DCAS is not a Silver Bullet for Nonblocking Algorithm Design

Simon Doherty[‡] [†] David L. Detlefs[†] Lindsay Groves[‡] Christine H. Flood[†]
Victor Luchangco[†] Paul A. Martin[†] Mark Moir[†] Nir Shavit[†] Guy L. Steele Jr.[†]

[‡]*Victoria University of Wellington, PO Box 600, Wellington, New Zealand*
[†]*Sun Microsystems Laboratories, 1 Network Drive, Burlington, Massachusetts, USA*

ABSTRACT

Despite years of research, the design of efficient nonblocking algorithms remains difficult. A key reason is that current shared-memory multiprocessor architectures support only single-location synchronisation primitives such as compare-and-swap (CAS) and load-linked/store-conditional (LL/SC). Recently researchers have investigated the utility of double-compare-and-swap (DCAS)—a generalisation of CAS that supports atomic access to two memory locations—in overcoming these problems. We summarise recent research in this direction and present a detailed case study concerning a previously published nonblocking DCAS-based double-ended queue implementation. Our summary and case study clearly show that DCAS does not provide a silver bullet for nonblocking synchronisation. That is, it does not make the design and verification of even mundane nonblocking data structures with desirable properties easy. Therefore, our position is that while slightly more powerful synchronisation primitives can have a profound effect on ease of algorithm design and verification, DCAS does not provide sufficient additional power over CAS to justify supporting it in hardware.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Software/Program Verification; E.1 [**Data**]: Data Structures

General Terms

Algorithms, Design, Theory, Verification

Keywords

Multiprocessors, nonblocking synchronization, concurrent data structures, linked lists, lock-free, DCAS, double-compare-and-swap

1. INTRODUCTION

The traditional approach to designing concurrent algorithms and data structures is to use locks to protect data from corruption by concurrent updates. The use of locks enables algorithm designers to develop concurrent algorithms based closely on their sequential counterparts. However, several well-known problems are associated with the use of locks including deadlock, performance degradation in cases of high contention, and priority inversion [7].

A variety of nonblocking progress conditions can be used to characterise implementations that avoid the problems associated with locks [11, 14]. In this paper, we are mostly concerned with lock-free implementations, though we briefly discuss alternative conditions in Section 6. A *lock-free* implementation guarantees that after a finite number of steps of any operation on the data structure, some operation completes.

Herlihy [11] showed that it is possible to implement any shared data structure in a lock-free[1] manner using so-called *universal constructions*, which transform sequential implementations into equivalent nonblocking ones. These universal constructions are based on universal synchronisation primitives, such as *compare-and-swap* (CAS) or the *load-linked/store-conditional* (LL/SC) pair, which are widely available in modern shared-memory multiprocessor architectures. However, the generality offered by universal constructions comes at a price: data structures implemented using them are generally too expensive to be considered practical. As a result, there has been significant interest in *direct* nonblocking implementations of important data structures.

There have been some isolated successes in designing practical nonblocking implementations of some simple data structures using synchronisation primitives widely available in today's architectures [2, 25, 29]. However, these results depend on nontrivial insights, and in general the design of efficient nonblocking data structures remains difficult. As explained further below, a key reason is that the synchronisation primitives supported by current multiprocessor architectures (such as CAS and LL/SC) provide access to only a single location at a time.

A typical way to use CAS is to read the contents of a location and to then use CAS to attempt to atomically change the location from the value read to some new value. If a concurrent operation changes the contents of the location in

[1]Herlihy also showed this result for a stronger nonblocking progress condition called wait-freedom.

the meantime, then the CAS fails to modify the location,[2] so no harm is done and the operation can be retried. However, CAS cannot detect changes to locations other than the one it accesses, so delicate and subtle algorithmic tricks are often required to prevent processes from observing partial results of operations that modify more than one location.

With the hope of making nonblocking algorithm design significantly easier, researchers have recently investigated the use of synchronisation primitives that access *multiple* locations atomically. Much of this work has focussed on the DCAS primitive, a natural generalisation of the CAS primitive. DCAS allows a thread to atomically compare two memory locations to respective "old" values and to store respective "new" values to those locations if both comparisons succeed. DCAS is defined precisely in Figure 1.

In Section 2, we discuss a number of DCAS-based results which represent significant progress in the design of nonblocking data structures. However, we also show that, despite these positive results, many of the DCAS-based algorithms we discuss suffer from important limitations, and are often complicated and difficult to verify. *We therefore believe that the availability of DCAS would* not *achieve the goal of making the design of scalable and efficient nonblocking algorithms easy.*

In Sections 3 through 5, we present a detailed case study that illustrates the difficulties encountered in designing and verifying efficient nonblocking data structures using DCAS. We consider a lock-free deque implementation, known as "Snark" [3]. The Snark algorithm is one of a series of deque implementations developed to examine the utility of the DCAS operation. Although it was published with a detailed, semi-formal proof of correctness, an attempt to verify it more formally using PVS—a semi-automated verification system—has since shown it to be incorrect [5]. Section 3 describes the Snark algorithm and its bugs, and Section 4 describes a corrected version of the Snark algorithm. The corrected version has been verified with the help of PVS; Section 5 provides an overview of the verification effort. The design, correction, and eventual verification of this algorithm was a substantial undertaking.

