LL/SC and Atomic Copy: Constant Time, Space Efficient Implementations Using Only Pointer-Width CAS

Guy E. Blelloch
Carnegie Mellon University, Pittsburgh, PA, USA
guyb@cs.cmu.edu

Yuanhao Wei
Carnegie Mellon University, Pittsburgh, PA, USA
yuanhao1@cs.cmu.edu

Abstract

When designing concurrent algorithms, Load-Link/Store-Conditional (LL/SC) is a very useful primitive since it avoids ABA problems. The full semantics of LL/SC are not supported in hardware by any modern architecture, so there has been a significant amount of work on simulations of LL/SC using CAS. However, all previous algorithms that are constant time either use unbounded sequence numbers (and thus base objects of unbounded size), or require $\Omega(MP)$ space to implement M LL/SC objects for P processes.

We present the first constant time implementation of LL/SC from bounded-sized CAS objects using only constant space overhead per LL/SC variable. In particular, our implementation uses $\Theta(M+kP^2)$ space, where k is the number of outstanding LL operations per process, and only requires pointer-width CAS operations. In most algorithms that use LL/SC, k is a small constant which reduces our additive space overhead to $\Theta(P^2)$. Our algorithm can also be extended to implement L word LL/SC objects in $\Theta(L)$ time for LL and SC, $O(1)$ time for VL, and $\Theta((M+kP^2)L)$ space.

To achieve these bounds, our main technical contribution is implementing a new primitive called Single-Writer Copy which takes a pointer to a word sized memory location and atomically copies its contents into another object. The restriction is that only one process is allowed to write/copy into the destination object at a time. The ability to read from one memory location and write to another atomically, and in constant-time, is very powerful and we believe this primitive will be useful in designing other algorithms.

2012 ACM Subject Classification Computing methodologies → Concurrent algorithms

Keywords and phrases LL/SC, Atomic Copy, CAS, Constant Time

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.5

Related Version A full version of the paper is available at `https://arxiv.org/abs/1911.09671`.

Funding This work was supported in part by NSF grants CCF-1901381, CCF-1910030, and CCF-1919223. Yuanhao Wei was also supported by a NSERC PGS-D Scholarship.

Acknowledgements We would like to thank our anonymous reviewers for their helpful comments.

1 Introduction

In lock-free, shared memory programming, it is well known that the choice of atomic primitives makes a big difference in terms of ease of programmability, efficiency, and even computability. Most processors today support a set of basic synchronization primitives such as Compare-and-Swap, Fetch-and-Add, Fetch-and-Store, etc. However, many useful primitives are not supported, which motivates the need for efficient software implementations of these primitives. In this work, we present constant time, space-efficient implementations of a widely used primitive called Load-Link/Store-Conditional (LL/SC) as well as a new primitive we call

Single-Writer Copy (**swcopy**). All our implementations use only pointer-width read, write, and CAS. In particular, restricting ourselves to pointer-width operations means that we do not use unbounded sequence numbers, which are often used in other LL/SC from CAS implementations [27, 26, 22]. Many other algorithms based on CAS also use unbounded sequence numbers (often alongside double-wide CAS) to get around the ABA problem and this is sometimes called the *IBM tag methodology* [24, 18]. Our LL/SC implementation can be used in these algorithms to avoid unbounded sequence numbers and double-wide CAS.

The Single-Writer Atomic Copy (**swcopy**) primitive allows processes to atomically read from one memory location and write the result to another. The memory location being read can be arbitrary, but the location being written to has to be a special **Destination** object. A **Destination** object supports three operations, **read**, **write**, and **swcopy** and it allows any process to **read** from it, but only a single process to **write** or **swcopy** into it. This primitive is very useful in announcement array based algorithms because it removes any delay between reading a value and announcing that value. A recent paper uses this primitive to solve various problems related to resource management, such as concurrent reference counting, in constant (expected) time [12].

In this work, we focus on bounded wait-free solutions. Roughly speaking, *bounded wait-freedom* ensures that *each* process makes progress within a bounded number of its own steps regardless of how it is scheduled. In particular, algorithms satisfying this property do not suffer from problems such as deadlock, livelock, and priority inversion. All algorithms presented in this paper take either $O(1)$ or $O(L)$ time (where L is the number of words spanned by the implemented object), which makes them also bounded wait-free. The correctness condition we consider is *linearizability*, which intuitively means that all operations appear to take effect at a single point.

In our results below, the time complexity of an operation is the number of instructions that it executes (both local and shared) in a worst-case execution and space complexity of an object is the number of words that it uses (both local and shared). Counting local objects/instructions is consistent with previous papers on the topic [5, 26, 22]. There has been a significant amount of prior work on implementing LL/SC from CAS [6, 27, 20, 22, 26] and we discuss them in more detail in Section 2.

Result 1 (Load-Link/Store-Conditional): *A collection of M LL/SC objects operating on L-word values shared by P processes, each performing at most k outstanding LL operations, can be implemented with:*

1. $\Theta(L)$ *time for LL and SC,* $O(1)$ *time for VL and CL,*
2. $\Theta((M + kP^2)L)$ *space,*
3. *single word (at least pointer-width) read, write, CAS.*

Our algorithm requires knowing k, the maximum number of outstanding LL/SC operations, and P in advance. In theory, k could be as large as M, but for most data structures implemented from LL/SC, such as Fetch-And-Increment [14] and various Universal Construction [16, 10, 1], k is at most 2. Assuming w-bits is enough to store a pointer, given a data structure implemented from w-bit LL/SC objects, Result 1 implies that it can be implemented from w-bit CAS objects while maintaining the same time complexities and using only $\Theta(kP^2)$ additional space across all instances of the data structure. In contrast, using previous approaches [5, 6, 20, 27] to implement the w-bit LL/SC objects from w-bit CAS objects would require $\Omega(P)$ space *per* LL/SC object.

Our main technical contribution is in implementing a **Destination** object supporting **read**, **write** and **swcopy** with the following bounds.

Result 2 (Single-Writer Copy): *A collection of M* **Destination** *objects, each storing single word values, shared by P processes can be implemented with:*

1. *$O(1)$ worst-case time for read, write, and swcopy,*
2. *$\Theta(M + P^2)$ space,*
3. *single word (at least pointer-width) read, write, CAS.*

To help implement the **Destination** objects, we implement a weaker version of LL/SC which allows the LL operation to fail if it is concurrent with a successful SC. Our version of weak LL/SC is a little different from what was previously studied [5, 20, 27], and we compare the two in more detail in Section 2. Our algorithm for weak LL/SC uses several known techniques which we also cover in Section 2.

