwald [5] has explored both directions by designing universal constructions based on DCAS and by considering implementations of specific objects using DCAS.

One object considered by Greenwald is a double-ended queue [9] (hereafter "deque"). This is an important object to study, as it generalizes the two most widely used concurrent objects, namely stacks and queues. In two previous papers, our group presented various improvements over Greenwald's deque implementations. The important advantages achieved by these implementations include allowing concurrent access to the two ends of the deque and allowing the deque to be dynamically resized. (See [1, 3] for details.) The latter advantage was first achieved in [3] by representing the deque as a doubly linked list, using an algorithm affectionately known as "Snark". This allows new values to be pushed onto the deque by allocating a new node and using DCAS to link the node into the doubly linked list; values are popped by using DCAS to unlink an end node from the list, and then freeing the node. The disadvantage of this approach is that it requires invocation of the storage allocator for every push and pop operation. This incurs an overhead and also causes the implementation to inherit any scalability problems of the storage allocator.

In this paper, we present a new deque implementation, which allows empty nodes to remain in the list to be reused later, while retaining all of the advantages over previous algorithms. This allows us to allocate and free nodes in bulk when the size of the deque changes significantly, and also to avoid invoking the storage allocator at all while the size remains relatively stable.

We designed our new deque implementation—which we have nicknamed "Hat Trick" because it employs tricks using pointers we call "hats"—using a two-step methodology we recently proposed. The idea is to first design an implementation assuming the existence of a garbage collector. This significantly simplifies the design process for two reasons. First, the problem of determining when a node can be freed is often difficult, even in sequential implementations, and garbage collection (GC) is widely recognised as being effective in relieving programmers of this responsibility. This difficulty—and therefore the benefit provided by GC in overcoming it—is even more pronounced in lock-free implementations, where we must be concerned not only with the state of the object, but also with the states of concurrently executing threads in deciding when nodes should be freed. The second reason is specific to lock-free implementations. A common difficulty in designing such implementations is that if a pointer value changes from a value A to another value B, and then subsequently changes back to A, a CAS or DCAS operation cannot detect that these changes occurred, and they can therefore succeed when they should have failed. This problem is commonly known as the "ABA problem", and often accounts for a significant amount of the complexity of a lock-free implementation. In some cases, GC gives us a "free" solution to this problem by preventing a node from being reallocated in the case that a thread still has a pointer to it. (Note that a thread that is trying to change a pointer value A to another value supplies A as

```
boolean CAS(val *addr0,
            val old0,
            val new0) {
  atomically {
    if (*addr0 == old0) {
      *addr0 = new0;
      return true;
    } else return false;
  }
}
```

```
boolean DCAS(val *addr0, val *addr1,
             val old0, val old1,
             val new0, val new1) {
  atomically {
    if ((*addr0 == old0) && (*addr1 == old1)) {
      *addr0 = new0; *addr1 = new1;
      return true;
    } else return false;
  }
}
```

Figure 1: Single and Double Compare-and-Swap Operations.

an argument to its CAS or DCAS operation, so it retains a copy of A until after the CAS or DCAS is executed, which prevents A from being prematurely collected and reallocated.)

The use of DCAS and the assumption of a garbage-collected environment together significantly simplify the task of designing lock-free implementations of shared objects (although our experience suggests that this is still far from trivial). However, such implementations are not applicable in environments that do not support GC. (A corollary of particular concern for our group is that such implementations cannot be used in the implementation of garbage collectors.) In [4], we proposed a methodology called Lock-Free Reference Counting (LFRC) for converting a large class of GC-dependent implementations to equivalent GC-independent ones, while preserving lock-freedom. In this paper we apply this methodology to achieve a GC-independent version of our bulk allocation deque implementation.

The rest of the paper is organized as follows: In Section 2, we present preliminary information such as definitions of DCAS, deques, and correctness conditions. In Section 3, we present our GC-dependent, lock-free bulk allocation deque implementation, and then describe how we applied the LFRC methodology to achieve a GC-independent version. Concluding remarks appear in Section 4. A proof overview is presented in an appendix.

2 Preliminaries

Our model of computation is described in detail in [3]; here we briefly repeat the most important aspects of this model, and also discuss the correctness conditions considered in this paper.

We assume a sequentially consistent shared-memory multiprocessor that supports CAS and DCAS operations; the semantics of these operations are specified in Figure 1.

The correctness condition we consider is linearizability [8], which essentially requires that each operation on the deque "appears" to take effect atomically at some point during its execution, and that the totally ordered sequence of operations produced by these "linearization points" is consistent with the specified sequential semantics.

Our deque implementation supports push_left, push_right, pop_left, and pop_right operations with the obvious sequential semantics (see [3] for a precise definition). In addition, in this paper, we introduce two new operations for removing surplus storage from each side of the deque. For the purposes of linearizability, the semantics of these operations is "skip". That is, they do not affect the state of the implemented deque abstract data type. However, they *are* intended to have side effects, and we need to somehow specify the desired properties of these side effects.