In Section 6, we present a very simple lock-free deque implementation based on 3CAS (a compare-and-swap operation that can access *three* locations atomically). The substantial gap in difficulty between the DCAS- and 3CAS-based algorithms supports our position that the additional power of DCAS does not ease algorithm design sufficiently to justify its inclusion in future architectures. Because this case study addresses only a single example, we do *not* take the position that 3CAS *is* sufficiently powerful. We address this further and discuss other alternative approaches to simplifying nonblocking algorithm design in Section 6.

We summarise our position and conclude in Section 7.

```
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;
  }
}
```

Figure 1: Semantics of the DCAS operation

2. PREVIOUS DCAS-BASED RESULTS

In this section we discuss previous work that investigates the effect of DCAS on the ease of designing efficient nonblocking algorithms and data structures. We summarise some useful advances in nonblocking methods and data structures. However, we also point out various limitations and shortcomings of the results we discuss. Furthermore, as discussed below, most of these results are complicated and require subtle correctness proofs. In later sections, we argue that many of these problems could be overcome using synchronisation support slightly stronger than DCAS.

2.1 Historical overview

In early work in this area, Massalin and Pu constructed a nonblocking operating system kernel using DCAS [24]. Greenwald and Cheriton built another kernel that uses several DCAS-based data structures, and described a way to support DCAS in hardware [7, 8]. This work stimulated further investigation into the effect of using DCAS on the difficulty of nonblocking algorithm design [1, 3, 4, 6, 23].

As discussed below, the use of DCAS in this research has resulted in a number of interesting advances in techniques for designing nonblocking algorithms and data structures. The study of DCAS-based algorithms has also led to practical algorithms that do *not* depend on DCAS, and are therefore applicable in current architectures. For example, Detlefs *et al.* [4] show how to use DCAS to effect storage reclamation for nonblocking data structures; this work subsequently led Herlihy *et al.* to a practical solution that uses only widely available synchronisation primitives [13].

2.2 Program Transformations Using DCAS

There has been some success in using DCAS to develop interesting program transformations. We discuss two such transformations below.

Greenwald [6] presents a technique called *two-handed-emulation*, which allows many sequential programs to be transformed into nonblocking concurrent versions in a routine (but not mechanical) fashion. Using Greenwald's technique, a process wishing to execute an operation on a data structure first registers a descriptor for that operation. This descriptor contains a "program counter" that indicates how much of the operation has been executed. Any process can help to complete the operation by using DCAS to modify the data structure while simultaneously testing and incrementing the program counter, thereby ensuring that each step of the operation is executed exactly once.

While many sequential algorithms can be straightforwardly transformed into nonblocking versions using this technique, direct application of two-handed emulation has important drawbacks. First, local variables present a problem: because different threads may execute different steps of an operation, local variables set by one step must sometimes be made available to other threads. Greenwald discusses

[2]CAS-based algorithm design is complicated by the fact that this is not strictly true; see our discussion of the ABA problem in Section 3.

some techniques for achieving this, but these techniques are subtle and complicated, which undermines the goal of providing simple means for achieving nonblocking data structures. Furthermore, these techniques essentially amount to converting local variables to shared ones, which introduces significant overhead.

Second, parallelism can be compromised unless the implementation obtained by applying two-handed emulation is substantially modified. For example, Greenwald constructs a basic hash table implementation in which processes register operations in the header of the hash table. Thus, every process wishing to insert an element must first complete any other active operation before beginning its own, even if the operations access distinct buckets. Greenwald refines this basic implementation to one in which processes register descriptors on each bucket (in the same way as a hash table might be implemented with a separate lock for each bucket). However, this refinement significantly complicates the *resize* operation on hash tables. The resulting algorithm is as complicated as many algorithms developed without two-handed emulation, and so an effort to formally verify its correctness would be a significant undertaking.

Detlefs *et al.* [4] present a storage reclamation technique called *lock-free reference counting* (LFRC). LFRC allows unused storage to be reclaimed by maintaining reference counts on each object. The key problem addressed in this work is that an object may be deallocated after a new pointer to it has been created, but before the object's reference count has been incremented to reflect this creation. LFRC addresses this problem by using DCAS to confirm that some pointer to the object exists while incrementing its reference count.

LFRC solves an important problem in nonblocking algorithm design—deciding when objects can be safely deallocated.[3] However, LFRC has drawbacks. First, an expensive DCAS is required for every load of a pointer, and space must be reserved for reference counts on every object. Second, LFRC cannot be applied to programs that use some pointer operations, such as pointer arithmetic. Finally, because LFRC is based on reference counts, it does not reclaim garbage that contains cycles. The authors of [4] suggest that in many cases, it is possible to modify applications so that they no longer produce garbage cycles. However, doing so is not always straightforward, especially when dealing with nonblocking algorithms (see [23] for a nontrivial example).

