

Obstruction-Free Synchronization: Double-Ended Queues as an Example

Maurice Herlihy
Computer Science Department
Brown University, Box 1910
Providence, RI 02912

Victor Luchangco Mark Moir
Sun Microsystems Laboratories
1 Network Drive, UBUR02-311
Burlington, MA 01803

Abstract

We introduce obstruction-freedom, a new nonblocking property for shared data structure implementations. This property is strong enough to avoid the problems associated with locks, but it is weaker than previous nonblocking properties—specifically lock-freedom and wait-freedom—allowing greater flexibility in the design of efficient implementations. Obstruction-freedom admits substantially simpler implementations, and we believe that in practice it provides the benefits of wait-free and lock-free implementations.

To illustrate the benefits of obstruction-freedom, we present two obstruction-free CAS-based implementations of double-ended queues (deques); the first is implemented on a linear array, the second on a circular array. To our knowledge, all previous nonblocking deque implementations are based on unrealistic assumptions about hardware support for synchronization, have restricted functionality, or have operations that interfere with operations at the opposite end of the deque even when the deque has many elements in it. Our obstruction-free implementations have none of these drawbacks, and thus suggest that it is much easier to design obstruction-free implementations than lock-free and wait-free ones. We also briefly discuss other obstruction-free data structures and operations that we have implemented.

1. Introduction

The traditional way to implement shared data structures is to use mutual exclusion (locks) to ensure that concurrent operations do not interfere with one another. Locking has a number of disadvantages with respect to software engineering, fault-tolerance, and scalability (see [8]). In response, researchers have investigated a variety of alternative synchronization techniques that do not employ mutual exclusion. A synchronization technique is *wait-free* if it ensures that every thread will continue to make progress in the face

of arbitrary delay (or even failure) of other threads. It is *lock-free* if it ensures only that some thread always makes progress. While wait-free synchronization is the ideal behavior (thread starvation is unacceptable), lock-free synchronization is often good enough for practical purposes (as long as starvation, while possible in principle, never happens in practice).

The synchronization primitives provided by most modern architectures, such as *compare-and-swap* (CAS) or *load-locked/store-conditional* (LL/SC) are powerful enough to achieve wait-free (or lock-free) implementations of any linearizable data object [9]. Nevertheless, with a few exceptions (such as queues [16]), wait-free and lock-free data structures are rarely used in practice. The underlying problem is that conventional synchronization primitives such as CAS and LL/SC are an awkward match for lock-free synchronization. These primitives lend themselves most naturally to *optimistic* synchronization, which works only in the absence of synchronization conflicts. For example, the natural way to use CAS for synchronization is to read a value v from an address a , perform a multistep computation to derive a new value w , and then to call CAS to reset the value of a from v to w . The CAS is successful if the value at a has not been changed in the meantime. Progress guarantees typically rely on complex and inefficient “helping” mechanisms, that pose a substantial barrier to the wider use of lock-free synchronization.

In this paper, we propose an alternative nonblocking condition. A synchronization technique is *obstruction-free* if it guarantees progress for any thread that eventually executes in isolation. Even though other threads may be in the midst of executing operations, a thread is considered to execute in isolation as long as the other threads do not take any steps. (Pragmatically, it is enough for the thread to run long enough without encountering a synchronization conflict from a concurrent thread.) Like the wait-free and lock-free conditions, obstruction-free synchronization ensures that no thread can be blocked by delays or failures of other threads. This property is weaker than lock-free synchronization, because it does not guarantee progress when

two or more conflicting threads are executing concurrently.

The most radical way in which our approach of implementing obstruction-free algorithms differs from the usual approach of implementing their lock-free and wait-free counterparts is that we think that ensuring progress should be considered a problem of engineering, not of mathematics. We believe that commingling correctness and progress has inadvertently resulted in unnecessarily inefficient and conceptually complex algorithms, creating a barrier to widespread acceptance of nonblocking forms of synchronization. We believe that a clean separation between the two concerns promises simpler, more efficient, and more effective algorithms.

