Practical Lock-Free and Wait-Free LL/SC/VL Implementations Using 64-Bit CAS

Maged M. Michael

IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA

Abstract. The ideal semantics of the instructions LL/SC/VL (Load-Linked, Store-Conditional, Validate) are inherently immune to the ABA problem which is a fundamental problem that affects most lock-free algorithms. This paper presents practical lock-free and wait-free implementations of arbitrary-sized LL/SC/VL variables using 64-bit CAS (Compare-and-Swap). The implementations improve on Jayanti and Petrovic's 64-bit wait-free implementations by reducing the space overhead per variable to a small constant, and not requiring advance knowledge of the maximum number of participating threads, while maintaining minimal amortized expected time and work complexities.

1 Introduction

A shared object is *lock-free* [3] if whenever a thread executes some finite number of steps toward an operation on the object, some thread must have completed an operation on the object during the execution of these steps. A lock-free shared object is also *wait-free* [2] if progress is also guaranteed per operation. Unlike conventional lock-based objects, lock-free objects are immune to deadlock and livelock, regardless of thread speeds, scheduling policies, and arbitrary termination, in addition to performance advantages such as tolerance to preemption.

A subtle problem that affects the design of most lock-free algorithms is the ABA problem. If not prevented, it can cause the corruption of lock-free objects as well as unrelated objects that happen to reuse dynamic memory removed from these objects, and it can cause the program to crash or return incorrect results. The ABA problem was first reported in the documentation of the Compare-and-Swap (CAS) instruction and its use for implementing lock-free freelists on the IBM System 370 [5]. CAS takes three arguments: the address of a memory location, an expected value, and a new value. If the memory location is found to hold the expected value, the new value is written to it, atomically. A Boolean return value indicates whether the write occurred. If CAS returns true, it is said to succeed, otherwise, it is said to fail.

The ABA problem occurs when a thread reads some value A from a shared variable, and then other threads write to the variable some value B, and then A again. Later, when the original thread checks if the variable holds the value A, using read or CAS, the comparison succeeds, while the intention of the algorithm designer is for such a comparison to fail in this case, and to succeed only if the

variable has not been written after the initial read. However, the semantics of read and CAS prevent them from distinguishing the two cases.

The theoretical semantics of the instructions LL/SC/VL (Load-Linked, Store-Conditional, Validate) make them inherently immune to the ABA problem. LL takes one argument: the address of a memory location, and returns its contents. SC takes two arguments: the address of a memory location and a new value. If the location was not written since the current thread last read it using LL, the new value is written to the memory location, atomically. A Boolean return value indicates whether the write occurred. VL takes one argument: the address of a memory location, and returns a Boolean value that indicates whether the memory location was not written since the current thread last read it using LL. If SC or VL returns true, it is said to succeed, otherwise, it is said to fail.

For practical architectural reasons, none of the architectures that support LL/SC (PowerPC, MIPS, Alpha) support the ideal semantics, and hence offer little or no help with preventing the ABA problem, and for most lock-free algorithms LL/SC with restricted semantics are used just to simulate CAS.

Until recently, implementations of LL/SC/VL variables [5, 1, 11] required atomic operations on both the implemented variable and an additional tag field. As most 32-bit architectures support 64-bit—as well as 32-bit—atomic instructions, these mechanisms are feasible to varying degrees in 32-bit applications running on 32-bit as well as 64-bit architectures. However, most current 64-bit architectures do not support atomic instructions on more than 64-bit blocks, thus it is no longer possible to pack a large tag with pointer-sized values in 64-bit applications.

Jayanti and Petrovic [7] address this problem by presenting wait-free implementations of 64-bit LL/SC/VL using 64-bit CAS. However, these implementations require space overhead per LL/SC/VL variable that is proportional to N, where N is the maximum number of threads that may operate on the LL/SC/VL variable. The implementations also require the use of N-sized arrays, which are problematic to implement without advance knowledge of the value of N or a conservative estimate of it. These requirements limit the practicality of these implementations to special cases where the maximum number of LL/SC/VL variables and the maximum number of threads in the program that may operate on these variables are known in advance to be small.

In this paper, we present lock-free and wait-free implementations of arbitrary-sized LL/SC/VL variables using 64-bit CAS (preliminary version in [8]). The implementations require only constant space overhead per LL/SC/VL variable (one word for the lock-free implementation and four words for the wait-free implementation), and in the worst case linear space in the number of participating threads per participating thread. The implementations do not require advance knowledge of the maximum number of participating threads.

In the wait-free implementation, LL and VL take constant time, and SC takes constant amortized expected time. In the lock-free implementation, VL takes constant time and SC takes constant amortized expected time, and in the absence of interfering successful SC operations, LL takes constant time. In both

implementations—regardless of contention—concurrent LL, VL, and unsuccessful SC operations do not interfere with each other. Using the work performance measure, the amortized expected complexity of any set of LL/SC/VL operations using either of our implementations is the same (except for the amortized and expected qualifiers) as those assuming hypothetical hardware support for ideal LL/SC/VL.

The rest of this paper is organized as follows. In Section 2, we discuss the memory reclamation technique used to support the LL/SC/VL implementations and other related issues. In Section 3, we present the lock-free implementation, and in Section 4, we present the wait-free implementation. We discuss the complexity of the implementations in Section 5, and conclude with Section 6.