As explained in Section 1, the implementation presented in this paper improves over previous ones by allowing the deque representation to include additional storage that is not currently

used for values. Thus, we can avoid unnecessary calls to the storage allocator by keeping storage previously used by values for the use of future values pushed onto the deque. This gives rise to the possibility that excessive storage will be retained by the implementation in the case that the size of the abstract deque becomes much smaller. For this reason, we provide remove_left and remove_right operations to remove all but a specified amount of storage from respective ends of the deque. We made the design decision that it would require excessive overhead to make these operations *guarantee* to ensure that at most the specified amount of storage is maintained. Instead, our implementation attempts to retain at most the specified amount of storage, and returns a boolean value, which roughly speaking indicates whether the operation was successful in ensuring that there is not excessive storage retained. However, the only guarantee made by our implementation is that if a remove operation executes without interference from any concurrent operations, then it will ensure that at most the specified amount of storage is retained. Even in this case, we cannot guarantee that the removed storage is actually freed, because if some thread is still accessing the chopped-off storage, then the storage cannot yet be collected or freed. This also implies that if a thread dies permanently, it may permanently prevent some storage from being freed. There is no way around this problem, short of sophisticated operating system support to identify and clean up after dead threads.

3 The Hat Trick Deque

In this section, we first describe our GC-dependent deque implementation, and then explain how we transformed it to an equivalent GC-independent version.

3.1 GC-Dependent Implementation

Our deque implementation is based on a doubly linked list representation. Each node contains left and right pointers denoted L and R, and a value field V, which can contain values pushed onto the deque, or one of the special values LX, LY, LN, RX, RY, and RN. (It is assumed that these values are never pushed onto the deque.) The list contains one node for each value in the deque, plus additional nodes that can be used for values in the future, and are also used for the synchronization involved with growing and shrinking the list.

The values currently in the deque are stored in consecutive nodes in the list, and there is always at least one additional node on each side of the values; each node other than those containing values is distinguished by one of the special values listed above. The node directly to the right (left) of the rightmost (leftmost) value is called the right (left) sentinel. Two shared pointers, RHat and LHat, always point to the right and left sentinels, respectively (except in one special case, which is described later). In the case that there are no values in the deque, the left and right sentinels are adjacent. Figure 2(a) depicts a representation of the empty deque (we will see an alternative representation soon). Our implementation is completely symmetric; we therefore restrict our presentation to the right side henceforth.

The sequence of special values obtained by following R pointers from the node containing the rightmost value (i.e., the values in the right sentinel and any nodes to the right of it) consists of zero or more "right null" values, distinguished by the RN value, followed by a "right terminator" value, distinguished by RX. To the right of the first RX node, there are zero or more nodes, which can be marked by RN, RX, or RY.[1] These nodes (if any) exist because of a previous shrinking

[1] Actually, stronger statements can be made—and are necessary for the correctness proof—about the values in the nodes to the right of the right sentinel, but this description is sufficient for a basic understanding of the algorithm.