To support our case, we have implemented several obstruction-free shared data structures that display properties not yet achieved by comparable lock-free implementations. In this paper, we present two obstruction-free double-ended queue (deque) implementations. Elsewhere [11], we describe a software transactional memory implementation used to construct an obstruction-free red-black tree [5]. To our knowledge, there are no lock-free implementations of any data structure as complicated as a red-black tree.

As an aside, we note that there is no “obstruction-free hierarchy” comparable to the “wait-free consensus hierarchy”: One can solve obstruction-free consensus using only read/write registers by derandomizing randomized wait-free consensus algorithms such as the one in [4].

Because obstruction-freedom does not guarantee progress in the presence of contention, we need to provide some mechanism to reduce the contention so that progress is achieved. However, lock-free and wait-free implementations typically also require such mechanisms to get satisfactory performance. We can use these same mechanisms with obstruction-free implementations, as we discuss below. Because obstruction-freedom guarantees safety regardless of the contention, we can change mechanisms, even dynamically, without changing the underlying nonblocking implementation.

One simple and well-known method to reduce contention is for operations to “back off” when they encounter interference by waiting for some time before retrying. Various choices for how long to wait are possible; randomized exponential backoff is one scheme that is effective in many contexts. Other approaches to reducing contention include queuing and timestamping approaches, in which threads agree amongst themselves to “wait” for each other to finish. While simplistic applications of these ideas would give rise to some of the same problems that the use of locks does, we have much more freedom in designing sophisticated approaches for contention control than when using locks, because correctness is not jeopardized by interrupting an operation at any time and allowing another operation to continue execution.

In fact, it is possible to design contention management mechanisms that guarantee progress to every operation that takes enough steps, provided the system satisfies some very weak (and reasonable) assumptions. Thus, the strong progress properties of wait-free implementations can be achieved in practice by combining obstruction-free implementations with appropriate contention managers. In scenarios in which contention between operations is rare, we will benefit from the simple and efficient obstruction-free designs; the more heavy-weight contention resolution mechanisms will rarely be invoked. In contrast, in most lock-free and wait-free implementations, the mechanisms that are used to ensure the respective progress properties impose significant overhead *even in the absence of contention*. A study of the performance of various contention managers in practice, and tradeoffs between system assumptions, progress guarantees and performance is beyond the scope of this paper.

In some contexts, explicit contention reduction mechanisms may even be unnecessary. For example, in a uniprocessor where threads are scheduled by time slice, relatively short obstruction-free operations will be guaranteed to run alone for long enough to complete. Similarly, in priority-scheduled uniprocessors, an operation runs in isolation unless it is preempted by a higher priority operation. (As an aside, it has been shown previously that the consensus hierarchy collapses in such systems [2, 17]. However, in these results, *correctness*, as well as progress, depends on the system assumptions.)

In Section 2, we discuss previous work on nonblocking dequeues. Section 3 presents a simple obstruction-free deque implementation on a linear array, and Section 4 extends this algorithm to circular arrays. A detailed formal proof for the extended algorithm is given in a full version of the paper [10]. We conclude in Section 5.

2. Related Work on Nonblocking Deques

In this section, we briefly summarize related work on nonblocking dequeues. Double-ended queues (dequeues) are formally defined in [6, 13]. Informally, dequeues generalize FIFO queues and LIFO stacks by supporting a sequence of values and operations for adding (pushing) a value to or removing (popping) a value from either end. Thus, implementing a shared deque combines all of the intricacies of implementing queues and stacks.

Arora, *et al.* proposed a limited-functionality CAS-based lock-free deque implementation [3]. Their deque allows only one process to access one end, and only pop operations to be done on the other. Thus, they did not face the difficult problem of concurrent pushes and pops on the same end of the deque. They further simplified the problem by allowing some concurrent operations to simply abort and report

failure.

Greenwald proposed two lock-free deque implementations [7]. Both implementations depend on a hardware DCAS (double compare-and-swap) instruction, which is not widely supported in practice, and one of them does not support noninterfering concurrent operations at opposite ends of the deque. Various members of our group have also proposed several lock-free deque implementations that depend on DCAS [1, 6, 14].