Result 3 (Weak Load-Link/Store Conditional): *A collection of M weak LL/SC objects operating on L-word values shared by P processes, each performing at most one outstanding wLL, can be implemented with:*

1. *$\Theta(L)$ time for wLL and SC, $O(1)$ time for VL and CL,*
2. *$\Theta((M + P^2)L)$ space,*
3. *single word (at least pointer-width) read, write, CAS.*

Our implementations of **swcopy** and LL/SC are closely connected. We begin in Section 4 by implementing a weaker version of LL/SC (Result 3). Then, in Section 5, we use this weaker LL/SC to implement **swcopy** (Result 2), and finally, in Section 6, we use **swcopy** to implement the full semantics of LL/SC (Result 1). As we shall see, once we have **swcopy**, our algorithm for regular LL/SC becomes almost identical to our algorithm for weak LL/SC.

2 Related Work

LL/SC Implementations. We consider three types of implementations, single-word LL/SC from single-word CAS, multi-word LL/SC from single-word LL/SC, and multi-word LL/SC from single-word CAS. Existing implementations of each type are summarized in Table 1. All the algorithm shown in the table are wait-free and have optimal time bounds. Aghazadeh and Woelfel [4] do not directly implement multi-word LL/SC from single-word LL/SC, but their single CAS universal construction (Figure 1 of [4]) can be used to achieve the bounds shown in Table 1.

All previous implementations of single or multi-word LL/SC from CAS have one of three drawbacks. They either (1) are not constant time [13, 19], (2) use unbounded sequence numbers [27, 26, 22, 23], or (3) require $\Omega(MP)$ space even in the common case where k is constant [6, 20, 27].

The simplest way to implement an LL/SC object is to tag a CAS object with an unbounded sequence number [27]. To avoid using unbounded sequence numbers, various algorithms were proposed that recycle these tags [27, 6, 20]. However, all these algorithms are only able to implement single-word LL/SC, and in some of these algorithms (i.e. [6] and Figures 4 and 7 of [27]), the size of the simulated LL/SC object is smaller than the size of the CAS objects that they use. Furthermore, these algorithms [27, 6, 20] are unable to efficiently implement a large number of LL/SC objects as they require at least $\Omega(PM)$ space.

To efficiently implement a large number of large LL/SC objects, a common technique is to use a level of indirection combined with some form of memory management. This means that the simulated LL/SC object stores a pointer to a buffer which contains the actual value of the LL/SC object. To perform an **SC**, the process first allocates a new buffer, initializes it with the desired value, and then tries to change the LL/SC object to point to

Table 1 Cost of implementing M LL/SC objects from either LL/SC or CAS, where k represents the maximum number of outstanding LL/SC operations per process, and L is the number of words in each simulated LL/SC object.

Single-word LL/SC from CAS	**Uses Unbounded Sequence Numbers**	**Time**	**Space**
Anderson and Moir [6], Figure 1	No	$O(1)$	$O(P^2M)$
Moir [27], Figure 4	Yes	$O(1)$	$O(P+M)$
Moir [27], Figure 7	No	$O(1)$	$O(P^2+PM)$
Jayanti and Petrovic [20]	No	$O(1)$	$O(PM)$
Jayanti and Petrovic [23]	Yes	$O(1)$	$O(P^2+PM)$
Multi-word LL/SC from LL/SC	**Unbounded Seq. #**	**Time**	**Space**
Aghazadeh and Woelfel [4]	No	$O(L)$	$O(MP^5L)$
Anderson and Moir [5], Figure 2	No	$O(L)$	$O(P^2ML)$
Jayanti and Petrovic [21]	No	$O(L)$	$O(PML)$
Multi-word LL/SC from CAS	**Unbounded Seq. #**	**Time**	**Space**
Michael [26]	Yes	$O(L)$[1]	$O((P^2k+M)L)$
Jayanti and Petrovic [22]	Yes	$O(L)$	$O((P^2+M)L+Pk)$
This Paper	No	$O(L)$	$O((P^2k+M)L)$

the new buffer with a CAS. This technique is used in Michael's algorithm [26], Jayanti and Petrovic's algorithm [22], as well as our algorithm. The general idea has also been used to implement various other objects such as descriptors [7], writable objects [2], and resetable TAS objects [3]. The main difference between these algorithms is in how they recycle buffers to ensure bounded space usage.

In Jayanti and Petrovic's algorithm, they allow **LL** operations to read from a buffer that has already been recycled and they provide a mechanism for the **LL** to detect if this is the case. Whenever an **LL** operation reads from a buffer that has already been recycled, their helping mechanism ensure that the **LL** operation is sent a consistent value that is safe to return. A consequence of reusing buffers prematurely is that the ABA problem could occur when a process tries to update the buffer pointer with a CAS. To prevent this, they tag each pointer with an unbounded sequence number.

Another approach is to have each **LL** operation acquire a hazard pointer [25] to the buffer before accessing it. The buffer will not be recycled as long as there is a hazard pointer to it. This approach is used by our algorithm, Michael's algorithm [26], and others [2, 3]. The main challenge is in acquiring a hazard pointer in constant time, which requires some form of helping. Michael's helping technique [26] makes use of unbounded sequence numbers which seem difficult to recycle because their relative ordering matters. Unlike other algorithms which require storing both a pointer and an unbounded sequence number in the same word, Michael's algorithm stores the sequence number separately. This reduces the likelihood of wrap around in practice. Aghazadeh, Golab and Woelfel present a clever helping technique that avoids unbounded sequence numbers using only registers [2]. However, using their technique to implement M LL/SC objects would require at least MP space. The helping technique we use is encapsulated by our **swcopy** algorithm.

[1] Amortized expected time

There has also been some work on algorithms that do not require knowing the number of processes in advance, for example Jayanti and Petrovic [23] and Doherty et al. [13]. Michael mentions that his algorithm from [26] can be extended to not require knowing k and P in advance by using ideas from Section 3.2 of the hazard pointers paper [25]. Our algorithm can be extended in the same way, but this extension would not maintain our time bounds as allocating a new hazard pointer takes $\Theta(P)$ time in the worst case.

