Pragmatic Primitives for Non-blocking Data Structures

Trevor Brown
University of Toronto

Faith Ellen
University of Toronto

Eric Ruppert
York University

ABSTRACT

We define a new set of primitive operations that greatly simplify the implementation of non-blocking data structures in asynchronous shared-memory systems. The new operations operate on a set of Data-records, each of which contains multiple fields. The operations are generalizations of the well-known load-link (LL) and store-conditional (SC) operations called LLX and SCX. The LLX operation takes a snapshot of one Data-record. An SCX operation by a process p succeeds only if no Data-record in a specified set has been changed since p last performed an LLX on it. If successful, the SCX atomically updates one specific field of a Data-record in the set and prevents any future changes to some specified subset of those Data-records. We provide a provably correct implementation of these new primitives from single-word compare-and-swap. As a simple example, we show how to implement a non-blocking multiset data structure in a straightforward way using LLX and SCX.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures—*Distributed data structures*

Keywords

load-link/store-conditional; non-blocking; multiset

1. INTRODUCTION

Building a library of concurrent data structures is an essential way to simplify the difficult task of developing concurrent software. There are many lock-based data structures, but locks are not fault-tolerant and are susceptible to problems such as deadlock [11]. It is often preferable to use hardware synchronization primitives like compare-and-swap (CAS) instead of locks. However, the difficulty of this task has inhibited the development of *non-blocking* data structures. These are data structures which guarantee that some operation will eventually complete even if some processes crash.

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PODC'13, July 22–24, 2013, Montréal, Québec, Canada
Copyright 2013 ACM 978-1-4503-2065-8/13/07 ...$15.00.

Our goal is to facilitate the implementation of high-performance, provably correct, non-blocking data structures on any system that supports a hardware CAS instruction. We introduce three new operations, *load-link-extended* (LLX), *validate-extended* (VLX) and *store-conditional-extended* (SCX), which are natural generalizations of the well known *load-link* (LL), *validate* (VL) and *store-conditional* (SC) operations. We provide a practical implementation of our new operations from CAS. Complete proofs of correctness appear in [7]. We also show how these operations make the implementation of non-blocking data structures and their proofs of correctness substantially less difficult, as compared to using LL, VL, SC, and CAS directly.

LLX, SCX and VLX operate on *Data-records*. Any number of types of Data-records can be defined, each type containing a fixed number of *mutable* fields (which can be updated), and a fixed number of *immutable* fields (which cannot). Each Data-record can represent a natural unit of a data structure, such as a node of a tree or a table entry. A successful LLX operation returns a snapshot of the mutable fields of one Data-record. (The immutable fields can be read directly, since they never change.) An SCX operation by a process p is used to atomically store a value in one mutable field of one Data-record *and finalize* a set of Data-records, meaning that those Data-records cannot undergo any further changes. The SCX succeeds only if each Data-record in a specified set has not changed since p last performed an LLX on it. A successful VLX on a set of Data-records simply assures the caller that each of these Data-records has not changed since the caller last performed an LLX on it. A more formal specification of the behaviour of these operations is given in Section 3.

Early on, researchers recognized that operations accessing multiple locations atomically make the design of non-blocking data structures much easier [5, 13, 17]. Our new primitives do this in three ways. First, they operate on Data-records, rather than individual words, to allow the data structure designer to think at a higher level of abstraction. Second, and more importantly, a VLX or SCX can depend upon multiple LLXs. Finally, the effect of an SCX can apply to multiple Data-records, modifying one and finalizing others.

The precise specification of our operations was chosen to balance ease of use and efficient implementability. They are more restricted than multi-word CAS [13], multi-word RMW [1], or transactional memory [17]. On the other hand, the ability to finalize Data-records makes SCX more general than k-compare-single-swap [15], which can only change one

word. We found that atomically changing one pointer and finalizing a collection of Data-records provides just enough power to implement numerous pointer-based data structures in which operations replace a small portion of the data structure. To demonstrate the usefulness of our new operations, in Section 5, we give an implementation of a simple, linearizable, non-blocking multiset based on an ordered, singly-linked list.

Our implementation of LLX, VLX, and SCX is designed for an asynchronous system where processes may crash. We assume shared memory locations can be accessed by single-word CAS, read and write instructions. We assume a safe garbage collector (as in the Java environment) that will not reallocate a memory location if any process can reach it by following pointers. This allows records to be reused.

Our implementation has some desirable performance properties. A VLX on k Data-records only requires reading k words of memory. If SCXs being performed concurrently depend on LLXs of disjoint sets of Data-records, they all succeed. If an SCX encounters no contention with any other SCX and finalizes f Data-records, then a total of $k+1$ CAS steps and $f+2$ writes are used for the SCX and the k LLXs on which it depends. We also prove progress properties that suffice for building non-blocking data structures using LLX and SCX.

2. RELATED WORK

Transactional memory [12, 17] is a general approach to simplifying the design of concurrent algorithms by providing atomic access to multiple objects. It allows a block of code designated as a transaction to be executed atomically, with respect to other transactions. Our LLX/VLX/SCX primitives may be viewed as implementing a restricted kind of transaction, in which each transaction can perform any number of reads followed by a single write and then finalize any number of words. It is possible to implement general transactional memory in a non-blocking manner (e.g., [11, 17]). However, at present, implementations of transactional memory in software incur significant overhead, so there is still a need for more specialized techniques for designing shared data structures that combine ease of use and efficiency.

Most shared-memory systems provide CAS operations in hardware. However, LL and SC operations have often been seen as more convenient primitives for building algorithms. Anderson and Moir gave the first wait-free implementation of small LL/SC objects from CAS using $O(1)$ steps per operation [3]. See [14] for a survey of other implementations that use less space or handle larger LL/SC objects.

Many non-blocking implementations of primitives that access multiple objects use the *cooperative technique*, first described by Turek, Shasha and Prakash [19] and Barnes [5]. Instead of using locks that give a process exclusive access to a part of the data structure, this approach gives exclusive access to *operations*. If the process performing an operation that holds a lock is slow, other processes can *help* complete the operation and release the lock.