2.3 Data Structures Using DCAS

Massalin and Pu [24] presented the earliest collection of concurrent DCAS-based data structures: LIFO stacks; FIFO queues; and linked lists. All have significant shortcomings: the stack implementation is not linearizable[4] [8]; the stack and queue are statically sized (i.e., their maximum size must be known and fixed in advance); and the queue does not handle boundary conditions in a nonblocking fashion. Later, Greenwald [8] describes several DCAS-based nonblocking data structure implementations, including array-based stacks and double-ended queues (deques), and dynamic-sized queues and doubly linked lists.

Deques generalise stacks and queues by allowing push and pop operations at both ends (see [1, 18] for a precise definition). Deques provide an excellent case study for nonblocking synchronisation because they involve all of the intricacies of stacks and queues, and offer the additional challenge of coordinating opposite-end pop operations competing to claim the last element. This task is further complicated by at least two desirable properties of deque implementations: exploiting the natural parallelism between opposite-end operations when the deque contains several elements, and supporting *dynamic-sized* deques that do not require an *a priori* bound on the size of the deque.

Greenwald's stack and queue implementations are extremely simple, exploiting the fact that every operation can be achieved by modifying only two locations (so each operation can be effected with a single DCAS operation). However, his deque has the disadvantages that it is statically sized and that operations at opposite ends of the deque interfere, even when it contains several elements.

Several DCAS-based deque implementations that improve on Greenwald's have been proposed [1, 3, 23], and are discussed below. Three of these implement dynamic-sized deques: a significant advantage for applications for which it is impossible or inconvenient to fix in advance the maximum deque size. In general, dynamic-sized data structures must reclaim storage that is no longer required; this is particularly challenging in nonblocking data structures. However, all of the dynamic-sized deques discussed below rely on garbage collection (GC) for reclamation. As a result, they are not universally applicable.[5]

Agesen *et al.* [1] present two deque implementations. The first, an array-based implementation, is very simple and uses one DCAS for each operation in the absence of contention. However, like Greenwald's implementation, it has the disadvantage of being statically sized. The second is a dynamic-sized implementation based on a doubly linked list. However, it requires two DCAS operations for every pop operation, and it "steals" a bit from every pointer to be used as a flag, which limits its applicability.

The Snark algorithm [3] improves on the dynamic-sized implementation of [1]: it uses only one DCAS for each uncontended operation and does not need a spare bit in pointers. Unfortunately, it is incorrect as published, despite a detailed manual proof. This demonstrates that the availability of DCAS does not immediately admit solutions that are simple enough to eliminate the significant effort required for proving the correctness of algorithms based only on single-location synchronisation primitives such as CAS. We address the Snark algorithm in a detailed case study in the following sections, describing bugs in the published algorithm, corrections to the algorithm, and the verification of the corrected version.

Martin *et al.* [23] present a dynamic-sized deque implementation that uses bulk allocation in order to reduce the overhead associated with allocating a new node for every push operation. They also present experiments that demonstrate that this optimisation can improve performance sub-

[3] In some previous work (e.g. [25]), objects are never truly deallocated but are instead placed on a special type-specific free-list, thereby permanently preventing their reuse for other types.

[4] Linearizability is the standard correctness condition for nonblocking data structure implementations [17]. An implementation is linearizable if each operation appears to take place atomically at some point between its invocation and its response.

[5] As discussed earlier, the recent line of research into DCAS-based algorithms has led to effective mechanisms for nonblocking memory management [4, 13]. However, using these techniques results in additional space and time overhead.

stantially. However, while bulk allocation is a natural optimisation, integrating it with a nonblocking doubly-linked-list deque implementation was a significant challenge, and the resulting algorithm is even more complicated than its predecessors.

In summary, while the investigation of the utility of DCAS has resulted in a number of advances in nonblocking algorithms and data structures, each of the results discussed above has serious drawbacks. We argue in Section 6 that DCAS is therefore not a silver bullet for making the design of nonblocking algorithms and data structures easy. We also argue that different synchronisation support would allow us to achieve substantially simpler algorithms that overcome these drawbacks. The case study presented below illustrates this point in detail.

3. THE SNARK ALGORITHM

This section briefly describes the Snark algorithm [3] and outlines two bugs in it. The purpose of this section is to show that implementations of data structures that use DCAS can suffer the same problems as implementations that use CAS, even for fairly mundane structures.

The declarations and initial state for the Snark algorithm are presented in Figure 2. Code for the original published version is presented in Figures 3 and 4. Only the right-side operations are presented here; the left-side operations are symmetric and can be found (with a full discussion of the algorithm) in [3].

The Snark algorithm uses a doubly linked list in which each node is connected to its neighbours through its `L` and `R` fields. The `V` field of a node contains the value represented by that node.