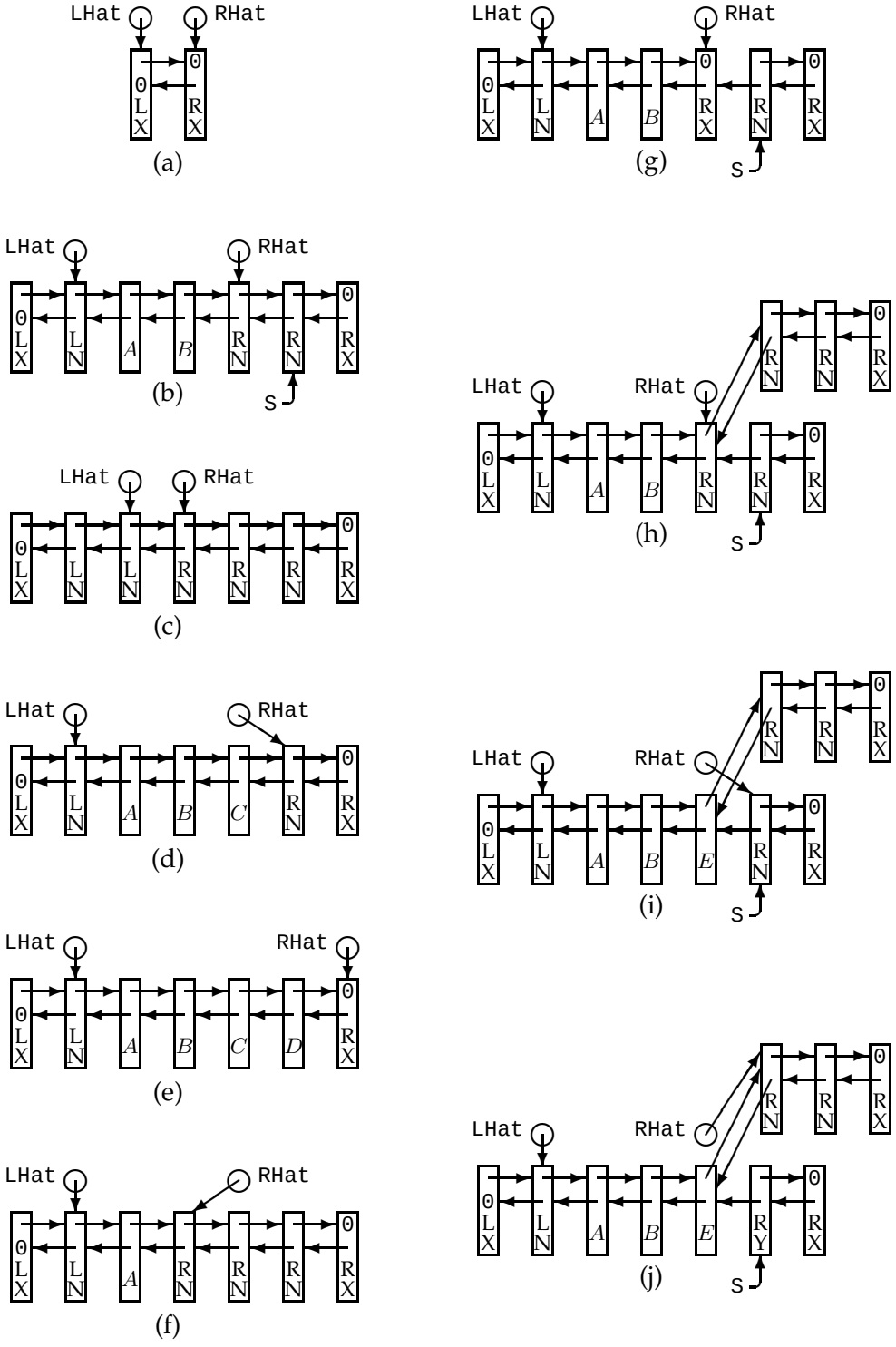

Figure 2: Example states. (a) An empty deque. (b) A "typical" deque containing values A and B. (c) An empty deque with spare nodes. (d) After push_right(C). (e) After push_right(D). (f) After pop_right. (g) After remove_right(0). (h) After add_right(2). (i) Delayed push_right DCAS has moved RHat onto a "spur". ((g) through (i) show states of the natural, but incorrect, algorithm; see text for details.) (j) After unspur_right.

operation, which is explained later. As described below, we use the terminating RX node to avoid further use of these nodes so that they can eventually become garbage.

Figure 2(b) shows a "typical" deque, containing two values A and B (ignore the "S" label for now). The right null nodes (i.e., those marked by RN) between the rightmost value and the first RX node are "spare" nodes, which can be used for new values that are pushed from the right. Because these spare nodes do not contain values, Figure 2(c) shows another representation of the empty deque, this one containing spare nodes onto which values can be pushed in the future.

As stated earlier, the algorithm is symmetric, so below we explain only the right-side operations. We begin by describing the basic operation of push_right and pop_right. Then we describe the operations used to add new nodes (add_right) and to remove unwanted nodes (remove_right). Finally, we complete the description of our GC-dependent algorithm by identifying a special case that can arise from concurrent push_right and add_right operations, and explaining how our algorithm deals with this case. The code for the Hat Trick algorithm is shown in Figure 3.

Basic operation of push_right

The basic idea behind the push_right operation is to turn the current right sentinel into a value node, and to change RHat to point to the node to the right of the current right sentinel, thereby making it the new right sentinel. This is achieved by reading RHat to locate the right sentinel (line 2); determining the node to the right of the right sentinel (line 3); and then using DCAS to atomically move RHat to the next node and to change the value in the previous right sentinel from RN to the new value (line 5). For example, starting from the state shown in Figure 2(b), push_right(C) results in the state shown in Figure 2(d). A subsequent push_right(D) would similarly result in the state shown in Figure 2(e). Thus, the RX terminator can become the sentinel. However, a further push_right would fail to find a spare node marked by RN, and so this simple scenario would not succeed in pushing the new value. In fact, there are several possible reasons that the simple push_right operation described above might not succeed, as discussed below.

First, the DCAS can fail due to concurrent operations, in which case, push_right simply retries. This can happen only if another operation succeeds in changing the deque during this push_right operation, so lock-freedom is not compromised by retrying. Furthermore, push_right might not succeed because it detects that there is no node available to become the new right sentinel (line 4), or because the value in the old sentinel is not RN (in which case the DCAS will fail). In this case it may be that we have exhausted the right spare nodes; push_right checks for this case at line 8 by checking if the right sentinel contains the terminator RX. If so, it calls add_right (line 9) to grow the list to the right before retrying. The add_right procedure—and the special case dealt with at line 11—are described later.

Basic operation of pop_right

The basic idea behind the pop_right operation is to locate the rightmost value node, and to turn this node into the right sentinel by using DCAS to atomically change its value to RN and to move the RHat to point to it. Thus, for example, a simple pop_right operation performed starting from the state shown in Figure 2(b) results in the state shown in Figure 2(f). A "normal" pop_right operation begins by reading RHat to locate the right sentinel (line 39), and then reading the L pointer of this node in order to locate the rightmost value node (line 40). It then reads the value stored in this node (line 42). It can be shown that this value could be RN, RX, LY, or RY only in the presence of concurrent successful operations, so pop_right retries if it read any of these values (line 43). If pop_right read LN or LX from the value field, then either the deque is empty (i.e., there

```
class Node {
    class Node *L, *R;
    valtype V;
    Node(v){L = R = NULL; V=v;} }


push_right(valtype v) {
1   while (true) {
2     rh = RHat;
3     rhR = rh→R;
4     if (rhR != NULL &&
5         DCAS(&RHat, &rh→V,
6           rh, RN, rhR, v))
7       return OKval;
8     else if (rh→V == RX) {
9       if (!add_right(some_number))
10        return FULLval;
11    } else unspur_right(); } }


add_right(int n) {
12 chain = alloc_right(n);
13 if (chain == NULL) return false;
14 while (true) {
15    rptr = RHat;
16    while (rptr !=NULL &&
17        (v = rptr→V) == RN)
18      rptr = rptr→R;
19    if (v == RY)
20      unspur_right();
21    else if (rptr != NULL &&
22        v == RX) {
23      chain→L = rptr;
24      rrptr = rptr→R;
25      if (DCAS(&rptr→R, &rptr→V,
26          rrptr, RX, chain, RN))
27        return true; } }


alloc_right(int n) {
28 last = new Node(RX);
29 if (last == NULL) return NULL;
30 for (i=0; i<n; i++) {
31    newnode = new Node(RN);
32    if (newnode == NULL) break;
33    newnode→R = last;
34    last→L = newnode;
35    last=newnode; }
36 last→L = NULL;
37 return newnode; }
```

```
pop_right() {
38 while (true) {
39    rh = RHat;
40    rhL = rh→L;
41    if (rhL!=NULL) {
42      result = rhL→V;
43      if (result != RN &&
44          result != RX &&
45          result != LY && result != RY)
46        if (result == LN ||
47            result == LX) {
48          if (DCAS(&RHat, &rhL→V,
49              rh, result, rh, result))
50            return EMPTYval;
51        } else if (DCAS(&RHat, &rhL→V,
52              rh, result, rhL, RN))
53            return result; } } }

remove_right(int n) {
54 chop = RHat;
55 for (i=0; i<n; i++) {
56    if (chop→V == RX)
57      return true;
58    chop = chop→R;
59    if (chop == NULL) return true; }
60 rptr = chop→R;
61 if (rptr == NULL) return true;
62 if (v = DCAS(&chop→V, &rptr→V,
63      RN, RN, RX, RY)) {
64    CAS(&chop→R, rptr, NULL);
65    break_right(rptr); } }
66 return v;

unspur_right() {
67 rh = RHat;
68 if (rh→V == RY) {
69    rhL = rh→L;
70    ontrack = rhL→R;
71    if (ontrack != NULL)
72      CAS(&RHat, rh, ontrack); } }

break_right(p) {
73 v = RY;
74 q = p→R;
75 while (v != RX && q != NULL) {
76    do {
77      v = q→V;
78    } until (CAS(&q→V,v,RY));
79    p→R = NULL;
80    p = q;
81    q = p→R; }
82 p→R = NULL; }
```