The cooperative technique was also used recently for a wait-free universal construction [8] and to obtain non-blocking binary search trees [10] and Patricia tries [16]. The approach used here is similar.

Israeli and Rappoport [13] used a version of the cooperative technique to implement multi-word CAS from single-word CAS (and sketched how this could be used to implement multi-word SC operations). However, their approach applies single-word CAS to very large words. The most efficient implementation of k-word CAS [18] first uses single-word CAS to replace each of the k words with a pointer to a record containing information about the operation, and then uses single-word CAS to replace each of these pointers with the desired new value and update the status field of the record. In the absence of contention, this takes $2k+1$ CAS steps. In contrast, in our implementation, an SCX that depends on LLXs of k Data-records performs $k+1$ single-word CAS steps when there is no contention, no matter how many words each record contains. So, our weaker primitives can be significantly more efficient than multi-word CAS or multi-word RMW [1, 4], which is even more general.

If k Data-records are removed from a data structure by a multi-word CAS, then the multi-word CAS must depend on every mutable field of these records to prevent another process from concurrently updating any of them. It is possible to use k-word CAS to apply to k Data-records instead of k words with indirection: Every Data-record is represented by a single word containing a pointer to the contents of the record. To change any fields of the Data-record, a process swings the pointer to a new copy of its contents containing the updated values. However, the extra level of indirection affects all reads, slowing them down considerably.

Luchangco, Moir and Shavit [15] defined the k-compare-single-swap (KCSS) primitive, which atomically tests whether k specified memory locations contain specified values and, if all tests succeed, writes a value to one of the locations. They provided an *obstruction-free* implementation of KCSS, meaning that a process performing a KCSS is guaranteed to terminate if it runs alone. They implemented KCSS using an obstruction-free implementation of LL/SC from CAS. Specifically, to try to update location v using KCSS, a process performs LL(v), followed by two collects of the other $k-1$ memory locations. If v has its specified value, both collects return their specified values, and the contents of these memory locations do not change between the two collects, the process performs SC to change the value of v. Unbounded version numbers are used both in their implementation of LL/SC and to avoid the ABA problem between the two collects.

Our LLX and SCX primitives can be viewed as multi-Data-record-LL and single-Data-record-SC primitives, with the additional power to finalize Data-records. We shall see that this extra ability is extremely useful for implementing pointer-based data structures. In addition, our implementation of LLX and SCX allows us to develop shared data structures that satisfy the non-blocking progress condition, which is stronger than obstruction-freedom.

3. THE PRIMITIVES

Our primitives operate on a collection of Data-records of various user-defined types. Each type of Data-record has a fixed number of mutable fields (each fitting into a single word), and a fixed number of immutable fields (each of which can be large). Each field is given a value when the Data-record is created. Fields can contain pointers that refer to other Data-records. Data-records are accessed using LLX, SCX and VLX, and reads of individual mutable or immutable fields of a Data-record. Reads of mutable fields are permitted because a snapshot of a Data-record's fields

is sometimes excessive, and it is sometimes sufficient (and more efficient) to use reads instead of LLXs.

An implementation of LL and SC from CAS has to ensure that, between when a process performs LL and when it next performs SC on the same word, the value of the word has not changed. Because the value of the word could change and then change back to a previous value, it is not sufficient to check that the word has the same value when the LL and the SC are performed. This is known as the ABA problem. It also arises for implementations of LLX and SCX from CAS. A general technique to overcome this problem is described in Section 4.1. However, if the data structure designer can guarantee that the ABA problem will not arise (because each SCX never attempts to store a value into a field that previously contained that value), our implementation can be used in a more efficient manner.

Before giving the precise specifications of the behaviour of LLX and SCX, we describe how to use them, with the implementation of a multiset as a running example. The multiset abstract data type supports three operations: GET(key), which returns the number of occurrences of key in the multiset, INSERT($key, count$), which inserts $count$ occurrences of key into the multiset, and DELETE($key, count$), which deletes $count$ occurrences of key from the multiset and returns TRUE, provided there are at least $count$ occurrences of key in the multiset. Otherwise, it simply returns FALSE.

Suppose we would like to implement a multiset using a sorted, singly-linked list. We represent each node in the list by a Data-record with an immutable field key, which contains a key in the multiset, and mutable fields: $count$, which records the number of times key appears in the multiset, and $next$, which points to the next node in the list. The first and last elements of the list are sentinel nodes with count 0 and with special keys $-\infty$ and ∞, respectively, which never occur in the multiset.

Figure 5 shows how updates to the list are handled. Insertion behaves differently depending on whether the key is already present. Likewise, deletion behaves differently depending on whether it removes all copies of the key. For example, consider the operation DELETE$(d, 2)$ depicted in Figure 5(c). This operation removes node r by changing $p.next$ to point to a new copy of $rnext$. A new copy is used to avoid the ABA problem, since $p.next$ may have pointed to $rnext$ in the past. To perform the DELETE$(d, 2)$, a process first invokes LLXs on p, r, and $rnext$. Second, it creates a copy $rnext'$ of $rnext$. Finally, it performs an SCX that depends on these three LLXs. This SCX attempts to change $p.next$ to point to $rnext'$. This SCX will succeed only if none of p, r or $rnext$ have changed since the aforementioned LLXs. Once r and $rnext$ are removed from the list, we want subsequent invocations of LLX and SCX to be able to detect this, so that we can avoid, for example, erroneously inserting a key into a deleted part of the list. Thus, we specify in our invocation of SCX that r and $rnext$ should be *finalized* if the SCX succeeds. Once a Data-record is finalized, it can never be changed again.

LLX takes (a pointer to) a Data-record r as its argument. Ordinarily, it returns either a snapshot of r's mutable fields or FINALIZED. If an LLX(r) is concurrent with an SCX involving r, it is also allowed to fail and return FAIL. SCX takes four arguments: a sequence V of (pointers to) Data-records upon which the SCX depends, a subsequence R of V containing (pointers to) the Data-records to be finalized, a mutable field fld of a Data-record in V to be modified, and a value new to store in this field. VLX takes a sequence V of (pointers to) Data-records as its only argument. Each SCX and VLX and returns a Boolean value.