2 Preliminaries

Memory Reclamation The memory reclamation problem is the problem of allowing dynamic blocks removed from lock-free objects to be freed, while guaranteeing that threads operating on these objects never access the memory of free blocks. The term *free* here is used in a broad sense, including reusing the block, dividing, coalescing, or unmapping its memory. Solutions for the memory reclamation problem have the side effect of partially—but not completely—preventing the ABA problem. In this paper, we make use of this feature of the hazard pointer memory reclamation method [9].

Briefly, the hazard pointer method uses single-writer shared pointers called hazard pointers. When a thread sets one of its hazard pointer to the address of a block, it in effect announces to other threads that if they happen to remove that block after the setting of the hazard pointer, then they must not free it as long as the hazard pointer continues to point to it. So, after a thread removes a block—and before it can free the block—it scans the list of hazard pointers and checks if any of them points to the block. Only if no match is found then the block is determined to be safe to free.

In a preferred implementation [9] using amortization, only constant amortized expected time is needed for processing each removed block until it is determined to be safe to free. A thread scans the hazard pointers after accumulating $H+\Theta(H)$ removed blocks, where H is the number of hazard pointers in the program. Then, the thread reads the H hazard pointers and organizes the non-NULL values read from them in an efficient private search structure such as a hash table with constant expected lookup time. Then, for each of the blocks that it has accumulated it searches the hash table for matching values. As described in [9], the procedure takes $O(H)$ expected time, and is guaranteed to identify $\Theta(H)$ blocks as safe to free. We use the hazard pointer method because it is portable across operating systems and architectures, it is wait-free, and it uses only pointer-sized instructions. Also, threads can join the method and retire dynamically, and acquire and release hazard pointers dynamically. Neither the number of participating threads N nor the number of hazard pointers H needs to be known in advance. See [9] for more details.

Definitions A thread p is said to hold an *active reservation* for LL/SC/VL variable $\mathcal{O}$ at time t, if p has performed LL($\mathcal{O}$) at time $t_0 < t$ and it is possible for p to perform SC($\mathcal{O}$,v) or VL($\mathcal{O}$) at some time $t_1 \geq t$ without performing LL($\mathcal{O}$) during the interval $[t, t_1]$. We define K as the highest number of active reservations that p needs to hold concurrently. Typically, K is a small constant.

3 Lock-Free LL/SC/VL Implementation

In the lock-free implementation (Figure 1) the LL/SC/VL variable $\mathcal{O}$ is represented by a pointer X. The current value of $\mathcal{O}$ is always held in the dynamic block currently pointed to by X. Whenever $\mathcal{O}$ is written, a new block holding the new value replaces the old block pointed to by X.

The subscript i in some function and variable names is used to distinguish among the reservations that may be held concurrently by the same thread.

The basic idea of the implementation is for $\text{LL}_{p,i}(\mathcal{O})$ to read the value in the current block pointed to by X and use a hazard pointer to protect the block from being reused prematurely, i.e., while p is holding the reservation for $\mathcal{O}$. Subsequently, if $\text{SC}_{p,i}(\mathcal{O},v)$ or $\text{VL}_{p,i}(\mathcal{O})$ find X pointing to the same block, then it must be the case that $\mathcal{O}$ was not written since $\text{LL}_{p,i}(\mathcal{O})$ was performed.

$\underline{\text{LL}_{p,i}(\mathcal{O})}$: The implementation of $\text{LL}_{p,i}(\mathcal{O})$ proceeds as follows. In line 2, thread p reads a pointer value from X into a persistent private variable $exp_{p,i}$, with the intention of reading $\mathcal{O}$'s value from $*exp_{p,i}$ (as in line 5). However, p cannot just proceed to read $*exp_{p,i}$, as it is possible that after line 2, another thread has replaced $exp_{p,i}$ (by performing a successful SC on $\mathcal{O}$) and then freed it before p manages to read its contents.

So, in line 3, p sets the hazard pointer $\text{hp}_{p,i}$ to $exp_{p,i}$ in order to prevent the block $exp_{p,i}$ from being freed before line 5 if it happens to be replaced by another thread. However, it is possible that $exp_{p,i}$ has already bean removed before p set the hazard pointer in line 3. Therefore, p checks X again in line 4. If $\text{X} \neq exp_{p,i}$, then it is possible that $exp_{p,i}$ was removed before p set $\text{hp}_{p,i}$ in line 3, and hence $exp_{p,i}$ might have been—or will be—freed before line 5. It is not safe to proceed to line 5 in this case, so p starts over from line 2.

If in line 4, p finds $\text{X} = exp_{p,i}$, then it is safe for p to read $*exp_{p,i}$ in line 5. This is true whether or not X has changed between line 2 and line 4. What matters is that at line 4, $\text{X} = exp_{p,i}$ (and hence $exp_{p,i}$ is not removed or free at that point) and $\text{hp}_{p,i} = exp_{p,i}$ already from line 3. Therefore, according to the hazard pointer method, from that point (line 4), if $exp_{p,i}$ is removed by another thread, it will not be freed as long as $\text{hp}_{p,i}$ continues to point to it, which is true as long as the current reservation is alive (i.e., beyond line 5).

$\text{LL}_{p,i}(\mathcal{O})$ is linearized [4] at line 4. At that point $\text{X} = exp_{p,i}$ and accordingly $\mathcal{O} = *exp_{p,i}$. By the hazard pointer method, the value of $*exp_{p,i}$ remains unchanged between lines 4 and 5. Therefore, $\text{LL}_{p,i}(\mathcal{O})$ returns the value of $\mathcal{O}$ at the time of the execution of line 4. Also, as described later, subsequent SC and VL operations—for the same reservation initiated by $\text{LL}_{p,i}(\mathcal{O})$—will succeed if and only if $\mathcal{O}$ has not been written since line 4 of $\text{LL}_{p,i}(\mathcal{O})$.

