Universal Constructions for Multi-Object Operations*

(Extended Abstract)

James H. Anderson and Mark Moir
Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175

Abstract

We present wait-free and lock-free universal constructions that allow operations to access multiple objects atomically. Such constructions provide functionality similar to nested critical sections in conventional, lock-based systems. In such a system, two critical sections might be nested, for example, to swap the contents of two shared buffers. Using our constructions, such a transfer can be done in a wait-free or a lock-free manner.

Our universal constructions are based upon multi-word synchronization primitives. In the first part of the paper, we present wait-free implementations of such primitives from one-word primitives. These implementations allow processes that access disjoint words to execute in parallel. Previous implementations of multi-word primitives either overly restrict parallelism, or provide only lock-free execution. We also present several implementations involving one-word universal primitives that allow our constructions to be applied with greater flexibility. In particular, we present time-optimal, wait-free implementations of *Load-Linked* and *Store-Conditional* from *Read* and *Compare-And-Swap*, and vice versa, and implementations that eliminate the need to deal with spurious *Store-Conditional* failures.

1 Introduction

This paper extends recent research on *universal* wait-free and lock-free constructions of shared objects [4, 5]. Such constructions are based upon strong primitives such as *Compare-And-Swap* (*CAS*) or *Load-Linked* (*LL*) and *Store-Conditional* (*SC*), and can be used to implement any object in a wait-free or a lock-free manner. In this paper, we give universal wait-free and lock-free constructions that extend the functionality of previous constructions by allowing multi-object operations. Such operations are allowed to execute in parallel, whenever possible, when applied to disjoint sets of objects. Multi-object operations can be used in much the same manner as nested critical sections in conventional lock-based systems. For example, two critical sections might be nested in such a system in order to transfer the contents of one shared buffer to another. Using our constructions, such a transfer can be done in a wait-free or a lock-free manner.

The multi-object constructions we present are based upon multi-word universal primitives. In the first part of the paper, we give efficient, wait-free implementations of such primitives from one-word primitives. These implementations allow operations on disjoint words to execute in parallel. In contrast, previous implementations of multi-word primitives either overly restrict parallelism, or provide only lock-free execution. We also present time-optimal implementations of one-word primitives that show that *CAS* is equivalent to *LL* and *SC* from a performance standpoint — this stands in contrast to the commonly-held belief that *LL* and *SC* necessarily result in more efficient object implementations.

Time complexity bounds for the implementations we present are summarized in Table 1. In this table, *VL* denotes a *Validate* operation, *MWCAS* denotes a multi-word *CAS*, *MWSC* denotes a multi-word *SC*, and *FSC* denotes a *SC* that may fail spuriously.[1] Other abbreviations are as defined above. Time bounds for each implementation are given in terms of N, the number of processes in the implementation, and M, the number of implemented words or objects.

In the following paragraphs, we present an overview of the three major sections of this paper. The first two of these sections contain results involving one-word and multi-word primitives. In the third of these sections, our multi-object constructions are presented.

Our results involving one-word primitives are presented in Section 2. Three key results are presented in this section: a wait-free implementation of *LL*, *SC*,

*Work supported, in part, by NSF Contract CCR 9216421.

[1] *SC* is usually implemented on top of a write-invalidate cache protocol. A *SC* may incorrectly fail in such an implementation if a cached word is selected for replacement by the cache protocol.

Primitives Used	Primitives Implemented	Worst-Case Time Complexity
Read, *CAS*	*LL*, *SC*, *VL*	$O(1)$, $O(1)$, $O(1)$
LL, *SC*	*Read*, *CAS*	$O(1)$, $O(1)$
LL, *FSC*	*LL*, *SC*, *VL*	$O(1)$, $O(1)$,[2] $O(1)$
LL, *SC*, *VL*	*Read*, *MWCAS*	$O(1)$, $O(N^3M)$
LL, *SC*, *VL*	*LL*, *MWSC*, *VL*	$O(1)$, $O(N^3M)$, $O(1)$
LL, *VL*, *MWSC*	Any multi-object operation	$O(NM^2)$, $O(1)$

Table 1: Summary of results.

and *VL* from *Read* and *CAS*; a wait-free implementation of *Read* and *CAS* from *LL* and *SC*; and an efficient implementation of *LL*, *SC*, and *VL* from *LL* and *FSC*. These results allow us to apply our multi-object implementations given *either Read* and *CAS or LL* and *SC*, and to ignore the possibility of spurious *SC* failures.

Although existing universal constructions can be used to convert between *CAS* and *LL*, *SC*, and *VL*, such constructions entail high overhead. Our implementations of these primitives are time-optimal, requiring constant time per operation. The best previous wait-free implementation of *LL*, *SC*, and *VL*, recently presented by Israeli and Rappoport in [7], requires $O(N)$ time per operation. It also requires N-bit shared variables, which severely limits its usefulness in practice. (Israeli and Rappoport did not present similar constructions for *CAS*.)

Our implementations of multi-word universal primitives are given in Section 3. Again, such primitives may be implemented using existing universal constructions, but at considerable expense. The use of such constructions would also limit parallelism: processes performing operations involving disjoint sets of words could not execute in parallel. The importance of parallelism in this context was first noted by Israeli and Rappoport [7].

