A Complete and Constant Time Wait-Free Implementation of CAS from LL/SC and Vice Versa*

Prasad Jayanti

Dartmouth College, Hanover, NH 03755, USA

Abstract. We consider three popular types of shared memory that support one of the following sets of operations: {CAS, read, write}, {LL, SC, VL, read, write}, or {RLL, RSC, read, write}. We present algorithms that, together with Moir's [Moi97], efficiently implement each shared memory above from any of the other two. Our implementations are wait-free and have constant time and space complexity. Thus, concurrent programs developed for one of the above memories can be ported to any other without incurring any increase in time complexity. Further, since our implementations are wait-free, a wait-free concurrent program remains wait-free even after porting.

This work is similar in spirit to [IR94, AM95, Moi97]. The main difference is that in these earlier works the write operation is not included in the set of implemented operations. Specifically, earlier works implement {CAS, read} from {LL, SC} and vice versa, but to our knowledge there are no existing implementations of either of {CAS, read, write} and {LL, SC, VL, read, write} from the other. Consequently, it is not possible to port concurrent programs between systems supporting {CAS, read, write} and {LL, SC, VL, read, write}. The implementations in this paper help overcome this drawback.

At first glance, adding *write* to the set of implemented operations might appear easy. However, there is ample evidence to suggest that simple modifications to earlier implementations do not work. Our implementations are therefore quite different from the ones in earlier works.

1 Introduction

This paper concerns asynchronous shared memory multiprocess systems. In such systems, processes execute their programs at arbitrary rates and communicate with each other by applying operations on shared memory. In general, the set of operations supported by shared memory varies from one system to another. Consequently, porting multiprocess applications is a nontrivial problem. Specifically, consider a multiprocess application developed for systems whose shared memory supports a set S_1 of operations. Suppose that we wish to port this application to another system whose shared memory supports a different set S_2 of operations.

* This work is partially supported by NSF RIA grant CCR-9410421.

Clearly, this is possible if (and only if) each operation in S_1 can be simulated using the operations in S_2. Thus, portability of programs from the first system to the second requires that we implement shared memory supporting operations in S_1 from shared memory supporting operations in S_2.

Henceforth the term *S-memory* denotes shared memory that supports operations in set S. An *implementation of S_1-memory from S_2-memory* specifies a set of procedures, one for each operation in S_1. To execute an operation $op \in S_1$ on the implemented S_1-memory, a process simply executes the procedure specified by the implementation for *op*. The procedure simulates *op* using operations in S_2. The value returned by the procedure is considered to be the response for *op*. We require the implementation to satisfy two properties:

1. Linearizability: If operations on the underlying S_2-memory are atomic, then operations on the implemented S_1-memory also appear to be atomic. That is, each operation on the implemented S_1-memory appears to take effect instantaneously at some point between its invocation and completion [HW90].

2. Wait-freedom: When a process p calls and executes any implementation procedure, the procedure terminates after a finite number of p's steps, regardless of whether remaining processes are fast, slow, or have crashed [Lam77, Her91].

The second property (wait-freedom) implies that, unlike in lock-based implementations, slow or crashed processes cannot impede the progress of faster processes. Further, suppose P is a wait-free protocol designed for systems with S_1-memory. If P is ported to a system with S_2-memory (using an implementation of S_1-memory from S_2-memory), the second property ensures that P remains a wait-free protocol even after porting.

The *time complexity* of an implementation is the number of steps that a process executes, in the worst-case, in order to return from an implementation procedure. The *space complexity* of an implementation of S_1-memory from S_2-memory is the number of words of S_2-memory needed to implement one word of S_1-memory. In general, the time or the space complexity of an implementation depends on n, the maximum number of processes that can concurrently access the implementation. We say the time or space complexity is *constant* if it is independent of n. The implementations in this paper have constant time and space complexity.

Shared memory types considered: We consider three common types of shared memory, described in the following. Our goal is to efficiently implement each type of shared memory from any other.

1. {CAS, read, write}-memory (CAS is an abbreviation for Compare-And-Swap.)

Figure 1 describes the effect of these operations (by convention, the first argument of an operation is the process invoking the operation and the last argument is the object, *i.e.* the memory word, on which the operation is invoked). CAS(p_i, u, v, O) checks if O's value is u. If so, it changes O' value to v and returns *true*; otherwise it leaves O's value unchanged at u and returns *false*. CAS is supported by Pentium and SPARC architectures.

$\underline{\text{Read}(p_i, O)}$
return value(O)
 	
$\underline{\text{Write}(p_i, v, O)}$
value(O) $= v$
	$\underline{\text{CAS}(p_i, \textit{old}, \textit{new}, O)}$
	if value(O) $= \textit{old}$
	    value(O) $:= v$
	    return *true*
	else return *false*

Fig. 1. Specification of {CAS, Read, Write} object O

$\underline{\text{LL}(p_i, O)}$
Pset(O) $:=$ Pset(O) $\cup \{p_i\}$
return value(O)
 	
$\underline{\text{VL}(p_i, O)}$
return $p_i \in$ Pset(O)
	$\underline{\text{SC}(p_i, v, O)}$
	if $p_i \in$ Pset(O)
	    value(O) $:= v$
	    Pset(O) $:= \emptyset$
	    return *true*
	else return *false*
		$\underline{\text{read}(p_i, O)}$
		return value(O)
		