Figure 3: Code for right Hat Trick operations. Left operations are symmetric.

are no values between the two sentinels) or pop_right read inconsistent values due to concurrent operations. In this case, pop_right uses DCAS to check whether the values read from RHat and the V field of the rightmost value node exist simultaneously in these locations (line 48). (Note that the last two arguments to the DCAS are the same as the second two, so the DCAS does not change any values; it just checks that the values are the same as the ones read previously.) If so, pop_right returns "empty" (line 50); otherwise it retries. Finally, if pop_right did find a value in the node to the left of the right sentinel, then it uses DCAS to attempt to atomically change this value to RN (thereby making this node available for subsequent push_right operations) and to move RHat to point to this node (line 51). If the DCAS succeeds in atomically removing the rightmost value and making the node that contained it become the new right sentinel, the value can be returned (line 53); otherwise, pop_right retries.

Interaction between operations on opposite ends

While the deque contains more than one value, the left operations behave symmetrically to the right ones, and there is no interaction between them. This is an advantage over some previous DCAS-based deque implementations (e.g., [5]), which do not allow left and right operations to execute concurrently without interfering with one another. However, when there are zero or one values in the deque, the pop_left and pop_right operations do interact. In particular, if these two operations execute concurrently while only one value is in the deque, then one of the operations should succeed in popping the value, while the other should receive an indication that the deque is empty (assuming no other concurrent operations). Our implementation handles this case correctly because the pop operations use DCAS to change the value they are attempting to pop to a null value (LN or RN, depending on the side we're popping from). Whichever of the two operations executes its DCAS first succeeds, and the other fails because there is no longer a non-null value.

The add_right procedure

We next explain the add_right procedure, which is used to add new null nodes to the right of the linked list for use by subsequent push_right operations. This procedure can be called directly if desired. It is also called by push_right if it determines that there are no more right null nodes available (by observing RX in the right sentinel at line 8). add_right takes an argument indicating the number of nodes to add; we do not address the issue of how this number is chosen here. add_right begins by calling alloc_right to construct a doubly linked chain of the required length; the rightmost node in this chain contains RX, and all others contain RN. alloc_right is straightforward because no other process can concurrently access the chain as it is constructed from newly allocated nodes.

Next, add_right attempts to splice in the new chain by replacing the R pointer in the right terminator node with a pointer to the new chain, and replacing the RX in the terminator node with RN, so that it becomes just another spare node, with the rightmost node of the new chain becoming the new terminator node. These replacements are achieved atomically using DCAS at line 25. In preparation for this, add_right first "walks" down the chain from the right sentinel past all of the RN nodes, looking for the RX terminator (lines 15 through 18). When it finds the RX terminator, add_right attempts to splice its new chain onto the existing one, as described above, by using DCAS (line 25). In preparation, add_right first makes the leftmost node of its new chain point back to the current RX terminator (line 23) so that, if the DCAS succeeds, the doubly linked list will be complete, and then reads the current R pointer (line 24).

Because of the possibility of concurrent operations, the search at lines 16 to 18 could encounter any value, real or special, before finding the RX terminator. In most such cases, add_right simply begins its search again (as usual, this does not compromise lock-freedom because the concurrent operations that caused this to happen must have succeeded). However, there is one special case in which we cannot simply retry (see lines 19 and 20); this case is explained later.

Next we explain the remove_right operation—which is used to remove all but a specified number of the spare right nodes—and then we explain the special case alluded to above, which can arise from concurrent push_right and add_right operations.

The remove_right operation

The remove_right operation is invoked with a number that indicates the maximum number of spare nodes that should remain on the right side of the list. If a remove_right operation fails to chop off part of the list due to concurrent operations, it may be that the decision to chop off the additional nodes was premature. Therefore, rather than insisting that remove_right retry until it is successful in ensuring there are no more spare nodes than specified, we allow it to return false in the case that it encounters concurrent operations, leaving the decision of whether to retry the remove_right operation in this case to the user. In fact, the decision of when to invoke the remove operations, and how many nodes are specified to remain, is also left to the user.