Michael proposed a simple and efficient lock-free, CAS-based deque implementation [15]. However, the technique used by this algorithm fundamentally causes all operations to interfere with each other. Therefore, it offers no insight into designing scalable nonblocking data structures in which noninterfering operations can proceed in parallel.

3. Obstruction-Free Deque Implementation

In this section we present our array-based obstruction-free deque implementation. This algorithm is extremely simple, but it is not a real deque in the sense that it does not really generalize queues: if we only push on one end and pop from the other, we will exhaust the space in the array and will not be able to push any more items. In the next section, we show how to extend the algorithm to “wrap around” in the array in order to overcome this problem.

The data declarations and right-side push and pop operations are shown in Figure 1; the left-side operations are symmetric with the right-side ones.

We assume the existence of two special “null” values LN and RN (left null and right null) that are never pushed onto the deque. We use an array A to store the current state of the deque. The deque can contain up to MAX values, and the array is of size $MAX+2$ to accommodate a leftmost location that always contains LN and a rightmost location that always contains RN. (These extra locations are not strictly necessary, but they simplify the code.) Our algorithm maintains the invariant that the sequence of values in $A[0].val \dots A[MAX+1].val$ always consists of at least one LN, followed by zero or more data values, followed by at least one RN. The array can be initialized any way that satisfies this invariant. To simplify our presentation, we assume the existence of a function `oracle()`, which accepts a parameter `left` or `right` and returns an array index. The intuition is that this function attempts to return the index of the leftmost RN value in A when invoked with the parameter `right`, and attempts to return the index of the rightmost LN value in A when invoked with the parameter `left`. The algorithm is linearizable [12] even if `oracle` can be incorrect (we assume that it at least always returns a value between 1 and $MAX+1$, inclusive, when invoked with the parameter `right` and always returns a value between 0 and MAX , inclusive, when invoked with the pa-

rameter `left`; clearly it is trivial to implement a function that satisfies this property). Stronger properties of the oracle are required to prove obstruction-freedom; we discuss these properties and how they can be achieved later.

As explained in more detail below, we attach version numbers to each value in order to prevent concurrent operations that potentially interfere from doing so. The version numbers are updated atomically with the values using a compare-and-swap (CAS) instruction.¹ As usual with version numbers, we assume that sufficient bits are allocated for the version numbers to ensure that they cannot “wrap around” during the short interval in which one process executes a single iteration of a short loop in our algorithm.

The reason our obstruction-free deque implementation is so simple (and the reason we believe obstruction-free implementations in general will be significantly simpler than their lock-free and wait-free counterparts) is that there is no progress requirement when any interference is detected. Thus, provided we maintain basic invariants, we can simply retry when we detect interference. In our deque implementation, data values are changed only at the linearization point of successful push and pop operations. To prevent concurrent operations from interfering with each other, we increment version numbers of adjacent locations (without changing their associated data values). As a result of this technique, two concurrent operations can each cause the other to retry: this explains why our implementation is so simple, and also why it is not lock-free. To make this idea more concrete, we describe our implementation in more detail below.

The basic idea behind our algorithm is that a `rightpush(v)` operation changes the leftmost RN value to v , and a `rightpop()` operation changes the rightmost data value to RN and returns that value (the left-side operations are symmetric, so we do not discuss them further except when dealing with interactions between left- and right-side operations). Each `rightpush(v)` operation that successfully pushes a data value (as opposed to returning “full”) is linearized to the point at which it changes an RN value to v . Similarly, each `rightpop` operation that returns a value v (as opposed to returning “empty”) is linearized to the point at which it changes the `val` field of some array location from v to RN. Furthermore, the `val` field of an array location does not change unless an operation is linearized as discussed above. The `rightpush` operation returns “full” only if it observes a non-RN value in $A[MAX].val$. Given these observations, it is easy to see that our algorithm is linearizable if we believe the following three claims (and their