The main result of Section 3 is a wait-free implementation of *MWCAS* from *LL*, *SC*, and *VL*. By a straightforward generalization of the one-word case, *MWCAS* can in turn be used to implement *LL*, *VL*, and *MWSC* (see Table 1). The problem of implementing such multi-word primitives has been considered previously by Barnes [2], by Israeli and Rappoport [7], and by Shavit and Touitou [8]. However, the implementations presented in these papers are only lock-free. A process in our implementation attempts to "lock", in a wait-free manner, each of the words that it accesses. A similar (albeit only lock-free) approach is used in [7] and [8].

The main problem encountered in obtaining a wait-free implementation of *MWCAS* is that of efficiently "helping" conflicting operations — such helping is central to most wait-free universal constructions. Good parallelism would seem to preclude one process from helping another that accesses a disjoint set of words. However, such helping is sometimes necessary because of transitivity. For example, if processes p, q, and r access words A and B, B and C, and C and D, respectively, then p may have to help r in order to make progress (because p must help q, which in turn conflicts with r). Our implementation of *MWCAS* deals with this problem by dynamically determining transitive conflicts. In addition, we have incorporated a number of optimizations that allow a *MWCAS* to terminate quickly if it can be linearized to a point where it fails. The performance benefits of allowing *CAS* operations to fail early were first recognized by Bershad in his work on operating system-based implementations of *CAS* [3].

In the last major section of the paper, Section 4, we use the primitives developed previously to obtain both lock-free and wait-free universal constructions of multi-object operations. Both constructions are based upon *LL*, *VL*, and *MWSC*, and are obtained by adapting the universal constructions based upon *LL* and *SC* presented by Herlihy in [5]. As in the implementation of *MWCAS*, the major problem that arises in our wait-free construction is that of ensuring good parallelism in the face of transitive conflicts. In this construction, a process handles such conflicts in two steps: it first applies all operations that transitively conflict with its operation to local copies of the affected objects; it then uses a *MWSC* primitive to attempt to "swing" shared pointers for these objects to point to the local copies just updated. Because of complications arising from transitive conflicts, there are substantial differences between the implementation of this help mechanism and that employed in Herlihy's original construction.

The remainder of this paper consists of the three sections outlined above, followed by concluding remarks in Section 5.

2 One-Word Primitives

In this section, we present efficient implementations of one-word synchronization primitives that allow our results (and others) to be applied with greater flexibility. We begin with a constant-time implementation of *LL*, *SC*, and *VL* using *Read* and *CAS*.[3] We then present a simple, constant-time implementation of *Read* and *CAS* from *LL* and *SC*. The latter construction assumes that *SC* does not fail spuriously. We conclude this section by using *LL* and *FSC* to implement *LL* and *SC*. This result allows us to use our constructions in systems where *SC*

[2]The *SC* operation terminates in $O(1)$ time after the most recent spurious *FSC*. See Section 2 for details.

[3]More accurately, we use shared registers that support atomic *Read* and *Write* operations, as well as shared registers that support *Read* and *CAS* operations. We similarly assume the availability of read/write registers in subsequent constructions.

```
type llsctype = record value: valtype; tag: 0..2N; pid: 1..N end
shared variable X: llsctype; A: array[1..N] of llsctype
private variable old, chk: llsctype; j: 1..N + 1; newtag: 0..2N

procedure LL()          procedure SC(val: valtype)                                        procedure VL()
   old := X;               if chk ≠ old then return false fi;                                return chk = old ∧ X = old
   A[p] := old;            read A[j].tag;
   chk := X;               if j = N then j := 1 else j := j + 1 fi;
   return old.value        select newtag : newtag ∉ {last N tags read} ∪ {last N tags selected} ∪ {last tag successfully CAS'd};
                           return CAS(X, old, (val, newtag, p))
```

Figure 1: Constant-time LL, SC, and VL using *Read* and CAS. Private variables are static between invocations.

```
shared variable X: valtype

procedure CAS(old, new: valtype)
   if LL(X) ≠ old then return false fi;
   if old = new then return true fi;
   return SC(X, new)
```

Figure 2: Constant-time CAS using LL and SC. LL trivially implements *Read*.

might fail spuriously.

Figure 1 depicts an N-process implementation of LL, SC, and VL that is based upon *Read* and CAS. Variable X contains the implemented variable, along with a tag and process identifier. To see how the latter two fields are used, consider the SC procedure. A process p performing a SC chooses a tag value, and then attempts to perform the SC by executing a CAS. As explained below, tags are selected in such a way that this CAS succeeds iff no successful CAS has occurred since the second read of X in p's LL procedure — i.e., iff no other process has performed a successful SC since p's previous LL.

A LL operation is linearized to occur at its first read of X if the values read from X differ, and at its second read of X otherwise. A SC is linearized to occur at its CAS, and a VL is linearized to occur when it reads X.

A key property in proving this implementation correct is that a process p does not prematurely "reuse" a tag, thereby causing a CAS by some process q to succeed when it should fail. To see that this cannot happen, note that if q executes CAS with $old = (x, v, p)$ for some x and v, then $A[q] = (x, v, p)$ holds between q's second read of X and q's CAS. Suppose p reuses tag v in this interval. Because p does not use any of the N most recently selected tags, it follows that p performs at least N successful SC operations before reusing v. Thus, p must have read $A[q]$ in the last N operations, and therefore does not reuse tag v. In [1], we show how the new tag can be selected in constant time. Thus, we have the following theorem.