		$\underline{\text{write}(p_i, v, O)}$
		value(O) $:= v$
		Pset(O) $:= \emptyset$

Fig. 2. Specification of {LL, SC, VL, read, write} object O

2. {LL, SC, VL, read, write}-memory

In this type of shared memory, the state of a memory word O is described by two quantities, *value*(O) and *Pset*(O). For each operation, its response and its effect on the state of the memory word are specified in Figure 1. Informally, Pset(O) is the set of processes that have performed LL since the time O was last modified. A write operation modifies the object's value and causes Pset to become empty. Like a read operation, LL returns the value of the object; in addition, the process performing LL is added to Pset. An SC either acts like a write operation or has no effect on the state of the object. Specifically, when p_i performs SC(p_i, v, O), if p_i is in Pset(O), SC changes O's value to v, causes Pset(O) to become empty, and returns *true*; otherwise it returns *false* without affecting O's state (correspondingly, we say SC succeeds or fails). Thus, process p_i's SC on O succeeds if no process modified O (by performing a successful SC or write) since the most recent time p_i performed LL on O. VL by p_i returns true if p_i is in Pset, *i.e.*, if no process modified O since the most recent time p_i performed LL on O.

These operations, as defined above, are not supported by any real multiprocessor. However, many recent wait-free algorithms are designed for a shared memory that supports these operations. Further, these operations are very flexible and often facilitate simple and efficient solutions to process synchronization. Thus, it is important to be able to port programs that are based on this shared memory to real machines supporting other types of shared memory.

3. {RLL, RSC, read, write}-memory

RLL and RSC are weaker versions of LL and SC operations. Their specification is not relevant to understanding the results in this paper. So we omit their specification, but it can be found in [Moi97].

These operations, first proposed in [JHB87], are supported by MIPS-II [Kan89], DEC Alpha [Sit92], and IBM PowerPC [IBM94].

The result and its significance: This paper presents two implementations:

1. {CAS, read, write} from {LL, SC, VL}.
2. {LL, SC, VL, read, write} from {LL, SC, VL}.

These implementations are presented in Sections 2 and 3 and have constant time and space complexity. The following two implementations are due to Moir [Moi97]:

1. {LL, SC, VL} from {CAS}.
2. {CAS} from {RLL, RSC}.

The first implementation has constant time complexity, and the second has constant time complexity provided that RSC does not experience spurious failures (see [AM95, Moi97] for a description of spurious failures). Both implementations have constant space complexity.

Composing our implementations with Moir's, each of the three types of shared memory described in the previous subsection has a constant time and space complexity implementation from any of the other two types of shared memory. It follows that any concurrent program designed for one type of shared memory can be ported to others without affecting its time complexity (the slow-down is only by a constant factor).

Previous work: Israeli and Rappoport implement {LL, SC, VL} from {CAS, read} [IR94]. Their implementation makes an unreasonable assumption that the shared objects used in the implementation have $\Omega(n)$ bits, where n is the number of processes that may concurrently access the implemented memory. Anderson and Moir implement {LL, SC, VL} from {CAS, read} and vice versa [AM95]. Moir implements {LL, SC, VL} from {CAS}, and {CAS} from {RLL, RSC} [Moi97]. The above implementations in [AM95] and [Moi97] have constant time and space complexity.

Bridging a gap in the previous work: In all of the previous works cited above, notice that the *write* operation is not included in the set of implemented operations. This is the conceptual gap that this paper fills.

Most existing concurrent programs that rely on LL/SC for synchronization seem to apply LL and SC on some memory words, and read and write on other memory words. In particular, it seems uncommon to come across programs in which all of LL, SC, read and write are applied on the same memory word. This pattern of operation usage is, however, not universal. For instance, in a snapshot algorithm that we are developing all of LL, SC, *and* write are applied on the same memory word. The same is true of another algorithm—a read-parallel universal construction—that we are developing. To port these and other such programs to a system supporting {CAS, read, write}-memory, we will need an implementation of {LL, SC, VL, write}-memory from {CAS, read, write}-memory, such as the one presented in this paper. In particular, such porting is not possible using the implementations in [AM95, Moi97] since these works implement an object that supports LL and SC, but not an object that supports LL, SC, and write.

We close this section with two remarks. First, each of {CAS, read, write}-memory and {LL, SC, VL, write}-memory can be implemented from the other using existing universal constructions (for instance, the constructions in [HS93, Moi97]). This approach is not satisfactory because the time complexity of the resulting implementation is not constant (the time complexity is $O(n)$, where n is the number of processes that can concurrently access the implementation).

Our second remark concerns the conceptual difficulty of the implementations designed in this paper. At first glance, it might *appear* that it should be fairly straightforward to include the write operation in the set of implemented operations. In fact, after implementing {CAS, read} from {LL, SC}, Anderson and Moir state that a write operation is straightforward to incorporate into their construction and give a two line algorithm to implement the write operation (see lines 6-10 after Theorem 2 on page 186 of [AM95]). We believe that their statement is incorrect: in [Jay98], we point out why their construction is incorrect. We emphasize that this is a very minor aspect of their paper and does not affect the main results of that paper. Nevertheless, it points out that implementing read and write operations in addition to other operations (such as CAS, LL, SC etc.) is subtler than it appears. In fact, in an expanded version of this paper [Jay98] we present ample evidence to suggest that this problem is nontrivial.

