Practical Implementations of Non-Blocking Synchronization Primitives

Mark Moir*

Department of Computer Science
The University of Pittsburgh
Pittsburgh, PA 15260

Abstract

This paper is concerned with system support for non-blocking synchronization in shared-memory multiprocessors. Many non-blocking algorithms published recently depend on the Load-Linked (LL), Validate (VL), and Store-Conditional (SC) instructions. However, most systems support either Compare-and-Swap (CAS) or a weak form of LL and SC that imposes several restrictions on the use of these instructions and does not provide the exact semantics expected and assumed by algorithm designers. These limitations currently render several recent non-blocking algorithms inapplicable in most systems. The results presented here eliminate this problem by providing practical means for implementing any algorithm that is based on these instructions on any multiprocessor that provides either CAS or a form of LL and SC that is sufficiently weak that it is provided by all current hardware implementations of these instructions. This is achieved in two steps. First, we propose a slight modification to the interface for LL, VL, and SC, which will not greatly impact programmers. We then exploit this modification to provide time-optimal, space-efficient implementations of the desired primitives using commonly available ones.

1 Introduction

Non-blocking synchronization has been of increasing interest recently, largely due to its ability to avoid the ill effects of locking such as convoying, deadlock, priority inversion, contention, and susceptibility to process delays and failures (see, for example, [5]). It is well recognized that "strong" synchronization primitives such as compare-and-swap (CAS) and load-linked/store-conditional (LL/SC) are necessary for general non-blocking synchronization [6]. As a result, most modern shared-memory multiprocessors support some form of strong synchronization primitive, and many non-blocking algorithms have been published that depend on such primitives.

Unfortunately, a significant gap remains between the primitives relied upon by designers of non-blocking algorithms, and the primitives provided in hardware. In particular, many machines provide either CAS or LL/SC, but not both. Furthermore, most hardware implementations of the LL/SC instructions do not fully implement the semantics expected and assumed by algorithm designers. As a result, several non-blocking algorithms developed recently (e.g. [2, 3, 4, 7, 10, 14]) are not directly applicable on current multiprocessors[1]. The results presented here eliminate this gap by providing time- and space-efficient, wait-free implementations of the primitives required by such algorithms, using primitives that are commonly implemented in hardware.

Specifically, we aim to provide implementations that allow any algorithm that is based on either CAS or LL/VL/SC to be applied on any shared-memory multiprocessor that provides either CAS or a restricted version of LL and SC, which we call RLL and RSC. The RLL/RSC pair has the following restrictions.

- a process may not access memory between an RLL and the subsequent RSC;
- no validate (VL) instruction is provided;
- RSC may occasionally fail when the normal semantics of LL/SC dictate that it should succeed; and
- variables accessed by RLL and RSC must be contained within one machine word.

*Work supported in part by an NSF CARREER Award, CCR 9702767. Email: moir@cs.pitt.edu.

[1] Of course, it is straightforward to implement LL and SC using locks, but this defeats the purpose of the non-blocking algorithms that use them.

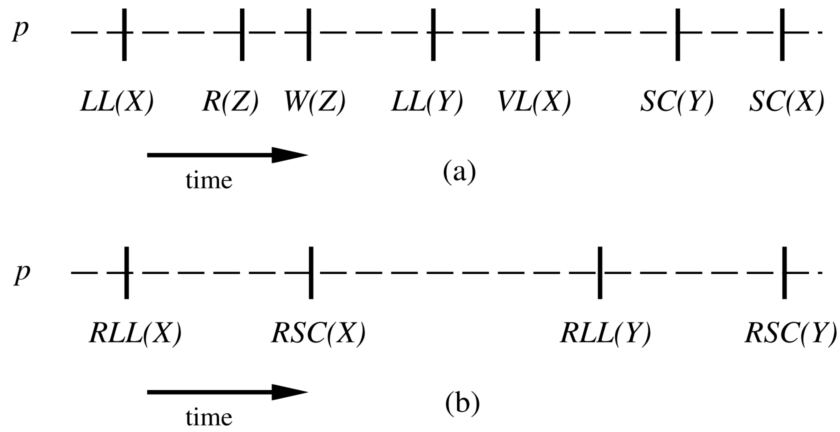

Figure 1: Comparison of uses of LL/VL/SC (a) and RLL/RSC (b) instructions.

The implications of the first two of these restrictions are illustrated in Figure 1. In the simple example shown in Figure 1(a), process p performs a LL on a variable X, reads and writes another variable Z, performs a LL on a third variable Y, validates X (using VL), and then performs a SC on Y and then on X. Thus, process p has two concurrent LL-SC sequences — one on X and one on Y. (A LL-SC sequence is a sequence of one LL, zero or more VLs and zero or one SC's on one variable.) Observe that, in the example shown in Figure 1(a), p violates both of the first two restrictions listed above. In contrast, the restricted RLL/RSC pair cannot be used this way. As illustrated in Figure 1(b), RLL and RSC must be used in simple pairs, with no memory accesses between an RLL and the subsequent RSC.