We begin by discussing a natural (but incorrect) approach to removing spare nodes, and then explain how this approach fails and how we address this problem.

remove_right first walks down the list from the right sentinel, counting the null nodes that will not be removed, as specified by the argument to remove_right (lines 54 to 59). If it reaches the end of the list (line 59), or a node containing RX (line 56), before counting the specified number of right null nodes, then it can simply return true, as there are no excess nodes to be excised. Otherwise, it reaches an RN node and we would like to chop off the remaining nodes. The natural approach is to simply use a DCAS to change the R pointer of this node to null, thereby making nodes to the right of this one available for garbage collection, and to change the RN to RX, thereby preserving the invariant that an RX terminator exists. However, careful examination of the resulting algorithm reveals a problem, illustrated by the scenario described below.

A special case: "spurs"

Consider a push_right(E) operation that runs alone from the state shown in Figure 2(b), and stops just before it executes its DCAS at line 5. If the DCAS is executed and succeeds, then the new value will be stored in the current right sentinel, and RHat will be changed to point to the node (labeled S in Figure 2(b)) to the right of the current right sentinel. However, note that the DCAS does not access the R pointer of the current right sentinel. Thus, the DCAS may succeed even if this pointer changes, and this is the root of the problem. Suppose now that remove_right(0) is invoked (using the above-described natural but incorrect approach), and it runs alone to completion, resulting in the state shown in Figure 2(g). If the DCAS of the previously suspended push_right executed now, it would fail because the value in the sentinel node is not RN. However, further suppose that add_right(2) is invoked at this point, and runs alone to completion. This results in the state shown in Figure 2(h). If the DCAS of the push_right operation executes at this point, it will succeed, resulting in the state shown in Figure 2(i). Observe that the RHat pointer has failed to move along the linked list, and has instead gone onto a "spur". This can result in incorrect behaviour because subsequent values pushed onto the right side of the deque can never be found by pop_left operations.

Our approach to dealing with this problem is not to avoid it, but rather to modify our algorithm so that we can detect and correct it. The idea is to separate the removal of nodes into two steps. In the first step, in addition to marking the node which will become the new right terminator with RX, we also mark its successor (S) with the special value RY. The RY value is stable: it marks a node as forever dead and prevents subsequent values from being pushed onto the node, and also prevents new chains of nodes from being added onto the node. Changing the two adjacent nodes to have RX and RY values respectively is achieved atomically using DCAS (line 62). This "logically" chops off the rest of the list, but the pointer to node S is still intact so, in the second step, we change this pointer to null, allowing the chopped off portion of the list to eventually be garbage collected (line 64).[2] The modified algorithm can be understood by replacing RN with RY in node S in Figures 2(g) through 2(i).

Marking node S with RY prevents further pushes from proceeding down the old chain (the DCAS at line 4 will fail because it will not find RN in node S). It also allows processes that are attempting to push values to detect that RHat has gone onto a spur and to rectify the problem by invoking unspur_right (line 11) before retrying the push operation. unspur_right simply verifies that RHat still points to a node labeled with RY (lines 67 and 68) and then follows the still-existing pointer from S back to its predecessor (line 69), determines the correct right-neighbour (line 70), and then uses CAS to move RHat to the correct node (line 72). Thus, Figure 2(j) shows the result of executing unspur_right from the state shown in Figure 2(i) (modified so that node S contains RY, as described above). unspur_right is simple because nothing except unspurring can happen from a spurred state: the RY prevents further push_rights from completing without first calling unspur_right and pop_rights naturally move off the spur. pop_lefts also pose no problem if they reach the node where the spur occurred, as they will see an RN in the first node of the newly added chain, and will correctly conclude that the deque is empty.

3.2 Obtaining a GC-independent Implementation

We have described our entire deque implementation, except for the break_right function invoked at line 65. The implementation described thus far (that is, *without* break_right) is correct in a garbage-collected environment, as the chains broken off the list (either by removing nodes or by adding new ones) become unreachable and can therefore be collected. However, as discussed in Section 1, we are also interested in an implementation that does not depend on garbage collection. We complete this section by explaining how our implementation can be transformed into one that does not depend on garbage collection.

Our approach is based on the general-purpose methodology called LFRC (Lock-Free Reference Counting) recently proposed by our group for transforming implementations that depend on garbage collection into equivalent ones that do not [4]. The only non-trivial aspect of applying this methodology is ensuring that no garbage cycles exist. (The methodology is based on reference counting, which cannot collect cycles in garbage.)

Clearly, as described so far, our implementation does allow cycles in garbage, because the chains cut off the list by remove_right are doubly linked. Therefore, in preparation for applying the LFRC methodology, we modified our implementation so that cycles are broken in chains that are cut off from the list. This is achieved (on the right side) by break_right, which is invoked after

[2]Observe that, if the process that invoked remove_right failed permanently between the DCAS at line 62 and the CAS at line 64—or indeed if it failed after the CAS, but before returning from remove_right, thereby keeping its local pointer rptr to the rest of the list—then the chopped off nodes would never be garbage collected. As discussed earlier, without sophisticated operating system support for cleaning up after a process dies, this behaviour is unavoidable.

successfully performing the DCAS at line 62. The intuition is that the process that broke off the chain is responsible for breaking its cycles.