2 {CAS, read, write} from {LL, SC, VL}

In this section, we present a constant time and space implementation of an object that supports CAS, Read, and Write operations from objects that support LL, SC, and VL operations. The implementation is designed in two steps. First, we define a new object that we call *dual stream LL/SC/VL object*, and implement it from a {LL, SC, VL} object. Second, we implement a {CAS, read, write} object from dual stream LL/SC/VL objects. Composing these two implementations, we obtain {CAS, read, write} from {LL, SC, VL}.

2.1 Definition of Dual Stream LL/SC/VL

A *dual stream LL/SC/VL object* supports two versions of LL, SC, and VL operations: LL_0, VL_0, SC_0, LL_1, VL_1, and SC_1. The operation SC_j by a process p succeeds if and only if no process performed a successful SC_0 or SC_1 since process p's latest LL_j. The important point is that the success of SC_j does *not* depend on when p performed $LL_{\overline{j}}$.

Formally, the state of an object O supporting these operations is described by $value(O)$, $Pset_0(O)$, and $Pset_1(O)$. Figure 3 gives a precise definition of the operations.

$\underline{\mathrm{LL}_j(p_i, O)}$	$\underline{\mathrm{SC}_j(p_i, v, O)}$	$\underline{\mathrm{VL}_j(p_i, O)}$
$\mathrm{Pset}_j(O) := \mathrm{Pset}_j(O) \cup \{p_i\}$	**if** $p_i \in \mathrm{Pset}_j(O)$	**return** $p_i \in \mathrm{Pset}_j(O)$
return value(O)	value$(O) := v$	
	$\mathrm{Pset}_0(O) := \emptyset$	
	$\mathrm{Pset}_1(O) := \emptyset$	
	return *true*	
	else return *false*	

Fig. 3. Specification of a dual stream LL/SC/VL object O

To illustrate the definition, consider an execution in which operations are applied in the following order: $LL_1(p_0, O)$, $LL_0(p_1, O)$, $SC_0(p_1, u, O)$, $LL_0(p_0, O)$, $SC_1(p_0, v, O)$, $SC_0(p_0, w, O)$. In this sequence, SC_0 by p_1 succeeds and causes SC_1 by p_0 to fail. The SC_0 by p_0 succeeds because there is no successful SC_0 or successful SC_1 since p_0's latest LL_0.

2.2 Implementing a dual stream LL/SC/VL object

We implement a dual stream LL/SC/VL object $\mathcal{O}$ from an object O that supports LL, SC, and VL operations. To understand the issues involved in getting this implementation right, consider the naive (and incorrect) implementation that works as follows: to implement $\mathrm{LL}_j(p_i, \mathcal{O})$, p_i simply performs LL on O; to implement $\mathrm{SC}_j(p_i, v, \mathcal{O})$, p_i simply performs $\mathrm{SC}(p_i, v, O)$; and to implement $\mathrm{VL}_j(p_i, \mathcal{O})$, p_i simply performs VL on O. To see that this implementation is incorrect, consider a run in which the following sequence of operations are applied on $\mathcal{O}$: $\mathrm{LL}_1(p_0, \mathcal{O})$, $\mathrm{LL}_0(p_1, \mathcal{O})$, $\mathrm{SC}_0(p_1, u, \mathcal{O})$, $\mathrm{LL}_0(p_0, \mathcal{O})$, $\mathrm{SC}_1(p_0, v, \mathcal{O})$. Clearly, by the specification of a dual stream object, SC_1 by p_0 (the last operation in the sequence) should fail. However, with the above implementation, this operation succeeds; this is because $\mathrm{LL}(p_0, O)$, performed in implementing $\mathrm{LL}_0(p_0, \mathcal{O})$ (the next to last operation in the above sequence), causes $\mathrm{SC}(p_0, v, O)$, performed in implementing $\mathrm{SC}_1(p_0, v, \mathcal{O})$ (the last operation in the sequence), to succeed.

The above exercise suggests the principal feature that any correct implementation must possess: the application of LL_j on $\mathcal{O}$ (for any $j \in \{0, 1\}$) by a process p_i should *not* cause a subsequent $SC_{\bar{j}}$ by p_i, that should otherwise fail, to succeed. We present such an implementation in Figure 4. Below, we provide intuition for how this implementation works.

Each process p_i maintains three local variables: $value_i$, $ActiveLL(i, 0)$, and $ActiveLL(i, 1)$. Informally, $value_i$ is the value of $\mathcal{O}$ that p_i obtained in its most recent LL operation (*i.e.*, LL_0 or LL_1) on $\mathcal{O}$. $ActiveLL(i, j)$ is a boolean variable. If $ActiveLL(i, j)$ is false, it means that p_i is sure that some process performed a successful SC on $\mathcal{O}$ after its (p_i's) most recent LL_j on $\mathcal{O}$. If $ActiveLL(i, j)$ is true, it means that p_i has no evidence that a successful SC happened after p_i's most recent LL_j. Now we describe how LL_j, VL_j, and SC_j operations on $\mathcal{O}$ are implemented.

O: {LL, SC, VL}-object
Initially, value(O) = value($\mathcal{O}$) and Pset(O) = $\emptyset$.

