Nonblocking k-compare-single-swap

Victor Luchangco
Sun Microsystems Laboratories
1 Network Drive
Burlington, MA 01803, USA
victor.luchangco@sun.com

Mark Moir
Sun Microsystems Laboratories
1 Network Drive
Burlington, MA 01803, USA
mark.moir@sun.com

Nir Shavit
Tel Aviv University
Tel Aviv 69978, Israel
shanir@cs.tau.ac.il

ABSTRACT

The current literature offers two extremes of nonblocking software synchronization support for concurrent data structure design: intricate designs of specific structures based on single-location operations such as compare-and-swap (CAS), and general-purpose multilocation transactional memory implementations. While the former are sometimes efficient, they are invariably hard to extend and generalize. The latter are flexible and general, but costly. This paper aims at a middle ground: reasonably efficient multilocation operations that are general enough to reduce the design difficulties of algorithms based on CAS alone.

We present an obstruction-free implementation of an atomic *k-location-compare single-swap* (KCSS) operation. KCSS allows for simple nonblocking manipulation of linked data structures by overcoming the key algorithmic difficulty in their design: making sure that while a pointer is being manipulated, neighboring parts of the data structure remain unchanged. Our algorithm is efficient in the common uncontended case: A successful k-location KCSS operation requires only two CAS operations, two stores, and $2k$ noncached loads when there is no contention. We therefore believe our results lend themselves to efficient and flexible nonblocking manipulation of list-based data structures in today's architectures.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Algorithms, Theory

Keywords

Multiprocessors, nonblocking synchronization, concurrent data structures, linked lists, obstruction-freedom

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SPAA'03, June 7–9, 2003, San Diego, California, USA.
Copyright 2003 Sun Microsystems, Inc. All rights reserved.
ACM 1-58113-661-7/03/0006 ...$5.00.

1. INTRODUCTION

The implementation of concurrent data structures is much easier if one can apply atomic operations to multiple memory locations [7]. Current architectures, however, support atomic operations only on small, contiguous regions of memory (such as a single or double word) [6, 18, 27, 29].

The question of supporting multilocation operations in hardware has in recent years been a subject of debate in both industrial and academic circles. Until this question is resolved, and to a large extent to aid in its resolution, there is an urgent need to develop efficient software implementations of atomic multilocation operations.

The current literature (see the survey in Section 5) offers two extremes of software support for concurrent data structure design. On one hand, there are intricate designs of specific constructs based on single-location synchronization primitives such as *compare-and-swap* (CAS). On the other hand, there are general-purpose implementations of multilocation software transactional memory. While the former lead to efficient designs that are hard to extend and generalize, the latter are very general but costly. This paper aims at the middle ground: reasonably efficient multilocation operations that offer enough generality to reduce the design difficulties of algorithms based on CAS alone.

1.1 K-compare single-swap

We present a simple and efficient nonblocking[1] implementation of an atomic *k-location-compare single-swap* (KCSS) operation. KCSS verifies the contents of k locations and modifies one of them, all as a single atomic operation. Our KCSS implementation, when executed without contention, requires only two CAS operations, two stores, and $2k$ noncached loads. It requires no memory barriers under the TSO memory model [29]. As we show, this compares favorably with all implementations in the literature.

The nonblocking progress condition our implementation meets is *obstruction-freedom*. Obstruction-freedom is a new progress condition, proposed by Herlihy, Luchangco, and Moir [14] to simplify the design of nonblocking algorithms by removing the need to provide strong progress guarantees in the algorithm itself (as required by wait-freedom or lock-freedom [11]). Simply put, obstruction-freedom guarantees a thread's progress if other threads do not actively interfere for a sufficient period. The definition is thus geared towards

[1]We use "nonblocking" broadly to include all progress conditions requiring that the failure or indefinite delay of a thread not prevent other threads from making progress, rather than as a synonym for "lock-free", as some authors prefer.

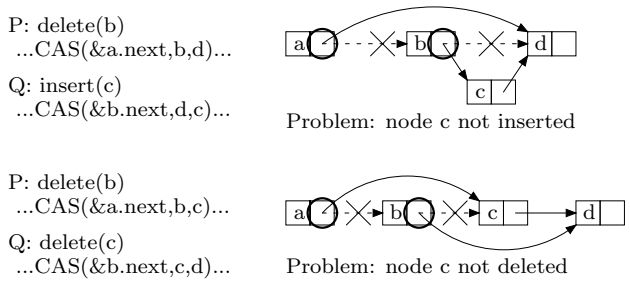

Figure 1: CAS-based list manipulation is hard (the examples slightly abuse CAS notation). In both examples, P is deleting b from the list. In the upper example, Q is trying to insert c into the list, and in the lower example, Q is trying to delete c from the list. Circled locations indicate the target addresses of the CAS operations; crossed out pointers are the values before the CAS succeeds.

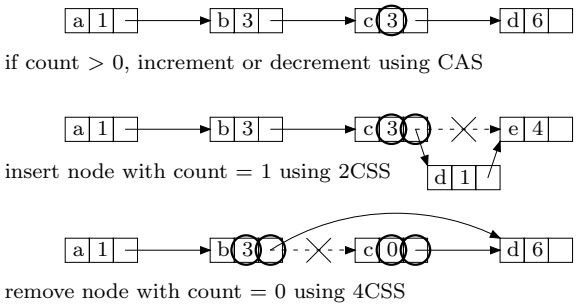

Figure 2: Illustration of a KCSS-based multiset implementation. Each element in the multiset (i.e., an element with nonzero multiplicity) is represented by a node in the list, which stores the element's multiplicity in a count field. Inserts or deletes of such elements respectively increment or decrement the count (top figure). Two- and four-location KCSS operations are used to add and remove nodes by swapping one pointer, while confirming nearby nodes have not changed (middle and bottom).

the uncontended case, handling contended cases through orthogonal contention management mechanisms. (Contention management mechanisms similar in spirit to the ones discussed in [15] are applicable to the algorithms presented here; we do not discuss contention management further in this paper.) Lock-based algorithms are not obstruction-free because a thread trying to acquire a lock can be blocked indefinitely by another thread that holds the lock. On the other hand, any lock-free algorithm is also obstruction-free because lock-freedom guarantees progress by some thread if some thread continuously take steps.