Types
    blocktype = valuetype = arbitrary-sized value
Shared variables representing each LL/SC/VL variable $\mathcal{O}$
    X: pointer to blocktype initially (X = $b \neq$ NULL) $\wedge$ (*b = $\mathcal{O}$'s initial value)
Per-thread shared variables (for thread p)
    for $i \in \{0, \ldots, K\text{-}1\}$ $\mathrm{hp}_{p,i}$: hazard pointer
Per-thread persistent private variables (for thread p)
    for $i \in \{0, \ldots, K\text{-}1\}$ $exp_{p,i}$: pointer to blocktype

$\underline{\mathrm{LL}_{p,i}(\mathcal{O}) : \text{valuetype}}$
1: repeat
2: $\quad exp_{p,i} := \mathrm{X}$
3: $\quad \mathrm{hp}_{p,i} := exp_{p,i}$
4: until (X = $exp_{p,i}$)
5: return *$exp_{p,i}$

$\underline{\mathrm{VL}_{p,i}(\mathcal{O}) : \text{boolean}}$
1: return (X = $exp_{p,i}$)

$\underline{\mathrm{SC}_{p,i}(\mathcal{O},v) : \text{boolean}}$
1: $b :=$ GetSafeBlock()
2: *$b := v$
3: $ret :=$ CAS(X,$exp_{p,i}$,b)
4: if (ret)
5: $\quad$ RetireNode($exp_{p,i}$)
6: else
7: $\quad$ KeepSafeBlock(b)
8: return ret

Fig. 1. Lock-free implementation of LL/SC/VL using pointer-sized CAS.

As long as the reservation initiated by $\mathrm{LL}_{p,i}(\mathcal{O})$ is alive, p must keep $\mathrm{hp}_{p,i}$ and $exp_{p,i}$ unchanged. Once p reaches a point where it will not be possible to issue $\mathrm{SC}_{p,i}(\mathcal{O})$ or $\mathrm{VL}_{p,i}(\mathcal{O})$ corresponding to the current reservation, then p can simply reuse $\mathrm{hp}_{p,i}$ and $exp_{p,i}$.

$\mathrm{SC}_{p,i}(\mathcal{O},v)$: The implementation of $\mathrm{SC}_{p,i}(\mathcal{O},v)$ proceeds as follows. In lines 1 and 2, p allocates a safe block and sets it to the new value v. A block b is safe if there is no thread q with any reservation j with $\mathrm{hp}_{q,j} = b$. That is, if this SC succeeds, then the new pointer value of X—i.e., b—will be different from all the $exp_{q,j}$ pointer values associated with all live reservations at the time.

In line 3, the CAS succeeds if and only if X = $exp_{p,i}$. When SC is issued it is guaranteed that $\mathrm{hp}_{p,i} = exp_{p,i}$ and that they were unchanged since the linearization of $\mathrm{LL}_{p,i}(\mathcal{O})$ that initiated the reservation. By the hazard pointer method, if the block $exp_{p,i}$ was replaced after $\mathrm{LL}_{p,i}(\mathcal{O})$, then no other thread could have subsequently allocated $exp_{p,i}$ as a safe block during the lifetime of the reservation. Therefore, the CAS succeeds if and only if, $\mathcal{O}$ was not written since $\mathrm{LL}_{p,i}(\mathcal{O})$. So, $\mathrm{SC}_{p,i}(\mathcal{O},v)$ is linearized at line 3.

If $\mathrm{SC}_{p,i}(\mathcal{O},v)$ succeeds (i.e., the CAS in line 3 succeeds), the old block removed from X (i.e., $exp_{p,i}$) cannot be reused immediately and its value should not be changed until it is determined to be safe to free by going through the hazard pointer method by calling RetireNode (defined in [9]). If $\mathrm{SC}_{p,i}(\mathcal{O},v)$ fails, then the block b can be reused immediately safely, as it was already safe in line 1 and it has not been observed by other threads since then.

The functions GetSafeBlock and KeepSafeBlock can be replaced with malloc and free, respectively, which can be implemented in user-level in an efficient completely lock-free manner [10]. Furthermore, our LL/SC/VL implementations have the feature that as long as N (the number of participating threads) is stable (which is an implicit requirement in Jayanti and Petrovic's implementations [7]), the number of blocks needed per thread remains stable, and so these functions can be completely private and without calling malloc and free. Each participating thread maintains two persistent private lists of *safe* and *not-safe-yet* blocks with combined maximum size equal to the batch size of the hazard pointer method. GetSafeBlock pops a block from the *safe* list. KeepSafeBlock pushes a block into that list. When the thread calls RetireNode and processes the blocks in the *not-safe-yet* list, it moves the blocks identified to be safe to the *safe* list.

$\underline{\text{VL}_{p,i}(\mathcal{O})}$: The implementation of $\text{VL}_{p,i}(\mathcal{O},v)$ simply checks if X is equal to $exp_{p,i}$. As argued regarding $\text{SC}_{p,i}(\mathcal{O},v)$, this is true if and only if $\mathcal{O}$ has not been written since the linearization of $\text{LL}_{p,i}(\mathcal{O})$. $\text{VL}_{p,i}(\mathcal{O})$ is linearized at line 1.