For example, in Figure 5(c), the DELETE$(d, 2)$ operation invokes SCX(V, R, fld, new), where $V = \langle p, r, rnext\rangle$, $R = \langle r, rnext\rangle$, fld is the next pointer of p, and new points to the node $rnext'$.

A terminating LLX is called *successful* if it returns a snapshot or FINALIZED, and *unsuccessful* if it returns FAIL. A terminating SCX or VLX is called *successful* if it returns TRUE, and *unsuccessful* if it returns FALSE. Our operations are wait-free, but an operation may not terminate if the process performing it fails, in which case the operation is neither successful nor unsuccessful. We say an invocation I of LLX(r) by a process p is *linked to* an invocation I' of SCX(V, R, fld, new) or VLX(V) by process p if r is in V, I returns a snapshot, and between I and I', process p performs no invocation of LLX(r) or SCX(V', R', fld', new') and no unsuccessful invocation of VLX(V'), for any V' that contains r. Before invoking VLX(V) or SCX(V, R, fld, new), a process must *set up* the operation by performing an LLX(r) linked to the invocation for each r in V.

3.1 Correctness Properties

An implementation of LLX, SCX and VLX is *correct* if, for every execution, there is a linearization of all successful LLXs, all successful SCXs, a subset of the non-terminating SCXs, all successful VLXs, and all reads, such that the following conditions are satisfied.

C1: Each read of a field f of a Data-record r returns the last value stored in f by an SCX linearized before the read (or f's initial value, if no such SCX has modified f).

C2: Each linearized LLX(r) that does not return FINALIZED returns the last value stored in each mutable field f of r by an SCX linearized before the LLX (or f's initial value, if no such SCX has modified f).

C3: Each linearized LLX(r) returns FINALIZED if and only if it is linearized after an SCX(V, R, fld, new) with r in R.

C4: For each linearized invocation I of SCX(V, R, fld, new) or VLX(V), and for each r in V, no SCX(V', R', fld', new') with r in V' is linearized between the LLX(r) linked to I and I.

The first three properties assert that successful reads and LLXs return correct answers. The last property says that an invocation of SCX or VLX does not succeed when it should not. However, an SCX can fail if it is concurrent with another SCX that accesses some Data-record in common. LL/SC also exhibits analogous failures in real systems. Our progress properties limit the situations in which this can occur.

3.2 Progress Properties

In our implementation, LLX, SCX and VLX are technically wait-free, but this is only because they may fail. So, we must state progress properties in terms of *successful* operations. The first progress property guarantees that LLXs on finalized Data-records succeed.

P1: Each terminating LLX(r) returns FINALIZED if it begins after the end of a successful SCX(V, R, fld, new)

with r in R or after another LLX(r) has returned FINALIZED.

The next progress property guarantees non-blocking progress of invocations of our primitives.

P2: If operations are performed infinitely often, then operations succeed infinitely often.

However, this progress property leaves open the possibility that only LLXs succeed. So, we want an additional progress property:

P3: If SCX and VLX operations are performed infinitely often, then SCX or VLX operations succeed infinitely often.

Finally, the following progress property ensures that *update* operations that are built using SCX can be made non-blocking.

P4: If SCX operations are performed infinitely often, then SCX operations succeed infinitely often.

When the progress properties defined here are used to prove that an application built from the primitives is non-blocking, there is an important, but subtle point: an SCX can be invoked only after it has been properly set up by a sequence of LLXs. However, if processes repeatedly perform LLX on Data-records that have been finalized, they may never be able to invoke an SCX. One way to prevent this from happening is to have each process keep track of the Data-records it knows are finalized. However, in many natural applications, for example, the multiset implementation in Section 5, explicit bookkeeping can be avoided. In addition, to ensure that changes to a data structure can continue to occur, there must always be at least one non-finalized Data-record. For example, in our multiset, *head* is never finalized and, if a node is reachable from *head* by following *next* pointers, then it is not finalized.

Our implementation of LLX, SCX and VLX in Section 4 actually satisfies stronger progress properties than the ones described above. For example, a VLX(V) or SCX(V, R, fld, new) is guaranteed to succeed if there is no concurrent SCX(V', R', fld', new') such that V and V' have one or more elements in common. However, for the purposes of the specification of the primitives, we decided to give progress guarantees that are sufficient to prove that algorithms that use the primitives are non-blocking, but weak enough that it may be possible to design other, even more efficient implementations of the primitives. For example, our specification would allow some spurious failures of the type that occur in common implementations of ordinary LL/SC operations (as long as there is some guarantee that not all operations can fail spuriously).

4. IMPLEMENTATION OF PRIMITIVES

The shared data structure used to implement LLX, SCX and VLX consists of a set of Data-records and a set of SCX-records. (See Figure 1.) Each Data-record contains user-defined mutable and immutable fields. It also contains a *marked* bit, which is used to finalize the Data-record, and an *info* field. The marked bit is initially FALSE and only ever changes from FALSE to TRUE. The *info* field points to an SCX-record that describes the last SCX that accessed the Data-record. Initially, it points to a *dummy* SCX-record. When an SCX accesses a Data-record, it changes the *info* field of the Data-record to point to its SCX-record. While this SCX is active, the *info* field acts as a kind of lock on the Data-record, granting exclusive access to this SCX, rather than to a process. (To avoid confusion, we call this *freezing*, rather than locking, a Data-record.) We ensure that an SCX S does not change a Data-record for its own purposes while it is frozen for another SCX S'. Instead, S uses the information in the SCX-record of S' to help S' complete (successfully or unsuccessfully), so that the Data-record can be unfrozen. This cooperative approach is used to ensure progress.

```
type Data-record
  ▷ User-defined fields
  m_1, ..., m_y   ▷ mutable fields
  i_1, ..., i_z   ▷ immutable fields
  ▷ Fields used by LLX/SCX algorithm
  info            ▷ pointer to an SCX-record
  marked          ▷ Boolean

type SCX-record
  V               ▷ sequence of Data-records
  R               ▷ subsequence of V to be finalized
  fld             ▷ pointer to a field of a Data-record in V
  new             ▷ value to be written into the field fld
  old             ▷ value previously read from the field fld
  state           ▷ one of {InProgress, Committed, Aborted}
  allFrozen       ▷ Boolean
  infoFields      ▷ sequence of pointers, one read from the
                  ▷ info field of each element of V
```

Figure 1: Type definitions for shared objects used to implement LLX, SCX, and VLX.