The RLL and RSC primitives used in this paper are sufficiently weak that, to our knowledge, they are provided by all hardware implementations of LL/SC-like primitives. For example, on the MIPS R4000 processor [12], LL and SC are implemented using a single bit `LLBit` per processor. This bit is set by LL, and a subsequent SC succeeds only if the bit is still set. `LLBit` is cleared by *any* cache invalidation. Thus, while the cache-coherence mechanism ensures that the SC fails if the address is written by another processor, it is quite possible for the SC to fail *spuriously*, i.e., even if no other processor has written that address. Furthermore, because there is only one `LLBit` per processor, it is impossible to have concurrent LL-SC sequences on the R4000. Other processors that provide LL and SC, such as the DEC Alpha [1] and the PowerPC [13], implement these instructions similarly and have similar restrictions.

It is easy to see why hardware designers prefer to provide weaker versions of LL and SC than those commonly assumed by algorithm designers. First, if concurrent LL-SC sequences are to be supported, then the processor must maintain an equivalent of the R4000's `LLBit` for *every* LL-SC sequence, and must also record the address of each word accessed. This would clearly be expensive to implement. The complexity of supporting concurrent LL-SC sequences in hardware is complicated even further by the possibility that multiple processes on one processor might use these instructions concurrently. Recent lower bound results by Valois [15] also suggest that it is prohibitively expensive to provide the full semantics of LL, VL, and SC in hardware. It is therefore natural to investigate the possibility of employing existing hardware versions to efficiently provide the desired semantics in software.

Our strategy for allowing any algorithm that is based on CAS or on LL, VL, and SC to be implemented in any machine that provides either CAS or RLL and RSC is based on time- and space-efficient implementations of CAS from RLL/RSC and of LL/VL/SC from CAS. (The techniques used in these implementations can also be combined to provide direct implementations of LL/VL/SC from RLL/RSC.) Several related results have been presented previously, but all have disadvantages that render them impractical in most settings. First, Israeli and Rappoport [10] present an implementation of LL/SC from CAS that depends upon unrealistic assumptions about the size of machine words and has high time complexity. Anderson and Moir [2] improve upon this by providing a constant-time implementation with realistic assumptions. However, as discussed later, this implementation has impractical space requirements if used to implement many variables. Finally, Valois [15] outlines implementations that improve on these space requirements. However, these results were intended primarily to establish upper bounds on space requirements, and are not practical.

In this paper, we first present two very simple implementations: the first implements CAS using RLL and RSC, and the second implements LL, VL, and SC using CAS. We then combine the techniques used in these two implementations to provide an implementation of LL, VL, and SC with the semantics desired by programmers, using the restricted RLL and RSC instructions that are implemented in modern shared memory multiprocessors.

These implementations all have constant time complexity (in the case of implementations based on RLL and RSC, this means that they terminate provided only finitely many spurious failures occur during each SC operation, and that they do so in constant time after the last spurious failure); have extremely low space overhead; and — in the case of the LL/VL/SC implementations — permit multiple LL-SC sequences to be executed concurrently. However, they also have two potential disadvantages. First, they employ tags that are treated as unbounded. If these tags wrap around while some process executes one LL-SC sequence, then the potential exists for the implementations to behave incorrectly. By increasing the size of the tags, the likelihood of such an error can be made vanishingly small. However, our implementations require that the tags are stored together with data values in a single machine word. This requirement gives rise to a trade-off between the size of the tags and the size of the data values that may be stored. We believe that, for many applications on many machines, these sizes can be chosen so that the tags are sufficiently large that errors cannot occur, while leaving sufficient room for the values that are stored by the application. For example, on a 64-bit machine, reserving 48 bits for the tag means that an error can occur only if a variable is modified 2^{48} times during one LL-SC sequence. (Even if a variable is modified a million times a second, this would take about nine years.) This choice would leave 16 bits for data, which would be sufficient for many applications. Nonetheless, it is certainly conceivable that, for some applications on some machines, an acceptable trade-off between tag size and data size could not found. Indeed, some applications may need to store data values that exceed the size of one machine word. To address these problems, we present two more implementations of LL, VL, and SC. The first uses bounded and reasonably small tags, thereby allowing more room for data, while simultaneously eliminating the possibility (however remote) of an error. This implementation still requires data values to be stored in one machine word. To overcome this restriction, we finally present an implementation of LL, VL, and SC that allows W-word values to be stored for any W, with each word containing a tag and a part of the data value. Thus, W can be chosen to be sufficiently large to accommodate data of the desired size, and tags that are large enough that they will not wrap around in the lifetime of most computer systems.