4 Wait-Free LL/SC/VL Implementation

In the LL/SC/VL implementation in Section 3, $\text{LL}_{p,i}(\mathcal{O})$ is not wait-free because thread p is always trying to capture the value of $\mathcal{O}$ from a specific block $exp_{p,i}$. However, by the time p reads the pointer value to $exp_{p,i}$ from X and is about to protect it using the hazard pointer $\text{hp}_{p,i}$, the block $exp_{p,i}$ might have already been removed and possibly freed, and p is forced to start over.

Unlike hazard pointers which prevent specific blocks from being freed, we introduce the notion of a *trap* which can capture *some block* (not a specific one) that satisfies certain criteria. In $\text{LL}_{p,i}(\mathcal{O})$, we use a trap to guarantee that in a constant number of steps, *some block* holding some value of $\mathcal{O}$ will be guaranteed not to be freed until p reads a value of $\mathcal{O}$ from it.

To be linearizable, LL needs to return a value that was held by $\mathcal{O}$ between LL's invocation and its response [4]. Therefore, a trap must avoid capturing a block that holds an old value of $\mathcal{O}$ that was overwritten before LL's invocation. For that purpose, we maintain a sequence number for each LL/SC/VL variable. The sequence number is incremented after every successful SC. When a thread sets a trap for a variable, it specifies the minimum acceptable sequence number.

4.1 Trap Functionality

Our wait-free LL/SC/VL implementation uses the following interface with the trap mechanism: $\text{SetTrap}_p(\mathcal{O},seq)$, $\text{ReleaseTrap}_p()$, $\text{GetCapturedBlock}_p()$, and $\text{ScanTraps}_p(b)$. Between every two calls to SetTrap_p there must be a call to ReleaseTrap_p. Thread p's trap is said to be active after a call to SetTrap_p and before the corresponding call to ReleaseTrap_p. $\text{GetCapturedBlock}_p$ is called only when p's trap is active. A block b is passed as argument of ScanTraps only after b has been removed from a LL/SC/VL variable. Blocks passed to ScanTraps are only freed (i.e., determined to be safe) by ScanTraps. The trap mechanism offers the following guarantees:

1. If thread p's call to GetCapturedBlock$_p$(), between calls to SetTrap$_p$($\mathcal{O}$,seq) and the corresponding ReleaseTrap$_p$(), returns b then either b = NULL or b holds the n^{th} value of $\mathcal{O}$, where $n \geq seq$.
2. If thread p calls SetTrap$_p$($\mathcal{O}$,seq) at time t, and block b is removed from $\mathcal{O}$ (before or after t), and b holds the n^{th} value of $\mathcal{O}$, and $n \geq seq$, and b is passed to ScanTraps$_q$ by some thread q after t, then b will not be freed before p calls ReleaseTrap$_p$(), and/or p's trap captures a block b' different from b.

First we present the wait-free LL/SC/VL functions assuming this trap functionality, then we describe the trap implementation in detail.

4.2 LL/SC/VL Functions

Figure 2 shows the structures and functions of the wait-free LL/SC/VL implementation. We start with the sequence number. An additional variable Seq is used per LL/SC/VL variable $\mathcal{O}$. Between every two consecutive successful SC operations on $\mathcal{O}$, Seq must be incremented exactly once. Therefore, if Seq = n, then there must have been either n or n+1 successful SC operations performed on $\mathcal{O}$ so far.

Two additional fields are added to the block structure. The field Var points to the LL/SC/VL variable, and the field Seq holds a sequence number. If the n^{th} successful SC on $\mathcal{O}$ sets X to b, then it must be the case that at that point b→Seq = n and b→Var = $\mathcal{O}$. The purpose of these two fields is to enable a trap to capture only a block that holds a value for a specific LL/SC/VL variable with a sequence number higher than some value.

LL$_{p,i}$($\mathcal{O}$): Lines 1–4 of LL$_{p,i}$($\mathcal{O}$) are similar to the lock-free implementation in Figure 1. In the absence of intervening successful SC operations by other threads between p's execution of line 1 and line 3, LL$_{p,i}$($\mathcal{O}$) returns at line 4. In such a case LL$_{p,i}$($\mathcal{O}$) is linearized at line 3. If intervening successful SC operations are detected in line 3, p sets a trap before it tries to read X again, to ensure completion in constant time even if more successful SC operations occur.

In line 5, p reads Seq into a local variable seq. At that point, there must have been so far either seq or seq+1 successful SC operations performed on $\mathcal{O}$. In line 6, p sets a trap for $\mathcal{O}$ with a sequence number seq. The trap guarantees that from that point until the release of the trap, either the trap has captured a block that contains a value of $\mathcal{O}$ with a sequence number greater than or equal to seq, or no block that contains a value of $\mathcal{O}$ with a sequence number greater than or equal to seq has been freed. Then, p proceeds to read X again into $exp_{p,i}$ in line 7, and in line 8 it sets hp$_{p,i}$ to $exp_{p,i}$. At this point, it must be the case that $exp_{p,i}$→Seq ≥ seq.