1.2 Manipulating linked data structures

KCSS is a natural tool for linked data structure manipulation; it allows a thread, while modifying a pointer, to check atomically that related nodes and pointers have not changed. An application of immediate importance is the implementation of nonblocking linked data structures with arbitrary insertions and deletions. As Figure 1 shows, naive approaches to implementing nonblocking insertion and deletion operations for a linked list using single-location CAS do not work. Although there exist effective (and rather ingenious) nonblocking algorithms for ordered list-based sets [8, 21], these algorithms do not generalize easily to arbitrary linked data structures. For example, it is not clear how to modify these algorithms to implement multisets.

By employing KCSS instead of CAS, we can simplify the design of arbitrary nonblocking linked-list operations. For example, Figure 2 illustrates how the use of KCSS can significantly simplify the design of a linked-list construct to support multiset operations (details of the illustrated algorithms are beyond the scope of this paper). For simplicity, the illustrated implementation uses a 4CSS operation to make sure the adjacent nodes have not changed during node removal; we can achieve the same purpose using KCSS operations that access only two locations at the cost of a slightly more intricate algorithm. However, adding a small number of additional locations to a KCSS operation is not prohibitive because the cost of verifying each additional location is only two noncached loads, a worthwhile tradeoff in many cases.

Toward building our KCSS algorithm, we provide a simple and novel implementation of *load-linked/store-conditional* (LL/SC) using CAS; this implementation improves on previous results in that it can accommodate pointer values on all common architectures. We believe this algorithm is of independent value: it extends the applicability of LL/SC-based algorithms to all common architectures that support CAS. This answers a question that remained open following the work of Moir [24].

Section 2 defines the semantics of the operations for which we present implementations in Section 3. In Section 4, we discuss various optimizations, generalizations, and extensions that are possible. In Section 5, we survey previous work on multilocation synchronization. We conclude in Section 6. A correctness proof appears in the appendix.

2. PRELIMINARIES

A k-location-compare single-swap (KCSS) operation takes k locations $a_1..a_k$, k expected values $e_1..e_k$, and a new value n_1. If the locations all contain the expected values, the KCSS operation atomically changes the first location a_1 from e_1 to n_1 and returns *true*; in this case, we say that the KCSS *succeeds*. Otherwise, the KCSS returns *false* and does not modify any memory location; in this case we say that it *fails*. In the next section, we present an implementation for KCSS using special read, load-linked (LL), store-conditional (SC) and snapshot operations that we have also implemented. In this section, we describe more precisely the interface and semantics of the various operations, the correctness requirements, and our assumptions about the system.

2.1 Semantics of operations

We now describe the semantics of the operations for which we provide implementations in the next section. We consider a collection of *locations*. At any point in time, each location has an *abstract value* from a set of *application values*. (As explained in the next section, our implementation requires some mild restrictions on this set.)

A KCSS($k, a[1..k], expvals[1..k], newval$) operation returns *false* iff for some $i \in [1, k]$, the abstract value of location $a[i]$ differs from $expvals[i]$. If this operation returns *true*, then it also changes the abstract value of location $a[1]$ to *newval*. The locations specified by a must all be distinct.

READ(a) and LL(a) operations return the abstract value of location a. An LL operation of thread p is said to be *outstanding* until p invokes an SC operation on the same location. The behaviour of all operations is undefined if LL or KCSS is invoked by process p while p has an outstanding LL operation on any location. (It is straightforward to remove this restriction, but retaining it simplifies our presentation.)

The behaviour of an SC(a) operation S by process p is undefined if it is invoked before any LL(a) operation by process p has completed, or if there is not a previous LL(a) operation L by process p such that there is no LL, SC or KCSS operation invoked by process p between L and S.[2] Otherwise, an SC(a) operation by process p returns *true* only if no other operation that changes the abstract value of location a has occurred since the preceding LL(a) operation by process p. (We say that a SC operation *succeeds* if it returns *true*, and *fails* otherwise.) To ensure that this operation is useful for implementing obstruction-free data structures, we further require that an SC(a) operation succeeds if no other operation that accesses location a takes a step between the invocation of p's preceding LL(a) operation and the completion of the SC(a) operation. (Observe that this specification allows a concurrent READ(a) operation to cause a SC(a) operation to fail; in fact, it would do so in our implementation. Although this possibility does not jeopardize obstruction-freedom, eliminating it would allow some concurrent operations to succeed that would otherwise fail, and thus, may be desirable. Our implementation can easily be modified to come close to this goal; see Section 4.)

A SNAPSHOT($m, a[1..m]$) operation returns an array $V[1..m]$ such that, for each $i \in [1, m]$, $V[i]$ is the abstract value of location $a[i]$. The locations specified by a must be distinct.

2.2 Correctness condition

We present obstruction-free, linearizable implementations of the operations described above in the next section. *Linearizability* [17] implies that each operation appears to take effect instantaneously at some point between its invocation and its response; this point is the operation's *linearization point*. *Obstruction-freedom* [14] requires that if a thread p executes an operation, and after some point p runs without interference for long enough, then that operation will terminate.

2.3 Interoperability with dynamic data structures and memory management

In our implementations of the above operations, each location initially holds its initial abstract value. Thus, locations can be dynamically allocated and initialized by a single thread, which is important for dynamic-sized data structures. Our implementations also allow a location to be freed if no operation that specifies this location as an argument is executing or will be invoked. Furthermore, they guarantee that there will always be a pointer to an object that could be accessed in the future. Thus, our operations do not affect memory management, and in particular, data structures based on our implementations "play nicely" with garbage collection[3] and nonblocking memory management techniques such as those in [13, 22].

2.4 System model

We assume a machine architecture that supports linearizable load, store, and CAS operations. It is straightforward to transform these algorithms to work in systems that provide LL and SC instead of CAS [24].[4]

The semantics of CAS is equivalent to the following atomic code fragment:

```
bool CAS(loc *a, value expval, value newval){
  atomically{
    if (*a != expval)
      return false;
    *a = newval;
    return true;
  }
}
```

Although we assume linearizability, our algorithms are correct on multiprocessors that provide only the TSO memory model [29], without adding memory barrier instructions; this is a side effect of the way we use CAS.