Local variables of each p_i:
$ActiveLL(i,j)$ ($j \in \{0,1\}$): boolean, initialized to *false*
$value_i$: arbitrarily initialized

$\mathrm{LL}_j(p_i, \mathcal{O})$

1. **if** $\neg ActiveLL(i,\bar{j})$
2. $\quad value_i := \mathrm{LL}(p_i, O)$
3. **else if** $\neg \mathrm{VL}(p_i, O)$
4. $\quad ActiveLL(i,\bar{j}) := false$
5. $\quad value_i := \mathrm{LL}(p_i, O)$
6. $ActiveLL(i,j) := true$
7. **return** $value_i$

$\mathrm{VL}_j(p_i, \mathcal{O})$

if $ActiveLL(i,j)$
$\quad$**return** $\mathrm{VL}(p_i, O)$
else return *false*

$\mathrm{SC}_j(p_i, \mathcal{O})$

if $ActiveLL(i,j)$
$\quad$**return** $\mathrm{SC}(p_i, O)$
else return *false*

Fig. 4. Implementing dual stream LL/SC/VL object $\mathcal{O}$ from {LL, SC, VL}-object O

To perform LL_j on the implemented object $\mathcal{O}$, a process p_i proceeds as follows. It checks the value of $ActiveLL(i,\bar{j})$. There are two cases, depending on the value of $ActiveLL(i,\bar{j})$, described as follows.

Case 1: Suppose $ActiveLL(i,\bar{j})$ is false. This means that p_i already knows that some process performed a successful SC on $\mathcal{O}$ after p_i's most recent $\mathrm{LL}_{\bar{j}}$. Thus, p_i has enough information in its local variable $ActiveLL(i,\bar{j})$ to know that it should fail a subsequent $\mathrm{SC}_{\bar{j}}$ or $\mathrm{VL}_{\bar{j}}$. In particular, to implement the current LL_j operation on $\mathcal{O}$, it is safe to perform an LL on the base object O. So it performs LL on O and stores the response in $value_i$ (Line 2), sets the local variable $ActiveLL(i,j)$ to *true* (since it has just performed LL_j on $\mathcal{O}$ and has no evidence that any process has since performed a successful SC on $\mathcal{O}$) (Line 6), and returns $value_i$ (Line 7).

Case 2: Suppose, on Line 1, $ActiveLL(i,\bar{j})$ is true. This means that p_i has so far no evidence that a successful SC operation happened after p_i's latest $\mathrm{LL}_{\bar{j}}$ on $\mathcal{O}$. To determine if such a successful SC happened, p_i performs VL on O (Line 3). There are two subcases, depending on whether VL returns true or false, and are described as follows.
Subcase 2 (i): Suppose VL returns false. This means that a successful SC happened after p_i's latest $\mathrm{LL}_{\bar{j}}$. So p_i sets $ActiveLL(i,\bar{j})$ to *false* (Line 4). This ensures that p_i has enough information in its local variable $ActiveLL(i,\bar{j})$ to know that it should fail a subsequent $\mathrm{SC}_{\bar{j}}$ or $\mathrm{VL}_{\bar{j}}$. Thus, it is now safe to implement the current LL_j on $\mathcal{O}$ by performing an LL on O. So p_i performs LL on O and stores the response in $value_i$ (Line 5), sets the local variable $ActiveLL(i,j)$ to *true* (Line 6), and returns $value_i$ (Line 7).
Subcase 2 (ii): Suppose VL (on Line 3) returns true. This means that no process

performed a successful SC on $\mathcal{O}$ between time τ when p_i performed its latest $\mathrm{LL}_{\bar{j}}$ on $\mathcal{O}$ and time τ' when VL is performed on Line 3. Then, p_i can pretend that its current LL_j takes effect at τ', the time when it performs VL on Line 3. Further, since there is no successful SC on $\mathcal{O}$ during the interval (τ, τ'), the value of $\mathcal{O}$ at τ' is the same as its value at τ, which is already present in the local variable $value_i$. Thus, there is no need for p_i to perform LL on O (in fact, as described in the next paragraph, the implementation would be incorrect if p_i performs an LL on O). So p_i sets the local variable $ActiveLL(i, j)$ to *true* (Line 6), and returns $value_i$ (Line 7).

As mentioned above, in Subcase 2 (ii), it is important that p_i does not perform LL on O. To see this, suppose that it does. That is, after VL returns true (on Line 3) at time τ', p_i performs LL on O in its next step, say, at time τ''. Suppose that a different process p_k performs a successful SC (SC_0 or SC_1) on $\mathcal{O}$ after τ' and before τ''. If p_i performs $\mathrm{SC}_{\bar{j}}$ at a later time, it should fail by the specification of a dual stream object. However, because of the LL on O at time τ'', it will succeed. This is why p_i does not perform LL on O in our implementation. This completes the discussion of how p_i performs LL_j on $\mathcal{O}$.

To perform VL_j on the implemented object $\mathcal{O}$, p_i proceeds as follows. If the local variable $ActiveLL(i, j)$ is false, then some successful SC on $\mathcal{O}$ occurred after p_i's latest LL_j operation; so p_i returns false. Otherwise, p_i performs VL on O to determine if a successful SC occurred on $\mathcal{O}$ since its latest LL_j, and returns the response from O. (As an optimization, if VL returns false, p_i could set its local variables $ActiveLL(i, 0)$ and $ActiveLL(i, 1)$ to false. But, in the interest of brevity of the algorithm, we did not include these extra lines. They do not affect the correctness.)

To perform SC_j on $\mathcal{O}$, p_i proceeds similarly. If $ActiveLL(i, j)$ is false, then some successful SC on $\mathcal{O}$ occurred after p_i's latest LL_j operation; so p_i returns false. Otherwise, p_i performs SC on O and returns the response from O. (An optimization similar to the one mentioned above for the VL_j implementation is also possible here, but we have omitted it in the interest of brevity.)