Our LL, VL, and SC implementations require a new parameter to be added to the usual interface for invoking these instructions. A pointer to a private variable of the invoking process is passed to LL, and the value written by LL to that private variable is subsequently passed to the VL and SC operations. We believe that, in most applications, the provision of these parameters will put little extra burden on programmers. However, as explained later, we exploit this modification in order to provide significantly more efficient implementations of LL, VL, and SC.

The remainder of this paper is organized as follows. Section 2 contains definitions and notation. In Section 3, we present the results mentioned above, and in Section 5, we present concluding remarks and recommendations. Due to space limitations, detailed proofs are deferred to the full version of the paper.

2 Preliminaries

Our programming notation is largely self-explanatory. We use C-like notation for dereferencing pointers, and all defined types are assumed to fit into one machine word except *vartype* in Figure 6. We use $\oplus$ and $\ominus$ to denote addition and subtraction modulo the range of the variable accessed.

Atomic code fragments are given in Figure 2 that specify the "normal" semantics of CAS and LL/VL/SC. RLL and RSC have semantics similar to those of LL and SC, but with the restrictions listed in Section 1.

In the full paper, we prove that each of our results yields a linearizable [9] implementation of the stated primitives. It is assumed that variables accessed by our implementations are not modified by other means. Each of our RLL/RSC-based implementations is wait-free provided only finitely many spurious failures occur during the execution of each operation. (As discussed in Section 5, our RLL/RSC-based implementations are structured so that repeated spurious failures are unlikely.) We measure the time complexity of these implementations as the worst case number of steps taken to complete after the last spurious failure.

We measure space overhead as the number of words that must be reserved in addition to the words to be accessed by our implementations. (This does not reflect the fact that our one-word implementations reduce the size of the values that can be stored by using some of the space in the implemented words for tags. It also does not count space used for the additional variables required for invoking LL, VL, and SC. These variables are just one word per LL-SC sequence, and would ordinarily be stored on the execution stack of the invoking process. Therefore they do not require additional space to be reserved.) We say that LL/VL/SC implementations that require data values to fit into one word with a tag implement *small* variables.

```
CAS(X, v, w) ≡ if X = v then X := w; return true
                          else return false fi

LL(X)        ≡ valid_X[p] := true; return X

VL(X)        ≡ return valid_X[p]

SC(X, v)     ≡ if valid_X[p] then
                   X := v;
                   for i := 0 to N − 1 do
                       valid_X[i] := false od;
                   return true
               else return false
               fi
```

Figure 2: Equivalent atomic code fragments specifying the "normal" semantics of CAS and LL/VL/SC. Fragments for LL, VL, and SC are for process p. $valid_X$ is a shared array of booleans associated with variable X. i is a private variable of process p. N is the total number of processes. The semantics of VL and SC are undefined if process p has not executed a LL since p's mosr recent SC.

```
type wordtype = record tag: tagtype; val: valtype end

procedure CAS(addr: pointer to wordtype;
                        old, new:valtype) returns boolean

    private register oldword, newword: wordtype

1:  oldword := *addr;
2:  if oldword.val ≠ old then return false fi;
3:  if old = new then return true fi;
4:  newword := (oldword.tag ⊕ 1, new);
    while true do
5:      if RLL(addr) ≠ oldword then return false fi;
6:      if RSC(addr,newword) then return true fi
    od
```

Figure 3: Constant-time, low-overhead, unbounded-tag implementation of CAS using RLL and RSC.

3 Unbounded Tag Algorithms

3.1 Implementing CAS using RLL/RSC

We begin this section by presenting our implementation of CAS from RLL and RSC. This implementation, which is shown in Figure 3, is guaranteed to terminate provided only finitely many spurious failures occur during each invocation of CAS.

Our CAS implementation accepts a pointer *addr* to the word being accessed, and old and new values for the word. Each word to be accessed by the CAS operation contains a tag and a data value. The tag is used to detect changes to the data field.

A CAS operation by process p begins by reading the current value of the word being accessed (line 1). At line 2, the CAS returns false if the value read at line 1 does not match the old value. Next, if the old and new values are equal, then the CAS returns true (line 3). In either case, it is easy to see that the CAS operation is correctly linearized to the point at which the word pointed to by *addr* is read (line 1).

If the old value matches the current value of the accessed word and the new value differs from the old (i.e. CAS did not return from line 2 or line 3), then p prepares to change the accessed word to contain the new value, along with a new tag (line 4). Then, in the loop at lines 5 and 6, p repeatedly tries to change the accessed word from the old tag and value to the new. At line 5, p performs a RLL on *addr*, and compares the value read to the value read at line 1. If these values differ, then p returns false. In this case, some successful RSC was performed on the accessed word during p's CAS operation. Thus, because the old and new values of any CAS operation that reaches line 6 are different (see line 3), the value of the accessed word differs from p's old value immediately after the first such RSC to occur after p executes line 1. Thus, p's CAS is correctly linearized as failing at that point. Otherwise, p attempts so write the new tag and value using RSC (line 6). If it is successful, then p's CAS returns true. Observe that, because p's RSC succeeds, no process modifies the accessed word between p's RLL and p's RSC. Thus, the value of the accessed word equals p's old value immediately before the RSC (as the tests at lines 2 and 5 both failed), and it equals p's new value immediately after (see line 4). Therefore, p's CAS is correctly linearized to the point at which p executes the successful RSC in this case.