Weak LL/SC Implementations. A variant of Weak LL/SC was introduced by Anderson and Moir [5] and also studied elsewhere [20, 27]. The version we consider is less restrictive than theirs because they require a failed **wLL** operation to return the process id of the **SC** operation that caused it to fail whereas we do not require failed **wLL** operations to return anything. While prior work is able to implement the stronger version of **wLL**, they either employ stronger primitives like LL/SC [5], use unbounded sequence numbers [27], require $O(MP)$ space for M Weak LL/SC objects [5, 20]. To match the bounds stated in Result 3, we implement our own version of weak LL/SC that is sufficient for our **swcopy** algorithm. Conveniently, the majority of our weak LL/SC algorithm from Section 4 can be reused when implementing full LL/SC in Section 6.

Atomic Copy. A similar primitive called **memory-to-memory move** was studied in Herlihy's classic wait-free hierarchy paper [15]. The primitive allows atomic reads and writes to any memory location and supports a **move** instruction which atomically copies the value at one memory location into another. Herlihy showed that this primitive has consensus number infinity. Our **swcopy** is a little different because it allows arbitrary atomic operations (e.g. Fetch-and-Add, Compare-and-Swap, Write, etc) on the source memory location as long as the source objects supports an atomic read. Another difference is that we restrict the destination of the copy to be single-writer. Herlihy's proof that **memory-to-memory move** has unbounded consensus number would also work with our **swcopy** primitive. This means the **Destination** objects defined in Section 5.1 also have consensus number infinity.

3 Preliminaries

A *Compare-and-Swap* (CAS) object stores a value that can be accessed through two operations, read and CAS. The read operation returns the value that is current stored in the CAS object. The CAS operation takes a new value and an expected value and atomically writes the new value into the CAS object if the value that is currently there is equal to the expected value. The CAS operation returns true if the write happens and false otherwise. We say a CAS operation is *successful* if it returns true and *unsuccessful* otherwise.

A *Load-Linked/Store-Conditional* (LL/SC) object stores a value and supports three operations: LL, VL (validate-link), and SC. Sometimes a CL (clear-link) operation is also supported which has no return value. Let O be an LL/SC object. An LL on O simply returns the current value of O. An SC on O performed by process p_i takes a new value and writes it into O if there has not been a CL on O or a successful SC on O since the last LL on O by process p_i. We say that an SC is *successful* if it writes a value into O and *unsuccessful* otherwise. Successful SC operations return true and unsuccessful ones return false. A VL operation by process p_i returns true if there has not been a CL on O or a successful SC on O since the last LL on O by process p_i. A *weak LL/SC* object supports wLL, VL, CL (optional), and SC, and it behaves like an LL/SC object except a wLL on O is allowed to return **empty** if the next SC on O by the same process is guaranteed to be unsuccessful. We

say that a **wLL** is unsuccessful if it returns **empty**. Otherwise, we say the **wLL** is *successful*. When a process performs an LL, the LL is considered to be *outstanding* until the process performs a CL or an SC on the same object. Similarly, when a process performs a successful wLL, the wLL is considered to be *outstanding* until the process performs a CL or an SC on the same object.

We work in the standard asynchronous shared memory model [8] with P processes communicating through base objects that are either registers, or CAS objects. Processes may fail by crashing. All base objects are word-sized and we assume they are large enough to store pointers into memory. We do not hide any extra bits in pointers.

In our model, an *execution* (or equivalently, *execution history*) is an alternating sequence of *configurations* and *steps* C_0, e_1, C_1, e_2, C_2, ..., where C_0 is an *initial configuration*. Each step is a shared operation on a base object. Configuration C_i consists of the values of every base object, and the state of every process after the step e_i is applied to configuration C_{i-1}.

If configuration C precedes configuration C' in an execution, the *execution interval* from C to C' is the set of all configurations and steps between C and C', inclusive. Similarly, the *execution interval* of an operation is the set of all configurations and steps from the first step of that operation to the last step of that operation. The execution interval for an *incomplete operation* is the set of all configurations and steps starting from the first step of that operation.

We say the implementation of an object is *linearizable* [17] if, for every possible execution and for each operation on that object in the execution, we can pick a configuration or step in its execution interval to be its linearization point, such that the operation appears to occur instantaneously at this point. In other words, all operations on the object must behave as if they were performed sequentially, ordered by their linearization points. If multiple operations have the same linearization point, then an ordering must be defined among these operations.

All implementations that we discuss will be *bounded wait-free*. This means that we can give an upper bound on how many steps it takes to complete each operation.

4 Weak LL/SC from CAS

As a subroutine, our **swcopy** operation makes use of multi-word weak LL/SC objects. Recall from Section 3 that weak LL/SC supports three operations **wLL**, **VL** and **SC**, and works the same way as regular LL/SC except that **wLL** is allowed to return **empty**.

In this section, we present a constant time algorithm for multi-word weak LL/SC which works when there is at most one outstanding **wLL** per process. We support a constant time CL operation which can be used to limit the number of outstanding **wLL** operations. Our algorithm assumes that **VL**, CL, and **SC** are only performed on an object if the process has an outstanding **wLL** on that object. This version is sufficient to implement the other algorithms in our paper. To implement M Weak LL/SC objects, each spanning L words, our algorithm takes $O((M + P^2)L)$ space. The proof of correctness for this algorithm can be found in the full version of the paper [11]. The high level idea is to use a layer of indirection and use an algorithm similar to Hazard Pointers [25] to get an upper bound on the memory usage. Many concurrent algorithms use the same high level idea [24, 2, 3].

Each **WeakLLSC** object is represented using a pointer, **buf**, to an L-word buffer storing the current value of the object. A **wLL** operation simply reads **buf** and returns the value that it points to. To perform an **SC**, the process first allocates a new L-word buffer, writes the new value in it, and then tries to write the address of this buffer into **buf** with a CAS. The expected value of this CAS is the value that the process read from **buf** during its previous **wLL**. The problem with this algorithm is that it uses an unbounded amount of space.