Theorem 1: LL, SC, and VL can be implemented with constant time complexity using *Read* and CAS. □

We now turn our attention to the implementation of *Read* and CAS using LL and SC, shown in Figure 2. *Read* is trivially implemented by LL. Process p performs a CAS by reading variable X using LL, and possibly performing a subsequent SC of X. If $X \neq old$ or if $X = old = new$, then p's CAS can be linearized to the point at which the LL occurs. Otherwise, if the SC is successful, then the CAS can be linearized to occur when the SC is executed. If the SC fails, then a successful SC by another process has occurred since p's previous LL. Because each successful SC changes the value of X, there is a point during p's CAS at which X differs from old; p's (failed) CAS can be linearized at that point. This construction yields the following theorem.

Theorem 2: *Read* and CAS can be implemented with constant time complexity using LL and SC. □

We should point out that ordinary *Read* and *Write* operations are straightforward to incorporate into the constructions of Figures 1 and 2. In particular, *Write*(new) can be implemented in the construction of Figure 1 like SC; the main difference is that X is updated by a *Write* rather than a CAS. *Write*(new) can be implemented in the construction of Figure 2 by the following code, which is similar to that given for CAS.

```
if LL(X) = new then return fi;
SC(X, new)
```

In Figure 3, we present an implementation of LL, SC, and VL from LL and FSC. In this implementation, tags are maintained that allow a process to identify a spurious FSC failure and to retry the FSC. Thus, if FSC does not fail infinitely often during one invocation of the implemented SC, then the implemented SC eventually terminates. Tags are maintained using a mechanism that is similar to that employed in Figure 1. This implementation yields the following theorem.

Theorem 3: LL and FSC can be used to implement wait-free LL and VL operations, and a SC operation that terminates provided only finitely many spurious FSC failures occur per SC invocation. □

```
shared variable X: record value: valtype; tag: 0..2N; pid: 1..N end
private variable old, chk: valtype; newtag: 0..2N

procedure LL()                procedure SC(new: valtype)
   old := X;                     if chk ≠ old then return false fi;
   A[p] := old;                  read A[j].tag;
   chk := LL(X);                 if j = N then j := 1 else j := j + 1 fi;
   return old.value              select newtag : newtag ∉ {last N tags read} ∪ {last N tags selected} ∪ {last tag successfully FSC'd};
                                 while true do
procedure VL()                      if FSC(X,(new, newtag)) then return true
   chk := LL(X);                    elseif LL(X) ≠ old then return false
   return old = chk                 fi
                                 od
```

Figure 3: Implementation of *LL*, *SC*, and *VL* using *LL* and *FSC*.

3 Multi-Word Primitives

In this section, we describe our implementation of *Read* and *MWCAS* for processes $1..N$ on words $1..M$ using *LL*, *SC*, and *VL*. A straightforward generalization of the construction in Figure 1 implements *LL*, *MWSC*, and *VL* using *Read* and *MWCAS*. Details are deferred to the full paper.

Herlihy's universal construction [5] can be used to implement any shared object using *LL* and *SC*. In particular, *MWCAS* can be implemented using Herlihy's construction by treating all M words as one object. However, this approach suffers from two drawbacks. First, concurrent *MWCAS* operations cannot execute in parallel, even if they access completely disjoint sets of words. Thus, this approach severely limits parallelism. Second, all M words must be copied for each operation, even if the operation accesses only one word. If M is large, this can be a significant disadvantage. Our implementation, shown in Figures 4 and 5, circumvents both of these problems. Before giving a detailed description of our implementation, a brief overview is in order.

The *Read* procedure takes an argument a, $1 \leq a \leq M$, and returns the value of word a. The *MWCAS* procedure takes four arguments: *nw*, *words*, *old*, and *new*. The *nw* argument is the number of words accessed by the *MWCAS* operation. The remaining arguments are lists, each containing *nw* values: *words* specifies the words (in ascending order) to be accessed by the *MWCAS* operation; *old* contains the old value for each word accessed; and *new* contains the new value for each word accessed. If each word accessed contains the corresponding *old* value, then each word is modified to contain the corresponding *new* value, and the *MWCAS* operation succeeds, returning *true*. Otherwise, the words are unchanged and the operation fails, returning *false*.

In order for a *MWCAS* operation to fail, it is sufficient to detect that just one word does not contain the corresponding *old* value. However, such an operation should succeed only if *all old* values are detected to match the words' current values at the same time. Unfortunately, a word might change values after it has been observed to "match", but before another word has been checked. To address this problem, our algorithm "locks" each word in sequence, each time checking that the corresponding *old* value matches the current value of the word. A word cannot be modified while locked, so if all words accessed by an operation are locked, then it is safe for that operation to succeed.

A process that wishes to modify a word may not do so while that word is "locked" by another process. In order to make the implementation wait-free, one process must therefore be able to "help" another process to complete its operation. This is achieved by having each process "announce" the parameters and state of its operation, so that another process can continue to execute a partially completed operation. *LL*, *VL*, and *SC* operations are used to ensure that each stage of each operation is executed exactly once. Techniques similar to this one have been used previously [2, 7, 8]. However, these implementations are only lock-free, not wait-free, so operations are not guaranteed to complete. We employ a technique that allows a process to detect concurrent operations with which it potentially interferes, and to help complete such operations. If a process is interfered with sufficiently often, it is eventually helped to complete its operation, so no starvation is possible. In the remainder of this section, we describe our implementation in more detail. A complete proof is deferred to the full paper.