We have seen that each CAS operation that terminates is correctly linearized. It remains to address wait-freedom. It is easy to see that a CAS operation fails to terminate only if it repeatedly reads the old tag and value at line 5, and the RSC at line 6 repeatedly fails. Assume towards a contradiction that a CAS operation by process p takes infinitely many steps without terminating. Because the RSC does not fail spuriously infinitely often during one CAS operation, this implies that the RSC fails infinitely often due to successful RSC's and that p infinitely often reads the same tag value at line 5. Because each successful RSC increments the tag, this implies that the tag value wraps around between each pair of consecutive executions of line 5. As discussed previously, provided the tag field is large enough, this would never happen in any practical system. It is also easy to see that a CAS operation terminates in constant time after the last spurious failure, provided the tag does not wrap around between consecutive executions of line 5.

This implementation yields the following result.

Theorem 1: RLL and RSC can be used to implement a CAS operation for small variables that is wait-free provided there are not infinitely many spurious failures during one CAS operation; that terminates in constant time after the last spurious failure; and that has no space overhead. □

3.2 Implementing LL/VL/SC

We now turn our attention to the implementation of LL, VL, and SC operations using CAS shown in Figure 4. As described in Section 1, these operations differ slightly from the standard ones in that they require the programmer to pass a pointer to a private word to LL, and to pass the value of that word (written by the LL implementation) to the subsequent VL and SC operations. These words are used by LL to store information that is later used by VL and SC. This obviates the need to search for information associated with the variable being accessed, thereby avoiding a fundamental space-time tradeoff that would render the implementation impractical. We now describe this implementation in more detail.

The LL operation copies (line 1) the word to be accessed (pointed to by the *addr* parameter) to the private word supplied (pointed to by the *keep* parameter), and returns the value read (line 2). The value stored in the private word is used later to detect changes to the accessed word. The LL operation is linearized to the point at which the read in line 1 is executed.

A VL operation by process p rereads the accessed word, and returns true if it still contains the same value as was read by the previous LL operation (which is pointed to by *keep*), and false otherwise. Because *keep* is a private variable of p, the two reads in line 3 can be considered as being executed atomically at the point at which p dereferences *addr*. Thus, the VL operation can be linearized to that point.

Finally, a SC operation by process p attempts to change the accessed word from the previous value (stored in *keep*) to the new value and a new tag (line 4). The new tag is determined by incrementing the previous tag. The SC operation returns true if the CAS is successful, and false otherwise. If the CAS is unsuccessful, then the value of the accessed word has changed since p's previous LL on this word was linearized, which implies that a successful CAS (and therefore a successful SC) was executed on this word. Thus, p's SC fails correctly, and can be linearized to the point at which the CAS fails. Otherwise, if the CAS succeeds, then no successful SC has been executed since p's previous LL (assuming the tags to not wrap around in that time), so p's SC succeeds correctly, and can be linearized to the point at which the CAS succeeds.

```
type wordtype = record tag: tagtype; val: valtype end

procedure LL(addr,keep: pointer to wordtype)
                                        returns valtype
1:  *keep := *addr;
2:  return keep→val

procedure VL(addr: pointer to wordtype; keep: wordtype)
                                        returns boolean
3:  return keep = *addr

procedure SC(addr: pointer to wordtype; keep: wordtype;
                    new:valtype) returns boolean
4:  return CAS(addr, keep, (keep.tag ⊕ 1, new))
```

Figure 4: Constant-time, low-overhead, unbounded-tag implementation of LL/VL/SC using CAS.

This implementation requires no additional space overhead, and allows processes to execute concurrent LL-SC sequences. Thus, we have the following result.

Theorem 2: CAS can be used to implement constant-time LL, VL, and SC operations for small variables with no space overhead. □

Theorems 1 and 2 together imply Theorem 3 below. However, directly combining the implementations of Figures 3 and 4 to achieve this result has the disadvantage that words to be accessed by LL, VL, and SC must contain two tags: one for the CAS implementation and one for the LL, VL, and SC implementation. Because both tags and the current value must fit into one machine word, this substantially reduces the time needed for the tags to wrap around. A direct implementation that uses only one tag is shown in Figure 5. This implementation is based on the same techniques as those used in Figures 3 and 4. For simplicity of presentation, the remaining implementations presented in this paper are based on CAS. In each case, the technique in Figure 3 can be used to acquire the same result using RLL and RSC.