If the trap set in line 6 does not capture a block before the setting of the hazard pointer in line 8, then $exp_{p,i}$ could not have been freed after line 7 and will not be freed as long as hp$_{p,i}$ continues to point to it. Therefore, if p finds in line 9 that no block has been captured yet by the trap, then it must be safe to read $exp_{p,i}$→Value in line 11. In such a case, LL$_{p,i}$($\mathcal{O}$) is linearized at line 7.

```
Types
    valuetype = arbitrary-sized value
    seqnumtype = 64-bit unsigned integer
    blocktype = record Value: valuetype; Seq: seqnumtype;
                       Var: pointer to LL/SC/VL variable end
Shared variables representing each LL/SC/VL variable O
    X: pointer to blocktype
    Seq: seqnumtype
        initially (X = b ≠ NULL) ∧ (b→Value = O's initial value) ∧
                  (b→Seq = Seq = 0) ∧ (b→Var = O)
Per-thread shared variables (for thread p)
    for i ∈ {0,...,K-1} hp_{p,i}: hazard pointer
Per-thread persistent private variables (for thread p)
    for i ∈ {0,...,K-1} exp_{p,i}: pointer to blocktype

    LL_{p,i}(O) : valuetype
1:  exp_{p,i} := X
2:  hp_{p,i} := exp_{p,i}
3:  if (X = exp_{p,i})
4:      return exp_{p,i}→Value
5:  seq := Seq
6:  SetTrap_p(O,seq)
7:  exp_{p,i} := X
8:  hp_{p,i} := exp_{p,i}
9:  b := GetCapturedBlock_p()
10: if (b = NULL)
11:     v := exp_{p,i}→Value
12: else
13:     v := b→Value
14:     exp_{p,i} := NULL
15: ReleaseTrap_p()
16: return v

    SC_{p,i}(O,v) : boolean
1:  if (exp_{p,i} = NULL) return FALSE
2:  b := GetSafeBlock()
3:  b→Value := v
4:  b→Var := O
5:  seq := exp_{p,i}→Seq
6:  b→Seq := seq+1
7:  CAS(Seq,seq-1,seq)
8:  ret := CAS(X,exp_{p,i},b)
9:  if (ret)
10:     ScanTraps_p(exp_{p,i})
11: else
12:     KeepSafeBlock(b)
13: return ret

    VL_{p,i}(O) : boolean
1:  return (X = exp_{p,i})
```

Fig. 2. Wait-free implementation of LL/SC/VL using 64-bit CAS.

This is correct for the following reasons: First, at line 7, X = $exp_{p,i}$ and $exp_{p,i}$→Value remains unchanged after line 7 (throughout the life of the reservation). Therefore, $\mathrm{LL}_{p,i}(\mathcal{O})$ returns the value of $\mathcal{O}$ at the time of p's execution of line 7. Second, by the guarantee of the memory reclamation method, $exp_{p,i}$ will not be freed after line 7 until the end of the reservation. Therefore, subsequent calls to SC and VL for the same reservation are guaranteed to succeed (by finding X = $exp_{p,i}$) iff $\mathcal{O}$ has not been written since p's execution of line 7.

If p finds in line 9 that the trap set in line 6 has indeed captured some block b (may or may not be the same as $exp_{p,i}$), then it might be possible that $exp_{p,i}$ has already been removed before setting the hazard pointer in line 8. Therefore,

it may not be safe to access $exp_{p,i}$. However, as long as the trap is not released, it is safe to access b. So, p proceeds to read $b\rightarrow$Value in line 13.

In this case (i.e., a block b was captured before line 9), $\mathrm{LL}_{p,i}(\mathcal{O})$ is linearized just before the SC that removed b from X, which is guaranteed to have occurred between p's execution of line 1 and line 9. If $b\rightarrow\mathrm{Seq} > seq$ then b must have been removed from X after p's execution of line 5. If $b\rightarrow\mathrm{Seq} = seq$ then b must have been removed from X after p's execution of line 1.

In line 14 $exp_{p,i}$ is set to NULL in order to guarantee the failure of subsequent $\mathrm{SC}_{p,i}(\mathcal{O},v)$ and $\mathrm{VL}_{p,i}(\mathcal{O})$ operations for the same reservation, as they should. At this point we are already certain that $\mathcal{O}$ has been written after the linearization point of $\mathrm{LL}_{p,i}(\mathcal{O})$, as only removed blocks get trapped.

After reading a valid value either in line 11 or line 13, it is safe to release the trap in line 15. Therefore, each participating thread needs only one trap, even if it may hold multiple reservations concurrently.

$\underline{\mathrm{SC}_{p,i}(\mathcal{O},v)}$: In line 1, p checks if $\mathrm{LL}_{p,i}(\mathcal{O})$ has already determined that this SC will certainly fail (by setting $exp_{p,i}$ to NULL). If so, $\mathrm{SC}_{p,i}(\mathcal{O},v)$ returns FALSE and is linearized immediately after its invocation.

Otherwise, $\mathrm{LL}_{p,i}(\mathcal{O})$ must have read its return value from $exp_{p,i}\rightarrow$Value and $exp_{p,i}$ has been continuously protected by $\mathrm{hp}_{p,i}$. Therefore, the same arguments for the lock-free implementation apply here too, and $\mathrm{SC}_{p,i}(\mathcal{O},v)$ is linearized at the time of applying CAS to X (line 8). However, care must be taken to maintain the invariant that when the $n+1^{th}$ successful SC on $\mathcal{O}$ is linearized, Seq $= n$.

In lines 2 and 3, as in the SC implementation in Section 3, p allocates a safe block b and sets its Value field to the new value v. Also, p sets $b\rightarrow$Var to $\mathcal{O}$ in line 4 to allow the trap implementation to associate b with $\mathcal{O}$.

In lines 5 and 6, p reads the value seq from $exp_{p,i}\rightarrow$Seq and then sets $b\rightarrow$Seq to seq+1 for the following reason. The current SC can succeed only if it replaces $exp_{p,i}$ in X with b in line 8. Since $exp_{p,i}$ holds the seq^{th} value of $\mathcal{O}$, then if the current SC succeeds, b will be holding the $(seq{+}1)^{th}$ value of $\mathcal{O}$. So, p sets $b\rightarrow$Seq to seq+1 to allow the trap implementation to associate b with the $(seq{+}1)^{th}$ value of $\mathcal{O}$.