The basic idea of break_right is straightforward: simply walk down the chain, setting the forward (i.e., the R pointer in the case of break_right) pointers to null. However, there are some subtleties involved with breaking these cycles; as a result, it was significantly more challenging to prepare our Hat Trick implementation for the LFRC methodology than it was to prepare the Snark algorithm, which was the first implementation to which we applied LFRC [3, 4]. The reason is that we need to deal with the possibility of concurrent accesses to the broken off chain while we are breaking it, because some processes may have been accessing it before it was cut off. Concurrent pop and push operations pose no problem, as their DCASs will not succeed when trying to push onto or pop from a cut-off node because the relevant hat no longer points to it. Also, these operations check for null pointers and take appropriate action, so there is no risk that setting the forward pointers to null will cause concurrent push and pop operations to dereference a null pointer, for example. However, dealing with concurrent remove and add operations is less trivial, as it is possible for both types of operation to modify the chain we are attempting to break up.

First, simply walking down the list setting forward pointers to null (presumably using the detection of a null forward pointer as a termination condition), does nothing to prevent another process from adding a new chain onto any node in the chain that contains an RX value. If this happens, the cycles in this newly added chain will never be broken, resulting in a memory leak. Also, this naive approach can result in multiple processes concurrently breaking links in the same chain in the case that one process executing remove_right succeeds at line 62 in chopping off some nodes within an already chopped-off chain. This results in unnecessary work and more difficulty in reasoning about correctness.

We address both of these problems with one technique: before setting the forward pointer of a node to null (line 79), we first use CAS to set the node's value field to RY (lines 76 to 78). The reason for using a CAS instead of an ordinary store is that we can determine what value we overwrote when storing RY. If the CAS changes an RX node to RY, then the loop terminates (see line 75). The reason that it's safe to terminate in this case is that either the RX was in the rightmost node of the chain (and changing the RX to an RY prevents a new chain being added to it subsequently), or some process set the value of this node to RX at line 62, in which case that process has the responsibility to break the cycles in the rest of the chain.

Ensuring that the cycles are broken in chopped off chains was the only challenging part of achieving a GC-independent deque implementation using the LFRC methodology. Otherwise, we just needed to follow the steps explained in [4].

4 Concluding Remarks

We have presented a new lock-free implementation of a double-ended queue. This DCAS-based implementation allows concurrent accesses to the two ends of the deque and significantly improves on the best previous implementation by allowing nodes to be reused, thereby avoiding unnecessary invocations of the storage allocator. In our implementation, deque operations that do not encounter contention and do not need to add more nodes to the list complete after a small number of loads and stores, and one DCAS.

We achieved our new implementation in two steps, following the methodology we recently proposed in which we first solve the easier problem of implementing the desired object under the assumption of garbage collection, and then applying our LFRC scheme to achieve an implementation that does not depend on garbage collection. This provides further evidence of the power of

this approach.

We have implemented the Snark [3] and Hat Trick deques in C++, using fine-grained mutual exclusion to implement DCAS. Although we have not conducted a thorough performance study, we have compared the performance of these two deque implementations by means of a simple benchmark run on a Sun E6500 with 16 400MHz UltraSPARC® II processors and 16GB memory running the Solaris™ 2.8 Operating Environment. We implemented all memory accesses (loads, stores, CASs, and DCASs) using Solaris™ mutexes to ensure that our implementations behaved correctly in the weak memory consistency model (recall that our algorithms are designed under the assumption of sequential consistency). This is undoubtedly excessive synchronization, so there is room for significant optimization.

In our benchmark, each of 16 threads invoked 50,000 operations in a tight loop (choosing at random between push and pop and between left and right); no remove operations were invoked in the case of the Hat Trick implementation. Our Hat Trick implementation completed the benchmark approximately 33% faster than the Snark implementation did. While these results indicate that our approach of avoiding unnecessary invocations of the storage allocator results in significant improvement, we believe the potential benefits are in fact much greater. First, while the Snark implementation invokes the allocator once for each push operation (i.e., about 400,000 times over the entire benchmark), the Hat Trick allocated only about 1200 nodes over the benchmark. The simple nature of our benchmark probably does not stress the storage allocator as much as a less regular application would. In particular, our benchmark only requires the allocator to repeatedly allocate and free objects of the same size, with no interference from other, unrelated invocations of the allocator, and with plenty of heap memory available. In a less regular setting, however, the allocator is likely to impose a greater overhead, which will negatively impact the performance of Snark much worse than that of Hat Trick. Second, we expect this improvement would be more pronounced in highly optimized versions of these algorithms, as time spent in the storage allocator would contribute a greater fraction of the overall running time in this case.

One possible criticism of our work is its reliance on the DCAS instruction, which is not widely supported in existing architectures. We believe our work in this direction is valuable for two reasons. First, enhanced synchronization support (such as DCAS or some other mechanism) will never be added to future architectures unless there is a convincing body of work showing how it could be exploited if it were added. Second, and of more immediate value, is that the insights gained through simplifying problems by assuming stronger synchronization support might transfer to algorithms that do not require such assumptions. Our work on DCAS-based lock-free algorithms has already lead to new results that can be implemented in existing architectures.

Acknowledgments: We are grateful to Victor Luchangco for useful feedback.

References

[1] O. Agesen, D. Detlefs, C. Flood, A. Garthwaite, P. Martin, M. Moir, N. Shavit, and G. Steele. DCAS-based concurrent deques. *Theory of Computing Systems*, 2002. To appear. A preliminary version appeared in the Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures.