Theorem 1. *There is a constant time complexity implementation of a dual stream LL/SC/VL object from a single object supporting LL, SC, and VL operations.*

2.3 Implementing {CAS, Read, Write}

In this section, we implement a {CAS, Read, Write} object $\mathcal{O}$ using a dual stream LL/SC/VL object O and a normal {LL, SC, VL} object R. In the following, we first describe the key ideas incorporated into the algorithm and then provide an informal description of the procedures implementing CAS, Read, and Write operations. The implementation is presented in Figure 5.

The key ideas: The following are the main ideas and conventions incorporated into the algorithm.

1. The value of the implemented object $\mathcal{O}$ is maintained in the base object O at all times.

2. The zeroth stream of O's operations (specifically, LL_0 and SC_0) are used to implement the CAS operation on $\mathcal{O}$. The first stream of O's operations (specifically, LL_1 and SC_1) are used to implement Read and Write operations on $\mathcal{O}$.

3. When a process wants to write a value V in the implemented object $\mathcal{O}$, it first "announces" its intent by attempting to get the value V into the base object R. We use LL and SC operations to ensure that if several processes attempt to announce concurrently, only one will succeed. For example, suppose W and W' are concurrent write operations on $\mathcal{O}$ that both attempt to announce their write values in R. Suppose W's announcement succeeds and W''s fails. After this, both W and W' proceed to install in O the write value announced in R. (Thus, after a Write operation attempts to announce its write value, regardless of whether its announcement is successful, it proceeds to help install in O whatever value was successfully announced in R.) Once a Write operation transfers the value in R to O, it clears the object R (by making its value $\perp$), thus making it possible for another Write operation to announce its write value.

Continuing with the above example, suppose that W and W' have together transferred W's announced value (which is in R) to O. We linearize W at the point where W's write value is transferred to O. As for W', it is linearized immediately before W (this amounts to pretending that W overwrites W' just after W' takes effect.) In other words, W' can be sure that it has taken effect as soon as it helps W take effect. This, in fact, is the incentive for W' to help install W's announced value (even though W''s announcement has failed).

4. Processes use LL_1 and SC_1 operations on O in order to transfer a value from R to O. Further, a process that is attempting to transfer a value V from R to O exercises the following caution: after performing LL_1 on O the process performs $\mathrm{SC}_1(V)$ on O if and only if O's value is not already V. Such care is necessary for two reasons: (1) to ensure that at most one successful SC_1 operation is performed (by all helping processes together) on behalf of any Write operation, and (2) to prevent a frivolous Write (*i.e.*, a Write operation that writes the same value as what the object already has) from erroneously causing a CAS to fail (this is explained in detail in the expanded version [Jay98]).

5. When a process p_i wants to apply $\mathrm{CAS}(p_i, \mathit{old}, \mathit{new}, \mathcal{O})$, it first obtains O's value by performing LL_0 on O. If O's value is different from *old*, p_i terminates its CAS returning *false*. On the other hand, if O's value is the same as *old* and if $\mathit{old} = \mathit{new}$, p_i pretends that its CAS changed O's value from *old* to *old*, and terminates returning *true*. If the above cases do not apply, it follows that O's value (as read by p_i's LL) is the same as *old* and $\mathit{old} \neq \mathit{new}$.

At this point, one strategy would be to have p_i perform an SC on O, attempting to change O's value to *new*. However, with such a strategy, successive CAS operations can repeatedly succeed, making it impossible for Write operations on $\mathcal{O}$ to take effect.

To prevent such starvation of Write operations, we adopt a different strategy in which p_i helps any pending Write operation on $\mathcal{O}$ before performing SC on

O: Dual stream LL/SC object supporting LL_0, SC_0, LL_1, SC_1.
Initially, $\mathrm{value}(O) = \mathrm{value}(\mathcal{O})$ and $\mathrm{Pset}_0(O) = \mathrm{Pset}_1(O) = \emptyset$.
R: {LL, SC, VL} object
Initially, $\mathrm{value}(R) = \perp$ and $\mathrm{Pset}(R) = \emptyset$

```
Read(p_i, 𝒪)
return LL_1(p_i, O)

Write(p_i, v, 𝒪)
InstallWrite(p_i)
[val, phase] := LL(p_i, R)
if val = ⊥
      SC(p_i, [v, 0], R)
InstallWrite(p_i)

CAS(p_i, old, new, 𝒪)
val := LL_0(p_i, O)
if val ≠ old return false
if old = new return true
InstallWrite(p_i)
return SC_0(p_i, new, O)

InstallWrite(p_i)
[val, phase] := LL(p_i, R)
if val = ⊥ return
if phase = 0
      val' := LL_1(p_i, O)
      if val' = val
            SC(p_i, [⊥, 0], R)
      else SC(p_i, [val, 1], R)
      [val, phase] := LL(p_i, R)
      if phase = 0 return
val' := LL_1(p_i, O)
if ¬VL(p_i, R) return
if val' ≠ val
      if ¬SC_1(p_i, val, O)
            val' := LL_1(p_i, O)
            if ¬VL(p_i, R) return
            if val' ≠ val
                  SC_1(p_i, val, O)
SC(p_i, [⊥, 0], R)
```

Fig. 5. Implementing object $\mathcal{O}$ that supports CAS, Read, and Write

O and completing its CAS operation. Specifically, p_i determines if there is a pending Write operation on $\mathcal{O}$ by checking if R has a non-$\perp$ value. If there is a pending Write operation W, p_i helps complete W by transferring the value in R to O (and then clearing R). After this, p_i completes its CAS operation by performing SC on O.