An SCX-record contains enough information to allow any process to complete an SCX operation that is in progress. V, R, fld and new store the arguments of the SCX operation that created the SCX-record. Recall that R is a subsequence of V and fld points to a mutable field f of some Data-record r' in V. The value that was read from f by the LLX(r') linked to the SCX is stored in *old*. The SCX-record has one of three states, InProgress, Committed or Aborted, which is stored in its *state* field. This field is initially InProgress. The SCX-record of each SCX that terminates is eventually set to Committed or Aborted, depending on whether or not it successfully makes its desired update. The dummy SCX-record always has *state* = Aborted. The *allFrozen* bit, which is initially FALSE, gets set to TRUE after all Data-records in V have been frozen for the SCX. The values of *state* and *allFrozen* change in accordance with the diagram in Figure 2. The steps in the algorithm that cause these changes are also indicated. The *infoFields* field stores, for each r in V, the value of r's *info* field that was read by the LLX(r) linked to the SCX.

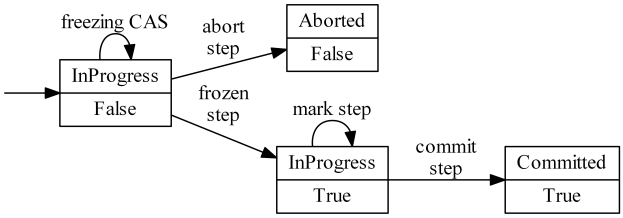

Figure 2: Possible [*state*, *allFrozen*] field transitions of an SCX-record.

We say that a Data-record r is *marked* when $r.marked =$ TRUE. A Data-record r is *frozen* for an SCX-record U if $r.info$ points to U and either $U.state$ is InProgress, or $U.state$ is Committed and r is marked. While a Data-record r is frozen for an SCX-record U, a mutable field f of r can be changed only if f is the field pointed to by $U.fld$ (and it can only be changed by a process helping the SCX that created U). Once a Data-record r is marked and $r.info.state$ becomes Committed, r will never be modified again in any way. Figure 3 shows how a Data-record can change between frozen and unfrozen. The three bold boxes represent frozen Data-records. The other two boxes represent Data-records that are not frozen. A Data-record r can only become frozen when $r.info$ is changed (to point to a new SCX-record whose state is InProgress). This is represented by the grey edges. The black edges represent changes to $r.info.state$ or $r.marked$. A frozen Data-record r can only become unfrozen when $r.info.state$ is changed.

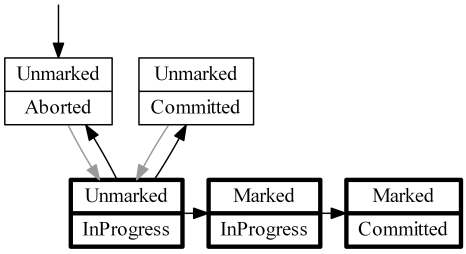

Figure 3: Possible transitions for the *marked* field of a Data-record and the *state* of the SCX-record pointed to by the *info* field of the Data-record.

4.1 Constraints

For the sake of efficiency, we have designed our implementation of LLX, VLX and SCX to work only if the primitives are used in a way that satisfies certain constraints, described in this section. We also describe general (but somewhat inefficient) ways to ensure these constraints are satisfied. However, there are often quite natural ways to ensure the constraints are satisfied without resorting to the extra work required by the general solutions.

Since our implementation of LLX, SCX and VLX uses helping to guarantee progress, each CAS of an SCX might be repeatedly performed by several helpers, possibly after the SCX itself has terminated. To avoid difficulties, we must show there is no ABA problem in the fields affected by these CAS steps.

The *info* field of a Data-record r is modified by CAS steps that attempt to freeze r for an SCX. All such steps performed by processes helping one invocation of SCX try to CAS the *info* field of r from the same old value to the same new value, and that new value is a pointer to a newly created SCX-record. Because the SCX-record is allocated a location that has never been used before, the ABA problem will not arise in the *info* field. (This approach is compatible with safe garbage collection schemes that only reuse an old address once no process can reach it by following pointers.)

A similar approach could be used to avoid the ABA problem in a mutable field of a Data-record: the new value could be placed inside a wrapper object that is allocated a new location in memory. (This is referred to as Solution 3 of the ABA problem in [9].) However, the extra level of indirection slows down accesses to fields.

To avoid the ABA problem, it suffices to prove the following constraint is satisfied.

- **Constraint**: For every invocation S of SCX(V, R, fld, new), new is not the initial value of fld and no invocation of SCX(V', R', fld, new) was linearized before the LLX(r) linked to S was linearized, where r is the Data-record that contains fld.

The multiset in Section 5 provides an example of a simple, more efficient way to ensure that this constraint is always satisfied.

To ensure property P4, we put a constraint on the way SCX is used. Our implementation of SCX(V, R, fld, new) does something akin to acquiring locks on each Data-record in V. Livelock could occur if different invocations of SCX do not process Data-records in the same order. To prevent this, we could define a way of ordering all Data-records (for example, by their locations in memory) and each sequence passed to an invocation of SCX could be sorted using this ordering. However, this could be expensive. Moreover, to prove our progress properties, we do not require that *all* SCXs order their sequences V consistently. It suffices that, if all the Data-records stop changing, then the sequences passed to later invocations of SCX are all consistent with some total order. This property is often easy to satisfy in a natural way. More precisely, use of our implementation of SCX requires adherence to the following constraint.

- **Constraint**: Consider each execution that contains a configuration C after which the value of no field of any Data-record changes. There must be a total order on all Data-records created during this execution such that, if Data-record r_1 appears before Data-record r_2 in the sequence V passed to an invocation of SCX whose linked LLXs begin after C, then $r_1 < r_2$.