We now discuss the update of Seq. Seq is monotonically increasing and always by increments of 1. Correctness requires that between every two successful SC operations, Seq must be incremented exactly once. That is, if this SC succeeds in line 8, then Seq must be equal to seq at that point. By an induction argument, if this SC is to succeed then at line 7 either Seq $= seq$-1 or Seq $= seq$. So, whether the CAS in line 7 succeeds or fails, immediately after line 7 Seq $= seq$, if this SC is to succeed. An optional optimization is to test for Seq $= seq$-1 before issuing CAS in line 7.

We then argue that if Seq is incremented by another thread between lines 7 and 8, then this SC must fail. If Seq $= n > seq$ at line 8, then some other thread q must have performed CAS on Seq with a new value n, then q must have observed X $= exp_{q,j}$ with $exp_{q,j}\rightarrow$Seq $= n$, which implies that at least n SC operations on $\mathcal{O}$ have already succeeded before line 8, then this SC cannot be the $(seq{+}1)^{th}$ successful SC on $\mathcal{O}$, and so this SC must fail.

We now move to line 8. As in the lock-free LL/SC/VL implementation, SC succeeds if and only if the CAS on X succeeds. Line 8 is the linearization point of SC if $exp_{p,i} \neq$ NULL. If SC succeeds, p passes the removed block $exp_{p,i}$ to ScanTraps so that it will be freed but not prematurely. If SC fails, then block b remains safe and can be freed or kept for future use without going through traps or hazard pointers.

VL$_{p,i}(\mathcal{O})$: As in the lock-free implementation in Section 3, X = $exp_{p,i}$ if and only if $\mathcal{O}$ has not been written since the linearization of LL$_{p,i}(\mathcal{O})$.

Wrap-around: Implementing sequence numbers as 64-bit variables makes wrap-around impossible for all practical purposes. Even if 1,000,000 successful SC operations are performed on $\mathcal{O}$ every second, the 64-bit sequence number would still not wrap around after 584,000 years!

4.3 Trap Implementation

The trap implementation is shown in Figure 3. Each participating thread p owns a trap record trap$_p$ and an additional hazard pointer traphp$_p$.

SetTrap$_p(\mathcal{O},seq)$: This function starts by setting trap$_p$→Var to $\mathcal{O}$ and setting trap$_p$→Seq to seq in lines 1 and 2, in order to indicate that the trap is set for blocks that contain a value of $\mathcal{O}$ with a sequence number not less than seq.

The purpose of the field trap$_p$→Captured is to provide other threads with a location to offer a block that matches the criteria of this trap. When p sets a trap, it cannot just set trap$_p$→Captured to NULL, otherwise a slow thread trying to set trap$_p$→Captured in relation to an earlier trap might offer the wrong block for the current trap. We need to guarantee that only a thread that intends to offer a block that satisfies the criteria of the current trap succeeds in setting trap$_p$→Captured.

For that purpose, p uses a unique tag every time it sets a trap. Since arbitrary 64-bit numbers might match pointer values captured by the trap, we use only non-pointer values for the tag. By convention and as required by most current processor architectures, the addresses of dynamic blocks must be 8-byte aligned. So, tag numbers not divisible by 8 are guaranteed not to match any block addresses. By convention, NULL = 0. Thus, a 64-bit word can hold 7×2^{61} different tag values. The variable tag_p keeps track of the tag values already used by trap$_p$. Similar to the sequence number, even if trap$_p$ is used 1,000,000 times per second to perform LL operations where each LL is interfered with by a successful SC, and this continues for 511,000 years, tag_p would still not wrap around. So, wrap-around is not a practical concern in this case either.

In line 3, p sets trap$_p$→Captured to tag_p, and in line 4 it sets traphp$_p$ to the same tag value. We discuss traphp$_p$ in detail when discussing ScanTraps.

Setting the trap takes effect only when p sets trap$_p$→Active to TRUE in line 5. Finally, in line 6, p prepares a new tag value in tag_p for the next trap.

ReleaseTrap$_p()$: In this function, p simply sets trap$_p$→Active to FALSE and is ready to reuse the trap if needed. It is essential that trap$_p$→Active = FALSE whenever p updates the other fields of trap$_p$ or traphp$_p$. Lines 2 and 3 are optional optimizations.

```
Types
    tagtype = 64-bit unsigned integer
    traptype = record Active: boolean; Var: pointer to LL/SC/VL variable;
        Seq: seqnumtype; Captured: tagtype or pointer to blocktype end
Per-thread shared variables (for thread p)
    trap_p: traptype                initially trap_p→Active = FALSE
    traphp_p: hazard pointer
Per-thread persistent private variables (for thread p)
    tag_p: tagtype                  initially tag_p = non-pointer value (e.g. 1)
    list_p: list of block addresses

    SetTrap_p(O,seq)
1:  trap_p→Var := O
2:  trap_p→Seq := seq
3:  trap_p→Captured := tag_p
4:  traphp_p := tag_p
5:  trap_p→Active := TRUE
6:  tag_p := next non-pointer value

    ReleaseTrap_p()
1:  trap_p→Active := FALSE
2:  trap_p→Captured := NULL
3:  traphp_p := NULL

    ScanTraps_p(list_p)
1:  for all q
2:      if ¬trap_q→Active skip
3:      tag := trap_q→Captured
4:      if (tag is a pointer value) skip
5:      var := trap_q→Var
6:      seq := trap_q→Seq
7:      b := list_p.lookup(var,seq)
8:      if (b = NULL) skip
9:      if CAS(trap_q→Captured,tag,b)
10:         CAS(traphp_q,tag,b)
11: RetireNode(list_p)

    GetCapturedBlock_p() : pointer to blocktype
1:  b := trap_p→Captured
2:  if (b is a pointer value) return b else return NULL
```

Fig. 3. Trap structures and functions.