[2] J. Anderson and M. Moir. Universal constructions for large objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317–1332, 1999. A preliminary version appeared in the Proceedings of the 9th Annual International Workshop on Distributed Algorithms, 1995.

[3] D. Detlefs, C. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. Steele. Even better DCAS-based concurrent deques. In *Proceedings of the 14th International Conference on Distributed Computing*, pages 59–73, 2000.

[4] D. Detlefs, P. Martin, M. Moir, and G. Steele. Lock-free reference counting. *Distributed Computing*, 2002. To appear. A prelminary version appeared in the Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing, 2001.

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

[6] David Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921–930, December 1977.

[7] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

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

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

[10] M. Moir. Transparent support for wait-free transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319, 1997.

[11] Susan Speer Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Cornell University, Ithaca, New York, July 1975. Department of Computer Science TR 75-251.

A Proof Sketch

The techniques of Owicki [11] as exemplified by Gries [6] may be used to prove properties of concurrent algorithms that operate on a shared data structure. The strategy is to (1) demonstrate that the primitive operations used to alter the data structure are linearizable and therefore may be regarded as occurring separately in some linear order; (2) formulate an invariant about the data structure, typically expressed as a conjunction of propositions called "the invariants"; (3) prove that each operation within the program text, if executed, necessarily preserves the invariant; (4) prove that the invariant implies the desired properties.

A.1 Definitions

The Hat Trick data structure consists of one or more *parts*: a *main list*, a number of *spurs* (*left spurs* and *right spurs*), and a number of *new chunks* (*left new chunks* and *right new chunks*). These are made up of nodes that each contain pointers R and L and a value V. A pointer may either be null or point to a node. Six *special values* are used internally by the algorithm (the *right special values* RN, RX, RY and the *left special values* LN, LX, LY); they are distinct from any value that the user may ask to push onto the deque. There are two globally shared pointer variables RHat and LHat.

The algorithm and data structure are left-right symmetric. We use a special convention to save space: any sentence preceded by "§" should be understood to imply a second sentence that is identical except for the exchange of left and right (thus occurrences of "R pointer" will be read as "L pointer" in the second sentence; "LHat" will be read as "RHat"; "RN as "LN"; "leftmost" as "rightmost"; and so on).

Each node Q belongs to exactly one part of the data structure, according to the following rules: (a) If Q→V is not a special value, then Q belongs to the main list. (b) §If Q→V = RY, then Q belongs to a right spur. (c) §If (Q→V = RX or Q→V = RN) and (Q→L = NULL or Q→L→R ≠ Q), then Q belongs to a right new chunk. (d) §If (Q→V = RX or Q→V = RN) and Q→L ≠ NULL and Q→L→R = Q and Q→L→V is a left special value, then Q belongs to the main list. (e) §If (Q→V = RX or Q→V = RN) and Q→L ≠ NULL and Q→L→R = Q and Q→L→V is not a left special value, then Q belongs to the same part of the data structure as node Q→L.

A *doubly linked list* is a nonempty set of nodes with the following properties: (i) §There is exactly one node in the set whose R pointer does not point to a node in the set; this node is called the *rightmost* node of the doubly linked list. (ii) §Every node of the set can be reached by starting from the rightmost node and then chasing L pointers some number of times. (iii) §For every node P of the set except the rightmost, P→R→L = P. It follows that the nodes in a doubly linked list are linearly ordered in a left-to-right ordering—or, equivalently, a right-to-left ordering that is exactly the reverse of the left-to-right ordering. We may speak of the values contained in the nodes of a doubly linked list as being ordered accordingly.

Two nodes P and Q are said to be *adjacent* if either P→R = Q or Q→L = P.

The data structure *represents* a deque that contains the ordered list of n values $(V_1, V_2, \ldots, V_n)$ for $n \geq 0$ if and only if all of the following are true: (1) The main list is a doubly linked list. (2) The values of the main list, taken from left to right, are as follows: an LX value in the leftmost node; then some number (possibly zero) of LN values; then the values $V_1, V_2, \ldots, V_n$ (if any), in order; then some number (possibly zero) of RN values; and finally an RX value in the rightmost node. (3) §RHat points to a node in the main list or a spur. (4) §RHat→L points to the node in the main list that contains the rightmost actual stored value V_n, or a left special value if $n = 0$ (that is, the deque is empty).

A push action *occurs* when a thread successfully executes the DCAS on line 5. A pop action

occurs when some thread successfully executes the DCAS on line 48 or the DCAS on line 51. A remove_right action *occurs* when some thread successfully executes the DCAS on line 62.