For example, if one was using LLX and SCX to implement an *unsorted* singly-linked list, this constraint would be satisfied if the nodes in each sequence V occur in the order they are encountered by following next pointers from the beginning of the list, *even if* some operations could reorder the nodes in the list. While the list is changing, such a sequence may have repeated elements and might not be consistent with any total order.

4.2 Detailed Algorithm Description and Sketch of Proofs

Pseudocode for our implementation of LLX, VLX and SCX appears in Figure 4. If x contains a pointer to a record, then $x.y := v$ assigns the value v to field y of this record, $\&x.y$ denotes the address of this field and all other occurrences of $x.y$ denote the value stored in this field.

THEOREM 1. *The algorithms in Figure 4 satisfy properties C1 to C4 and P1 to P4 in every execution where the constraints of Section 4.1 are satisfied.*

The detailed proof of correctness [7] is quite involved, so we only sketch the main ideas here.

An LLX(r) returns a snapshot, FAIL, or FINALIZED. At a high level, it works as follows. If the LLX determines that r is not frozen and r's *info* field does not change while the LLX reads the mutable fields of r, the LLX returns the values read as a snapshot. Otherwise, the LLX helps the SCX that last froze r, if it is frozen, and returns FAIL or

```
1  LLX(r) by process p
2  ▷ Precondition: r ≠ Nil.
3     marked1 := r.marked                                          ▷ order of lines 3–6 matters
4     rinfo := r.info
5     state := rinfo.state
6     marked2 := r.marked
7     if state = Aborted or (state = Committed and not marked2) then    ▷ if r was not frozen at line 5
8        read r.m1, ..., r.my and record the values in local variables m1, ..., my
9        if r.info = rinfo then                                    ▷ if r.info points to the same
10          store ⟨r, rinfo, ⟨m1, ..., my⟩⟩ in p's local table     ▷ SCX-record as on line 4
11          return ⟨m1, ..., my⟩
12    if (rinfo.state = Committed or (rinfo.state = InProgress and Help(rinfo))) and marked1 then
13       return Finalized
14    else
15       if r.info.state = InProgress then Help(r.info)
16       return Fail

17 SCX(V, R, fld, new) by process p
18 ▷ Preconditions: (1) for each r in V, p has performed an invocation Ir of LLX(r) linked to this SCX
                     (2) new is not the initial value of fld
                     (3) for each r in V, no SCX(V', R', fld, new) was linearized before Ir was linearized
19    Let infoFields be a pointer to a newly created table in shared memory containing,
         for each r in V, a copy of r's info value in p's local table of LLX results
20    Let old be the value for fld stored in p's local table of LLX results
21    return Help(pointer to new SCX-record(V, R, fld, new, old, InProgress, False, infoFields))

22 Help(scxPtr)
23    ▷ Freeze all Data-records in scxPtr.V to protect their mutable fields from being changed by other SCXs
24    for each r in scxPtr.V enumerated in order do
25       Let rinfo be the pointer indexed by r in scxPtr.infoFields
26       if not CAS(r.info, rinfo, scxPtr) then                    ▷ freezing CAS
27          if r.info ≠ scxPtr then
28             ▷ Could not freeze r because it is frozen for another SCX
29             if scxPtr.allFrozen = True then                     ▷ frozen check step
30                ▷ the SCX has already completed successfully
31                return True
32             else
33                ▷ Atomically unfreeze all nodes frozen for this SCX
34                scxPtr.state := Aborted                          ▷ abort step
35                return False
36    ▷ Finished freezing Data-records (Assert: state ∈ {InProgress, Committed})
37    scxPtr.allFrozen := True                                    ▷ frozen step
38    for each r in scxPtr.R do r.marked := True                  ▷ mark step
39    CAS(scxPtr.fld, scxPtr.old, scxPtr.new)                     ▷ update CAS
40    ▷ Finalize all r in R, and unfreeze all r in V that are not in R
41    scxPtr.state := Committed                                   ▷ commit step
42    return True

43 VLX(V) by process p
44 ▷ Precondition: for each Data-record r in V, p has performed an LLX(r) linked to this VLX
45    for each r in V do
46       Let rinfo be the info field for r stored in p's local table of LLX results
47       if rinfo ≠ r.info then return False      ▷ r changed since LLX(r) read info
48    return True          ▷ At some point during the loop, all r in V were unchanged
```

Figure 4: Pseudocode for LLX**,** SCX **and** VLX**.**

FINALIZED. If the LLX returns FAIL, it is not linearized. We now discuss in more detail how LLX operates and is linearized in the other two cases.

First, suppose the LLX(r) returns a snapshot at line 11. Then, the test at line 7 evaluates to TRUE. So, either $state$ = Aborted, which means r is not frozen at line 5, or $state$ = Committed and $marked_2$ = FALSE. This also means r is not frozen at line 5, since $r.marked$ cannot change from TRUE to FALSE. The LLX reads r's mutable fields (line 8) and rereads $r.info$ at line 9, finding it the same as on line 4. In Section 4.1, we explained why this implies that $r.info$ did not change between lines 4 and 9. Since r is not frozen at line 5, we know from Figure 3 that r is unfrozen at all times between line 5 and 9. We prove that mutable fields can change only while r is frozen, so the values read by line 8 constitute a snapshot of r's mutable fields. Thus, we can linearize the LLX at line 9.

Now, suppose the LLX(r) returns FINALIZED. Then, the test on line 12 evaluated to TRUE. In particular, r was already marked when line 3 was performed. If $rinfo.state$ = InProgress when line 12 was performed, HELP($rinfo$) was called and returned TRUE. Below, we argue that $rinfo.state$ was changed to Committed before the return occurred. By Figure 3(a), the $state$ of an SCX-record never changes after it is set to Committed. So, after line 12, $rinfo.state$ = Committed and, thus, r has been finalized. Hence, the LLX can be linearized at line 13.

When a process performs an SCX, it first creates a new SCX-record and then invokes HELP (line 21). The HELP routine performs the real work of the SCX. It is also used by a process to help other processes complete their SCXs (successfully or unsuccessfully). The values in an SCX-record's old and $infoFields$ come from a table in the local memory of the process that invokes the SCX, which stores the results of the last LLX it performed on each Data-record. (In practice, the memory required for this table could be greatly reduced when a process knows which of these values are needed for future SCXs.)