When the deque is not empty, `LeftHat` (resp. `RightHat`) points to the leftmost (resp. rightmost) node that contains an unpopped value. Snark uses sentinel nodes on either end of the deque to allow operations to detect when the deque is empty (as explained below). Figure 5 illustrates a deque containing three elements. Observe that the inward pointers of the sentinels are self pointers. Snark maintains the following key property, which characterises a state that represents a nonempty deque:

Property 1: If the deque is not empty, then the following property holds:

> `LeftHat`→`L` ≠ `LeftHat` and
> `RightHat`→`R` ≠ `RightHat` and
> `LeftHat`→`L`→`R` = `LeftHat`→`L` and
> `RightHat`→`R`→`L` = `RightHat`→`R`

Property 1 implies that if `LeftHat`→`L` = `LeftHat` or `RightHat`→`R` = `RightHat`, then the deque is empty. Thus, a self pointer in the outward field of a node pointed to by `LeftHat` or `RightHat` can be used to detect when the deque is empty. In fact, the Snark algorithm guarantees that if one hat points to a node with such a self pointer, then so does the other:

Property 2: If the deque is empty, then:

> `LeftHat`→`L` = `LeftHat` and
> `RightHat`→`R` = `RightHat`

The above implies that the empty deque can be represented by a variety of different state configurations. Figure

```
struct Node {valtype V; Node *L; Node *R}

Node *Dummy, *LeftHat, *RightHat;

initially
  Dummy != null and
  Dummy->L == Dummy and Dummy->R == Dummy and
  LeftHat == Dummy and RightHat == Dummy
```

Figure 2: State variables and initialisation for the Snark algorithm.

```
pushRight(val v) {
  nd = new Node();
  if (nd == null) return "full";
  nd->R = Dummy;
  nd->V = v;
  while (true) {
    rh = RightHat;
    rhR = rh->R;
    if (rhR == rh) {
      nd->L = Dummy;
      lh = LeftHat;
      if (DCAS(&RightHat, &LeftHat,
                 rh, lh, nd, nd))
        return "ok";
    } else {
      nd->L = rh;
      if (DCAS(&RightHat, &rh->R,
                 rh, rhR, nd, nd))
        return "ok";
    }
  }
}
```

Figure 3: Original pushRight operation [3].

```
val popRight() {
  while (true) {
    rh = RightHat;
    lh = LeftHat;
    if (rh->R == rh) return "empty";
    if (rh == lh) {
      if (DCAS(&RightHat, &LeftHat,
                 rh, lh, Dummy, Dummy))
        return rh->V;
    } else {
      rhL = rh->L;
      if (DCAS(&RightHat, &rh->L,
                 rh, rhL, rhL, rh)) {
        result = rh->V;
        rh->R = Dummy;
        return result;
      }
    }
  }
}
```

Figure 4: Original popRight operation [3].

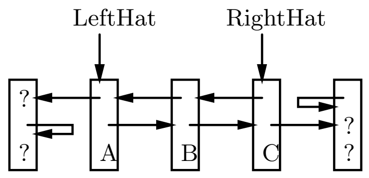

Figure 5: A deque containing three elements.

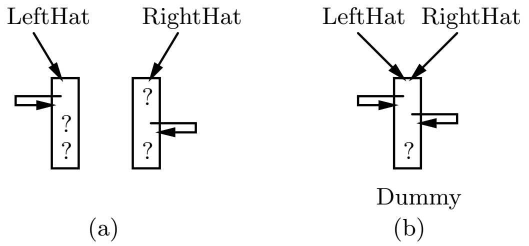

Figure 6: Empty deque states. (a) Generic empty state. (b) Special case empty state.

6(a) illustrates one such state. Snark uses a global constant `Dummy` such that `Dummy`→`L` = `Dummy`→`R` = `Dummy` always holds. Observe that Figure 6(b) shows a special case of the empty deque representations shown in Figure 6(a). This representation is used as an initial state, and can also be reached by removing the last element from the list.

The `pushRight` operation (Figure 3) works in a straightforward way: the DCAS at line 12 attempts to change an empty doubly linked list to a list containing the value being pushed, using `Dummy` as a sentinel on both sides; the DCAS at line 17 attempts to insert a new node (containing the appropriate value) onto the right side of a nonempty list, using `Dummy` as the new right sentinel.

Like `pushRight`, `popRight` (Figure 4) applies one of two DCAS operations to modify the doubly linked list: the DCAS at line 7 is meant to be applied when there is only one element in the deque, setting `LeftHat` and `RightHat` to `Dummy` so that both hats point to a node with appropriate self pointers (thereby making the deque empty). The DCAS at line 12 removes the node pointed to by `RightHat` from the doubly linked list by storing a self pointer into its `L` field and shifting `RightHat` onto the next node to the left. (Note that the removed node becomes the new right sentinel. The transition is illustrated by Figures 5 and 7.)