We first describe the shared data structures and how they are used. The parameters of each operation by process p are copied into $PARAM[p]$ (lines 7 to 10). This allows other processes to help process p's operation to complete. Process p writes an access matrix entry $AM[p][w]$ to indicate that p's operation accesses word w (line 9) or that process p is helping an operation that accesses word w (line 31). For each word w, $MEM[w]$ contains the current or soon-to-be-current value of word w, and $LOCK[w]$ is used to lock word w. Each process p has a status variable $STAT[p]$, which represents the state of p's current operation, if any.

MWCAS operations proceed in "phases". As shown in Figure 6, each operation has an *init* phase, a *lock* phase, and an *unlock* phase, executed in that order. An

```
type  param_type = record nw: 1..M; words: array[1..M] of 1..M; old, new: array[1..M] of val_type end;
      stat_type = record stat: {init, lock, modify, unlock}; flag: {succ, fail, diff, help}; proc: 1..N  end;
      lock_type = record owner: 0..N; index: 1..M end;
      access_type = record help: 0..N; old: indirect ∪ val_type; index: 1..M end

shared variable  MEM: array[1..M] of val_type init initial values for implemented words;   /* Implemented words */
                 LOCK: array[1..M] of lock_type init (0, 1);                               /* One lock for each word */
                 PARAM: array[1..N] of param_type init (1, (1, ..., 1), (0, ..., 0));      /* Parameters to each operation */
                 STAT: array[1..N] of stat_type init (init, succ, 1);                      /* Status of each operation */
                 AM: array[1..N] of array[1..M] of access_type                             /* Access matrix */


procedure Read(a: 1..M) returns valtype

v := MEM[a];                                 /* Read most recent value (possibly not yet current) */
x := LL(&LOCK[a]);                           /* v is current unless word is locked by a process r, ... */
if x.owner = 0 ∨ LL(&STAT[x.owner]).stat ≠ modify then return v fi;   /* ... which is in the modify phase */
old := PARAM[x.owner].old[x.index];          /* Get previous value, which may still be current */
if ¬VL(&LOCK[a]) then return v fi;
if VL(&STAT[x.owner]) then return old else return v fi   /* Only return old value if v still not current */


procedure MWCAS(nw: 1..M; addr: array of 1..M; old, new: array of val_type) returns boolean

PARAM[p].nw := 1; list := {};                /* Don't violate invariants while changing PARAM; No AM entries yet */
for j := 1 to nw do                          /* Announce parameters to operation */
    PARAM[p].words[j], PARAM[p].old[j], PARAM[p].new[j] := addr[j], old[j], new[j];
    AM[p][addr[j]] := (p, old[j], j);  insert(list,addr[j])      /* Initalize access matrix */
od;
PARAM[p].nw := nw;
STAT[p] := (lock, fail, p);  Help(p, p, 0, 0);                   /* Start operation; Begin by helping self */
for each j ∈ list do AM[p][j] := (0, 0, 1) od;                  /* Stop other processes from helping this process */
st := STAT[p];
if st.flag ≠ diff ∨ nw > 1 then                                  /* If words were potentially locked ... */
    for j := 1 to nw do                                          /* ... then unlock them */
        v := LL(&LOCK[addr[j]]);                                 /* Invalidate late locks */
        if v.owner = p ∨ (v.owner = 0 ∨ STAT[v.owner] ∉ {lock, modify}) then
            if ¬SC(&LOCK[addr[j]], (0, 1)) then                  /* Undo late lock if it happened before invalidated */
                if LL(&LOCK[addr[j]]).owner = p then
                    SC(&LOCK[addr[j]], (0, 1))
                fi
            fi
        fi;
        for k := 1 to N do                                       /* Help processes whose operations touch this word ... */
            pr := AM[k][addr].help;
            if pr ≠ 0 then                                       /* ... or are waiting indirectly for this word */
                if (LL(STAT[pr])).stat = lock then
                    list := {};  Do_Locking(pr, pr, 0);
                    for each j ∈ list do AM[p][j] := (0, 0, 1) od   /* Stop other processes from helping this process */
                fi
            fi
        od
    od;
    if st.flag = help ∧ LL(&STAT[st.proc]) = modify then Do_Modifying(st.proc) fi
fi;                                          /* If killer's modify phase is not yet complete, help it finish */
STAT[p] := (init, fail, p);                  /* Initialize STAT[p] for next time */
return st.flag = succ


procedure Help(pr: 1..N; i: 1..N; last_locked: 0..M; index: 0..M)

if (LL(STAT[i])).stat = lock then
    if index > 0 ∧ ¬VL(&LOCK[last_locked]) then return fi;     /* Previously locked word was unlocked */
    Do_Locking(pr, i, index)                 /* Help process i continue (or start) its locking phase */
fi;
if (LL(STAT[i])).stat = modify then Do_Modifying(i) fi; return   /* Help i's modify phase, if necessary */
```