Consider an invocation of HELP(U) by process p to carry out the work of the invocation S of SCX(V, R, fld, new) that is described by the SCX-record U. First, p attempts to freeze each r in V by performing a *freezing CAS* to store a pointer to U in $r.info$ (line 26). Process p uses the value read from $r.info$ by the LLX(r) linked to S as the old value for this CAS and, hence, it will succeed only if r has not been frozen for any other SCX since then. If p's freezing CAS fails, it checks whether some other helper has successfully frozen the Data-record with a pointer to U (line 27).

If every r in V is successfully frozen, p performs a *frozen step* to set $U.allFrozen$ to TRUE (line 37). After this frozen step, the SCX is guaranteed not to fail, meaning that no process will perform an abort step while helping this SCX. Then, for each r in R, p performs a *mark step* to set $r.marked$ to TRUE (line 38) and, from Figure 3, r remains frozen from then on. Next, p performs an *update CAS*, storing new in the field pointed to by fld (line 39), if successful. We prove that, among all the update CAS steps on fld performed by the helpers of U, only the first can succeed. Finally, p unfreezes all r in V that are not in R by performing a *commit step* that changes $U.state$ to Committed (line 41).

Now suppose that, when p performs line 27, it finds that some Data-record r in V is already frozen for another invocation S' of SCX. If $U.allFrozen$ is FALSE at line 29, then we can prove that no helper of S will ever reach line 37, so p can abort S. To do so, it unfreezes each r in V that it has frozen by performing an *abort step*, which changes $U.state$ to Aborted (line 34), and then returns FALSE (line 35) to indicate that S has been aborted. If $U.allFrozen$ is TRUE at line 29, it means that each element of V, including r, was successfully frozen by some helper of S and then, later, a process froze r for S'. Since S cannot be aborted after $U.allFrozen$ was set to TRUE, its state must have changed from InProgress to Committed before r was frozen for another SCX-record. Therefore, S was successfully completed and p can return TRUE at line 31.

We linearize an invocation of SCX at the first update CAS performed by one of its helpers. We prove that this update CAS always succeeds. Thus, all SCXs that return TRUE are linearized, as well as possibly some non-terminating SCXs. The first update CAS of SCX(V, R, fld, new) modifies the value of fld, so a read(fld) that occurs immediately after the update CAS will return the value of new. Hence, the linearization point of an SCX must occur at its first update CAS. There is one subtle issue about this linearization point: If an LLX(r) is linearized between the update CAS and commit step of an SCX that finalizes r, it might not return FINALIZED, violating condition C3. However, this cannot happen, because, before the LLX is linearized on line 13, the LLX either sees that the commit step has been performed or helps the SCX perform its commit step.

An invocation I of VLX(V) is executed by a process p after p has performed an invocation of LLX(r) linked to I, for each r in V. VLX(V) simply checks, for each r in V, that the $info$ field of r is the same as when it was read by p's last LLX(r) and, if so, VLX(V) returns TRUE. In this case, we prove that each Data-record in V does not change between the linked LLX and the time its $info$ field is reread. Thus, the VLX can be linearized at the first time it executes line 47. Otherwise, the VLX returns FALSE to indicate that the LLX results may not constitute a snapshot.

We remark that our use of the cooperative method avoids costly recursive helping. If, while p is helping S, it cannot freeze all of S's Data-records because one of them is already frozen for a third SCX, then p will simply perform an abort step, which unfreezes all Data-records that S has frozen.

We briefly sketch why the progress properties described in Section 3.2 are satisfied. It follows easily from the code that an invocation of LLX(r) returns FINALIZED if it begins after the end of an SCX that finalized r or another LLX sees that r is finalized. To prove the progress properties P2, P3 and P4, we consider two cases.

First, consider an execution where only a finite number of SCXs are invoked. Then, only finitely many SCX-records are created. Each process calls HELP(U) if it sees that $U.state$ = InProgress, which it can do at most once for each SCX-record U. Since every CAS is performed inside the HELP routine, there is some point after which no process performs a CAS, calls HELP, or sees a SCX-record whose $state$ is InProgress. A VLX can fail only when an $info$ field is modified by a concurrent operation and an LLX can only fail for the same reason or when it sees a SCX-record whose $state$ is InProgress. Therefore, all LLXs and VLXs that begin after this point will succeed, establishing P2 and P3. Moreover, P4 is vacuously satisfied in this case.

Now, consider an execution where infinitely many SCXs are invoked. To derive a contradiction, suppose only finitely

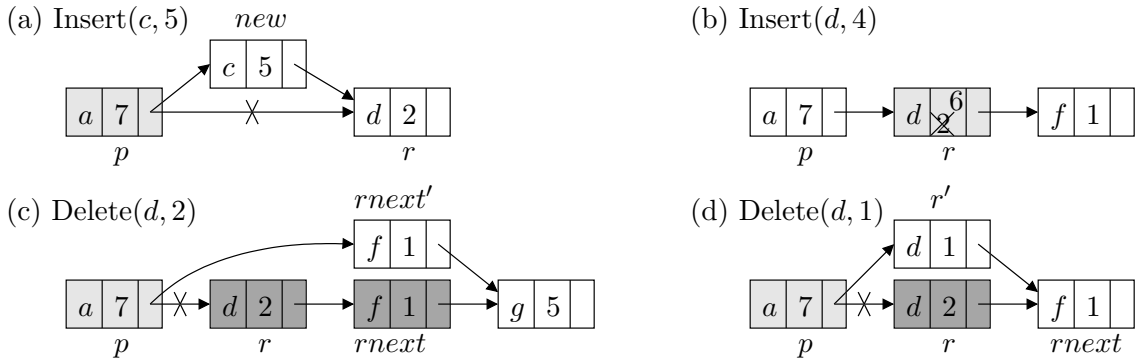

Figure 5: Using SCX to update a multiset. LLXs of all shaded nodes are linked to the SCX. Darkly shaded nodes are finalized by the SCX. Where a field has changed, the old value is crossed out.

many SCXs succeed. Then, there is a time after which no more SCXs succeed. The constraint on the sequences passed to invocations of SCXs ensures that all SCXs whose linked LLXs begin after this time will attempt to freeze their sequences of Data-records in a consistent order. Thus, one of these SCXs will succeed in freezing all of the Data-records that were passed to it and will successfully complete. This is a contradiction. Thus, infinitely many of the SCXs do succeed, establishing properties P2, P3 and P4.