3. OUR IMPLEMENTATIONS

We now describe our implementations of the READ, LL, SC, SNAPSHOT, and KCSS operations. We begin by explaining a restricted version of the LL, SC, and READ operations, which is correct if we need only these operations. We then explain how LL can be modified slightly to support a simple SNAPSHOT operation. Finally we explain how we implement KCSS using LL, SC, and SNAPSHOT.

Recall that an SC(a, v) operation by process p should succeed only if no other operation that modifies location a is linearized between the linearization points of p's preceding LL(a) operation and p's SC(a, v) operation. To overcome the ABA problem [23], previous implementations of LL/SC from CAS (e.g., [24]) have required special "tags" or "version numbers" to be stored together with the application value in a location that can be accessed by CAS. This requirement severely restricts the range of values that can be stored by those SC implementations, and in particular, makes these implementations inapplicable for storing pointers in many architectures.

Our goal is to design implementations that place much milder restrictions on the set of application values, in particular so that our implementations can access pointers on all common multiprocessor architectures. Below we specify these restrictions, which are too weak to allow tag/version

[2]This means that LL and SC should be used in pairs on the same location. Moir presents an efficient technique for generalizing LL/SC implementations so that LL/SC sequences can be executed concurrently on different locations [24]; this technique can be applied to the results presented here, so we do not address this restriction further in this paper.

[3]The garbage collector would need to be modified slightly to distinguish between pointers and tagged ids, which are described in the next section.

[4]In this case, LL and SC should be directly used to replace the use of CAS in our implementations; native LL and SC cannot replace our implementations of the LL and SC operations because our implementations of these operations are designed to be compatible with the SNAPSHOT operation.

number techniques, and then explain how we can achieve our implementations despite these weaker restrictions.

Each location can store either an application value or a *tagged process id*. The abstract value of a location that contains an application value is always that value; when the location contains a tagged id, it is a little more complicated, as we explain below. A tagged process id (*tagged id* for short) contains a process id and a tag.

The only restriction we place on application values is that we have some way to distinguish them from tagged ids. One simple way to achieve this when the application value of interest is a pointer is to "steal" the low-order bit to mark tagged ids: we can arrange that all locations are aligned on even byte boundaries so that the low-order bit of every pointer is zero (locations that will be targets of CAS instructions are usually required to be word-aligned anyway).

For convenience, we treat tags as if they were unbounded integers. In today's 64-bit architectures, we can use one bit to distinguish tagged ids, 15 bits for the process id and 48 bits for the tag, which, as discussed elsewhere [24], is more than enough to avoid the ABA problem that potentially arises as the result of tags wrapping around.

3.1 LL and SC

We now explain a simplified version of our implementations of the LL and SC operations. The code is shown in Figure 3. For the purposes of this simplified version, the reader should ignore the `tid` field of the location record (i.e., a location record is simply a memory location that contains an application value or a tagged id), and any code that accesses it, namely line 9.

In order to implement LL(a) and SC(a, v) operations for process p, we need a way to determine whether the abstract value of location a has changed since the LL(a) operation was linearized. Our approach is to have p's LL(a) operation store a previously unused tagged id in location a (line 8). We ensure that the tagged id is new by having p maintain a local tag, which it increments each time it needs a new tagged id (line 5). As explained below, we do not allow any operation that changes the abstract value of location a to be linearized while the tagged id of another process is in that location. Thus, if the SC(a, v) operation changes the contents of location a from the tagged id stored by the preceding LL(a) of the same process to v (i.e., the CAS in line 11 succeeds), then it changes the abstract value of location a to v while ensuring that the abstract value of location a has not changed since the previous LL(a) operation, as required.

Of course, to guarantee obstruction-freedom, it is not sufficient to prevent other operations from being linearized between the linearization points of p's LL(a) and SC(a, v) operations: we must guarantee that a thread that runs without interference will make progress. Therefore, it must be possible for a concurrent operation to remove p's tagged id from location a (thereby causing p's SC to fail), without changing the abstract value of location a; this is achieved by the auxiliary RESET operation, which is explained below. To make this possible, before attempting to store a new tagged id in location a, p's LL(a) operation first stores the application value it intends to replace with its tagged id (line 6) in a special location VAL_SAVE[p] (line 7). (Recall that p can have at most one outstanding LL operation, so a single location is sufficient.) For now, we can consider the abstract value of a location that contains a tagged id of process p to be

```
typedef struct loc_s {
  taggedid_t tid; // used for SNAPSHOT
  value_t val;    // atomically CASable
} loc_t;

void RESET(loc_t *a){
1:  value_t oldval = a->val;
2:  if(TAGGED_ID(oldval))
3:  CAS(&a->val, oldval, VAL_SAVE[ID(oldval)]); }

value_t LL(loc_t *a){
4:  while (true) {
5:    INC_MY_TAGGED_ID;          // increment local tag
6:    value_t val = READ(a);
7:    VAL_SAVE[MY_ID] = val;
8:    if (CAS(&a->val, val, MY_TAGGED_ID)) {
9:      a->tid = MY_TAGGED_ID;   // needed for SNAPSHOT
10:     return val;
      }
    }
}

bool SC(loc_t *a, value_t newval){
11: return CAS(&a->val, MY_TAGGED_ID, newval);
}

value_t READ (loc_t *a){
12: while (true) {
13:   value_t val = a->val;
14:   if (!TAGGED_ID(val)) return val;
15:   RESET(a);
    }
}
```

Figure 3: The code for LL and SC.

VAL_SAVE[p].[5] Thus, it is easy to see that when p's LL(a) operation replaces the application value in location a with a tagged id (line 8), the abstract value of location a does not change. Similarly, another operation that uses CAS to remove p's tagged id can correctly determine the abstract value of location a in order to replace p's tagged id with the correct abstract value by reading VAL_SAVE[p] (line 3). (Process p does not change VAL_SAVE[p] while any location contains its tagged id. Also, there is no ABA problem when either p or another process removes p's tagged id from location a, because p uses a fresh tagged id each time it stores a tagged id in location a and only process p stores a tagged id with p's process id.)