Our goal is to recycle buffer objects so that we use at most $O(M + P^2)$ of them. We recycle buffers with a variant of Hazard Pointers that is *worst-case* constant time rather than *expected* constant time. Before accessing a buffer, a **wLL** operation has to first protect it by writing its address to an announcement array. To make sure that its announcement happened "in time", the **wLL** operation re-reads **buf** and makes sure it is the same as what was announced. If **buf** has changed, then the **wLL** operation can return **empty** because it must have been concurrent with a successful **SC** and it can linearize immediately before the linearization point of the **SC**. If **buf** is equal to the announced pointer, then the buffer has been protected and the **wLL** operation can safely read from it.

A **VL** operation by process p_i simply checks if **buf** is equal to the buffer announced by its previous **wLL** operation. If so, it returns true, otherwise, it returns false. A **CL** operation by process p_i simply clears any buffer announced by p_i.

For the purpose of the **SC** operation, each process maintains two lists of buffers: a free list (**flist**) and a retired list (**rlist**). In an **SC** operation, the process allocates by removing a buffer from its local free list. If the CAS instruction performed by the **SC** is successful, it adds the old value of the CAS to its retired list. Each process's free list starts off with $2P$ buffers and we maintain the invariant that the number of buffers in the free list plus the number of buffers in the retired list always equals $2P$. When the free list becomes empty and the retired list hits $2P$ buffers, the process moves some buffers from the retired list to the free list. To decide which buffers are safe to reuse, the process scans the announcement array (the scan doesn't have to be atomic) and moves a buffer from the retired list to the free list if it was not seen in the array. In a later paragraph, we show how this step can be performed in worst-case $O(P)$ time. Since the process sees at most P different buffers in the announcement array during its scan, its free list's size is guaranteed to be at least P after this step. A process performs this expensive step at most once every P **SC** operations because the process has at least P free buffers after this step.

Pseudo-code is shown in Listing 1. In the pseudo-code, we use **A[i].read** and **A[i].write** to read from and write to the announcement array **A**. Since each element of the announcement array is a pointer type, **read** and **write** are trivially implemented using the corresponding atomic instruction. We wrap these instructions in function calls so that the code can be reused in Section 6. The argument from the previous paragraph also implies that **flist** cannot be empty on line 43, so we do not run the risk of dereferencing an invalid pointer on line 44. In the pseudo-code, we use **T*** to denote a pointer to an object of type T and **Value[L]** to denote an array of L word-sized values. If **var** is a variable, **&var** is used to denote the address of that variable.

Initialization. Each **WeakLLSC** object starts off pointing to a different Buffer object and each free list is initialized with $2P$ distinct Buffers. Buffers in the free lists are not pointed to by any of the **WeakLLSC** objects and no Buffer appears in two free lists. This property is maintained as the algorithm executes.

Linear-time set difference. In the pseudo-code, both **rlist** and **reserved** represent lists of buffers. The operation **rlist \ reserved** on line 57 computes the difference between the two lists. What makes the original hazard pointers algorithm *expected* rather than *worst-case* constant time is that it uses a hash table to perform this step. Another approach in the literature is to have each process maintain an array of size B where B is the number of buffers [2, 3]. The array is used to keep track of the buffers that appear in **reserved**. Since $B \in \Theta(M + P^2)$, having an array per process requires too much space in our case, so instead,

Listing 1 Amortized constant time implementation of L-word Weak LL/SC from CAS. Code for process p_i is shown.

```
1  shared variables:
2    Buffer* A[P]; // announcement array
3
4  local variables:
5    list<Buffer*> flist;
6    list<Buffer*> rlist;
7    // initial size of flist is 2P
8    // rlist is initially empty
9
10 struct Buffer {
11   // Member Variables
12   Value[L] val;
13   int pid;
14   bool seen;
15
16   void init(Value[L] initialV) {
17     copy initialV into val
18     pid = -1; seen = 0; } };
19
20 struct WeakLLSC {
21   // Member Variables
22   Buffer* buf;
23
24   // Constructor
25   WeakLLSC(Value[L] val) {
26     buf = new Buffer();
27     buf->init(val); } // non-atomic
28
29   void CL() { A[i].write(NULL); }
30   optional<Value[L]> wLL() {
31     Buffer* tmp = buf;
32     A[i].write(tmp);
33     if(buf == tmp)
34       return tmp->val; //non-atomic
35     else return empty; }
36
37   bool VL() {
38     Buffer* old = A[i].read();
39     return buf == old; }
40
41   bool SC(Value[L] newV) {
42     Buffer* old = A[i].read();
43     Buffer* newBuf = flist.remove();
44     newBuf->init(newV);
45     bool b = CAS(&buf, old, newBuf);
46     if(b) retire(old);
47     else flist.add(newBuf);
48     A[i].write(NULL);
49     return b; }
50
51   void retire(Buffer* old) {
52     rlist.add(old);
53     if(rlist.size() == 2*P) {
54       list<Buffer*> reserved = [];
55       for(int j = 0; j < P; j++)
56         reserved.add(A[j].read());
57       newlyFreed = rlist \ reserved;
58       rlist.remove(newlyFreed);
59       flist.add(newlyFreed); }}};
```

we borrow space from the buffers to perform the marking step. This is done by adding enough space for a process id, **pid**, and a bit, **seen**, to each **Buffer** object. To perform the set difference **rlist** \ **reserved**, the process first visits each buffer **B** in **rlist** and prepares the buffer by setting **B.pid** to its own process id and setting **B.seen** to false. Then, the process loops through **reserved** and for each buffer, it sets **seen** to true if **pid** equals its own process id. Next, the process loops through **rlist** again and constructs a list of buffers that have not been seen. This list is the result of the set difference. Finally, the process has to reset everything by setting **B.pid** to $\perp$ for each **B** in **rlist**.

Deamortization. So far, the algorithm we have described takes *amortized* constant time. To deamortize it, each process can maintain two sets of retired list and free lists. Each time the process removes from one free list, it performs a constant amount of work towards populating the other. This is a commonly used technique for deamortizing garbage collection cost [9]. A different deamortization approach was developed by Aghazadeh, Golab and Woelfel [2] in the context of a different problem.

Space complexity. The algorithm uses P shared space for the announcement array, $O(P^2)$ local space for all the retired and free lists, and $O((M+P^2)L)$ shared space for all the buffers and **WeakLLSC** objects. Therefore, its total space usage is $O((M+P^2)L)$. In addition, it only uses pointer-width read, write, CAS as atomic operations, so it fulfills the claims in Result 3.