There are two problems with the algorithm as presented in [3] (and reproduced in Figures 3 and 4). Both problems cause the pop operations to behave incorrectly. First, a pop operation can return `empty` even if the deque is never empty during the execution of the pop operation. To see why, observe that a pop operation returns `empty` if the test at line 5 succeeds. The idea behind this is that if `RightHat`→`R` = `RightHat` then the deque is empty (see Property 1). However, pop operations attempt to confirm this by testing the local variable `rh`, which does *not* ensure that `RightHat` still contains the value read from it earlier (in line 3 of Figure 4). The following describes an erroneous execution:

- A process p invokes `popRight` while the deque is not empty. It loads its `rh` variable and is then delayed.
- While p is delayed, other processes complete `pushRight` and `popLeft` operations so that the node referenced by p's `rh` variable is popped from the deque by a `popLeft` *without the deque being empty in that period.*
- p resumes execution and performs the test at line 5, finding `rh`→`R` = `rh` (because `rh` has been removed by a `popLeft`), and returns `empty`.

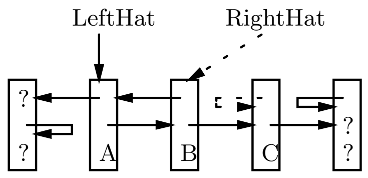

Figure 7: After DCAS at line 12 of popRight. Dashed lines show changed values.

As the deque was never empty during p's operation, this execution is not linearizable.

Second, Snark allows a node to be removed from the doubly linked list twice, causing its value to be returned twice. A more detailed description of this bug can be found in [5]. Briefly, it is possible to construct executions in which the following happens (the expression p.`x` means the value of process p's local variable `x`):

- Process p invokes `popRight` when the deque contains more than one element and runs alone until it is about to execute the DCAS at line 12, but is delayed before it does so.
- Other processes execute `pushRight` and `popLeft` operations so that p.`rh` = `LeftHat` and the deque contains more than one element. This can be achieved without modifying p.`rh`→`L`.
- Some process q invokes and completes an execution of `popLeft`, and this operation removes the node referenced by p.`rh`. This also happens without modifying p.`rh`→`L`.
- Other processes execute `popRight` operations so that once again, p.`rh` = `RightHat`. The deque is now empty. Finally, p executes its DCAS, which succeeds because p.`rh` = `RightHat` and p.`rh`→`L` = p.`rhL`, and p returns p.`rh`→`V`, which has already been returned by q.

The above-described bug is an instance of the ABA problem [26], which is well known in the context of CAS-based algorithms. ABA describes the following phenomenon: a process may read value A from a location and subsequently use CAS to attempt to change the contents of the location from A to some other value. Usually, the desired effect is that the CAS should succeed only if the value of the location has not changed since the previous read. However, CAS confirms only that the expected value A is in the location *at the time of the CAS*. It is possible that, between the read and the CAS, other processes change the value in the location from A to B and subsequently back to A again, and that the CAS therefore succeeds when we want it to fail. DCAS is subject to exactly the same phenomenon, extended to two locations, and that is the cause of the bug described above: p reads a value in `RightHat`, and its subsequent DCAS succeeds even though the `RightHat` has moved off that node and back onto it in the interim.

```
1  val popRight() {
2    while (true) {
3      rh = RightHat;
4      rhL = rh->L;
5      if (rh->R == rh) {
6        if (RightHat == rh) return "empty";
7      } else {
8        if (DCAS (&RightHat, &rh->L,
9                    rh, rhL, rhL, rh)) {
10         result = rh->V;
11         if (result != "claimed"){
12           if (CAS(&rh->V, result, "claimed")) {
13             rh->R = Dummy;
14             return result;
15           } else return "empty";
16         } else return "empty";
17       }
18     }
19   }
20 }
```

Figure 8: Corrected popRight operation.

It is interesting to note that garbage collection (GC) eliminates *some instances* of the ABA problem. GC eliminates instances of the ABA problem in which the value read is a pointer that cannot be reintroduced to the location in question before the object to which the pointer refers has been reclaimed by the garbage collector and subsequently reallocated. The premature reclamation that would cause this problem is prevented by GC because the process that intends to perform a CAS (or DCAS) has a copy of the pointer in its local variable, and therefore the object is not reclaimed and reallocated until after the CAS attempt. While this is a very useful trick in some cases, we caution the reader against assuming that GC eliminates *all* instances of the ABA problem. As illustrated above, the ABA problem can arise even in implementations that rely on GC.

4. A CORRECTED VERSION

Both of the above-described bugs can be fixed by modifying the pop operations; the push operations are left untouched. Figure 8 presents the modified `popRight` operation. One modification has been made to the original Snark algorithm that is not related to either bug: lines 6 to 9 of the original algorithm (Figure 4) have been removed. As noted above, the test at line 6 and the subsequent DCAS at line 7 were intended to address the special case in which the deque contains only one element. However, the DCAS at line 12 handles this case correctly. Therefore, the single-element deque need not be treated as a special case, and so the code at lines 6 to 9 can be eliminated. As a result of this eliminated special case, there is no longer any need for processes to read or modify the opposite hat (`LeftHat` in the case of processes popping from the right), so line 4 can also be eliminated from the popRight operation in Figure 4. This modification eliminates contention between pop operations and operations on the opposite end of the deque, and should therefore improve performance.