GetCapturedBlock$_p$(): In this function, p simply checks if any other thread has replaced the tag value it has put in trap$_p$→Captured in SetTrap$_p$ with a pointer value (block address). If so, GetCapturedBlock returns the address of the captured block, otherwise it returns NULL to indicate that no block has been captured yet by this trap.

ScanTraps$_p$($list_p$): Every successful SC$_{p,i}$ requires scanning the traps of the other thread for criteria matching the removed block $exp_{p,i}$ that held the value of $\mathcal{O}$ just before the success of SC. The ScanTraps function can be implemented in several ways. A simple implementation would be to scan the traps upon every successful SC, but this would take linear time per successful SC.

We show an amortized implementation that takes constant amortized expected time per successful SC. In this implementation, instead of calling ScanTraps for every successful SC, p accumulates replaced blocks in a persistent private list $list_p$. When $list_p$ contains enough removed blocks for amortization

($O(N)$ blocks or less if the trap structures are integrated with the hazard pointer method), p proceeds to scan the trap structures of others participating threads. First, p organizes the addresses of the blocks in $list_p$ in an efficient private search structure (hash table) that allows constant expected lookup time. If $list_p$ contains multiple blocks that hold values for the same LL/SC/VL variable, the lookup can just return the block with the highest sequence number.

Now we describe the scanning process. For each participating thread q, p starts by checking $\text{trap}_q \rightarrow$Active in line 2. If it is FALSE, then it is certain that q does not have an active trap that was set before any of the blocks in $list_p$ were removed. Therefore, p can skip this trap and move on to the next one if any.

If $\text{trap}_q \rightarrow$Active is TRUE, p proceeds to line 3 and reads $\text{trap}_q \rightarrow$Captured into the local variable tag. If tag is a pointer value, then it is either NULL or actually a block's address and not a tag. If it is NULL, then the trap must have already been released. If it is an actual block, then it must be the case that trap_q has already captured a block. Therefore p need not be concerned about providing a block to trap_q whether or not $list_p$ contains blocks that match the criteria of trap_q. So, whether trap_q has been released or has captured a block, p can skip this trap and move on to the next one if any.

If tag is a non-pointer value, then trap_q has not captured a block and so p needs to proceed to lines 5 and 6, to read $\text{trap}_q \rightarrow$Var and $\text{trap}_q \rightarrow$Seq. In line 7, p performs a lookup in $list_p$ (in constant expected time) for a block that matches the criteria of trap_q. If none are found, then p moves on to the next trap if any.

If p finds a block b in $list_p$ that matches the criteria of trap_q, then p tries to install b in $\text{trap}_q \rightarrow$Captured using CAS in line 9. If the CAS fails then it must be the case that either the trap has been released or some other thread has installed a block in $\text{trap}_q \rightarrow$Captured, and so p can move on to the next trap if any. If the CAS (in line 9) succeeds, then trap_q has captured b. In this case p needs to ensure that b is not freed before the release of trap_q. Therefore it tries to set traphp_q to b in line 10 using CAS.

If the CAS in line 10 fails, then q must have already released the trap. Therefore, neither b nor possibly the other blocks in $list_p$ that matched the criteria in trap_q need to be prevented from being free (at least as far as trap_q is concerned). So, p moves on to the next trap if any.

If the CAS in line 10 succeeds, then the hazard pointer method guarantees that b will not be freed as long as traphp_q remains unchanged, i.e., not before trap_q is released.

After scanning the trap records of all participating threads, p is guaranteed that all the blocks in $list_p$ can be passed safely to the hazard pointer method (through RetireNode) for ultimate determination of when they are safe to be freed. If a block b has been captured by one or more traps then it will not be freed as long as any of these traps has not been released. If a block b has not been captured by any traps, then either it did not match the criteria of any of the traps set before its removal, or it did match the criteria of one or more traps but in each case either the trap has been released or some other block has been captures by the trap. In any case, it is safe to pass b to RetireNode.

The trap structures can be maintained dynamically and in a wait-free manner similar to the hazard pointer structures in [9, Figure 4]. Threads can participate (i.e., acquire and release trap structures) dynamically and in a wait-free manner, and do not require advance knowledge of the maximum value of N. Furthermore, the trap structures can be integrated in the hazard pointer structures, and the blocks accumulated in $list_p$ can be counted together with the blocks accumulated for the sake of amortization in the hazard pointer method. For example, The blocks in $list_p$ can be counted with other removed blocks awaiting processing by the hazard pointer method (i.e., those in the *not-safe-yet* list mentioned in Section 3). When the combined count of these two lists reaches the (dynamically determined) batch size for the hazard pointer method, blocks in $list_p$ scan the trap structures, and then blocks in both lists scan the hazard pointers, all in a wait-free manner and with constant amortized expected time per freed block.

5 Complexity and Implementation Issues

Time and Work Complexities In the wait-free LL/SC/VL implementation LL, VL, and unsuccessful SC operations take constant time, and successful SC operations take constant amortized expected time. In the lock-free LL/SC/VL implementation, SC and VL are wait-free. Successful SC operations take constant amortized expected time, and VL and unsuccessful SC operations take constant time. LL takes constant time in the absence of intervening successful SC operations.