5 Single-Writer Atomic Copy

The copy primitive, **swcopy**, can be used to atomically read a value from some source memory location and write it into a **Destination** object. Our **Destination** objects are single-writer and we allow the source memory location to be modified by any instruction (e.g. write, fetch-and-add, swap, CAS, etc). The sequential specifications of **swcopy** and **Destination** objects are given below.

▶ **Definition 1.** *A* **Destination** *object supports 3 operations* **read**, **write** *and* **swcopy** *with the following sequential specifications:*

- **read()***: returns the current value in the* **Destination** *object (initially* $\perp$*).*
- **write(Value v)***: sets* **v** *as the current value of the* **Destination** *object.*
- **swcopy(Value* addr)***: reads the value pointed to by* **addr** *and sets it as the current value of the* **Destination** *object.*

Processes can perform **read** *operations at any time, but no two* **write**/**swcopy** *operations can be concurrent.*

Destination objects are very useful in announcement array based algorithms where it is beneficial to read and announce atomically. Section 5.1 describes our implementation and we prove correctness in Section 5.2.

5.1 Algorithm for Single-Writer Atomic Copy

In this section, we show how to implement **Destination** objects that support **read**, **write**, and **swcopy** in $O(1)$ time and $O(M+P^2)$ space (where M is the number of **Destination** objects). Our algorithm only requires pointer-width read, write and CAS instructions.

We represent a **Destination** object **D** internally using a triplet, **D.val**, **D.ptr**, and **D.old**. When there is no **swcopy** in progress, **D.val** stores the current value of the **Destination** object. When there is a copy in progress, **D.ptr** stores a pointer to the location that is being copied from. Operations that see a copy in progress will help complete the copy. Finally, **D.old** stores the previous value of the **Destination** object. The variables **D.val** and **D.ptr** are stored together in a multi-word **WeakLLSC** object (defined in Section 4). This allows us to read from and write to them atomically as well as prevent any potential ABA problems. The downside is that the only way to read **D.val** or **D.ptr** is through a **wLL** operation which can repeatedly fail due to concurrent **SC** operations. For this reason, we keep **D.old** in a separate word, so that the readers can return **D.old** if they fail too many times on **wLL**. Readers will only perform **SC** operations that change **D.ptr** from not **NULL** to **NULL**. Therefore, the writer's **wLL** will be successful whenever **D.ptr** is **NULL**. We will maintain the invariant that **D.ptr** is **NULL** whenever there is no ongoing **swcopy**. We also ensure that **D.ptr** changes exactly twice during each **swcopy**. The first change writes a valid pointer and the second change resets it back to **NULL**.

A **swcopy(Value* src)** on **Destination** object **D** begins by backing up the current value from **D.val** into **D.old**. At this point, **D.ptr** is guaranteed to be **NULL**, so the writer can successfully read **D.val** with a **wLL**. The **swcopy** proceeds by writing **src** into **D.ptr** with

Listing 2 Atomic copy (single-writer). Code for process p_i.

```
1  struct Data {
2    Value val;
3    Value* ptr;};

5  struct Destination {
6    // Member Variables
7    Value old;
8    WeakLLSC<Data> data;
9    // data is initially ⟨⊥, NULL⟩

11   void swcopy(Value *src) {
12     // This wLL() cannot fail
13     old = data.wLL().val;
14     data.SC(⟨⊥, src⟩);
15     Value val = *src;
16     optional<Data> d = data.wLL();
17     if(d != empty && d.ptr != NULL)
18       data.SC(⟨val,NULL⟩);
19     else if(d != empty)
20       data.CL(); }

21 void write(Value new_val) {
22   // This wLL() cannot fail
23   old = data.wLL().val;
24   data.SC(⟨new_val, NULL⟩); }

26 Value read() {
27   optional<Data> d = data.wLL();
28   if(d == empty) {
29     d = data.wLL();
30     if(d == empty) return old;}
31   if(d.ptr == NULL) {
32     data.CL();
33     return d.val; }
34   Value v = *(d.ptr);
35   if(data.SC(⟨v, NULL⟩)) return v;
36   d = data.wLL();
37   data.CL();
38   if(d != empty && d.ptr == NULL)
39     return d.val;
40   return old; } };
```

an **SC**. Finally, it reads the value **v** pointed to by **src** and tries to write (**v**, **NULL**) into (**D.val**, **D.ptr**) with an **SC**. It is not a problem if the **SC** fails because that means another process has helped complete the copy.

To **read** from **D**, a process begins by trying to read the pair (**D.val**, **D.ptr**) with a **wLL**. If it fails on this **wLL** twice, then it is safe to return **D.old** because the value of D has been updated at least once during this **read**. Now we focus on the case where one of the **wLL**s succeed and reads (**D.val**, **D.ptr**) into local variables (**val**, **ptr**). If **ptr** is NULL, then **val** stores the current value, which the **read** returns. If **ptr** is not NULL, then there is a concurrent **swcopy** operation and the **read** tries to help by reading the value **v** referenced by **ptr** and writing (**v**, **NULL**) into (**D.val**, **D.ptr**) with an **SC**. If the **SC** is successful, then the **read** returns **v**. Otherwise, the process performs one last **wLL**. If it is successful and sees that **D.ptr** is **NULL**, then it returns **D.val**. Otherwise, it is safe to return **D.old**.

The **write** operation is the most straightforward to implement. Since each **Destination** object only has a single writer, a **write** operation simply uses a **wLL** and an **SC** to store the new value into **D.val**. There cannot be any successful **SC** operations concurrent with the **wLL** because the other processes can only succeed on an **SC** during a **swcopy** operation. Therefore, the **wLL** and **SC** performed by the **write** will both always succeed. The **write** operations also needs to keep **D.old** up to date so it updates it before performing the **SC**.

Pseudo-code is shown in Listing 2. Calls to CL are inserted appropriately so that there is at most one outstanding **wLL** per process. By simple inspection of the pseudo-code, we can verify that each operation takes constant time. To implement M **Destination** objects, it uses M **WeakLLSC** objects, each spanning two words, and $O(M)$ pointer-width read, write, CAS objects. Using the algorithm from Result 3 to implement the **WeakLLSC** objects, we get an overall space usage of $O(M + P^2)$, which satisfies the properties in Result 2.