This completes the description of the LL(a) and SC(a, v) operations, except that we have not explained the READ(a) operation, which is used by LL(a) (line 6).

3.2 READ

The READ operation must determine the abstract value of location a. It first reads location a (line 13). If the value read is an application value, then this was the abstract value of location a when line 13 was executed, so it can be returned (line 14). Otherwise, the abstract value of location a when line 13 was executed was VAL_SAVE[p] where p is the process whose id is in the tagged id read at line 13. Simply reading that location would not necessarily provide the correct abstract value of location a because p might have changed the contents of this location since the READ(a) operation executed line 13. However, because there can be no ABA problem on tagged ids, the READ(a) operation could

[5] We later change this interpretation slightly to accommodate the KCSS operation, as described in Section 3.4.

read VAL_SAVE[p] and then reread location a to confirm that the same tagged id is still in location a. In this case, it could correctly linearize a read of the abstract value of location a at any point between the two reads of location a. If we wanted to support only LL, SC, and READ operations, this would be correct and would allow a location to be read without causing a concurrent LL/SC sequence on the same location to fail. However, in Figure 3, if a READ operation encounters a tagged id, it calls RESET in order to attempt to set location a back to its abstract value. As explained later, this is necessary to support the SNAPSHOT and KCSS operations that are presented next.

3.3 SNAPSHOT

A well-known nonblocking technique, originally suggested by Afek et al. [1], for obtaining an atomic snapshot of a number of locations is the following: We repeatedly "collect" (i.e., read each location individually and record the values read) the values from the set of locations until we encounter a collect in which none of the values collected has changed since it was read in the previous collect. In this case, it is easy to see that, when the first value is read in the last collect, all of the values read during the previous collect are still in their respective locations. The only tricky detail is how to determine that a value has not changed since the last time it was read. Because of the ABA problem, it is not sufficient to simply determine that the two values read were the same: the location's value may have changed to a different value and then changed back again between these two reads. As explained below, we can determine a value has not changed using the tid field (which we have been ignoring until now) associated with each location. This field serves the same purpose as the tags (or version numbers) discussed earlier. However, our implementation does not require them to be modified atomically with the val field, and therefore does not restrict applicability, as discussed earlier.

The code for SNAPSHOT is presented in Figure 4. First, observe that the basic structure (if we ignore tags for a moment longer) is essentially as described above: we collect the set of values twice (lines S7 and S8) and retry if any of the values changed between the first read and the second (line S10). Observe further that COLLECT_VALUES uses READ to read the value of each location. Thus, it ensures that the abstract value it reads from a location a is stored in location a itself. As described earlier, for the abstract value of a location to change, some process must install a fresh tagged id in that location and subsequently change that tagged id to the new abstract value. This entire sequence must occur between the READ in the first collect and the READ in the second. Therefore, line 9 of the LL operation, which stores the fresh tagged id in the tid field of the location, must be executed between the first and second reads of the tid field by the SNAPSHOT operation, which will therefore retry (see lines S6 and S9). This argument is simple, but it depends on the fact that READ resets a location that contains a tagged id. In Section 4, we explain how our implementation can be modified to avoid this requirement.

3.4 KCSS

Our KCSS implementation, shown in Figure 5, is built using the operations described above. The implementation itself is very simple, but the linearization argument is trickier. The basic idea is to use LL and SC to change the value

```
value_t[1..m] COLLECT_VALUES(int m, (loc_t *) a[1..m]){
    value_t V[1..m];
S1: for (i = 1; i <= m; i++) V[i] = READ(a[i]);
S2: return V;
}

tag_t[1..m] COLLECT_TAGGED_IDS(int m, (loc_t *) a[1..m]){
    taggedid_t T[1..m];
S3: for (i = 1; i <= m; i++) T[i] = a[i]->tid;
S4: return T;
}

value_t[1..m] SNAPSHOT(int m, (loc_t *)[1..m] a){
    taggedid_t  TA[1..m], TB[1..m];
    value_t VA[1..m], VB[1..m];
S5: while (true) {
S6:   TA[1..m] = COLLECT_TAGGED_IDS(m,a);
S7:   VA[1..m] = COLLECT_VALUES(m,a);
S8:   VB[1..m] = COLLECT_VALUES(m,a);
S9:   TB[1..m] = COLLECT_TAGGED_IDS(m,a);
S10:  if (for all i, (TA[i] == TB[i]) &&
                     (VA[i] == VB[i]))
S11:    return VA;
  }
}
```

Figure 4: The SNAPSHOT code.

```
bool KCSS(int k, (loc_t *) a[1..k],
          value_t expvals[1..k], value_t newval){
    value_t  oldvals[1..k];
K1: while (true) {
K2:   oldvals[1] = LL(a[1]);
K3:   oldvals[2..k] = SNAPSHOT(k-1,a[2..k]);
K4:   if (for some i, oldvals[i] != expvals[i])
K5:     SC(a[1], oldvals[1]);
K6:     return false;
      // try to commit the transaction
K7:   if (SC(a[1], newval)) return true;
    } // end while
}
```

Figure 5: The KCSS code.

of location a[1] from expvals[1] to newval (lines K2 and K7), and to use SNAPSHOT to confirm that the values in locations a[2..k] match expvals[2..k] (lines K3 and K4). If any of the k values is observed to differ from its expected value (line K4), then the KCSS and returns *false*, as required (line K6). However, before returning, it attempts to restore a[1] to expvals[1] using SC (line K5), so that the previous LL operation is no longer outstanding, and thus, the process may subsequently invoke another LL or KCSS operation.