The first bug—a popping process returning `empty` even when the deque is never empty during the operation—is solved by the extra test on line 6 of Figure 8. It is a property of the Snark algorithm that once a node has a self pointer in either of its `L` or `R` fields, that field will always contain a self pointer. Therefore, if a process is about to execute line 6, we know that `rh`$\rightarrow$`R` = `rh` still holds, so by Property 1 above, if `rh` = `RightHat` holds, then the deque is empty.

The second bug—that nodes can be removed from the deque more than once—is solved by lines 10 to 16 of Figure 8. The basic idea is to *allow* a node to be removed from the data structure more than once, but to make processes compete to return the value contained in it.[6]

After a process p removes a node from the list, p reads the node's `V` field (line 10), and attempts to use CAS to atomically replace the value with a special `claimed` value (line 12), unless the value is already `claimed` (line 11). If this CAS is successful, p returns the replaced value (line 14). If the value already contains `claimed` or if the CAS at line 12 fails, then some other process has already claimed the value from this node, so p returns `empty`. Because `claimed` is a special value that is never pushed, only one process can succeed in its CAS on a given node. That process can safely return the value in the node; processes that fail the CAS return `empty`.

It may seem strange that a process returns `empty` when it finds that some other process has claimed the value of the node it removed from the list. However, it can be shown that if two processes remove the same node, then the deque is empty when the second successful DCAS is executed and that this DCAS occurs during the execution of both operations. Thus, failing processes can return `empty` without having to retry their whole operation (thereby avoiding the contention that could be caused by that retry).

5. VERIFICATION OVERVIEW

The correctness of the modified algorithm has been verified using the PVS verification system [28]. It is straightforward to see that the algorithm is lock-free, so the proof effort focussed on linearizability. A complete description of the verification effort is in preparation, and is beyond the scope of this paper. Here we give an overview of the structure of the verification and the chief difficulties encountered. Because the modified algorithm is complicated, its verification required substantial human input and insight into the algorithm. As described in the next section, a slightly more powerful synchronization primitive enables a much simpler implementation that is easy to verify.

The verification of the corrected Snark algorithm is achieved by modelling both the algorithm and the specification of a deque using *I/O automata*, and using *simulation* proof techniques to prove that the algorithm automaton implements the specification automaton. An *I/O automaton* [20, 21] is a labelled transition system whose labels, called *actions*, are classified as *external* (those that represent invocations and responses of operations) or *internal* (those that represent internal steps of operations). A *trace* of an automaton is the sequence of external actions in some execution of that automaton. A concrete automaton C, which models the algorithm, *implements* a specification automaton S if every trace of C is a trace of S (so the automata are externally indistiguishable).

One way to show that C implements S is to demonstrate

[6]This bug could also be eliminated by using the standard technique of adding "version numbers" [25] to the `LeftHat` and `RightHat` pointers. However, this would further restrict the applicability of the algorithm because it would require DCAS to access the pointer and the version number together atomically.

a forward simulation from C to S. A *forward simulation* [22] is a relation between the states of C and the states of S such that for every step of C, there is a corresponding sequence (possibly empty) of steps of S that preserves the relation between the states in the two automata (i.e., if the pre-states satisfy the relation then so too do the post-states). The sequence of actions in S that corresponds to any step of C must depend only on that step (i.e., the pre-state, the action, and the post-state), and should include an external action if and only if that action is the action in the step of C under consideration. With a forward simulation from C to S, it is easy to show by induction on the length of an arbitrary execution of C that C implements S.

In the simple and natural abstract deque specification automaton S, the deque is represented by the sequence of values in the deque, and, in addition to the external invocation and response actions, S has a single internal action for each operation, which represents that operation taking effect atomically. For example, when the deque is not empty, the internal action for process p's `popRight` operation atomically removes the rightmost item from the sequence, and records it for p's response action to return. Unfortunately, as discussed below, there is no forward simulation from the (corrected) Snark automaton C to S.

Suppose there is a forward simulation from C to S. If two processes executing pop operations remove the same node from the doubly linked list, one of the two—call it q—will eventually claim and return the value from the node. Therefore, we must include the internal action of q's pop operation in the sequence of steps of S corresponding to some step in the execution of C. We cannot do so at any step before q claims the value, because we do not yet know which process will claim it. However, a sequence of pop operations that occurs after the node is removed, but before q claims the value, may cause the deque to become empty. This would require us to include q's internal action in the sequence of steps corresponding to some step of C that occurs *before* q claims the value. Thus, there is no forward simulation from C to S.

To overcome this problem, the verification introduces an *intermediate automaton* I, which is carefully designed to allow a forward simulation from C to I, but also to admit a verification that I implements S using a proof technique known as *backward simulation* [22]. In contrast to forward simulation proofs, backward simulations construct an execution of the specification automaton with the same trace as an arbitrary finite execution of the algorithm automaton (in our case the intermediate automaton I) by starting at the last state of the execution and working *backwards* from that state. Backward simulations are significantly more challenging than forwards ones. For this reason, the intermediate automaton I was designed to be as close as possible to the specification automaton S, differing just enough to allow a forward simulation from the algorithm automaton C to I. Specifically, I is like S in that operations act on an abstract sequence of values rather than a doubly linked list, and values are atomically added to and removed from that sequence; but it is like the Snark algorithm in that one operation may "steal" a value removed from the sequence by another operation, causing the operation whose value is stolen to return "empty". I is constructed so that values can be stolen only if the abstract deque is empty at some point during the execution of both operations. The backward simulation from I to S shows that all traces of I are also traces of S, and the forward simulation from C to I shows that all traces of C are also traces of I. Thus, by transitivity, C implements S.