5.2 Correctness Proof

We begin by defining the linearization points of **write** and **swcopy**. The linearization point of **read** is more complicated, so we will defer its definition until later. Each **write** operation is linearized on line 24. For **swcopy** operations, we will prove in Claim 4 that there exists exactly one **SC** instruction S during the **swcopy** that sets **data.ptr** to **NULL** and that this instruction is either executed by line 18 of the **swcopy** or line 35 of a concurrent **read** R. If S is executed by line 18, then the **swcopy** is linearized when it executes line 15. Otherwise, the **swcopy** is linearized on line 34 of R. We show in Claim 5 that this linearization point is contained in the execution interval of the **swcopy**. Note that partially complete **swcopy** operations without an **SC** instruction setting **data.ptr** to **NULL** are not linearized.

For the purposes of this proof, we will focus on an execution E consisting of operations on a single **Destination** object **D**. For simplicity, we will write **data.ptr** instead of **D.data.ptr** and **swcopy** instead of **D.swcopy**. At each configuration C in E, we define the *current value* of **D** to be the value written by the last modifying operation (either a **write** or a **swcopy**) linearized before C. To show that the algorithm in Listing 2 is correct, it suffices to show that the value returned by each **read** operation is the value of **D** at some step during the **read**. The **read** is linearized at that step.

We first prove two useful claims about the structure of the algorithm. Throughout the proof, it is important to remember that there can only be one **write** or **swcopy** operation active at any time. We say that a pointer is *valid* if it is not **NULL**.

▷ Claim 2. Suppose the **SC** performed by a **read** operation is successful, then **data.ptr** was valid at all configurations between line 31 of the **read** and the **SC**.

Proof. Let R be a **read** operation with a successful **SC** operation S on line 35. Let L be the successful **wLL** operation corresponding to S. L was either executed on line 27 of R or line 29 of R. Since S is successful, **data.ptr** cannot have changed between L and S. If **data.ptr** was **NULL** in this interval, then the if statement on line 31 would have evaluated to true, and S would not have been executed. Therefore, **data.ptr** is valid at all configurations between L and S, which includes all configurations between line 31 of R and S. ◁

▷ Claim 3. Suppose the **SC** on line 18 of a **swcopy** operation is successful, then **data.ptr** is valid at all configurations between line 14 of the **swcopy** and the **SC**.

Proof. Let Y be a **swcopy** operation with a successful **SC** operation S on line 18. For S to be executed, the if statement on line 17 must evaluate to true, which means that the **wLL** operation L on line 16 must have been successful. Since S is a successful **SC**, **data.ptr** cannot have changed between L and S. Again, due to the if statement on line 17, **data.ptr** is valid in this interval.

It remains to show that **data.ptr** is valid between lines 14 and 16. Suppose for contradiction that **data.ptr** is **NULL** in this interval. The only operation that can change **data.ptr** to be valid is on line 14 of **swcopy**, so **data.ptr** would have remained **NULL** until the end of Y. This contradicts the fact that **data.ptr** is valid between L and S. Therefore **data.ptr** is valid at all configurations between line 14 of Y and S. ◁

The following two claims show that the linearization points of each **swcopy** operation is well-defined and lie within its execution interval.

▷ Claim 4. There is exactly one successful **SC** instruction during a **swcopy** Y that sets **data.ptr** to **NULL** and this **SC** instruction is either from line 18 of Y or line 35 of some **read**. Furthermore, this **SC** instruction is executed after the first **SC** of Y.

Proof. Let Y_i be the ith **swcopy** operation in E. The order is well defined because there can be only one **swcopy** operation active at a time. We proceed by induction on i, alternating between two different propositions. Let P_i be the proposition that **data.ptr** equals **NULL** at the start of Y_i. Let Q_i be the proposition that Claim 4 holds for Y_i. P_1 acts as our base case and for the inductive step, we show that P_i implies Q_i and that Q_i implies P_{i+1}.

For the base case, we know that **data.ptr** is initialized to **NULL** and it can only be changed to something that is valid by the first **SC** of a **swcopy** operation. Therefore, **data.ptr** remains **NULL** until the first **swcopy** operation.

To show that P_i implies Q_i, we use the same argument to argue that **data.ptr** is **NULL** between the first **wLL**/**SC** pair performed by Y_i. By Claim 2, no **SC** operation from a **read** can succeed between the first **wLL**/**SC** pair of Y_i. This means the first **SC** performed by Y_i (on line 14) is guaranteed to succeed and set **data.ptr** to something valid. Between the first **SC** of Y_i and the end of Y_i, the only two operations that could possibly change Y_i are the **SC** on line 18 of Y_i and line 35 of a **read** operation. During this interval, if there are no successful **SC** operations from line 35, then the **SC** on line 18 of Y_i is guaranteed to execute and succeed. This shows that there is at least one successful **SC** from line 18 or line 35 between the first **SC** and the end of Y_i. By Claim 3, the **SC** on line 18 cannot succeed if **data.ptr** is **NULL**, and similarly for the **SC** on line 35 (Claim 2). Since the **SC**s on lines 18 and 35 both set **data.ptr** to **NULL**, at most one such **SC** can succeed between the first **SC** of Y_i and the end of Y_i. Therefore, P_i implies Q_i.

All that remains is to show that Q_i implies P_{i+1}. From Q_i, we know that **data.ptr** gets set to **NULL** between the first **SC** of Y_i and the end of Y_i. It will remain **NULL** until the first **SC** of Y_{i+1}, which means it is **NULL** at the beginning of Y_{i+1}. ◁

▷ Claim 5. The linearization point of each **swcopy** operation Y lies between the first **SC** and the end of Y.

Proof. A **swcopy** operation Y is either linearized at line 15 of its own operation or line 34 of a **read** operation R. Clearly, this lemma holds in the former case, so we focus on the latter.

By Lemma 4, we know that the **SC** operation S on line 35 of R happens between the first **SC** of Y and the end of Y. This means that the successful **wLL** operation L corresponding to S must have happened after the first **SC** of Y and before S. From the code, we can see that line 34 of R (which is the linearization point of Y) happens between L and S. By transitivity, the linearization point of Y happens between the first **SC** of Y and the end of Y. ◁

The next claim is useful for arguing that **data.ptr** is **NULL** at all configurations during a **write** operation and at all configurations between the beginning and the first **SC** of a **swcopy**.

▷ Claim 6. **data.ptr** can only be valid between the first **SC** of a **swcopy** and the end of the **swcopy**.