Figure 4: *Read*, *MWCAS*, and *Help* procedures.

```
procedure Do_Locking(pr: 1..N; i: 1..N; index: 0..M)
nw := PARAM[i].nw;                                   /* Determine how many words i's operation touches */
for j := index + 1 to nw do                          /* Help to lock each word of process i's operation */
    done, addr := false, PARAM[i].words[j];
    if AM[p][addr].help = 0 then
        AM[p][addr] := (pr, indirect, 1); insert(list, addr)      /* Announce indirect waiting */
    fi;
    while ¬done do                     /* Keep trying until successful (or VL fails - see below) */
        v := LL(&LOCK[addr]);                                                   /* Read lock */
        if ¬VL(&STAT[i]) then return                /* Don't help anymore if phase is complete */
        elseif v.owner = i then done := true                         /* Check if already locked */
        elseif ¬VL(ST[pr]) then return             /* Original process has completed locking phase */
        elseif v.owner ≠ 0 ∧ STAT[v.owner] ∈ {lock, modify} then  /* If locked by a process that needs help... */
            Help(pr, v.owner, addr, v.index)                         /* ... then help that process */
        else
            old := PARAM[i].old[j];        /* Fail without further locking if old value differs from current */
            val := MEM[addr];
            if VL(&LOCK[addr]) then
                if val ≠ old then SC(&STAT[i], (unlock, diff, i)); return           /* Operation fails */
                elseif ¬VL(&STAT[i]) then return                   /* Quit if phase already completed */
                elseif SC(&LOCK[addr], (i, j)) then                /* Otherwise, try to lock the word */
                    done, PV[addr] := true, PARAM[i].old[j]               /* Note that PV is auxilliary */
                fi
            fi
        fi
    od
od;
SC(&STAT[i], (modify, fail, i)); return                                    /* Start modify phase */


procedure Do_Modifying(i: 1..N)                   /* Write new values for successful MWCAS operation */
nw := PARAM[i].nw;                                   /* Determine how many words i's operation touches */
for j := 1 to nw do                                             /* For each word of operation ... */
    addr := PARAM[i].words[j];                                 /* ... get parameters for this word */
    old := PARAM[i].old[j];
    new := PARAM[i].new[j];
    if old ≠ new ∧ LL(&MEM[addr]) = old then       /* If this word is changing, prepare to write new value */
        for h := 1 to N skip i do                                       /* For each other process ... */
            val := AM[h][addr];          /* ... if overlapping operation is in progress, prepare to make it fail */
            if val.help = h ∧ val.old ≠ indirect ∧ (LL(&STAT[h])).stat = lock then
                if ¬VL(STAT[i]) then return fi;                 /* Return if phase is already complete */
                if AM[h][addr] = val then                    /* Ensure it's still ok to fail h's operation */
                    if val.old = old then SC(&STAT[h], (unlock, help, i))        /* h fails after i succeeds */
                    else SC(&STAT[h], (unlock, fail, i))                         /* h fails before i succeeds */
                    fi
                fi
            fi
        od;
        if ¬VL(STAT[i]) then return fi;                         /* Return if phase is already complete */
        SC(&MEM[addr], new)                                                    /* ... update the word */
    fi
od;
SC(&STAT[i], (unlock, succ, i)); return                   /* Operation is complete; unlock words */
```

Figure 5: *Do_locking* and *Do_Modifying* procedures.

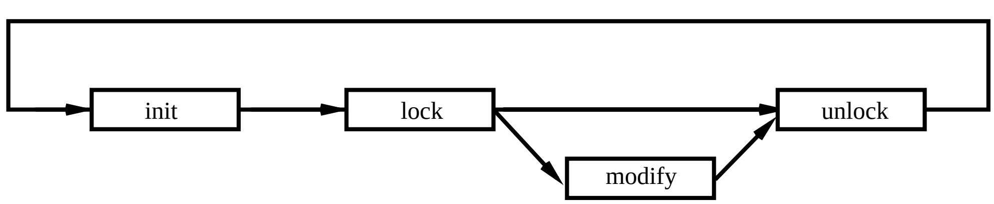

Figure 6: Phases of a *MWCAS* operation.

operation that successfully modifies one or more words also has a *modify* phase between its *lock* and *unlock* phases. The *init* phase and the *unlock* phase for an operation by process p are executed by p in the *MWCAS* procedure. Steps in the *lock* and *modify* phases may be executed "on behalf of" p by any process in the *Do_Locking* and *Do_Modifying* procedures, respectively. Process p performs a *MWCAS* operation by announcing its parameters and filling in the appropriate access matrix entries (lines 7 to 10) and then setting its status to *lock* and calling the *Help* procedure. The *Help* procedure executes, as necessary, p's *lock* phase and then p's *modify* phase. Below, we describe each phase in detail.

The *lock* phase attempts to lock each word accessed by p's operation in turn (lines 28 to 42). To ensure that p is eventually helped if it repeatedly fails to lock a word w, p first sets $AM[p][w]$ (lines 30 and 31). Process p also maintains a local *list*[4] of AM entries that have been set so that they may be cleared after p's operation is complete (line 12 or 21). If a word is locked by another process r that has not completed its *lock* or *modify* phase, then *Help* is called (line 35) on behalf of process r in order to release r's lock on that word. Otherwise, before locking a word, two checks are done. The first is to ensure that the current value of the accessed word matches the *old* value for the operation being executed. If not, the operation can fail immediately, without attempting to do further locking (line 39). The second check (line 40) is to ensure that a process does not belatedly attempt to lock a word on behalf of a process that has already completed its *lock* phase. If all the words of an operation are successfully locked, then the status of the operation is changed to *modify* (line 42). At this point, the operation is guaranteed not to fail.

In the *modify* phase, each memory word accessed by the successful operation is modified, if necessary, to its *new* value (line 55), and then the status of the operation is changed to *unlock* (line 56). The operation is linearized to the point at which line 56 is executed. Thus, $MEM[w]$ contains the current value of word w except for the interval between $MEM[w]$ being modified (line 55) and the end of the operation's *modify* phase (line 56). This observation is important for implementing an efficient *Read* operation, which is described later.