A.2 Invariants and Supporting Properties

1. §For every node Q, if Q is not in a new chunk, then Q→R does not point to a node in a new chunk.
2. §For every node Q, if Q→V ≠ LY and Q→V ≠ LN and Q→R ≠ NULL, then Q→R→L = Q.
3. §For every node Q, if Q→V = RX and Q→R ≠ NULL, then Q→R→L = Q.
4. §For every node Q, if Q→V = RX and Q→R ≠ NULL, then Q→R→V = RY.
5. §For every node Q, if Q→V = RN or Q→V is not a special value, then Q→R ≠ NULL.
6. §Once the value of a node has become RY, that value never changes again.
7. §Once the value of a node has become RY, the L pointer of that node is non-null and never changes again.
8. §Once the value of a node has become RY, the R pointer of that node can change only by becoming null, after which it never changes again.
9. Each part of the data structure is a doubly linked list.
10. There is exactly one main list.
11. §Exactly one node of the main list contains the value RX, and this node is the rightmost node of the main list.
12. §The leftmost node of a right spur has the value RY; the rightmost node, if different from the leftmost node, has the value RX; and all other nodes (if any) have the value RN.
13. §The rightmost node of a right new chunk has the value RX; all other nodes (if any) have the value RN.
14. §The L pointer of the leftmost node of a right spur points to a node that is in a spur or in the main list.
15. §The R pointer of the rightmost node of the main list either is a null pointer or points to the RY node of a right spur.
16. §The R pointer of the rightmost node of a right spur either is a null pointer or points to the RY node of a right spur.
17. §The R pointer of the rightmost node of a right new chunk is always a null pointer.
18. §If the L pointer of a node A points to a node B, but the R pointer of B does not point to A, then either the value of A is RY and A is the leftmost node of a right spur, or the value of A is RN and A is the leftmost node of a right new chunk.
19. §If the R pointer of a node is null, then the value of the node is RX or RY.
20. Once two nodes are adjacent, they always remain adjacent.

21. A node that belongs to the main list or a spur will never belong to a new chunk at a later time.

22. A node that belongs to a spur will never belong to the main list at a later time.

23. The data structure represents a deque.

24. The data structure represents a deque produced by starting with an empty deque and performing on it a series of push, pop, and remove_right operations corresponding exactly, in order, to the series of push, pop, and remove_right actions that have occurred.

A.3 Proofs

The proof technique has been fruitful for us, exposing more than one error in early versions of our algorithm when some aspect of the proof failed to go through. As an example of the technique, we give here an almost complete proof of one of the properties: Once the value of a node has become RY, that value never changes again.

The operations that can change the data structure are on lines 5, 23, 25, 51, 62, 64, 72, 78, 79, and 82 (The DCAS on line 48 does not alter the data structure; it merely verifies two values simultaneously.)

Of these, only 5, 25, 51, 62, and 78 can alter a value.

Line 5 alters a value only if it is RN. In the dual routine push_left, line 5 alters a value only if it is LN.

Line 25 alters a value only if it is RX (or, in the dual routine add_left, if it is LX).

Line 51 will be executed only if the variable "result" does not contain a special value, because of the if statements that govern it; therefore this operation alters a value only if it is not a special value. The same holds in the dual routine pop_left.

Line 62 alters two values, but only if both are RN (in the dual routine add_left, if both are LN).

Line 78, if successful, causes a value to become RY; so if the value was RY before the operation, it will still be RY after the operation. (In practice, line 78 is never executed in circumstances where q→V has the value RY, but that fact is not needed for this proof.)

In the dual routine break_left, things get more complicated; line 78, if successful, causes a value to become LY, so it is necessary to prove that q→V cannot have the value RY when that line is executed. We sketch this proof and leave the details to the reader: (a) on entry to break_left, p points to a node that was once in a left spur; (b) another property (which needs to be proved separately) is that a spur node always remains a spur node; (c) induction on the while loop at line 75 shows that q always points to a node of a left spur; (d) a node whose value is RY always belongs to a right spur, by definition; (e) therefore q→V $\neq$ RY at line 78 of break_left.

Therefore no operation on the data structure will invalidate the property in question.

About the Authors

Paul Martin joined Sun Microsystems Laboratories in 1993 as founding PI and later co-PI of the Speech Applications project. He is currently a member of the Knowledge Technology Group, adapting and extending natural language and knowledge representation tools. Additionally, Paul works with the Scalable Synchronization Group, exploring lock-free interaction for concurrent processes. Before joining Sun, Paul pursued his research interest in using human language to communicate with computers in IBM's Austin Information Retrieval Tools group, in the Human Interface and Natural Language groups at MCC, and at SRI's Artificial Intelligence Center. He received his Ph.D. from Stanford University for research conducted at Stanford's AI Lab and at Xerox PARC, and his B.S. from North Carolina State in CS and EE.

Mark Moir received the B.Sc. (Hons.) degree in Computer Science from Victoria University of Wellington, New Zealand in 1988, and the Ph.D. degree in Computer Science from the University of North Carolina at Chapel Hill, USA in 1996. From August 1996 until June 2000, he was an assistant professor in the Department of Computer Science at the University of Pittsburgh. In June 2000, he joined Sun Microsystems Laboratories, where he is now the Principal Investigator of the Scalable Synchronization Research Group.

Dr. Moir's main research interests concern practical and theoretical aspects of concurrent, distributed, and real-time computing. His current research focuses on mechanisms for non-blocking synchronization in shared-memory multiprocessors.

Guy Steele is a Distinguished Engineer at Sun Microsystems Laboratories in Burlington, Massachusetts. He is responsible for research in programming languages, parallel algorithms, implementation strategies, and architectural and software support. He has worked with James Gosling and Bill Joy on the detailed specification of the Java programming language. He has published more than two dozen papers on the subject of the Lisp language and Lisp implementation, including a series with Gerald Jay Sussman that defined the Scheme dialect of Lisp.

He is an ACM Fellow and a Fellow of the AAAI; he was awarded the ACM Grace Murray Hopper Award in 1988 and a Gordon Bell Prize in 1990. He designed the original EMACS command set and was the first person to port TEX.

Prior to joining Sun, he was a senior scientist at Thinking Machines Corporation, a member of technical staff at Tartan Laboratories, and an assistant professor at Carnegie-Mellon University. He is a co-author of three books on programming languages: *Common Lisp: The Language*, *C: A Reference Manual*, and *The High Performance Fortran Handbook*. In March 2001, Guy was elected to the National Academy of Engineering