Proof. **data.ptr** is initially **NULL** and the only instruction that sets **data.ptr** to something valid is the first **SC** of a **swcopy** instruction. By Claim 4, we know that after this **SC** instruction and before the end of the **swcopy**, **data.ptr** is set back to **NULL**. Therefore, **data.ptr** can only be valid between the first **SC** of a **swcopy** and the end of the **swcopy**. ◁

Finally, we prove the main claim.

▷ Claim 7. If **data.ptr** is **NULL**, then **data.val** stores the current value of **D**.

Proof. We will prove this by induction on the execution history E. The fields of **D** are initialized so that **data.ptr** stores **NULL** and **data.val** stores the initial value of **D**. Therefore this claim holds for the initial configuration. Suppose, for induction, that this claim holds for some configuration C, we need to show that it holds for the next configuration C'. If **D.data.ptr** is valid in C', then the claim is vacuously true, so suppose **D.data.ptr** is **NULL** at C'. Let S be the step between C and C'. There are four cases for S; either (1) S is a successful **SC** operation from line 18, (2) S is a successful **SC** operation from 35, (3) S is a successful **SC** operation from line 24, or (4) S is not a successful **SC** on **data**.

In the first case, S is executed by a **swcopy** operation Y, which is linearized on line 15 of Y. The value written into **data.val** by S is equal to the value of the source location at the linearization point of Y. There cannot be any **swcopy** or **write** operation linearized between the linearization point of Y and S, so **data.val** stores the current value of **D** at C'.

For the second case, we first show that S is concurrent with a **swcopy** operation. Due to the if statement on line 31, S can only be successful if **data.ptr** is valid. By Claim 6, **data.ptr** can only be valid during a **swcopy** operation, which means that S must occur during some **swcopy** operation Y. By Claim 4, we know that Y is linearized on line 34 of the **read** operation that executed S. Since there can only be one **swcopy** or **write** at a time, there cannot be any other **swcopy** or **write** operation linearized between the linearization point of Y and S. Since the value written into **data.val** by S is equal to the value of the source location at the linearization point of Y, **data.val** stores the current value of **D** at C'.

For case (3), S is the linearization point of a **write** operation and S writes the value of that **write** operation into **data.val**. This means **data.val** stores the current value of **D** at C'.

Finally, for the fourth case, suppose S is not a successful **SC** on **data**. This means the value of **data.val** will remain unchanged between C and C'. By the inductive hypothesis, **data.val** stores the current value of **D** at C, so it suffices to show that there are no **write** or **swcopy** operation linearized at S. By Claims 2 and 3, **data.ptr** is valid at the linearization point of a **swcopy** operation. Since **data.ptr** is **NULL** both before and after S, no **swcopy** operation can be linearized at S. To show that no **write** operations can be linearized at S, it suffices to show that the **SC** at the linearization point of a **write** operation is always successful. Let W be a **write** operation by process p. The only **SC** operations on **data** that can be concurrent with W are from **read** operations. By Claim 6, **data.ptr** is **NULL** for the duration of W, and by Claim 2, no **SC** from a **read** operation can succeed during W. Therefore, both the **wLL** and the **SC** performed by W are guaranteed to succeed. ◁

Suppose R is a completed **read** operation that returns v. As previously noted, to prove that Listing 2 is a linearizable implementation of a **Destination** object, it suffices to show that there exists a step during R such that the value of the **Destination** object at that step is equal to v. We linearize R at that step. If there are multiple operations linearized at the same step, **read** operations are always linearized last. Note that there cannot be multiple **write** or **swcopy** operations linearized at the same step.

There are five possible return points for a **read** operation. If R returns on lines 35 or 39, then on lines 35 or 36 (respectively), we know that **data.val** equals v and **data.ptr** equals **NULL**. If R returns on line 33, then either on line 27 or line 29, **data.val** equals v and **data.ptr** equals **NULL**. By Claim 7, **data.val** stores the current value of the **Destination** object whenever **data.ptr** is **NULL**, so for these three return points there exists a step during R such that v is the current value.

Now suppose R returns on lines 30 or 40 (i.e. the case where R reads and returns the value in **D.old**). There must have been two successful **SC**s, S_1 and S_2, on **D.data** during R. In the case where R returns on line 30, these two successful **SC** operations occurred during

the **wLL**s on lines 27 and 29. In the case where R returns on line 40, S_1 was the one that caused the **SC** on line 35 to fail and S_2 occurred during the **wLL** on line 36. By Claims 2 and 3, there cannot be two successful **SC**s from lines 18 or 35 in a row without a successful **SC** from line 14 of **swcopy** or line 24 of **write** in between. Therefore, there must have been a successful **SC** either from line 14 of **swcopy** or line 24 of **write** during R. We'll use S to denote this **SC** operation. In both cases, the line immediately before S updates **D.old** by first performing a **wLL** on **data**. By Claim 6, **data.ptr** equals **NULL** during this **wLL** operation and since the only **SC** operations that could potentially cause it to fail are by **read** operations, by Claim 2, this **wLL** is guaranteed to succeed. By Claim 7, **data.val** stores the current value v' at the time of this **wLL** operation. This value gets written into **old**, so **old** stores the current value immediately after this step. Since there is only a single **write** or **swcopy** at a time, **old** still contains the current value immediately before S. R reads and returns the value of **old** at its last step so there are two cases. Either R reads v' from **old** or it reads something newer. If R reads v', then it returns the current value of **D** at the step immediately before S (which happens during R). If R reads something newer, then **old** must have been updated between S and the end of R. This can only happen on line 13 or on line 23, and we've already argued that **old** stores the current value of **D** on these two lines. Therefore, in either case, R returns a value that was the current value of **D** at some point during R.

6 LL/SC from CAS

Now we have all the tools to implement LL/SC from CAS (Result 1). In this section, we present an algorithm that works whenever there is at most one outstanding LL per process. In the full version of this paper [11], we show how to generalize this to support k outstanding LLs per process.

This algorithm is almost identical to our algorithm for weak LL/SC from CAS (Section 4). To ensure that the **LL** operation always succeeds, we use **swcopy** to atomically read and announce the current buffer (lines 31 and 32 of Listing 1). This means that the announcement array needs to be an array of **Destination** objects (from Section 5.1) rather than raw pointers. Other than that, the algorithm remains the same. Note that **Destination** objects internally use Weak LL/SC objects, which in turn use an announcement array. The announcement array used by the Weak LL/SC objects is different from the one used by the LL/SC objects. Listing 3 shows just the difference between this algorithm and the weak LL/SC algorithm from Listing 1.