4.3 Additional Properties

Our implementation of SCX satisfies some additional properties, which are helpful for designing certain kinds of nonblocking data structures so that query operations can run efficiently. Consider a pointer-based data structure with a fixed set of Data-records called *entry points*. An operation on the data structure starts at an entry point and follows pointers to visit other Data-records. (For example, in our multiset example, the head of the linked list is the sole entry point for the data structure.) We say that a Data-record is *in the data structure* if it can be reached by following pointers from an entry point, and a Data-record r is *removed from the data structure* by an SCX if r is in the data structure immediately prior to the linearization point of the SCX and is not in the data structure immediately afterwards.

If the data structure is designed so that a Data-record is finalized when (and only when) it is removed from the data structure, then we have the following additional properties.

PROPOSITION 2. *Suppose each linearized* SCX(V, R, fld, new) *removes precisely the Data-records in R from the data structure.*

- *If* LLX(r) *returns a value different from* FAIL *or* FINALIZED*, r is in the data structure just before the* LLX *is linearized.*
- *If an* SCX(V, R, fld, new) *is linearized and new is (a pointer to) a Data-record, then this Data-record is in the data structure just after the* SCX *is linearized.*
- *If an operation reaches a Data-record r by following pointers read from other Data-records, starting from an entry point, then r was in the data structure at some earlier time during the operation.*

The first two properties are straightforward to prove. The last property is proved by induction on the Data-records reached. For the base case, entry points are always reachable. For the induction step, consider the time when an operation reads a pointer to r from another Data-record r' that the operation reached earlier. By the induction hypothesis, there was an earlier time t during the operation when r' was in the data structure. If r' already contained a pointer to r at t, then r was also in the data structure at that time. Otherwise, an SCX wrote a pointer to r in r' after t, and just after that update occurred, r' and r were in the data structure (by the second part of the proposition).

The last property is a particularly useful one for linearizing query operations. It means that operations that search through a data structure can use simple reads of pointers instead of the more expensive LLX operations. Even though the Data-record that such a search operation reaches may have been removed from the data structure by the time it is reached, the lemma guarantees that there *was* a time during the search when the Data-record was in the data structure. For example, we use this property to linearize searches in our multiset algorithm in Section 5.

5. AN EXAMPLE: MULTISET

We now give a detailed description of the implementation of a multiset using LLX and SCX. We assume that keys stored in the multiset are drawn from a totally ordered set and $-\infty < k < \infty$ for every key k in the multiset. As described in Section 3, we use a singly-linked list of nodes, sorted by key. To avoid special cases, it always has a sentinel node, *head*, with key $-\infty$ at its beginning and a sentinel node, *tail*, with key ∞ at its end. The definition of Node, the Data-record used to represent a node, and the pseudocode are presented in Figure 6.

SEARCH(key) traverses the list, starting from *head*, by reading *next* pointers until reaching the first node r whose key is at least *key*. This node and the preceding node p are returned. GET(key) performs SEARCH(key), outputs r's count if r's key matches *key*, and outputs 0, otherwise.

An invocation I of INSERT($key, count$) starts by calling SEARCH(key). Using the nodes p and r that are returned, it updates the data structure. It decides whether *key* is already in the multiset (by checking whether $r.key = key$) and, if so, it invokes LLX(r) followed by an SCX linked to r to increase $r.count$ by *count*, as depicted in Figure 5(b). Otherwise, I performs the update depicted in Figure 5(a): It invokes LLX(p), checks that p still points to r, creates a node, *new*, and invokes an SCX linked to p to insert *new* between p

```
type Node
    ▷ Fields from sequential data structure
    key      ▷ key (immutable)
    count    ▷ occurrences of key (mutable)
    next     ▷ next pointer (mutable)
    ▷ Fields defined by LLX/SCX algorithm
    info     ▷ a pointer to an SCX-record
    marked   ▷ a Boolean value

shared Node tail := new Node(∞, 0, Nil)
shared Node head := new Node(−∞, 0, tail)
```

```
1  Get(key)
2      ⟨r, −⟩ := Search(key)
3      if key = r.key then
4          return r.count
5      else return 0
```

```
6  Search(key)
7      ▷ Postcondition: p and r point to
           Nodes with p.key < key ≤ r.key.
8      p := head
9      r := p.next
10     while key > r.key do
11         p := r
12         r := r.next
13     return ⟨r, p⟩
```

```
14 Insert(key, count)      ▷ Precondition: count > 0
15     while True do
16         ⟨r, p⟩ := Search(key)
17         if key = r.key then
18             localr := LLX(r)
19             if localr ∉ {Fail, Finalized} then
20                 if SCX(⟨r⟩, ⟨⟩, &r.count, localr.count + count) then return
21         else
22             localp := LLX(p)
23             if localp ∉ {Fail, Finalized} and r = localp.next then
24                 if SCX(⟨p⟩, ⟨⟩, &p.next, new Node(key, count, r)) then return
```

```
26 Delete(key, count)      ▷ Precondition: count > 0
27     while True do
28         ⟨r, p⟩ := Search(key)
29         localp := LLX(p)
30         localr := LLX(r)
31         if localp, localr ∉ {Fail, Finalized} and r = localp.next then
32             if key ≠ r.key or localr.count < count then return False
33             else if localr.count > count then
34                 if SCX(⟨p⟩, ⟨r⟩, &p.next, new
                       Node(r.key, localr.count − count, localr.next)) then
                       return True
35             else ▷ assert: localr.count = count
36                 if LLX(localr.next) ∉ {Fail, Finalized} then
37                     if SCX(⟨p, r, localr.next⟩, ⟨r, localr.next⟩,
                           &p.next, new copy of localr.next) then return True
```

Figure 6: Pseudocode for a multiset, implemented with a singly linked list.

and r. If p no longer points to r, the LLX returns Fail or Finalized, or the SCX returns False, then I restarts.