Theorem 3: RLL and RSC can be used with no space overhead to implement for small variables constant-time LL and VL operations, and a SC operation that is wait-free provided only finitely many spurious failures occur during one invocation of SC, and that terminates in constant time after the last spurious failure. □

3.3 LL/VL/SC on Large Variables

The implementations presented so far require data values to reside within one machine word together with tags. For many applications, this is acceptable because a relatively small range of data values must be stored (for example array indexes). For other applications, this might be unacceptable because pointers or other large

```
type wordtype = record tag: tagtype; val: valtype end

procedure LL(addr,keep: pointer to wordtype)
                                        returns valtype
    *keep := *addr;
    return keep→val

procedure VL(addr: pointer to wordtype; keep: wordtype)
                                        returns boolean
    return keep = *addr

procedure SC(addr: pointer to wordtype; keep: wordtype;
                    newval:valtype) returns boolean

    private register oldword, newword: wordtype

    oldword := keep;
    newword := (keep.tag ⊕ 1, newval);
    while true do
        if RLL(addr) ≠ oldword then return false fi;
        if RSC(addr,newword) then return true fi
    od
```

Figure 5: Constant-time, low-overhead, unbounded tag implementation of LL/VL/SC using realistic LL/SC.

data items must be stored.

In this subsection, we present an implementation that seeks to overcome this disadvantage. This implementation allows data values to be spread over multiple words. This has the dual advantages of allowing larger tags to be used (thereby further reducing the likelihood of a tag value wrapping around during one LL-SC sequence), and allowing data values of arbitrary size to be stored. While these implementations do require additional space overhead, the size of this overhead is determined by the worst-case number of concurrent LL-SC sequences, the number of processes, and the size of the implemented words — not by the *number* of words implemented.

Our implementation for large variables is presented in Figure 6. This implementation provides a weaker form of LL, which we call WLL. In the event that a subsequent SC is sure to fail, WLL may simply return the identifier of some process that performed a successful SC during the execution of the WLL operation. In this case, WLL is not required to return a value of the implemented variable. WLL was first introduced by Anderson and Moir in [3][2] and is sufficient for most applications that use LL/SC instructions because the

[2] The implementation of WLL, VL, and SC presented in [3] implements only one W-word variable. A naive generalization of this implementation for T W-word variables requires $\Theta(NWT)$ space overhead; the implementation given here requires only $\Theta(NW)$ space overhead, regardless of the number of words implemented. Also, the implementation in [2] is based on stronger LL and SC instructions than those used here.

```
type segmenttype = record tag: tagtype; val: valtype end;
     headertype = record tag: tagtype; pid: 0..N − 1 end;
     vartype = record hdr: headertype;
                      data: array[0..W − 1] of segmenttype
               end;

shared variable A: array[0..N − 1][0..W − 1] of valtype;

proc Copy(addr: pointer to vartype; hdr: headertype;
          save: pointer to array[0..W − 1] of valtype)
                                   returns {succ, 0..N − 1}

   local variable i: 0..W − 1; h: headertype;
                  y, z: segmenttype;

   for i := 0 to W − 1 do
      y := addr→data[i];
      if y.tag = hdr.tag ⊖ 1 then
         z := (hdr.tag, A[hdr.pid][i]);
         CAS(&(addr→data[i]), y, z);
         y := z
      fi;
      h := addr→hdr; if h ≠ hdr then return h.pid fi;
      if save ≠ null then save[i] := y.val fi
   od;
   return succ

proc WLL(addr: pointer to vartype;
         keep: pointer to tagtype;
         retval: pointer to array[0..W − 1] of valtype)
                                   returns {succ, 0..N − 1}

   local variable x: headertype

   x := addr→hdr;
   *keep := x.tag;
   return Copy(addr, x, retval)

proc VL(addr: pointer to vartype;
                      keep: tagtype) returns boolean

   return (addr→hdr).tag = *keep

proc SC(addr: pointer to vartype; keep: tagtype;
        newval: pointer to array[0..W − 1] of valtype)
                                         returns boolean

   local variable oldhdr, newhdr: headertype; i: 0..W − 1

   oldhdr := addr→hdr;
   if oldhdr.tag ≠ *keep then return false fi;
   for i := 0 to W − 1 do
      A[p][i] := newval[i]
   od;
   newhdr := (oldhdr.tag ⊕ 1, p);
   if ¬CAS(addr, oldhdr, newhdr) then return false fi;
   Copy(addr,newhdr,null);
   return true
```

Figure 6: $\Theta(W)$-time, low-overhead, unbounded-tag implementation of W-segment WLL/VL/SC using CAS.

computation between a LL and a subsequent failing SC is usually wasted and must be repeated. By checking the return value from WLL, this wasted computation may be avoided in the event that the subsequent SC is certain to fail anyway.