If the SC in line K7 succeeds, then we know that the abstract value of location a[1] is expvals[1] for the entire interval between the linearization point of the LL in line K2 and the linearization point of the SC in line K7. In particular, this holds at the linearization point of the SNAPSHOT called in line K3, when the abstract values of a[2..k] match expvals[2..k]. Thus, we can linearize the successful KCSS operation at that point. This is where the linearization argument becomes slightly tricky: The actual value in location a[1] does not change to newval until the SC in line K7 is linearized. However, the *abstract* value of that location changes at the linearization point of the KCSS operation, which occurs earlier. Therefore, if any other operation observes the abstract value of that location between the linearization points of the SNAPSHOT in line K3 and the SC in line K7, it will see the wrong abstract value and the implementation will not be linearizable. To prevent this problem, we require READ

to reset a location, rather than simply reading the VAL_SAVE entry of the process whose tagged id is in the location, and then confirming that the tagged id is still in the location (as described earlier). This ensures that no process observes the wrong abstract value in the interval between the SNAPSHOT and the successful SC. As described in the next section, we can relax this requirement somewhat; we have presented our implementations without these modifications in order to keep the presentation simple and clear.

4. OPTIMIZATIONS, EXTENSIONS, AND GENERALIZATIONS

From the basic ideas we have presented in this paper, numerous possible optimizations, extensions, and generalizations are possible. We describe a few of them here.

4.1 Optimizing READ

Our READ operation can be optimized by observing that if the CAS in line 3 of Figure 3 succeeds, then we have already determined the abstract value of the location being accessed, which can be returned immediately without rereading.

4.2 Improving concurrency

As stated earlier, we can modify our implementation so that READ does not always have to reset a location that contains a tagged id: in some cases, reading a value from the VAL_SAVE location of the process whose tagged id is encountered, and then confirming that the tagged id is still in the location is sufficient to determine the correct abstract value. This does *not* work, however, in cases in which we linearize a modification to the location accessed by a LL/SC pair at a point other than the linearization point of the SC operation. In the operations we have presented, this is the case only for LL/SC sequences that are part of a higher-level KCSS operation. Therefore, if we extend the interface of LL so that the invoker can specify whether or not this is a "dangerous" use of LL/SC, then this information could be stored in the tagged id. Thus, READ could reset only when it encounters such LL/SC sequences, while allowing other, simpler uses of LL/SC to proceed concurrently.

This modification would complicate the SNAPSHOT implementation slightly. Recall that the argument given earlier for the linearizability of SNAPSHOT operations depends on READ always resetting a location if it contains a tagged id. This can be overcome by having SNAPSHOT also collect the tagged ids from locations for which it has determined values without resetting the location. As this would be done only if the tagged id is in the location on behalf of a nondangerous LL/SC sequence, the abstract value of the location does not change before that tagged id is removed from the location, so it is sufficient for SNAPSHOT to confirm that it has not.

4.3 DCSS and CAS

To implement a *double-compare single-swap* (DCSS) operation (i.e., KCSS with $k = 2$), we can replace the SNAPSHOT of $k - 1 = 1$ location in our KCSS implementation with a simple READ. Similarly, for a CAS on these locations, which is simply a KCSS operation with $k = 1$, the snapshot can be eliminated entirely.

In some cases, such as the multiset example mentioned earlier, locations that support only read, CAS and DCSS operations are sufficient. In these cases, we can eliminate the tid field (and the code that accesses it), as this field is necessary only for the SNAPSHOT operation. We can also implement CAS by using the native CAS instruction, resetting the location if it contains a tagged id.

4.4 Optimizations to SNAPSHOT and KCSS

The implementation of SNAPSHOT can be improved at the cost of muddying the presentation. For example, the tags collected at line S9 can be used for the first set of tags in the next iteration (we collect the tags again in the next iteration at line S6). Also, Harris [9] has noted that one can eliminate a complete sequence of reads from the snapshot, at the cost of a slightly more complex proof. We can also improve the performance of the KCSS by breaking the snapshot abstraction (for example, there is no need to take an entire snapshot if one of the early values read does not match the expected value).

4.5 Single-modification transactions

We chose the KCSS API to demonstrate our ideas because its semantics is easy to state and understand. However, as we will show in the full paper, the ideas presented here can easily be extended to support *transactions* that modify only a single location. The basic idea is to have transactional loads record the information collected in the first half of the snapshot in our KCSS implementation, and transactional commit do the second half of the snapshot to determine if any of the values read had been modified by a concurrent operation since being read by the transactional load. Interestingly, the implementation of this stronger semantics would actually be somewhat more efficient than using READ and KCSS for the same purpose, because the READs and the first half of the snapshot in KCSS are collapsed together into the transactional load. It would also be straightforward to provide a transactional "validate" operation that rechecks the values read so far in the transaction.

We believe that the ability provided by KCSS to "confirm" the abstract value of some locations, while modifying another, will significantly reduce the impact of ABA issues on algorithm designers. However, such issues may still arise in some cases, and implementing the transactions as discussed above would completely relieve designers of the burden of dealing with this problem.

5. RELATED WORK ON MULTILOCATION SYNCHRONIZATION

The idea to support atomic multilocation synchronization operations dates at least back to the Motorola MC68030 chip [25], which supported a double-compare-and-swap operation (DCAS); this operation generalizes CAS to allow atomic access to two locations. DCAS has also been the subject of recent research [2, 5, 7]. In a seminal paper [16], Herlihy and Moss suggested *transactional memory*, a broader transactional approach where synchronization operations are executed as optimistic atomic transactions in hardware.

Barnes was the first to introduce a software implementation of a k-location read-modify-write [4]. His implementation, as well as those of Turek, Shasha and Prakash [28], and Israeli and Rappoport [19], is based on the *cooperative method*: threads recursively help all other threads until the operation completes. Unfortunately, this introduced significant redundant "helping": threads do the work of other

threads on unrelated locations because a chain of dependencies among operations exists. As shown by Attiya and Dagan [3], this kind of "helping" overhead for lock-free multilocation operations is, in the worst case, unavoidable.

Shavit and Touitou [26] coined the term *software transactional memory* (STM) and presented the first lock-free implementation of an atomic multilocation transaction that avoided redundant "helping" in the common case, and thus significantly outperformed other lock-free algorithms [4, 19, 28]. However, their STM was restricted to "static" transactions, in which the set of memory locations to be accessed is known in advance, and its correctness proof was significantly complicated by the need to argue about the lock-free properties of the helping mechanism.