This algorithm uses $O((M + P^2)L)$ pointer-width read, write, CAS objects just like in Listing 1, but it also uses P **Destination** objects for the announcement array. From Result 2, we know that P **Destination** objects can be implemented in constant time and $O(P^2)$ space, so this algorithm achieves the bounds in Result 1.

6.1 LL/SC Correctness Proof (outline)

In the proof of correctness for our **WeakLLSC** algorithm, the key property is that at the linearization point of a successful **wLL** operation on a **WeakLLSC** object **X** by process p_i, both **A**[**i**] and **X.buf** point to the same buffer. We linearize our **LL** operation from Listing 3 so that the same property holds. At the linearization point of the **swcopy** operation on line 7, **A**[**i**] and **buf** are equal, so we linearize the **LL** at this point. **SC** and **VL** operations are linearized just as they were in the **WeakLLSC** algorithm. In the full version of the paper, we describe how to adapt our **WeakLLSC** proof to work for this algorithm.

Listing 3 Amortized constant time implementation of L-word LL/SC from CAS. Code for process p_i is shown. The algorithm is exactly the same as Listing 1 except for the parts that are shown. Note that the type of the announcement array has changed, so the way we **read** from and **write** to the announcement array is also different.

```
1  Destination<Buffer*> A[P];

3  struct LLSC {
4    Buffer* buf;
5    ...
6    Value[L] LL() {
7      A[i].swcopy(&buf);
8      Buffer* tmp = A[i].read();
9      return tmp->val; }
10 };
```

7 Conclusion and Discussion

We introduced a new primitive called **swcopy** and described how to implement it efficiently. We used this primitive to implement constant time LL/SC from CAS in a way that is both space efficient and avoids the use of unbounded sequence numbers. We believe the **swcopy** primitive can simplify the design of many other concurrent algorithms and make reasoning about them more modular.

We restricted the **Destination** objects in Section 5 to be single-writer because it was sufficient for the use cases we considered. It's possible to generalize this interface to support writes and copy operations that are concurrent with each other. However, it is unclear what the desired behavior should be in this case. One option would be to give atomic copy "store" semantics where the value of the **Destination** object is determined by the last **write** or **copy** to that location. Another option would be to give atomic copy "CAS" semantics where the **copy** is only successful if the **Destination** object stores the expected value. The right choice of definition will likely depend on the potential application.

Another interesting extension of the atomic copy primitive is to have it apply a function on the value being copied before writing it into the destination. This is similar to a Read-Modify-Write instruction except the read and the write are on two different memory locations.

Composing our LL/SC from CAS algorithm with Jayanti and Petrovic's multi-word LL/SC from single-word LL/SC algorithm [21] yields an implementation of multi-word LL/SC from CAS that uses $\Theta(P^2k + PML)$ space. This space complexity is sometimes better than the space complexity of our algorithm. In particular, If $k = M = O(1)$ and $L = \Theta(P)$, then the combined algorithm uses only $\Theta(P^2)$ space whereas our algorithm uses $\Theta(P^3)$ space.

References

1 Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Symposium on Theory of Computing (STOC)*, pages 538–547, 1995.

2 Zahra Aghazadeh, Wojciech Golab, and Philipp Woelfel. Making objects writable. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 385–395, 2014.

3 Zahra Aghazadeh and Philipp Woelfel. Space-and time-efficient long-lived test-and-set objects. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 404–419. Springer, 2014.

4 Zahra Aghazadeh and Philipp Woelfel. Upper bounds for boundless tagging with bounded objects. In *International Symposium on Distributed Computing (DISC)*, pages 442–457, 2016.

5 James H Anderson and Mark Moir. Universal constructions for large objects. In *International Workshop on Distributed Algorithms (WDAG)*, pages 168–182. Springer, 1995.

6 James H Anderson and Mark Moir. Universal constructions for multi-object operations. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 184–193, 1995.

7 Maya Arbel-Raviv and Trevor Brown. Reuse, don't recycle: Transforming lock-free algorithms that throw away descriptors. *arXiv preprint arXiv:1708.01797*, 2017.

8 Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

9 Henry G Baker Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

10 Greg Barnes. A method for implementing lock-free shared-data structures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 261–270, 1993.

11 Guy E. Blelloch and Yuanhao Wei. LL/SC and atomic copy: Constant time, space efficient implementations using only pointer-width CAS, 2019. `arXiv:1911.09671`.

12 Guy E. Blelloch and Yuanhao Wei. Concurrent reference counting and resource management in wait-free constant time, 2020. `arXiv:2002.07053`.

13 Simon Doherty, Maurice Herlihy, Victor Luchangco, and Mark Moir. Bringing practical lock-free synchronization to 64-bit applications. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 31–39. ACM, 2004.

14 Faith Ellen and Philipp Woelfel. An optimal implementation of fetch-and-increment. In *International Symposium on Distributed Computing (DISC)*, pages 284–298. Springer, 2013.

15 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

16 Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

17 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

18 IBM. IBM System/370 Extended Architecture, Principles of Operation, publication no. sa22-7085. 1983.

19 Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 151–160. ACM, 1994.

20 Prasad Jayanti and Srdjan Petrovic. Efficient and practical constructions of LL/SC variables. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 285–294. ACM, 2003.

21 Prasad Jayanti and Srdjan Petrovic. Efficient wait-free implementation of multiword LL/SC variables. In *IEEE International Conference on Distributed Computing Systems (ICSCS)*, pages 59–68. IEEE, 2005.

22 Prasad Jayanti and Srdjan Petrovic. Efficiently implementing a large number of LL/SC objects. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 17–31. Springer, 2005.

23 Prasad Jayanti and Srdjan Petrovic. Efficiently implementing LL/SC objects shared by an unknown number of processes. In *International Workshop on Distributed Computing (IWDC)*, pages 45–56. Springer, 2005.

24 Maged M Michael. ABA prevention using single-word instructions. *IBM Research Division, RC23089 (W0401-136), Tech. Rep*, 2004.

25 Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

26 Maged M Michael. Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In *International Symposium on Distributed Computing (DISC)*, pages 144–158. Springer, 2004.

27 Mark Moir. Practical implementations of non-blocking synchronization primitives. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 219–228, 1997.