¹A `CAS(a, e, n)` instruction takes three parameters: an address a , an expected value e , and a new value n . If the value currently stored at address a matches the expected value e , then CAS stores the new value n at address a and returns *true*; we say that the CAS *succeeds* in this case. Otherwise, CAS returns *false* and does not modify the memory; we say that the CAS *fails* in this case.

```

type element = record val: valtype; ctr: int end

A: array[0..MAX+1] of element initially there is some k in [0,MAX]
    such that A[i] = <LN,0> for all i in [0,k]
    and A[i] = <RN,0> for all i in [k+1,MAX+1].

rightpush(v) // v is not RN or LN
RH0: while (true) {
RH1:   k := oracle(right); // find index of leftmost RN
RH2:   prev := A[k-1]; // read (supposed) rightmost non-RN value
RH3:   cur := A[k]; // read (supposed) leftmost RN value
RH4:   if (prev.val != RN and cur.val = RN) { // oracle is right
RH5:     if (k = MAX+1) return "full"; // A[MAX] != RN
RH6:     if CAS(&A[k-1],prev,<prev.val,prev.ctr+1>) // try to bump up prev.ctr
RH7:       if CAS(&A[k],cur,<v,cur.ctr+1>) // try to push new value
RH8:         return "ok"; // it worked!
    } // end if (prev.val != RN and cur.val = RN)
} // end while

rightpop()
RP0: while (true) { // keep trying till return val or empty
RP1:   k := oracle(right); // find index of leftmost RN
RP2:   cur := A[k-1]; // read (supposed) value to be popped
RP3:   next := A[k]; // read (supposed) leftmost RN
RP4:   if (cur.val != RN and next.val = RN) { // oracle is right
RP5:     if (cur.val = LN and A[k-1] = cur); // adjacent LN and RN
RP6:     return "empty"
RP7:     if CAS(&A[k],next,<RN,next.ctr+1>) // try to bump up next.ctr
RP8:       if CAS(&A[k-1],cur,<RN,cur.ctr+1>) // try to remove value
RP9:         return cur.val // it worked; return removed value
    } // end if (cur.val != RN and next.val = RN)
} // end while

```

Figure 1. Obstruction-free deque implementation: Data declarations and right-side operations

symmetric counterparts):

- At the moment that line RH7 of a `rightpush(v)` operation successfully changes `A[k].val` for some `k` from `RN` to `v`, `A[k-1].val` contains a non-`RN` value (i.e., either a data value or `LN`).
- At the moment that line RP8 of the `rightpop` operation successfully changes `A[k-1].val` for some `k` from some value `v` to `RN`, `A[k].val` contains `RN`.
- If a `rightpop` operation returns “empty”, then at the moment it executed line RP3, `A[k].val=RN` and `A[k-1].val=LN` held for some `k`.

Using the above observations and claims, a proof by simulation to an abstract deque in an array of size `MAX` is straightforward. Below we briefly explain the synchronization techniques that we use to ensure that the above claims hold. The techniques all exploit the version numbers in the array locations.

The empty case (the third claim above) is the simplest: `rightpop` returns “empty” only if it reads the same value from `A[k-1]` at lines RP2 and RP5. Because every CAS that modifies an array location increments that location’s version number, it follows that `A[k-1]` maintained the same value throughout this interval (recall our assumption about version numbers not wrapping around). Thus, in particular, `A[k-1].val` contained `LN` at the moment that line RP3 read `RN` in `A[k].val`.

The techniques used to guarantee the other two claims are essentially the same, so we explain only the first one. The basic idea is to check that the neighbouring location (i.e., `A[k-1]`) contains the appropriate value (line RH2; see also line RH4), and to increment its version number (without changing its value; line RH6) between reading the location to be changed (line RH3) and attempting to change it (line RH7). If any of the attempts to change a location fail, then we have encountered some interference, so we can simply restart. Otherwise, it can be shown easily that the neighbouring location did not change to `RN` between

the time it was read (line RH2) and the time the location to be changed is changed (line RH7). The reason is that a `rightpop` operation—the only operation that changes locations to RN—that was attempting to change the neighbouring location to RN would increment the version number of the location the `rightpush` operation is trying to modify, so one of the operations would cause the other to retry.