Recently, Herlihy, Luchangco, Moir and Scherer introduced an obstruction-free implementation of a general STM that supports "dynamic" transactions [15]. Their implementation avoids helping altogether, thereby reducing the complexity of the algorithm and eliminating the overhead of redundant helping. As explained earlier, the design of obstruction-free operations is geared towards being simple and efficient in the uncontended case, where helping would not be needed, while allowing contended cases to be handled through orthogonal contention management mechanisms instead of helping mechanisms. While KCSS can be implemented using [15] with only slightly higher time complexity and twice the memory overhead of our `KCSS` implementation, the resulting algorithm would be a complex software application that would require that fresh memory be allocated per executed transaction or that a specialized built-in memory management scheme be introduced with the algorithm.

Nonblocking implementations of a simpler `KCAS` operation, that is, k-location compare-and-swap on nonadjacent locations, were presented by Herlihy et al. [12] and by Harris et al. [10]. However, implementing KCSS using these algorithms requires at least $2k+1$ CAS operations, making them prohibitively expensive.

Our `KCSS` implementation exploits the weaker semantics of `KCSS` and the weaker progress requirement of obstruction freedom to achieve an algorithm that provides the simplicity of [10, 12] at a significantly lower cost.

6. CONCLUDING REMARKS

We have presented a simple and efficient nonblocking implementation of a dynamic collection of locations that supports `READ`, `LL`, `SC`, `SNAPSHOT` and `KCSS` operations. We have also explained a simple extension by which we can support transactions that modify at most one location. These operations form a powerful set of tools for designing relatively simple obstruction-free implementations of important shared data structures such as linked lists.

Because of the novel way in which we solve the ABA problem, our implementation is more efficient, more flexible, and more widely applicable for implementing linked data structures than the techniques used in recent direct implementations of lock-free linked lists. That we were able to achieve this implementation with relatively simple algorithms furthers our belief that the introduction of obstruction-freedom is an important development in the search for simple, efficient and scalable nonblocking shared data structures.

As stated earlier, contention control mechanisms are required to ensure progress with obstruction-free algorithms; future work includes evaluating contention control mechanisms specifically for the implementations presented here.

Acknowledgments: We thank Paul Martin for valuable comments on initial versions of our algorithm, and Tim Harris for his many comments and for suggesting optimizations to the snapshot algorithm. This work was done while Nir Shavit was visiting Sun Labs.

7. REFERENCES

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993.

[2] 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*, 35:349–386, 2002. A preliminary version appeared in the Proc. 12th ACM Symposium on Parallel Algorithms and Architectures.

[3] H. Attiya and E. Dagan. Universal operations: Unary versus binary. In *Proc. 15th Annual ACM Symposium on Principles of Distributed Computing*, May 1996.

[4] G. Barnes. A method for implementing lock-free shared data structures. In *Proc. 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, June 1993.

[5] D. Detlefs, P. Martin, M. Moir, and G. Steele. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002. A preliminary version appeared in the Proc. 20th Annual ACM Symposium on Principles of Distributed Computing, 2001.

[6] A. Glew and W. Hwu. A feature taxonomy and survey of synchronization primitive implementations. Technical Report CRHC-91-7, University of Illinois at Urbana-Champaign, Dec. 1990.

[7] M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Aug. 1999.

[8] T. Harris. A pragmatic implementation of non-blocking linked lists. In *Proc. 15th International Symposium on Distributed Computing*, 2001.

[9] T. Harris. Personal communication, Dec. 2002.

[10] T. Harris, K. Fraser, and I. Pratt. A practical multi-word compare-and-swap operation. In *Proc. 16th International Symposium on Distributed Computing*, 2002.

[11] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 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 structures. In *Proc. 16th International Symposium on Distributed Computing*, 2002.

[14] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. 23rd International Conference on Distributed Computing Systems*, 2003.

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

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

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

[18] Intel. Pentium processor family user's manual: Vol 3, architecture and programming manual, 1994.

[19] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proc. 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 151–160, 1994.

[20] N. Lynch and F. Vaandrager. Forward and backward simulations – part I: Untimed systems. *Information and Computation*, 121(2):214–233, 1995.

[21] M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proc. 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, 2002.

[22] M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proc. 21st Annual ACM Symposium on the Principles of Distributed Computing*, 2002.

[23] 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.

[24] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proc. 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, 1997.

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

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

[27] R. Sites. Alpha architecture reference manual, 1992.

[28] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: Making lock-based concurrent data structure algorithms nonblocking. In *Proc. 11th ACM Symposium on Principles of Database Systems*, pages 212–222, 1992.

[29] D. Weaver and T. Germond. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.

APPENDIX

A. CORRECTNESS PROOF

In this appendix, we argue that the operations in Section 3 have the correct semantics (as defined in Section 2). We first argue that they are linearizable, and then that they are obstruction-free. Our proof is intended to be informal enough to avoid undue burden on the reader, but precise enough that it can be easily translated to a more formal and detailed proof.

A.1 Linearizability proof

For each operation, we define its *linearization point*, that is, a point during the operation's execution at which it appears to take effect. The linearization points are as follows:

- A `READ` operation linearizes to its last execution of line 13.
- An `LL` operation linearizes to its last (and only) successful CAS operation.
- An `SC` operation linearizes to its CAS operation.
- A `SNAPSHOT` operation linearizes to an arbitrary point between the two `COLLECT_VALUES` operations invoked in the `SNAPSHOT`'s last iteration of the loop.
- A `KCSS` operation that succeeds linearizes to the linearization point of its last `SNAPSHOT` operation. A `KCSS` operation that fails because the comparison in statement K4 of Figure 5 fails with $i = 1$ is linearized to the linearization point of the last `LL` operation in statement K2. A `KCSS` operation that fails because the comparison in statement K4 of Figure 5 fails with $i \neq 1$ is linearized to the linearization point of the `KCSS` operation's last `SNAPSHOT` operation.