6. R consists of two fields: a *value* field and a *phase* field. The phase field has either 0 or 1. When there is no pending Write on $\mathcal{O}$, R contains $[\perp, 0]$ (*i.e.*, $R \cdot \mathrm{value} = \perp$ and $R \cdot \mathrm{phase} = 0$). A Write operation W on $\mathcal{O}$ announces its write value v in R by changing R's state from $[\perp, 0]$ to $[v, 0]$. While helping W, a process p_j changes R's state from $[v, 0]$ to either $[v, 1]$ or $[\perp, 0]$, as described below. While attempting to transfer R's value v to O, p_j first reads O. If O's value is v (this can happen either because some other process has already transferred v from R to O or because the previous value of O is v), then there is no need to change O. In this case, p_j attempts to clear R by swinging its value from $[v, 0]$ to $[\perp, 0]$. On the other hand, if p_j reads a value other than v in O, then p_j believes that v should be written to O to help W take effect. To communicate its belief to other processes, p_j attempts to swing R from $[v, 0]$ to $[v, 1]$ (thus, a value of 1 in the phase field indicates that some process observed in O a value different from v after v was announced in R). Later, after v is transferred to O (by p_j or some other helper), R is swung from $[v, 1]$ to $[\perp, 0]$.

Informal description of procedures: We now describe the procedures implementing Read, Write, and CAS operations, and the auxiliary procedure InstallWrite.

Read($p_i, \mathcal{O}$) procedure: Since $\mathcal{O}$'s value is always contained in O, the implementation of Read($p_i, \mathcal{O}$) is simple: p_i performs LL_1 on O and returns the value obtained.

InstallWrite procedure: The implementations of both Write and CAS operations call the auxiliary procedure InstallWrite. This procedure helps any pending Write operation to run to completion. Below we explain how this is accomplished.

A process p_i executes InstallWrite as follows. It begins by reading R into $[\mathit{val}, \mathit{phase}]$ (Line 1). If val is $\perp$, then there is no pending Write on $\mathcal{O}$; so p_i returns (Line 2). Otherwise, there is a pending Write operation on $\mathcal{O}$. Let W denote this pending Write. In the rest of the procedure, the sole objective of p_i is to help W run to completion.

Suppose phase is 0. Then, p_i reads O (Line 4). If O's value is the same as val (the write value announced by W in R), then even without modifying O one can pretend that W has taken effect. So, to effect the completion of W, p_i attempts to swing R from $[\mathrm{val}, 0]$ to $[\perp, 0]$ (Line 6). On the other hand, if O's value is different from val, then to make W take effect it is necessary to modify O's value to val. To communicate to other processes the need to modify O, p_i attempts to swing R from $[\mathrm{val}, 0]$ to $[\mathit{val}, 1]$ (Line 7). Just after p_i completes executing the if-then-else statement on Lines 5-7, p_i can be certain that either W has completed or R's phase is currently 1. So it reads R again into $[\mathit{val}, \mathit{phase}]$ (Line 8). If phase is 0, it deduces that W has completed; since p_i's only goal is to help W run to completion, p_i returns from the procedure. On the other hand, if phase is 1, then some Write operation is pending; this pending write may be W or some subsequent write operation W'. In any case, if phase is 1, p_i cannot be certain that W has completed and so it cannot return yet from the procedure.

When p_i reaches Line 10, it is certain that when it last read R into $[\mathit{val}, \mathit{phase}]$ (on Line 1 or on Line 8), some Write operation W' was pending and phase was 1. (Notice that W' is either W or another Write that followed W.) Since phase is 1, it means that some process had noticed O's value to be different from val. So p_i prepares to transfer R's value to O. To do this, p_i performs LL_1 to read O's value into val' (Line 10) and then validates R (Line 11). If VL returns false on Line 11, it means that R was modified since the time p_i performed LL on R. It follows that some process changed R's value from $[\mathit{val}, 1]$ to $[\perp, 0]$ since the time p_i performed LL on R. It follows that W', and therefore W, have completed. So p_i returns from the procedure.

Suppose, on Line 11, VL returns true. Then, p_i knows that W' has not completed (but it may have taken effect). If $\mathit{val}' = \mathit{val}$ (on Line 12), p_i deduces that either some process already installed W''s write value in O or O's value is val even without any process installing W''s write value in O. In the former case, W' took effect. In the latter case, it is possible to pretend that W' took effect (overwriting *val* with *val*). Hence, p_i attempts to mark the completion of W' by swinging R from $[val, 1]$ to $[\perp, 0]$ (Line 18).

Suppose $val' \neq val$ (on Line 12). Then p_i deduces that W' has not taken effect. So it attempts to write *val* to O by performing SC_1 (Line 13). If this SC succeeds, then W' has taken effect, so it proceeds to Line 18 to clear R. Otherwise, some other process performed a successful SC either on behalf of W' or on behalf of a CAS operation. If the latter is the case, then p_i must try again to install W''s write value in O. So it performs LL_1 on O (Line 14), and then on Lines 15-17 performs the same actions as it did on Lines 11-13 (for the same reasons). Clearly, if SC_1 succeeds on Line 17, p_i can be certain that W' has taken effect. Interestingly, the same conclusion (that W' has taken effect) can be drawn even if SC_1 fails on Line 17. This is because the CAS operation is implemented in such a way that, while a Write on $\mathcal{O}$ is pending, at most one CAS succeeds. Thus, if SC_1 fails on Line 13 and on Line 17, p_i can be certain that some other process installed W''s write value in O. So it proceeds to Line 18 to clear R.