3.1. Oracle Implementations

The requirements for the `oracle` function assumed in the previous section are quite weak, and therefore a number of implementations are possible. We first describe the requirements, and then outline some possible implementations. For linearizability, the only requirement on the oracle is that it always returns an index from the appropriate range depending on its parameter as stated earlier; satisfying this requirement is trivial. However, to guarantee obstruction-freedom, we require that the oracle is *eventually accurate if repeatedly invoked in the absence of interference*. By “accurate”, we mean that it returns the index of the leftmost RN when invoked with `right`, and the index of the rightmost LN when invoked with `left`. It is easy to see that if any of the operations executes an entire loop iteration in isolation, and the `oracle` function returns the index specified above, then the operation completes in that iteration. Because the oracle has no obligation (except for the trivial range constraint) in the case that it encounters interference, we have plenty of flexibility in implementing it. One simple and correct implementation is to search the array linearly from one end looking for the appropriate value. Depending on the maximum deque size, however, this solution might be very inefficient. One can imagine several alternatives to avoid this exhaustive search. For example, we can maintain “hints” for the left and right ends, with the goal of keeping the hints approximately accurate; then we could read those hints, and search from the indicated array position (we’ll always be able to tell which direction to search using the values we read). Because these hints do not have to be perfectly accurate at all times, we can choose various ways to update them. For example, if we use CAS to update the hints, we can prevent slow processes from writing out-of-date values to hints, and therefore keep hints almost accurate all the time. It may also be useful to loosen the accuracy of the hints, thereby synchronizing on them less often. In particular, we might consider only updating the hint when it is pointing to a location that resides in a different cache line than the location that really contains the leftmost RN for example, as in this case the cost of the inaccurate hint would be much higher.

4. Extension to Circular Arrays

In this section, we show how to extend the algorithm in the previous section to allow the deque to “wrap around” the array, so that the array appears to be circular. In other words, $A[0]$ is “immediately to the right” of $A[\text{MAX}+1]$. As before, we maintain at least two null entries in the array: we use the array $A[0.. \text{MAX}+1]$ for a deque with at most MAX elements. The array can be initialized arbitrarily provided it satisfies the main invariant for the algorithm, stated below. One option is to use the initial conditions for the algorithm in the previous section.

We now describe the new aspects of the algorithm. Code for the right-side operations of the wrap-around deque implementation are shown in Figure 2. The left-side operations are symmetric, and we do not discuss them further except as they interact with the right-side operations. All arithmetic on array indices is done modulo $\text{MAX}+2$.

There are two main differences between this algorithm and the one in the previous section. First, it is more difficult to tell whether the deque is full; we must determine that there are exactly two null entries. Second, `rightpush` operations may encounter LN values as they “consume” the RN values and wrap around the array (similarly, `leftpush` operations may encounter RN values). We handle this second problem by enabling a `rightpush` operation to “convert” LN values into RN values. This conversion uses an extra null value, which we denote DN, for “dummy null”. We assume that LN, RN and DN are never pushed onto the deque.

Because the array is circular, the algorithm maintains the following invariants instead of the simpler invariant maintained by the algorithm in the previous section:

- All null values are in a contiguous sequence of locations in the array. (Recall that the array is circular, so the sequence can wrap around the array.)
- The sequence of null values consists of zero or more RN values, followed by zero or one DN value, followed by zero or more LN values.
- There are at least two different types of null values in the sequence of null values.

Thus, there is always at least one LN or DN entry, and at least one RN or DN entry.