Each `READ`, `LL`, `SNAPSHOT` or `KCSS` operation consists of a while loop containing an *attempt* to complete the operation. An attempt is *conclusive* if the operation returns in that attempt; otherwise, the attempt is *inconclusive* and another attempt is made. Each completed operation has exactly one conclusive attempt (its last attempt), and the linearization of the operation occurs in that attempt. Note that each attempt of an `LL` or `KCSS` operation is associated with a unique tagged id.

For `SNAPSHOT` and `KCSS` operations, it is not possible to determine the linearization point at the time of the linearization point: whether the attempt is conclusive depends on later events. To simplify the proof and avoid the need for backward simulation style arguments [20], we consider only *complete* execution histories, that is, ones in which all abstract operations have completed.

Based on the linearization points defined above, it is easy to determine the abstract value of any location a at any point in a complete execution history:

- If the value field of a contains an application value, then the abstract value of a is that application value.
- If the value field of a is a tagged id that corresponds to a successful `KCSS` operation whose linearization point is past (i.e., after the linearization point of the `SNAPSHOT` invocation in its last attempt), then the abstract value of a is the `newval` argument passed to that `KCSS` operation.
- Otherwise, the abstract value of a is `VAL_SAVE`$[p]$, where p is the process whose tagged id is in the value field of a.

Before arguing that every operation is linearizable, we recall the restriction that a thread may not invoke `LL` or `KCSS` while it has any outstanding `LL` operation. Given this restriction, it is easy to prove the following claim.

Claim 1. For any thread p executing an `LL` operation, if p has not yet successfully CAS'd its tagged id into the value field of a, or if p is executing a `KCSS` operation whose `LL` has not done so, then no location's value field contains a tagged id of p.

We now argue that the auxiliary RESET operation does not change the abstract value of any location.

LEMMA 1. *RESET does not change the abstract value of any location.*

PROOF. If, after line 1, oldvals is an application value, or if the RESET's CAS operation fails, then RESET does not change the value field of the location, so the abstract value is unchanged.

If the value is a tagged id and the RESET successfully changes the value field, then the tagged id cannot be associated with the conclusive attempt of a successful KCSS operation because the SC of that attempt will fail. Thus, immediately preceding the CAS operation, the abstract value of the location is VAL_SAVE[ID(oldvals)], so it is not changed by the CAS operation. (Because statement 3 of the RESET operation first reads VAL_SAVE[ID(oldvals)] and then attempts the CAS, it is conceivable that VAL_SAVE[ID(oldvals)] changes between the read and the successful CAS. However, this is not possible, because once a process has installed a tagged id into a location, it does not modify its VAL_SAVE entry again until after it has ensured that the tagged id has been removed. If the tagged id has been removed, then the CAS under consideration fails.) □

For READ and LL operations, we must verify that they return, and do not change, the abstract value of the location.

LEMMA 2. *The READ(a) operation returns the abstract value of a at its linearization point, and does not change the abstract value of any location.*

PROOF. A READ operation is linearized at the load (of the value field) of its conclusive attempt. At that time, the field contained an application value, so its value was the abstract value for that location. The only operation within a READ operation that may write a location is the RESET operation, which by Lemma 1, does not change the abstract value of any location. □

LEMMA 3. *The LL(a) operation returns the abstract value of a at its linearization point, and does not change the abstract value of any location.*

PROOF. An LL operation is linearized at the CAS operation of its conclusive attempt. Immediately before the CAS, the value field of a contains the application value val (returned by the preceding READ operation), so its abstract value is val, which the LL operation will return.

To see that an LL operation does not change the abstract value of any location, note that, by Lemma 2, the READ operation does not change the abstract value of any location. By Claim 1, no location has a tagged id of the thread executing the LL operation, so changing the corresponding entry in the VAL_SAVE array does not change the abstract value of any location. Finally, immediately after the CAS, the value field of a contains the tagged id corresponding to this attempt, and the corresponding entry in the VAL_SAVE array contains val (written on the previous line), so the abstract value of a does not change. □

LEMMA 4. *Suppose process p executes LL(a), and its next LL, SC or KCSS operation is SC(a, newval). If the SC operation returns false, then it does not change the abstract value of the location. If the SC operation returns true then the abstract value of a has not changed since the linearization point of the LL operation, and at the linearization point of the SC operation, the abstract value becomes newval.*

PROOF. If the SC operation returns *false* then no memory location is changed, so the abstract value of a is unchanged.

If the SC operation returns *true*, then immediately before the CAS operation, the value field of a contains the current tagged id of p, which is the tagged id written by the preceding LL operation. Because a tagged id is written into a location only once, the value field must have been unchanged since it was written by the LL operation. Thus, during that entire interval, the abstract value of a was the value stored in VAL_SAVE[p]. Because VAL_SAVE[p] is changed only by LL operations of p, the abstract value of a did not change in the interval. Immediately after the CAS, the value field contains the application value *newval*, as required. □

LEMMA 5. *A SNAPSHOT operation returns an array of the abstract values of the specified locations at its linearization point, and does not change the abstract value of any location.*

PROOF. Consider any conclusive attempt of the SNAPSHOT operation. Because the attempt is conclusive, the values returned for all locations by the two COLLECT_VALUES operations of this attempt are the same. We argue that in the interval between these COLLECT_VALUES operations, the abstract values of all these locations was always the value returned by these operations.

Consider any location a read by the SNAPSHOT operation. Let the value returned by the COLLECT_VALUES operations be v. Because COLLECT_VALUES uses READ, Lemma 2 implies that the abstract value of a was v at some point during each COLLECT_VALUES operation. Note also that, at the linearization point of each READ(a) operation, the value field of a contains its abstract value. Suppose, towards a contradiction, that the abstract value of a changed between the linearization points of the two READ(a) operations that read from location a in the two COLLECT_VALUES operations. Then some successful SC or KCSS operation that changed the abstract value of a must be linearized at some point between the two READ operations. Before the linearization point of such an SC or KCSS operation, an LL(a) operation must have been linearized, leaving its tagged id in the value field of a. This LL operation must be linearized after the first READ(a), and it must terminate before the second READ(a). Thus, it must store its tagged id into the tag field of a between the two READ operations. Therefore, the tag field of a changed between the two COLLECT_TAGGED_IDS operations. Because a tag field is never written more than once with the same tagged id, the two COLLECT_TAGGED_IDS operations must return different tags for a, contradicting the assumption that this attempt is conclusive. □

LEMMA 6. *If a KCSS($a[1..k]$, $expvals[1..k]$, newval) operation fails, then for some $i \in [1,k]$, the abstract value of $a[i]$ was not $expvals[i]$ at the linearization point of the operation.*

PROOF. If the KCSS operation fails, then for some $i \in [1,k]$, oldvals[i] $\neq$ $expvals[i]$.