Suppose process p modifies word $MEM[w]$ in the *modify* phase of an operation of process p. Then a concurrent *MWCAS* operation that accesses word w can safely fail because either its *old* value does not match before the change, or it does not match afterwards. Before modifying word w (line 55), process p first checks each other process q to see if it has a *MWCAS* operation that is attempting to lock word w (lines 48 and 49). If so, the status of q's operation is changed to *unlock*, causing it to fail. If q's *old* value for word w differs from p's, then q's operation can be linearized to fail immediately (line 53). However, if q has the same *old* value for word w as p does, then q's operation cannot fail until p's operation is linearized. In this case, q's status is changed to $(unlock, help, p)$ (line 52). When process q executes line 22, q calls *Do_Modifying* to ensure that p's operation has been linearized before q returns. In this case, q's operation is linearized immediately after p's operation — i.e., when p's status is changed to *unlock*.

Despite these optimizations, there is a risk that some process p repeatedly fails to lock $LOCK[w]$ because some other process q repeatedly locks $LOCK[w]$, but does not help to complete the operation that p is executing. This can arise either if q repeatedly performs operations whose *old* and *new* values for word w are equal, or if q's operation repeatedly fails after locking $LOCK[w]$. In either case, it is necessary to ensure that p's operation is helped to complete.

In our implementation, each operation terminates because, after completing an operation, each process q helps to complete the *lock* phase of any operation with which q's operation might have interfered (lines 19 to 21). From a performance standpoint, it is desirable for a process q to help an operation only if q actually interfered with that operation. However, because this interference can be caused by other processes acting "on behalf of" q, there is an overhead associated with detecting exactly which operations were interfered with by q's operation. We have chosen to assume that q's operation interferes with every operation that concurrently accesses a word accessed by q's operation, with one common exception: if a one-word operation fails because its *old* value did not match the value of the word accessed (line 39), then it can be shown that the lock associated with that word is not modified on behalf of that operation, so no helping is necessary. We believe that one-word operations are likely to be invoked much more frequently than multi-word operations. Therefore, this optimization is a useful compromise between conservatively assuming that each operation interferes with every word it accesses, and incurring the overhead of determining which operations were interfered with.

In the *unlock* phase (lines 14 to 18), a process q ensures that for each word w in q's operation, $LOCK[w]$ is not locked (and will not later become locked) on behalf of process q. Because of the possibility that some process p is about to execute line 41 on behalf of process q, process q must $SC(LOCK[w])$ in order to ensure that p's SC will fail, even if $LOCK[w]$ is not currently locked by process q. If q's SC fails, then it is possibly as a result of p executing line 41, thereby locking $LOCK[w]$ on behalf of q. Thus, at lines 17 and 18, process q checks again to ensure that $LOCK[w]$ is not locked by process q. This completes the discussion of the *MWCAS* oper-

[4] The operations used to access this local list are easily implemented in optimal time, and are therefore not presented here.

ation. We now describe the *Read* operation.

The *Read*(a) procedure (lines 1 to 6) assigns $v := MEM[a]$. As mentioned above, v is the current value of word a unless some process p has modified $MEM[a]$ and has not yet completed its *modify* phase when v is read. The *Read* operation detects this case (line 3), and determines the previous value of $MEM[a]$ (which is still the current value of word a) by reading the parameters to p's operation (line 4). The *VL* operation is used to ensure that the *old* value read is still correct (lines 5 and 6). If it is not, then it can be shown that process p's operation has completed, so it is safe to return v.

In the full paper, we show that process p can attempt to lock each word at most $O(N)$ times before p's operation is completed. Using this property, we obtain Theorem 4. A straightforward generalization of the one-word implementation of *LL*, *SC*, and *VL* presented in Section 2 yields Theorem 5.

Theorem 4: *Read* and *MWCAS* can be implemented with worst-case time complexity $O(1)$ and $O(N^3M)$, respectively, from *LL*, *SC*, and *VL*. □

Theorem 5: *LL*, *MWSC*, and *VL* can be implemented with worst-case time complexity $O(1)$, $O(N^3M)$, and $O(1)$ respectively, from *LL*, *SC*, and *VL*. □

4 Multi-Object Constructions

In this section, we first describe a relatively simple lock-free construction for implementing multi-object operations. We then present a wait-free construction for multi-object operations in more detail. Both constructions use *LL*, *VL*, and *MWSC*.

The lock-free construction is a generalization of Herlihy's single-object, lock-free construction [5]. A pointer to each object affected by a multi-object operation is loaded using *LL*. A local copy of each object is made, and the multi-object operation is applied to the copies. Finally, a *MWSC* operation is used to attempt to "install" the new versions of the affected objects. This is repeated until the *MWSC* is successful. This lock-free construction is presented in detail in the full paper.

We now turn our attention to the wait-free, universal construction shown in Figure 7.[5] We first describe the major data structures used in our construction, and then describe how a process performs a multi-object operation. We conclude this section with a brief description of the time complexity analysis for this construction. Complete proofs appear in the full paper.