In Figure 6, each variable consists of a header word, which contains a tag and a process identifier, and W segments (words), each of which contains a tag and a data value. A SC by process p writes a new value to a variable by changing its header word to contain a new tag and p's identifier, and by then writing each of the data segments for that variable. The value written to each segment is the new tag, together with part of the new data value being stored.

Because a process may fail or be delayed after changing the header word for a variable and before writing all of the segments, processes must be able to "help" each other to complete a successful SC in order to obtain consistent values to return from WLL. To this end, each value to be stored by a SC of process p is first stored by p in a shared array $A[p]$. This allows other processes to determine the values to be written to the segments of the accessed variable. The *Copy* procedure (lines 1 to 9) is used both to write the new values in a SC operation and to help another process to do so if necessary. *Copy* takes as parameters a pointer to the variable being accessed, a new header for that variable, and a pointer to an array in which to collect the values copied or read. The purpose of *Copy* is to ensure that all of the values of the SC associated with the new header have been copied to the segments of the implemented word, and, provided the header of the variable does not change during the copying, to save the values copied so that they can be returned by the WLL procedure. We now describe the *Copy* procedure in more detail.

To read and/or copy the ith word, the *Copy* procedure reads the current value of that word (line 2), and checks to see if a new value should be copied by comparing tags (line 3). If so, *Copy* prepares a new value for that word consisting of the new tag and the ith word of the value stored in A by the process that invoked the SC (line 4). *Copy* then uses CAS to change the ith word from the old value to the new. If some other process has already changed this word, then the CAS will simply fail. In either case, the new value has been copied, so the purpose of *Copy* is achieved. At line 6, *Copy* records that the value has been changed so that the correct value will later be saved (see line 8). Having ensured that the new value has been copied for the ith word, *Copy* checks to see whether a new SC has been performed since the one that wrote the current header (line 7). If so, then there is no need to continue copying the old values, so *Copy* returns the process identifier of a process that has done a successful SC, indicating that it did not save a consistent value. Otherwise, if an array has been provided for saving the values copied (i.e., *Copy* was called from *WLL*), the value for the ith value is saved (line 8) before proceeding to the next word. If all values are successfully copied without another SC intervening, then *Copy* returns *succ* (line 9), indicating that all values were successfully copied, and that a consistent value was saved.

Having described the *Copy* procedure, we now explain how WLL, VL, and SC operations for W-word variables are implemented. The WLL procedure reads the current header of the word being accessed (line 10), records the tag it read in the *keep* variable provided (line 11), and then calls *Copy* (line 12) to ensure that all of the values associated with the header read have been copied, and to obtain those values in an array specified by the calling process (*retval*). If *Copy* returns *succ*, then WLL returns *succ*, indicating that a consistent value was saved in the array provided. Otherwise, WLL returns a process identifier, indicating that a subsequent SC is certain to fail. The VL procedure simply checks to see if the the tag recorded by the previous WLL still matches the tag stored in the header word of the accessed variable. This determines whether a successful SC has been performed on this variable since the LL read the header.

The SC operation by process p first checks to see if a successful SC has been executed since p's previous LL (lines 14 and 15). If so, it fails immediately. Otherwise, it proceeds to "announce" the value it will attempt to store by copying it to $A[p]$ (lines 16 and 17). Then, SC prepares a new header consisting of the previous tag plus one and p's process identifier (line 18), and attempts to change the old header to the new using CAS. If this attempt fails, then some other process has performed a successful CAS (and therefore SC) during p's execution of SC, so p returns false, indicating that the SC failed. Otherwise, if p does successfully change the header, then p calls *Copy* to ensure that all of the new values are copied to the segments of the accessed word (line 20), and returns true (line 21). The call to *Copy* is necessary because p may need to reuse $A[p]$ for a subsequent SC, so p cannot simply rely on subsequent WLL operations to do the copying.

In the full paper, we present a formal linearizability proof for this algorithm. In this proof, each WLL and VL operation is linearized to the point at which it reads the header of the accessed word (lines 10 and 13 respectively), and each SC operation is linearized to the point at which it performs the CAS (line 19). This proof establishes the following result.

Theorem 4: CAS can be used to implement WLL, VL, and SC operations for an unlimited number of W-word variables with time complexity $\Theta(W)$, $\Theta(1)$, and $\Theta(W)$, respectively, and $\Theta(NW)$ space overhead. □

4 Bounded Tag Algorithms

All of the implementations presented in the previous section use unbounded tags to detect changes to data that is accessed by the implemented operations. In reality, the tags are stored in bounded variables that wrap around to zero when they exceed their maximum values. This wraparound does not cause our implementations to behave incorrectly unless the tag of some word cycles through all possible values in the time that one process completes one LL-SC sequence. Provided these tags are large enough, this could never happen for most applications and systems. However, because these tags must reside in one shared word with the data value, a trade-off arises between the size of the tag and the size of the data value. In particular, these implementations limit the size of the data values stored to being smaller than one machine word. The first implementation uses reasonably-small, bounded tags, leaving more room for data within each word.