Both the forward and backward simulation proofs were achieved using the PVS verification system. PVS provides a language (a typed high-order logic) for making assertions, and a theorem prover that provides help in constructing proofs and checking their validity. The human effort of the verification consisted of defining the automata and the two simulation relations using the PVS language, constructing proofs that the relations satisfy the requirements of a simulation relation (forward for proving that C implements I and backward for proving that I implements S), and guiding the prover in validating these proofs. In addition to preventing the simple human mistakes that are common in manual proofs, the use of PVS substantially reduced the burden of checking mundane details of the proof, particularly in rechecking these details when earlier parts of the proof changed as the proof was being developed. Nonetheless, PVS did not make it straightforward to construct and validate the proofs: Considerable insight was required to select the right specification automaton, to define the correct simulation relations, and to guide the theorem prover in verifying that the relations indeed meet the requirements for forward and backward simulations, respectively.

6. DISCUSSION

The case study presented above clearly illustrates that, even using DCAS, the design and verification of nonblocking implementations of some mundane data structures is still difficult enough to provide publishable results. Below we discuss some alternative directions for putting this cottage industry out of business.

All of the complication of Snark can be avoided by using a more powerful synchronisation primitive than DCAS. For example, a compare-and-swap that operates on *three* independent words (3CAS) can be used to atomically make *both* pointers in a node become self pointers. The `popRight` operation is shown in Figure 9; again the `popLeft` operation is symmetric and the push operations are unchanged. By using 3CAS in this way, it becomes impossible for a node to be popped twice, and the verification becomes straightforward because the entire effect of a pop operation on shared variables happens in a single atomic step. The operation can be linearized to that point, allowing a simple forward simulation proof to the natural specification automaton.

Another approach that can significantly simplify nonblocking algorithm design is to consider a weaker nonblocking progress property. The algorithms discussed in Section 2 are all *lock-free.* That is, processes are prevented from making progress only by the progress of other processes. An *obstruction-free* algorithm [14, 15] only guarantees that an operation completes if it does not encounter interference from a concurrent operation for a sufficient period of time. This property is weaker than lock-freedom, because it does not require progress in the case of repeated interference between two or more operations. The approach advocated by the authors of [14, 15] is to combine obstruction-free implementations with orthogonal "contention managers", which attempt to facilitate the conditions needed for progress.

The weaker requirements of obstruction-freedom admit implementations that are substantially simpler and more

```
val popRight() {
  while (true) {
    rh = RightHat;
    rhL = rh->L;
    rhR = rh->R;
    if (rhR == rh) {
      if (RightHat == rh) return "empty";
    } else {
      if (3CAS (&RightHat, &rh->L, &rh->R,
                rh, rhL, rhR, rhL, rh, rh)) {
        return result;
    }
  }
}
```

Figure 9: popRight operation using 3CAS.

efficient (at least in the absence of contention) than their lock-free counterparts, and this has resulted in significant progress in the implementation of nonblocking data structures, especially dynamic-sized ones [12, 14, 15, 19]. However, these implementations still depend on nontrivial insights, and we do not believe that the obstruction-free approach alone will simplify the design of nonblocking algorithms and data structures sufficiently that additional synchronisation support is no longer desired.

The design of efficient nonblocking algorithms would be significantly simplified by the use of synchronisation primitives that are stronger than DCAS. For example, as discussed above, a simple and correct dynamic-sized deque implementation similar in spirit to Snark can be constructed with the use of 3CAS (i.e., compare-and-swap that accesses *three* locations). This phenomenon is not isolated to the Snark algorithm. The authors of [23] describe a naive deque implementation admitting bulk allocation in which the data structure can enter pathological states. Dealing with these pathological states using DCAS complicated their algorithm significantly, but this could have been avoided easily using a 3CAS. Similarly, the dynamic-sized deque implementation of [3] would not need to "steal" a bit from pointers if 3CAS could be used to access the flag in a separate location.

Our intention here is not to start a new cottage industry of algorithms based on 3CAS, but merely to point out that the ability to access more locations can admit substantially simpler solutions to a given problem. In general, it is easy to imagine a need for compare-and-swap operations that can access even more than 3 locations: If we want to "compose" two operations (for example, to support the atomic transfer of a value from one deque to another), then we need a compare-and-swap that accesses the sum of the number of locations accessed by the two operations being composed. Therefore a general NCAS operation has been considered by some researchers [8, 10, 12, 27].