<u>Write($p_i, v, \mathcal{O}$) procedure</u>: To implement Write($p_i, v, \mathcal{O}$), process p_i proceeds as follows. First, p_i executes InstallWrite (Line 1), thereby helping any pending Write operation to complete and R to clear. Next p_i attempts to announce its write value in R (Lines 2-4). At the end of Line 4, either p_i's Write or some other process' Write announced a value in R. Let W denote the Write that successfully announced in R. By executing InstallWrite on Line 5, p_i helps W run to completion. After Line 5, p_i can be certain that its own Write (which may or may not be W) has taken effect. This is justified as follows. If p_i's Write is W, then it has taken effect (because of Line 5). Otherwise, p_i's Write is concurrent with W. Since W has taken effect, p_i's Write can be linearized just before W.

<u>CAS($p_i, old, new, \mathcal{O}$) procedure</u>: To implement CAS($p_i, old, new, \mathcal{O}$), process p_i proceeds as follows. First, it reads $\mathcal{O}$'s value into a local variable *val* by performing LL on O (Line 1). If $val \neq old$, p_i fails CAS (Line 2). If $val = old = new$, then there is no need to modify O; p_i simply pretends that its CAS succeeded (Line 3). Otherwise, p_i helps any pending Write to run to completion (without such help, Write operations will starve). Finally, p_i attempts to swing the value in O from *old* to *new* by performing SC_0 on O (Line 5). If the SC succeeds, it is obvious that p_i's CAS succeeded, so p_i returns *true*. If the SC fails, it is legitimate for p_i to fail its CAS. The justification is as follows. If p_i's SC fails on Line 5, it follows that some process performed either a write or a successful SC on O sometime between p_i's LL on Line 1 and p_i's SC on Line 5. Further, since O's value was *old* (when p_i performed Line 1) and in our algorithm each successful SC on O changes O's value, p_i can be certain that at some point between p_i's Line 1 and Line 5 O's value was different from *old*. Since p_i's CAS can be linearized at that point, it is legitimate for p_i to fail its CAS.

The theorem below follows from the above implementation and Theorem 1.

Theorem 2. *There is a constant time complexity implementation of an object supporting CAS, read, and write operations from two objects supporting LL, SC, and VL operations.*

3 {LL, SC, VL, Read, Write} from {LL, SC, VL}

In this section, we implement an object $\mathcal{O}$ that supports LL, SC, VL, Read, and Write operations using a dual stream LL/SC/VL object O and a normal {LL, SC, VL} object R. This implementation is presented in Figure 6. There are many similarities, yet significant differences, between this implementation and the one in the previous section. Below, we describe the main ideas on which the implementation is based:

1. O has two fields: a *value* field and a *tag* field. The value of the implemented object $\mathcal{O}$ is maintained in O's value field at all times. When an SC operation on $\mathcal{O}$ modifies O, it writes 2 in O's tag field. When a Write operation on $\mathcal{O}$ modifies O, it writes either 0 or 1 in O's tag field.

2. We ensure $\text{Pset}(\mathcal{O}) = \text{Pset}(O)$.

3. The zeroth stream of O's operations are used to implement the LL, SC, and VL operations on $\mathcal{O}$. The first stream of O's operations are used to implement Read and Write operations on $\mathcal{O}$.

4. As in the previous implementation, R is the object where a Write operation on $\mathcal{O}$ attempts to announce its write value. R has two fields: a *value* field and a *tag* field. When there is no pending Write operation on $\mathcal{O}$, R's value field has $\perp$. A Write operation attempts to announce its write value V by swinging R from $[\perp, tag]$ to $[V, \overline{tag}]$. Subsequently, when clearing R, processes swing R from $[V, \overline{tag}]$ to $[\perp, \overline{tag}]$. Thus, R's tag alternates between 0 and 1 with each successive announcement.

5. The tag fields of O and R make it possible for a process to determine whether or not the announced value in R was already transferred to O. In particular, the implementation has the following key property: If R's value is not $\perp$ (*i.e.*, there is a pending Write), then O's tag and R's tag are equal if and only if the announced value in R has already been applied to O (*i.e.*, the pending Write has already taken effect).

6. Consider a Write operation W that has announced its write value in R. Although many processes may help W run to completion by attempting to transfer its announced value from R to O, the implementation ensures that exactly one process performs a successful SC on O on behalf of W. In particular, even if $\mathcal{O}$'s value is V and W's write value is also V, the implementation makes sure that a successful SC is performed on O (by some process) on behalf of W. (This is in sharp contrast to the previous implementation of {CAS, Read, Write} where we took great care to avoid applying a Write operation if it does not change $\mathcal{O}$'s value.) In the current implementation, a successful SC is performed on O on behalf of every Write in order to ensure the invariant $\text{Pset}(\mathcal{O}) = \text{Pset}(O)$ (to see this, note that a Write operation on $\mathcal{O}$ should cause $\text{Pset}(\mathcal{O})$ to become empty.