Our bounded-tag implementation of LL, VL, and SC is shown in Figure 7. This implementation uses a feedback mechanism similar to one used by Anderson and Moir in [2][3]. This mechanism allows processes to choose new tags from a bounded range so that there is no possibility of "prematurely" reusing a tag. This ensures that a CAS does not succeed when it should fail, thereby performing the role of the unbounded tags of the previous implementations. The bounded tag implementation has the advantage that the tags are relatively small — leaving more room in each word for data — and yet there is no possibility of an error due to tags wrapping around.

The implementation in Figure 7 assumes a known bound k on the number of concurrent LL-SC sequences of any process. The feedback mechanism is based on the $N * k$ array A. Process p uses $A[p]$ to "announce" the tags it has read. Because p might execute up to k concurrent LL-SC sequences, $A[p]$ contains k "slots". Each process maintains a private stack S to manage these slots. By reading A, processes can avoid reusing tags that have recently been read by other processes. Each process manages its tags by using a private queue Q. We now describe this implementation in more detail.

The LL procedure first selects a slot to use for the new LL-SC sequence by popping a slot from S (line 1). This slot is recorded for later use in the *keep* word provided. Then, LL reads the word being accessed (line 2), and announces the tag it has read (line 3). Next, LL rereads the word being accessed, and records in *keep* whether or not the word has changed. If it has changed, then the subsequent SC will fail (line 8). Otherwise, it is guaranteed that the tag read was announced before the second read of the tag. In the full paper, we show that this is important for ensuring that the tag is not reused before this LL-SC sequence is completed. Finally, LL returns the value read (line 5). The VL procedure returns false if the accessed word has changed since it was first read in the preceding LL, and true otherwise (line 6).

The implementation in Figure 7 also provides a CL operation. This allows the programmer to indicate that the current LL-SC has been aborted. This might be necessary in some applications because this implementation requires that only k concurrent LL-SC sequences are executed by each process. If an LL-SC sequence is aborted, then this must be accounted for by the implementation. The CL operation simply returns the slot being used by this LL-SC sequence to the pool of unused slots by pushing it back onto the stack S (line 7).

The SC procedure immediately returns false if the accessed word changed during the preceding LL (line 8). At line 9, SC reads a tag and moves that tag to the back of its tag queue Q. Because the queue contains $2Nk+1$ tags and only two tags are removed per LL-SC sequence, this ensures that this tag will not be reused in the next Nk LL-SC sequences. Line 10 increases j (modulo Nk) to ensure that, after every Nk SC operations by process p, p has read all elements of A. In the full paper, we show that, because the tags read at line 9 are moved to the back of the tag queue, this ensures that p never reuses a tag prematurely, thereby causing a CAS that should fail to succeed. In lines 11 and 12, SC selects a new tag from the front of its tag queue Q, and returns the slot used for this LL-SC sequence to the stack S. Then, SC chooses the bit to be written with the new value so that it differs from the previous bit. This allows the tag chosen to be the same as the last tag p successfully SC'd, which enables us to use a substantially simpler tag selection mechanism than the one used in [3], and to use fewer tags. Finally, SC attempts to change the implemented variable from the old value to the new using CAS. If this succeeds, then the SC succeeds (the tag selection mechanism guarantees that no successful SC has been performed since the previous LL in this case); otherwise it fails (a successful SC must have been performed).

Constant-time operations for the stack S are trivially implemented. By maintaining Q as a doubly-linked list, and by having a static index table with pointers to each tag, the operations on Q can also be implemented in constant time. Thus, we have the following result.

Theorem 5: CAS can be used to implement constant-time LL, VL, and SC operations that allow k concurrent LL-SC sequences on an unlimited number of small variables with $\Theta(N^2k)$ space overhead. □

[3]The implementation in [2] uses a stronger version of LL and SC which allow memory to be accessed between an LL and the subsequent SC. Also, using this implementation to provide LL, VL, and SC operations for T words would require $\Theta(N^2T)$ space overhead, while the implementation presented here has only $\Theta(N^2k)$ space overhead, where k is the maximum number of concurrent LL-SC sequences by any process.

```
type wordtype = record tag: 0..2Nk; bit: boolean;
                       pid: 0..N − 1; val: valtype end;
     keeptype = record slot: 0..k − 1; fail: boolean end

shared variable A: array[0..N − 1][0..k − 1] of wordtype;

private variable
   S: stack of 0..k − 1 initially {0, ···, k − 1};
   Q: queue of 0..2Nk initially {0, ···, 2Nk};
   j: 0..Nk − 1

procedure LL(addr: pointer to wordtype;
             keep: pointer to keeptype) returns valtype

   local variable old: wordtype

   keep→slot := pop(S);
   old := *addr;
   A[p][keep→slot] := old;
   keep→fail := (*addr ≠ old);
   return old.val

procedure VL(addr: pointer to wordtype;
             keep: pointer to keeptype) returns boolean
   return ¬keep→fail ∧ (*addr = A[p][keep→slot])

procedure CL(keep: pointer to keeptype);
   push(S, keep→slot)

procedure SC(addr: pointer to wordtype;
             keep: pointer to keeptype; newval: valtype)
                                        returns boolean

   local variable t, newtag: 0..2Nk; newbit: boolean

   if keep→fail then return false fi;
   t := A[j div k][j mod k].tag; delete(Q, t); enqueue(Q, t);
   j := j ⊕ 1;
   newtag := dequeue(Q); enqueue(Q, newtag);
   push(S, keep→slot);
   newbit := ¬A[p][keep→slot].bit;
   return CAS(addr, A[p][keep→slot],
                          (newtag, newbit, p, newval))
```