The major data structures are *OBJ*, an array of pointers to the current versions of the implemented objects; *ANC*, which is used to "announce" operations so that they may be helped; and *AM*, an access matrix that is similar to the one used in Section 3. $ANC[p]$ contains a function that performs p's operation, the parameters to the operation, a bit that is used to detect completion of the operation, and the index of the first object accessed by the operation (or 0 if p does not have a current operation). $AM[p][n]$ contains *op* if p's current operation accesses object n, *help* if p is helping operations that access object n, and *none* otherwise.

A process p performs a multi-object operation by invoking *Do_Op*. In lines 9 and 10,[6] p's row of the access matrix *AM* is initialized to show which objects p's operation accesses. At line 11, p computes a bit that differs from p's bit in the first object accessed by p's operation. This bit is later used (lines 13 and 25) to determine whether p's operation has been completed. At line 12, p's *ANC* entry is filled with the function and parameters for p's operation, the bit computed at line 11, and the index of the first object accessed by p's operation.

The loop at lines 13 to 30 is repeated until the test at line 13 fails, indicating that p's operation has been successfully completed. This test is performed twice to avoid a race condition similar to the one described by Herlihy in [5]. Inside this outer loop, p detects operations that conflict with its own and attempts to perform these operations along with its own by making local copies of the affected objects, applying the operations to the local copies, and finally using *MWSC* to "install" the new versions of the objects.

The conflicting operations are detected by calling *TC* (line 16) to compute the "transitive closure" of conflicts. The transitive closure is computed *after* the pointers to the affected objects have been loaded using *LL*. This ensures that if process p's *MWSC* fails twice because of intermediate *MWSC* operations on some object w, then the process q that causes the second failure must perform its *LL* of $OBJ[w]$ — and therefore compute its transitive closure — *after* p's *ANC* entry has been written. Thus, q's transitive closure contains all words accessed by p's operation, so q applies p's operation.

Computing the transitive closure after loading the pointers presents a difficulty: the pointers to be loaded *are* those in the transitive closure. To get around this apparent contradiction, we *LL* the object pointers known to be in the transitive closure (line 14) and then recompute the closure (line 16). This is repeated until the closure does not include any more objects than were previously loaded (checked at line 17). Because objects are not removed from p's transitive closure (recorded in p's row of *AM* at line 17) until p's operation is com-

[5] In this figure, line numbers are included for reference only: they are not intended to denote atomic statements.

[6] The loop at line 9 has been simplified for ease of presentation; this loop, and others like it at lines 5, 14, and 26, can actually be implemented so that its time complexity is proportional to the number of times the following **if** condition is satisfied, and not necessarily to M. Also, the set operations, such as those in lines 4, 15, and 19 can be implemented without $O(M)$ or $O(N)$ loops.

```
type objtype  = record contents: contype; retval: array[1..N] of rettype; bit: array[1..N] of boolean end;
     anctype  = record func: functype; par: partype; bit: boolean; first: 0..M end

shared variable  OBJ: array[1..M] of *objtype; ANC: array[1..N] of anctype; AM: array[1..N][1..M] of {op, help, none};
                 COPY: array[0..N] of array[1..M] of objtype

initially  (∀p, n : 1 ≤ p ≤ N ∧ 1 ≤ n ≤ M :: ANC[p].first = 0 ∧ AM[p][n] = none ∧
               new[p][n] = &COPY[p][n] ∧ OBJ[n] = &COPY[0][n] ∧ COPY[0][n].contents = initial value of nth object)

private variable  i, h, k, first: 1..M; j: 1..N; bit: boolean; old, new: array[1..M] of *objtype; proc: set of 1..N;
                  tclist: set of 1..M; retval: array[1..M] of rettype; objl: array[1..M] of 1..M


procedure TC(objno: 1..M)
   private variable m: 1..M; n: 1..N

   for n := 1 to N do                                            /* Check each process to see if... */
      if LL(&ANC[n].first) ≠ 0 then                              /* ... it has an active operation ... */
         if AM[n][objno] ≠ none ∧ n ∉ proc then    /* ... that conflicts with this word, and has not been added ... */
            proc := proc ∪ {n};                                  /* If so, then include this process's words... */
            for m := 1 to M do
               if AM[n][m] ≠ none then
                  if ¬VL(&ANC[n].first) then end_for fi;
                  if m ∉ tclist then tclist := tclist ∪ {m}; TC(m) fi      /* ... and transitive conflicts */
               fi
            od
         fi
      fi
   od;
   return


procedure Do_Op(numobjs: 1..M; obj: array[1..M] of 1..M; func: functype; par: partype) returns array[1..M] of rettype

   for i := 1 to M do if AM[p][i] ≠ none then AM[p][i] := none fi od;      /* Initialize AM row to reflect operation */
   for i := 1 to numobjs do AM[p][obj[i]] := op od;
   bit := ¬OBJ[obj[1]]->bit[p];                                                /* Compute termination bit */
   ANC[p].func, ANC[p].par, ANC[p].bit, ANC[p].first := func, par, bit, obj[1];      /* Announce operation */
   while (OBJ[obj[1]]->bit[p] = bit) ∨ (OBJ[obj[1]]->bit[p] = bit) do               /* Operation is not done yet */
      repeat
         for i := 1 to M do if AM[p][i] ≠ none then old[i] := LL(&OBJ[i]) fi od;     /* Load pointers in closure */
         proc, tclist, same, k, fail := {}, {}, true, 1, false;
         TC(obj[1]);                                                           /* Recompute transitive closure */
         for each i ∈ tclist do if AM[p][i] = none then AM[p][i] := help; same := false fi od
      until same;                                 /* Repeatedly compute transitive closure until no more is added */
      for each i ∈ tclist do                                          /* Make local copies of local affected objects */
         memcpy(new[i], old[i], sizeof(objtype)); word[k], k := i, k + 1; if ¬VL(&OBJ[i]) then fail := true; exit_for fi
      od;
      if ¬fail then
         for j := 1 to N do                         /* Perform operations covered by transitive closure on local copies */
            if LL(&ANC[j].first) ≠ 0 then
               cover, h, first, func, par := true, 1, ANC[j].first, ANC[j].func, ANC[j].par;
               if ANC[j].bit ≠ new[first]->bit[j] then
                  for i := 1 to M do if AM[j][i] = op then
                     if i ∈ tclist then h, objl[h] := h + 1, i else cover := false fi
                  fi od;
                  if cover ∧ VL(&ANC[j].first) then func(new, objl, par); new[first]->bit[j] := ¬new[first]->bit[j] fi
               fi
            fi
         od;                                                                 /* Try to make local copies current */
         if MWSC((&OBJ[word[1]], ..., &OBJ[word[k − 1]]), (new[word[1]], ..., new[word[k − 1]])) then
            for i := 1 to k − 1 do new[word[i]] := old[word[i]] od; exit_while                 /* Reclaim old copies */
         fi
      fi
   od;
   ANC[p].first := 0;
   for k := 1 to numobjs do retval[k] := OBJ[obj[i]]->retval[p] od;          /* Retrieve return values from each object */
   return retval
```