O: Dual stream LL/SC/VL object
  Initially, value(O) = [value($\mathcal{O}$), 0] and $\text{Pset}_0(O) = \text{Pset}_1(O) = \emptyset$.
R: {LL, SC, VL} object
  Initially, value(R) = $[\perp, 0]$ and Pset(R) = $\emptyset$

$\underline{\text{LL}(p_i, \mathcal{O})}$
return $\text{LL}_0(p_i, O)$

$\underline{\text{SC}(p_i, v, \mathcal{O})}$
$\text{InstallWrite}(p_i)$
return $\text{SC}_0(p_i, [v, 2], O)$

$\underline{\text{VL}(p_i, \mathcal{O})}$
return $\text{VL}_0(p_i, O)$

$\underline{\text{Write}(p_i, v, \mathcal{O})}$
1. $\text{InstallWrite}(p_i)$
2. $[val, tag] := \text{LL}(p_i, R)$
3. **if** $val = \perp$
4. $\quad \text{SC}(p_i, [v, \overline{tag}], R)$
5. $\text{InstallWrite}(p_i)$

$\underline{\text{Read}(p_i, \mathcal{O})}$
return $\text{LL}_1(p_i, O)$

$\underline{\text{InstallWrite}(p_i)}$
1. $[val, tag] := \text{LL}(p_i, R)$
2. **if** $val = \perp$ **return**
3. $[val', tag'] := \text{LL}_1(p_i, O)$
4. **if** $\neg\,\text{VL}(p_i, R)$ **return**
5. **if** $tag' \neq tag$
6. $\quad$ **if** $\neg\,\text{SC}_1(p_i, [val, tag], O)$
7. $\quad\quad [val', tag'] := \text{LL}_1(p_i, O)$
8. $\quad\quad$ **if** $\neg\,\text{VL}(p_i, R)$ **return**
9. $\quad\quad$ **if** $tag' \neq tag$
10. $\quad\quad\quad \text{SC}_1(p_i, [val, tag], O)$
11. $\text{SC}(p_i, [\perp, tag], R)$

Fig. 6. Implementing object $\mathcal{O}$ that supports LL, SC, VL, Read, and Write

This, together with the need to maintain the invariant Pset($\mathcal{O}$) = Pset(O), implies that a successful SC must be performed on O on behalf of every write on $\mathcal{O}$).

With the above concepts, it is fairly easy to understand the implementation. The procedures for LL, VL, and Read operations on $\mathcal{O}$ are self-explanatory.

The auxiliary procedure InstallWrite helps any pending Write operation to run to completion. A process p_i executes this procedure as follows. It performs LL on R (Line 1). If R's value field has $\perp$, then no Write is pending, so p_i returns (Line 2). Otherwise, let W denote the pending Write operation. On Line 3, p_i reads O's value by performing LL_1. If VL on R returns false, it means that some process cleared R since the time p_i LL'd it on Line 1; this implies W completed, so p_i returns (Line 4). If $tag' = tag$ (Line 5), it means that some process has already transferred W's announced value from R to O; so p_i proceeds to Line 11 to clear R. If $tag' \neq tag$ (Line 5), it means that W has not yet taken effect. So p_i applies SC_1 on O, attempting to install in O the value and the tag that W announced in R (Line 6). If this attempt fails, p_i tries again (Lines 7-10), taking the same precautions that it took on Lines 3-6. Regardless of whether its SC succeeds or fails on Line 10, p_i can be certain that W's announcement has been successfully transferred to O by some process (the justification for this conclusion is similar to the one provided in the previous implementation). So, it attempts to clear R (Line 11).

The write operation on $\mathcal{O}$ works exactly as in the previous implementation with one exception: when announcing in R, it flips the tag bit of R.

To implement SC operation on $\mathcal{O}$, p_i first helps any pending Write (Line 1). (If it were not for this line, InstallWrite's assumption—that the pending Write operation that InstallWrite is helping to complete would have taken effect even if InstallWrite's SC failed both on Line 6 and on Line 10— would be incorrect). Having helped any pending Write, p_i performs an SC operation on O in order to effect its SC operation on the implemented object $\mathcal{O}$ (Line 2).

The theorem below follows from the above implementation and Theorem 1.

Theorem 3. *There is a constant time complexity implementation of an object supporting LL, SC, VL, Read, and Write operations from two objects supporting LL, SC, and VL operations.*

Acknowledgement: I thank the DISC '98 referees for their helpful comments.

References

[AM95] J. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 184–194, August 1995.

[Her91] M.P. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, 1991.

[HS93] M. P. Herlihy and N. Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 111–120, 1993.

[HW90] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.

[IBM94] IBM. *The Power PC Architecture: A specification for a new family of RISC processors.* Morgan-Kaufmann, 1994.

[IR94] A. Israeli and L. Rappoport. Disjoint-Access-Parallel implementations of strong shared-memory primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 151–160, August 1994.

[Jay98] P. Jayanti, 1998. See http://www.cs.dartmouth.edu/~prasad.

[JHB87] E. Jensen, G. Hagensen, and J. Broughton. A new approach to exclusive data access in shared-memory multiprocessors. Technical Report Technical Report UCRL-97663, Lawrence Livermore National Laboratory, 1987.

[Kan89] G. Kane. *MIPS RISC Architecture.* Prentice-Hall, Englewood Cliffs, N.J., 1989.

[Lam77] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.

[Moi97] 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, August 1997.

[Sit92] R. Site. *Alpha Architecture Reference Manual.* Digital Equipment Corporation, 1992.

This article was processed using the LaTeX macro package with LLNCS style