An invocation I of Delete($key, count$) also begins by calling Search(key). It invokes LLX on the nodes p and r and then checks that p still points to r. If r does not contain at least $count$ copies of key, then I returns False. If r contains exactly $count$ copies, then I performs the update depicted in Figure 5(c) to remove node r from the list. To do so, it invokes LLX on the node, $rnext$, that $r.next$ points to, makes a copy $rnext'$ of $rnext$, and invokes an SCX linked to p, r and $rnext$ to change $p.next$ to point to $rnext'$. This SCX also finalizes the nodes r and $rnext$, which are thereby removed from the data structure. The node $rnext$ is replaced by a copy to avoid the ABA problem in $p.next$. If r contains more than $count$ copies, then I replaces r by a new copy r' with an appropriately reduced count using an SCX linked to p and r, as shown in Figure 5(d). This SCX finalizes r. If an LLX returns Fail or Finalized, or the SCX returns False then I restarts.

A detailed proof of correctness appears in [7]. It begins by showing that this multiset implementation satisfies some basic properties.

Invariant 3. *The following are true at all times.*

- *head always points to a node.*
- *If a node has key ∞, then its next pointer is* Nil.
- *If a node's key is not ∞, then its next pointer points to some node with a strictly larger key.*

It follows that the data structure is always a sorted list.

We prove the following lemma by considering the SCXs performed by update operations shown in Figure 5.

Lemma 4. *The Data-records removed from the data structure by a linearized invocation of* SCX*(V, R, fld, new) are exactly the Data-records in R.*

This lemma allows us to apply Proposition 2 to prove that there is a time during each Search when the nodes r and p that it returns are both in the list and $p.next = r$.

Each Get and each Delete that returns False is linearized at the linearization point of the Search it performs. Every other Insert or Delete is linearized at its successful SCX. Linearizability of all operations then follows from the next invariant.

Lemma 5. *At every time t, the multiset of keys in the data structure is equal to the multiset of keys that would result from the atomic execution of the sequence of operations linearized up to time t.*

To prove the algorithm is non-blocking, suppose there is some infinite execution in which only finitely many operations terminate. Then, eventually, no more Insert or Delete operations perform a successful SCX, so there is a time after which the pointers that form the linked list stop changing. This implies that all calls to the Search subroutine must terminate. Since a Get operation merely calls Search, all Get operations must also terminate. Thus, there is some collection of Insert and Delete operations that take steps forever without terminating. We show that each such operation sets up and performs an SCX infinitely often. For any Insert or Delete operation, consider any iteration of the loop that begins after the last successful SCX changes the list. By Lemma 4 and Proposition 2, the nodes p and r reached by the Search in that iteration were in the

data structure at some time during the SEARCH and, hence, throughout the SEARCH. So when the INSERT or DELETE performs LLXs on p or r, they cannot return FINALIZED. Moreover, they must succeed infinitely often by property P2, and this allows the INSERT or DELETE to perform an SCX infinitely often. By property P4, SCXs will succeed infinitely often, a contradiction.

Thus, we have the following theorem.

THEOREM 6. *The algorithms in Figure 6 implement a non-blocking, linearizable multiset.*

6. CONCLUSION

The LLX, SCX and VLX primitives we introduce in this paper can also be used to produce practical, non-blocking implementations of a wide variety of tree-based data structures. In [6], we describe a general method for obtaining such implementations and use it to design a provably correct, non-blocking implementation of a chromatic tree, which is a relaxed variant of a red-black tree. Furthermore, we provide an experimental performance analysis, comparing our Java implementation of the chromatic search tree to leading concurrent implementations of dictionaries. This demonstrates that our primitives enable efficient non-blocking implementations of more complicated data structures to be built (and added to standard libraries), together with manageable proofs of their correctness.

Our implementation of LLX, SCX and VLX relies on the existence of efficient garbage collection, which is provided in managed languages such as Java and C#. However, in other languages, such as C++, memory management is an issue. This can be addressed, for example, by the new, efficient memory reclamation method of Aghazadeh, Golab and Woelfel [2].

Acknowledgements.

Funding was provided by the Natural Sciences and Engineering Research Council of Canada.

7. REFERENCES

[1] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling multi-object operations. In *Proc. 16th ACM Symposium on Principles of Distributed Computing*, pages 111–120, 1997.

[2] Z. Aghazadeh, W. Golab, and P. Woelfel. Brief announcement: Resettable objects and efficient memory reclamation for concurrent algorithms. In *Proc. 32nd ACM Symposium on Principles of Distributed Computing*, 2013.

[3] J. H. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proc. 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 184–193, 1995.

[4] H. Attiya and E. Hillel. Highly concurrent multi-word synchronization. *Theoretical Computer Science*, 412(12–14):1243–1262, Mar. 2011.

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

[6] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. Manuscript available from http://www.cs.utoronto.ca/~tabrown.

[7] T. Brown, F. Ellen, and E. Ruppert. Pragmatic primitives for non-blocking data structures. Manuscript available from http://www.cs.utoronto.ca/~tabrown.

[8] P. Chuong, F. Ellen, and V. Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proc. 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 335–344, 2010.

[9] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In *Proc. 13th IEEE Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 185–192, 2010.

[10] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proc. 29th ACM Symposium on Principles of Distributed Computing*, pages 131–140, 2010. Full version available as Technical Report CSE-2010-04, York University.

[11] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), May 2007.

[12] 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*, pages 289–300, 1993.

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

[14] P. Jayanti and S. Petrovic. Efficiently implementing a large number of LL/SC objects. In *Proc. 9th International Conference on Principles of Distributed Systems*, volume 3974 of *LNCS*, pages 17–31, 2005.

[15] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. *Theory of Computing Systems*, 44(1):39–66, Jan. 2009.

[16] N. Shafiei. Non-blocking Patricia tries with replace operations. In *Proc. 33rd International Conference on Distributed Computing Systems*, 2013. To appear.

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

[18] H. Sundell. Wait-free multi-word compare-and-swap using greedy helping and grabbing. *International Journal of Parallel Programming*, 39(6):694–716, Dec. 2011.

[19] 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.