The conventional performance measure for lock-free algorithms is work, the total number of steps executed by N threads to perform some number r of operations. In our implementations LL, VL, and unsuccessful SC operations do not interfere with each other and all operations take at most constant amortized expected time in the absence of contention. So, the amortized expected work complexity of our implementations is the same (except for the amortized and expected qualifiers) as that of a hypothetical hardware implementation of ideal LL/SC/VL. For example, consider r LL operations, r LL/SC pairs, and as many LL/SC operations as needed to result in r successful SC operations. Assuming hardware support for ideal LL/SC/VL, the work is $O(r)$, $O(r)$, and $O(r.N)$, respectively. Using either of our LL/SC/VL implementations, the amortized expected work is $O(r)$, $O(r)$, and $O(r.N)$, respectively.

Space Complexity The worst-case space complexity of our LL/SC/VL implementations consists of two components: (1) space overhead per LL/SC/VL variable, and (2) space per participating thread.

Component (1): For both implementations the space overhead per variable is a small constant, one word (variable X) for the lock-free implementation and four words (variables X and Seq, and the Var and Seq fields of the current block) for the wait-free implementation. Component (2): For both implementations, the space per participating thread is $O(N.K)$.

Component (1) is obviously reasonable and constitutes a clear and significant improvement over the space complexity of Jayanti and Petrovic's implementations [7]. Whether the number of LL/SC/VL variables in a program is large or not is no longer a concern.

We now argue that in the vast majority of cases, component (2) is—or can be made—within acceptable practical limits. If the maximum number of threads (active at the same time) in the program is in the order of hundreds or less, then the space overhead is within acceptable limits for 64-bit applications. Otherwise, we argue that optimizations can be applied to the hazard pointer method and to the trap structures and functions in order to reduce the expected space overhead to acceptable levels.

If a program may have a large number of threads active concurrently, a two-track (or more) organization can be used for hazard pointer and trap structures, in order to keep the number of participating threads N acceptable. High priority threads that operate frequently on LL/SC/VL variables use the first track, while threads that operate infrequently on LL/SC/VL variable use the other tracks. Threads in the first track keep their hazard pointer and trap structures between operations on lock-free objects. The other threads participate in the LL/SC/VL mechanism and hazard pointer method dynamically (still in a wait-free manner as in [9, Figure 4]) and use the other track(s). These threads acquire hazard pointer and trap structures dynamically before operating on lock-free objects and then release them after they are done with the operation.

Let N_1 be the maximum number of threads using the first track concurrently. Let n_2 be the number of threads that are participating concurrently in the LL/SC/VL mechanism using the second track at some time t. Let N_2 be the maximum value of n_2. Then, N is at most $N_1 + N_2$. As threads that use the second track are expected to operate infrequently on LL/SC/VL variables, the value of N is expected to be much less than the actual total number of threads. The amortized expected time for the operations of the first track threads remains constant, while for threads using the second track it is $O(n_2)$ (to acquire hazard pointer and trap structures dynamically as in [9]). Again, as second track threads are not expected to operate frequently on LL/SC/VL variables, then n_2 is expected to be small.

Supporting Reads and Writes: By supporting arbitrary-sized LL/SC/VL variables, it is straightforward to extend our implementations to support LL/SC/VL variables that also support reads and writes, by using Jayanti's constructions of LL/SC/VL/Read/Write from LL/SC/VL [6].

Using Restricted LL/SC: On architectures which support restricted LL/SC rather than CAS, CAS(a,e,v) can be implemented as follows [12]:

do { if (LL(a) $\neq$ e) return FALSE } until SC(a,v); return TRUE

Supporting Persistent Reservations: Our implementations can be easily extended to support what we call *persistent reservations*, which may be needed for lock-free traversal. With persistent reservations, a successful SC does not end a reservation, but rather a thread can perform multiple successful SC operations

during the lifetime of the same reservation. This can be achieved without increase in the time, work, or space complexity of the implementations, by using a total of $K+1$ hazard pointers per thread and by swapping hazard pointer labels after successful SC operations, if reservation persistence is desired.

6 Conclusion

Ideal LL/SC/VL offer a complete solution for the ABA problem. In this paper, we presented practical lock-free and wait-free implementations of LL/SC/VL that improve on Jayanti and Petrovic's implementations [7] by limiting the space requirements to practically acceptable levels, and eliminating the need for advance knowledge of the number of threads that may operate on these variables, while still using only 64-bit CAS, and maintaining low time and work complexities and low latency.

References

1. J. H. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 184–193, 1995.
2. M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, Jan. 1991.
3. M. P. Herlihy. A methodology for implementing highly concurrent objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, Nov. 1993.
4. M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.
5. *IBM System/370 Extended Architecture, Principles of Operation*, 1983. Publication No. SA22-7085.
6. P. Jayanti. A complete and constant time wait-free implementation of CAS from LL/SC and vice versa. In *Proceedings of the Twelfth International Symposium on Distributed Computing*, pages 216–230, Sept. 1998.
7. P. Jayanti and S. Petrovic. Efficient and practical constructions of LL/SC variables. In *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Distributed Computing*, pages 285–294, July 2003.
8. M. M. Michael. ABA prevention using single-word instructions. Technical Report RC 23089, IBM T. J. Watson Research Center, Jan. 2004.
9. M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004. Earlier version in *21st PODC*, pages 21–30, July 2002.
10. M. M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, June 2004.
11. M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, Aug. 1997.
12. *PowerPC Microprocessor Family: The Programming Environment*, 1991.