Figure 7: Wait-free, multi-object, universal construction.

pleted, it can be shown that the test at line 18 can fail at most $M-1$ times over the execution of p's operation.

After the conflicts have been detected, and the object pointers loaded, p makes a local copy of each of the affected objects. After copying each object, the pointer to that object is validated. If the *VL* fails, then the loop at lines 13 to 30 is restarted. Because the *MWSC* at line 29 would fail if executed in this case, unnecessary computation is avoided by restarting the loop immediately. This also avoids applying an operation to an out-of-date copy of the object. Having made local copies of the affected objects, p checks each process j (line 22) to see if it has a current operation (line 23) that has not been completed (line 25) and that only accesses objects within p's closure (lines 26 and 27). In performing this last check, p also compiles a list *objl* of the objects accessed by j's operation. If all of these checks succeed, then p calls the function pointed to by $ANC[j].func$, which applies j's operation to p's local copies and also modifies the $retval[j]$ field of some or all of the objects accessed, if return values are required. Process p then toggles j's bit in the first object accessed by j's operation to record that j's operation has been applied to p's local object copies. Finally, at line 29, p attempts to install its local object copies as the new current objects.

We conclude this section by briefly describing the time complexity analysis, which appears in the full paper. The key lemma is that during an operation by process p, p's *MWSC* can fail at most twice on account of any object, and therefore at most $M+1$ times in total. We also show that *TC* is called at most $O(M)$ times during p's operation and that the time complexity of calling *TC* (line 16), including all recursive calls, is $O(MN)$. All other terms in the time complexity are dominated by these terms. Thus, the construction in Figure 7 yields the following result. This gives the same asymptotic time complexity as Herlihy's construction [5] for the single-object case (that is, when $M = 1$). In fact, it can be shown that even when $M > 1$, if no multi-object operations conflict with a single-object operation, then that operation is completed in $O(N)$ time.

Theorem 6: Using *LL*, *VL*, and *MWSC*, wait-free, multi-object operations can be implemented with time complexity $O(NM^2)$. □

5 Concluding Remarks

Previous wait-free and lock-free object implementations allow operations to access only one object, which may preclude their use in some settings. Our implementations overcome this limitation by allowing operations to access multiple objects simultaneously in a wait-free or a lock-free manner. Our implementations are designed to permit operations on distinct sets of objects to execute in parallel, wherever possible.

The optimizations employed in our implementations yield very low time complexity in all but pathological circumstances that should rarely occur in practice. For example, the time complexity of a *MWCAS* by process p approaches the worst case only if p helps many concurrent operations through their *modify* phases. However, when contention is high, *MWCAS* operations are more likely to fail, and failing operations do not execute *modify* phases. Also, the best-case time complexity is significantly lower than the worst case — $O(1)$ for failing *MWCAS* operations and $O(N)$ for successful ones.

References

[1] J. Anderson and M. Moir, "Universal Constructions for Large Objects", submitted to Ninth International Workshop on Distributed Algorithms, September 1995.

[2] G. Barnes, "A Method for Implementing Lock-Free Shared Data Structures", *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1993, pp. 261-270.

[3] B. Bershad, "Practical Considerations for Non-Blocking Concurrent Objects", *Proceedings of the 13th international Conference on Distributed Computing Systems*, May 1993, pp. pages 264-274.

[4] M. Herlihy, "Wait-Free Synchronization", *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 1, 1991, pp. 124-149.

[5] M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Objects", *ACM Transactions on Programming Languages and Systems*, Vol. 15, No. 5, 1993, pp. 745-770.

[6] M. Herlihy and J. Wing, "Linearizability: A Correctness Condition for Concurrent Objects", *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 3, 1990, pp. 463-492.

[7] A. Israeli and L. Rappoport, "Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives", *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, August 1994, pp. 151-160.

[8] N. Shavit and D. Touitou, "Software Transactional Memory", these proceedings.