Another alternative for synchronisation support is *transactional memory* [16], which allows processes to make "tentative" changes to shared memory and to then commit those changes atomically. When two transactions seek to modify overlapping sets of memory locations, a successful commit by one transaction will cause the other transaction to fail without modifying memory. Therefore, achieving a correct implementation of a nonblocking data structure using transactional memory is in principle not much more difficult than achieving a sequential implementation. Transactional memory is more flexible than atomic multi-location update operations such as DCAS and 3CAS, and it does not suffer from the ABA problem because if a location is modified after a transaction reads from it and before that transaction attempts to commit, then the transaction fails and does not modify memory.

Recent experience with software implementations of transactional memory [9, 15] shows that transactional memory can be used to design simple nonblocking implementations that outperform their lock-based counterparts. For example, Herlihy *et al.* [15] present an implementation of a concurrent linked-list-based set that is based on their *dynamic software transactional memory* implementation. They also present experimental results showing that this implementation outperforms a coarse-grained locking implementation, even at low levels of contention. Harris and Fraser [9] describe another software transactional memory implementation, which they use to implement a hash table. This transactional hash table is always competitive with a fine-grained locking hash table implementation and in some situations outperforms it substantially.

Software implementations of NCAS and of transactional memory using only single-location synchronisation primitives such as CAS or LL/SC exist [10, 12, 27]. However, despite recent progress towards making these implementations more efficient and more widely applicable [10, 12], they are still complicated, and each multi-location operation or transaction entails multiple CAS operations. It remains to be seen whether these implementations (or future improved implementations) can serve as the basis for practical nonblocking algorithms and data structures. If not, hardware support for these primitives may be needed before practical and efficient nonblocking data structures are easy to design.

7. CONCLUDING REMARKS

We have argued that DCAS does not provide a "silver bullet" for nonblocking algorithms and data structures: algorithms based on DCAS share many of the disadvantages of previous algorithms based on single-location synchronisation primitives such as CAS and LL/SC. In particular, the algorithms are often complicated and subtle, requiring careful correctness proofs. Furthermore, many of the algorithms are achieved by relying on aspects of the execution environment, which severely restricts their applicability. While we have also argued that slightly stronger synchronisation support does significantly simplify the design of nonblocking algorithms, our intention in this paper is not to take a position on what hardware support *should* be included in future architectures. Instead, we hope to shift the focus from DCAS and encourage researchers to consider the utility and feasibility of other, stronger alternatives.

Apart from the lessons summarised above, the recent line of research into DCAS-based algorithms has been valuable for at least two other reasons. First, the techniques learned for effecting operations while strictly limiting the number of locations accessed are valuable even if stronger support for synchronisation is available. Limiting the number of locations accessed by common operations improves performance and scalability, but the ability to access more locations to deal with exceptional circumstances avoids the complications we have observed in work based solely on CAS or DCAS.

Second, as discussed earlier, the DCAS-based Lock-Free Reference Counting methodology for nonblocking memory management [4] provided insights that led directly to the development of practical memory management mechanisms that are applicable in current architectures [13].

8. REFERENCES

[1] O. Agesen, D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, M. Moir, N. Shavit, and Guy L. Steele Jr. DCAS-based concurrent deques. In *Theory of Computing Systems*, volume 35, 2002.

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

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

[4] D. Detlefs, P. Martin, M. Moir, and G. L. Steele, Jr. Lock-free reference counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, August 2001.

[5] S. Doherty. Modelling and verifying non-blocking algorithms that use dynamically allocated memory. Master's thesis, Victoria University Wellington, April 2003. `http://www.mcs.vuw.ac.nz/~sdoherty`.

[6] M. Greenwald. Two-handed emulation: How to build non-blocking implementations of complex data structures using DCAS. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, 2002.

[7] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the 2nd Symposium on Operating System Design and Implementation*, pages 123–136, 1996.

[8] M. B. Greenwald. *Non-Blocking Synchronisation and System Design*. PhD thesis, Stanford University, August 1999.

[9] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

[10] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 14th International Conference on Distributed Computing*, 2002.

[11] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[12] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free software NCAS and transactional memory. Unpublished manuscript, 2002.

[13] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structure. In *Proceedings of the 16th International Symposium on Distributed Computing*, October 2002.

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

[15] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, 2003.

[16] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of 20th Annual International Symposium on Computer Architecture*, 1993.

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

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

[19] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the ACM Symposium on Parallel Architectures and Algorithms*, 2003.

[20] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Mateo, CA, 1996.

[21] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, August 1987.

[22] N. A. Lynch and F. W. Vaandrager. Forward and backward simulations – part I: untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[23] P. A. Martin, M. Moir, and Guy L. Steele, Jr. DCAS-based concurrent deques supporting bulk allocation. Technical Report TR-2002-111, Sun Microsystems Laboratories, 2002.

[24] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical report, Columbia University, New York, June 1991.

[25] M. Michael and M. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[26] S. Prakash, Y. Lee, and T. Johnson. A non-blocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43(5):548–559, 1994.

[27] N. Shavit and D. Touitou. Software transactional memory. In *Distributed Computing, Special Issue*, volume 10, pages 99–116, 1997.

[28] The PVS Specification and Verification System, `http://pvs.csl.sri.com/`.

[29] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.