Instead of invoking `oracle(right)` directly, the push and pop operations invoke a new auxiliary procedure, `rightcheckedoracle`. In addition to an array index k , `rightcheckedoracle` returns `left` and `right`, the contents it last saw in $A[k-1]$ and $A[k]$ respectively. It guarantees `right.val = RN` and `left.val != RN`. Thus, if it runs in isolation, `rightcheckedoracle` always returns the correct index, together with contents of the

```

/* Returns k,left,right, where left = A[k-1] at some time t, and right = A[k]
   at some time t' > t during the execution, with left.val != RN and right.val = RN.
*/
rightcheckedoracle()
RO0: while (true) {
RO1:  k := oracle(right);
RO2:  left := A[k-1];           // order important for check
RO3:  right := A[k];           //   for empty in rightpop
RO4:  if (right.val = RN and left.val != RN)           // correct oracle
RO5:    return k,left,right;
RO6:  if (right.val = DN and !(left.val in {RN, DN})) // correct oracle, but no RNs
RO7:    if CAS(&A[k-1], left, <left.val, left.ctr+1>)
RO8:      if CAS(&A[k], right, <RN, right.ctr+1>)       // DN -> RN
RO9:        return k,<left.val, left.ctr+1>, <RN, right.ctr+1>;
    } // end while

rightpush(v)                               // !(v in {LN, RN, DN})
RH0: while (true) {
RH1:  k,prev,cur := rightcheckedoracle();       // cur.val = RN and prev.val != RN
RH2:  next := A[k+1];
RH3:  if (next.val = RN)
RH4:    if CAS(&A[k-1], prev, <prev.val, prev.ctr+1>)
RH5:      if CAS(&A[k], cur, <v, cur.ctr+1>)       // RN -> v
RH6:        return "ok";
RH7:  if (next.val = LN)
RH8:    if CAS(&A[k], cur, <RN, cur.ctr+1>)
RH9:      CAS(&A[k+1], next, <DN, next.ctr+1>);     // LN -> DN
RH10: if (next.val = DN) {
RH11:  nextnext := A[k+2];
RH12:  if !(nextnext.val in {RN, LN, DN})
RH13:    if (A[k-1] = prev)
RH14:      if (A[k] = cur) return "full";
RH15:  if (nextnext.val = LN)
RH16:    if CAS(&A[k+2], nextnext, <nextnext.val, nextnext.ctr+1>)
RH17:      CAS(&A[k+1], next, <RN, next.ctr+1>);     // DN -> RN
    } // end if (next.val = DN)
  } //end while

rightpop()
RP0: while (true) {
RP1:  k,cur,next := rightcheckedoracle();       // next.val = RN and cur.val != RN
RP2:  if (cur.val in {LN, DN} and A[k-1] = cur)   // depends on order of RO2 & RO3.
RP3:    return "empty";
RP4:  if CAS(&A[k], next, <RN, next.ctr+1>)
RP5:    if CAS(&A[k-1], cur, <RN, cur.ctr+1>)       // v -> RN
RP6:      return cur.val;
  } // end while

```

Figure 2. Wraparound deque implementation: Right-side operations.

appropriate array entries that prove that the index is correct. If no RN entry exists, then by the third invariant above, there is a DN entry and an LN entry; `rightcheckedoracle` attempts to convert the DN into an RN before returning.

Other than calling `rightcheckedoracle` instead of `oracle(right)` (which also eliminates the need to read and check the `cur` and `next` values again), the only change in the `rightpop` operation is that, in checking whether the deque is empty, `cur.val` may be either LN or DN, because there may be no LN entries.

Because the array is circular, a `rightpush` operation cannot determine whether the array is full by checking whether the returned index is at the end of the array. Instead, it ensures that there is space in the array by checking that `A[k+1].val = RN`. In that case, by the third invariant above, there are at least two null entries other than `A[k]` (which also contains RN), so the deque is not full. Otherwise, `rightpush` first attempts to convert `A[k]` into an RN entry. We discuss how this conversion is accomplished below.

When a `rightpush` operation finds only one RN entry, it tries to convert the next null entry—we know there is one by the third invariant above—into an RN. If the next null entry is an LN entry, then `rightpush` first attempts to convert it into a DN entry. When doing this, `rightpush` checks that `cur.val = RN`, which ensures there is at most one DN entry, as required by the second invariant above. If the next null entry is a DN entry, `rightpush` will try to convert it into an RN entry, but only if the entry to the right of the one being converted (the `nextnext` entry) is an LN entry. In this case, it first increments the version number of the `nextnext` entry, ensuring the failure of any concurrent `leftpush` operation trying to push a value into that entry. If the `nextnext` entry is a deque value, then the `rightpush` operation checks whether the right end of the deque is still at `k` (by rereading `A[k-1]` and `A[k]`), and if so, the deque is full. If not, or if the `nextnext` entry is either an RN or DN entry, then some other operation is concurrent with the `rightpush`, and the `rightpush` operation retries.