Figure 7: Constant-time, low-overhead, bounded tag implementation of LL/VL/SC using CAS.

5 Concluding Remarks

We have presented several time-optimal, space-efficient implementations of CAS and of LL, VL, and SC. Our implementations employ more realistic instructions and have substantially lower space overhead than similar previous results. (All of our implementations — with the possible exception of the one in Figure 7 — have practical space requirements for most systems and applications.) We have also proposed a slight modification to the LL, VL, and SC operations that admits substantially simpler and more efficient implementations. Our results allow several previously-inapplicable algorithms to be used in existing systems with minimal changes.

Our RLL/RSC-based implementations all terminate provided only finitely many spurious failures occur per operation. These implementations have a very small window between each RLL and the subsequent RSC, which makes spurious failures unlikely and, accordingly, repeated spurious failures extremely unlikely. Also, our first three implementations are disjoint access parallel [10]. Roughly, this means that memory contention is not introduced by these implementations. While our other two implementations are *not* disjoint access parallel, we believe that it is unlikely that they will introduce excessive contention because accesses to common variables are not concentrated in any one area.

These results have several implications for the design of future processors. First, Greenwald and Cheriton [5] dismiss software transactional memory (STM) [14] as an alternative for implementing multi-word synchronization primitives because it is inapplicable in existing systems, and conclude that double-word CAS should be provided in hardware. We have shown that STM *can* be implemented in existing systems; we therefore believe that software implementations still have the potential to obviate the need for more complicated hardware (although more algorithmic and experimental work is required to determine whether software-based approaches can be truly practical). Finally, Michael and Scott [11] have recently suggested that CAS, rather than LL/SC, should be implemented on distributed shared memory multiprocessors. Our results indicate to architects that the choice between CAS and LL/SC (in its various forms) will not greatly impact programmers or program complexity. This frees them to focus on issues of cost and performance (such as those discussed in [11]) in deciding which instructions to support.

Acknowledgements: We thank John Valois for his comments on an earlier draft of this paper.

References

[1] *Alpha Architecture Handbook*, Digital Equipment Corporation, 1992.

[2] J. Anderson and M. Moir, "Universal Constructions for Multi-Object Operations", *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, 1995, pp. 184-194.

[3] J. Anderson and M. Moir, "Universal Constructions for Large Objects", *Proceedings of the Ninth International Workshop on Distributed Algorithms*, 1995, pp. 168-182.

[4] 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.

[5] M. Greenwald and D. Cheriton, "The Synergy Between Non-Blocking Synchronization and Operating System Structure", *Proceedings of the Second Symposium on Operating System Design and Implementation*, 1996, pp. 123-136.

[6] M. Herlihy, "Wait-Free Synchronization", *ACM Transactions on Programming Languages and Systems*, 13(1), 1991, pp. 124-149.

[7] M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Objects", *ACM Transactions on Programming Languages and Systems*, 15(5), 1993, pp. 745-770.

[8] M. Herlihy and J. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures", *Proceedings of the 20th International Symposium in Computer Architecture*, 1993, pp. 289-300.

[9] M. Herlihy and J. Wing, "Linearizability: A Correctness Condition for Concurrent Objects", *ACM Transactions on Programming Languages and Systems*, 12(3), 1990, pp. 463-492.

[10] 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*, 1994, pp. 151-160.

[11] M. Michael and M. Scott, "Implementation of Atomic Primitives on Distributed Shared Memory Multiprocessors", *Proceedings of the 1st Annual Symposium on High Performance Computer Architecture*, 1995, pp. 221-231.

[12] *MIPS R4000 Microprocessor User's Manual*, MIPS Computer Systems, Inc., 1991.

[13] *PowerPC 601 RISC Microprocessor User's Manual*, Motorola, Inc., 1993.

[14] N. Shavit and D. Touitou, "Software Transactional Memory", *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, 1995, pp. 204-213.

[15] J. Valois, "Space Bounds for Transactional Synchronization", unpublished manuscript, January 1997.