If $i = 1$ then the linearization point of this operation is the preceding LL operation, which returned oldvals[1]. By Lemma 3, this value is the abstract value of $a[1]$ at that point, which therefore is not $expvals[1]$.

If $i \in [2,k]$ then the linearization point of this operation is the preceding SNAPSHOT operation, which returned

oldvals[i]. By Lemma 5, this value is the abstract value of $a[i]$ at that point, which therefore is not *expvals*[i]. □

LEMMA 7. *Immediately before the linearization point of a successful KCSS*($a[1..k]$, *expvals*$[1..k]$, *newval*) *operation, the abstract value of each specified location had the corresponding value from the expvals array, and the abstract value of* $a[1]$ *becomes newval at the linearization point.*

PROOF. The definition of the KCSS operation implies that the abstract value of $a[1]$ becomes *newval* at the linearization point of a successful KCSS operation.

Because the KCSS linearizes at the linearization point of its last SNAPSHOT, we know that at that point, the abstract values of locations $a[2]..a[k]$ are the values returned by the SNAPSHOT, which we check in the loop at line K4 are equal to the corresponding values in the *expvals* array. Because the subsequent SC operation succeeds, by Lemma 4, we know the abstract value of $a[1]$ is the value returned by the preceding LL operation—which we also check is *expvals*[1]—for the entire interval between the LL and SC operations. □

A.2 Obstruction-freedom proof

We now argue that all the operations are obstruction-free. We also argue that an SC(a) operation by thread p succeeds whenever no other operations that access the location take any steps between p's invocation of the preceding LL(a) operation and the completion of the SC operation. This latter property is required so that these operations are useful for implementing obstruction-free data structures.

That RESET and SC are obstruction-free is straightforward to see, as they have no loops.

LEMMA 8. *If a RESET*(a) *operation executes without any other threads accessing location a then immediately after the RESET operation, the value field of a contains an application value.*

PROOF. If the value field of a contains an application value before the RESET operation, then the RESET operation does not change it. If the value field of a contains a tagged id, then the RESET operation CAS's that field from its old value to the value in the corresponding entry of the VAL_SAVE array, which always contains an application value. Because no other thread changes the value field of a, this CAS always succeeds, so after the RESET operation, the value field of a contains an application value. □

LEMMA 9. *READ is obstruction-free.*

PROOF. Suppose thread p executes a READ(a) operation, and that after some point in that operation execution, only thread p takes steps. If p's current attempt at that point is inconclusive then it executes another attempt, during which no other thread takes steps. If p sees an application value in the value field of a then this new attempt will be conclusive. Otherwise, p invokes RESET(a), after which, the value field of a contains an application value. Thus, p's next attempt will succeed as long as no other thread accesses a. □

LEMMA 10. *If a READ*(a) *operation executes without any other threads accessing location a then immediately after the READ operation, the value field of a contains the value returned by the READ operation, which is an application value.*

PROOF. The READ(a) operation returns only if the value it read in the value field of a was an application value, which it returns. Because no other thread accesses a, immediately after the READ operation, the value field of a still contains that application value. □

LEMMA 11. *LL is obstruction-free.*

PROOF. Suppose that thread p executes an LL(a) operation, and that after some point during the execution of this operation, only thread p takes steps. If p's current attempt at that point is inconclusive then it executes another attempt, during which no other thread takes steps. By Lemma 10, immediately after the READ operation on line 6 of this attempt, the value field of a contains the value returned by that READ. Therefore, the CAS in line 8 succeeds, and LL operation terminates. □

LEMMA 12. *SNAPSHOT is obstruction-free.*

PROOF. Suppose that thread p executes a SNAPSHOT operation, and that after some point during the execution of that operation, only thread p takes steps. If p's current attempt at that point is inconclusive then it executes another attempt, during which no other thread takes steps. By Lemma 10 and inspection of the code, after the first COLLECT_TAGGED_IDS and COLLECT_VALUES operations of this attempt, the tag and value fields contain the values returned by those operations. Thus, the second COLLECT_TAGGED_IDS and COLLECT_VALUES operations will return the same values and this attempt will be conclusive. □

LEMMA 13. *If p executes an LL*(a) *operation without any other threads accessing location a then immediately after the LL operation, the value field of a contains p's MY_TAGGED_ID value.*

PROOF. This is straightforward because the LL operation terminates only if p successfully CAS's the value field of a to its MY_TAGGED_ID value. □

LEMMA 14. *If a thread p invokes LL*(a) *and then SC*(a) *with no intervening LL, SC or KCSS operations (but possibly other operations), and no thread (including p) takes any steps for any operation that accesses a in the interval between p's LL*(a) *and SC*(a) *operations, then the SC operation succeeds.*

PROOF. By Lemma 13, immediately after the LL(a) operation, the value field of a contains p's MY_TAGGED_ID value. Because no thread accesses a until p's subsequent SC operation, this field does not change in that interval, and because p does not invoke LL, SC or KCSS, p's MY_TAGGED_ID value also does not change in that interval. Thus, the CAS in the SC operation succeeds. □

LEMMA 15. *KCSS is obstruction-free.*

PROOF. Suppose only thread p executing a KCSS operation takes steps after some point. If its current attempt is inconclusive then it executes another attempt, during which no other thread takes steps. Thus, the only abstract operation that takes steps between the LL and SC operations of this attempt is the SNAPSHOT of this attempt, which does not access the location of the LL and SC operations. So, by Lemma 14, the SC operation succeeds, and the KCSS operation terminates. □