Assuming the invariants above, it is easy to see that this new algorithm is linearizable in exactly the same way as the algorithm in the previous section, except that a `rightpush` operation that returns “full” linearizes at the point that `nextnext` is read (line RH11). Because we subsequently confirm (line RH13) that `A[k-1]` and `A[k]` have not changed since they were last read, we know the deque extends from `A[k+2]` to `A[k-1]` (with `A[k-1]` as its rightmost value), so that `A[k]` and `A[k+1]` are the only nonnull entries, and thus, the deque is full.

Proving that the invariants above are maintained by the new algorithm is nontrivial, and we defer a complete proof to the full paper [10]. The main difficulty is verifying that

when a `rightpush` actually pushes the new value onto the deque (line RH5), either the `next` entry is an RN entry, or it is a DN entry and the `nextnext` entry is an LN entry. This is necessary to ensure that after the push, there are still at least two null entries, one of which is an RN or DN entry. One key to the proof is to note that the value of an entry is changed only by lines RO8, RH5, RH9, RH17, RP5, and their counterparts in the left-side operations. Furthermore, these lines only change an entry if the entry has not changed since it was most recently read. These lines are annotated in Figure 2 with how they change the value of the entry.

Time complexity The obvious measure of the time complexity of an obstruction-free algorithm (without regard to the particular contention manager and system assumptions) is the worst-case number of steps that an operation must take in isolation in order to be guaranteed to complete. For our algorithms, this is a constant plus the obstruction-free time complexity of the particular oracle implementation used.

5. Concluding Remarks

We have introduced obstruction-freedom—a new non-blocking condition for shared data structures that weakens the progress requirements of traditional nonblocking conditions, and as a result admits solutions that are significantly simpler and more efficient in the typical case of low contention. We have demonstrated the merits of obstruction-freedom by showing how to implement an obstruction-free double-ended queue that has better properties than any previous nonblocking deque implementation of which we are aware. We are also exploring other obstruction-free algorithms and techniques. Based on our progress to date, we are convinced that obstruction-freedom is a significant breakthrough in the search for scalable and efficient non-blocking data structures. Apart from ongoing work on other obstruction-free algorithms, an important part of our work is in investigating the use of various mechanisms to manage contention in order to allow obstruction-free implementations to make progress even under contention. There has been much work on similar issues, for example in the context of databases. We are working on evaluating existing and new schemes for this purpose. The beauty of obstruction-freedom is that we can modify and experiment with the contention management mechanisms without needing to modify (and therefore reverify) the underlying non-blocking algorithm. In contrast, work on different mechanisms for guaranteeing progress in the context of lock-free and wait-free algorithms has been hampered by the fact that modifications to the “helping” mechanisms has generally required the proofs for the entire algorithm to be done again.

References

- [1] O. Agesen, D. Detlefs, C. Flood, A. Garthwaite, P. Martin, M. Moir, N. Shavit, and G. Steele. DCAS-based concurrent dequeues. *Theory of Computing Systems*, 2003. 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. Wait-free synchronization in multiprogrammed systems: Integrating priority-based and quantum-based scheduling. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 123–132, 1999.
- [3] 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*, pages 119–129, 1998.
- [4] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, 1990.
- [5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1st edition, 1989.
- [6] D. Detlefs, C. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. Steele. Even better DCAS-based concurrent dequeues. In *Proceedings of the 14th International Conference on Distributed Computing*, pages 59–73, 2000.
- [7] M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, August 1999.
- [8] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, pages 123–136, 1996.
- [9] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.
- [10] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In preparation, 2003.
- [11] M. Herlihy, V. Luchangco, and M. Moir. Software transactional memory for supporting dynamic data structures. In preparation, 2003.
- [12] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
- [13] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison-Wesley, 1968.
- [14] P. Martin, M. Moir, and G. Steele. DCAS-based concurrent dequeues supporting bulk allocation. Technical Report TR-2002-111, Sun Microsystems Laboratories, 2002.
- [15] M. Michael. Dynamic lock-free dequeues using single-address, double-word cas. Technical report, IBM TJ Watson Research Center, January 2002.
- [16] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing*, pages 267–276, 1996.
- [17] S. Ramamurthy, M. Moir, and J. Anderson. Real-time object sharing with minimal system support. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 233–242, 1996.