8-19-2005

Efficient Wait-Free Algorithms for Implementing LL/SC Objects

Srdjan Petrovic
Dartmouth College

Efficient Wait-Free Algorithms for Implementing LL/SC Objects

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

in

Computer Science

by

Srdjan Petrovic

DARTMOUTH COLLEGE

Hanover, New Hampshire

August 19th, 2005

Examining Committee

(chair) Prasad Jayanti

Thomas H. Cormen

Maurice Herlihy

Douglas McIlroy

Charles K. Barlowe, Ph.D.
Dean of Graduate Studies

Abstract

Over the past decade, a pair of instructions called load-linked (LL) and store-conditional (SC) have emerged as the most suitable synchronization instructions for the design of lock-free algorithms. However, current architectures do not support these instructions; instead, they support either CAS (e.g., UltraSPARC, Itanium, Pentium) or restricted versions of LL/SC (e.g., POWER4, MIPS, Alpha). Thus, there is a gap between what algorithm designers want (namely, LL/SC) and what multiprocessors actually support (namely, CAS or restricted LL/SC). To bridge this gap, this thesis presents a series of efficient, wait-free algorithms that implement LL/SC from CAS or restricted LL/SC.

Acknowledgments

Prasad Jayanti, my Ph.D. advisor and now a great friend and mentor, I thank you for taking me on as a student. It is impossible for me to do justice, in any number of words, to your excellent guidance, friendship, and enthusiasm; I can only hope that I did not abuse your kindness and support. You have taught me so many things, including the research process and its excitement, writing papers, giving talks, teaching, mentoring students, reviewing papers, ..., the list does not end. All along the way, you have remained one of my closest and dearest friends. Prasad's wife, Aparna, and children Siddhartha and Sucharita have become my second family; I will always remember the careless times that I have spent at their home, and the basketball games we've played together.

I thank Tom Cormen, for his friendship and for his valuable suggestions regarding the writing of my thesis. Maurice Herlihy, I feel honored to have you on my thesis commitee: thank you for taking the time and for the encouragement. Doug McIlroy, I thank you sharing the enthusiasm for my research area and for reading so carefully through my thesis and sharing your comments.

In the five years that I have spent at Dartmouth, I had the good fortune of meeting and befriending several wonderful people, I thank them all. Udayan, Alin,

BJ, and Tim, thank you for the fun times.

I would never get so far without the support of my parents. Mama and Tata, I feel lucky to have you. Thank you for loving me and supporting me so steadfastly and unconditionally: I love you. I would also like to thank Mummy and Papa, my wife's parents, for their blessings and support.

Geeta, my love, you are my strength and inspiration. I thank you for the four most beautiful years of my life. I look forward to the journey that's ahead of us. Ila, my daughter, you have been a constant source of joy ever since you were born: I love you.

Contents

Chapter 1

Introduction

In shared-memory multiprocessors, multiple processes running concurrently on different processors cooperate with each other via shared data structures (e.g., queues, stacks, counters, heaps, trees). Atomicity of these shared data structures has traditionally been ensured through the use of locks. To perform an operation, a process obtains the lock, updates the data structure, and then releases the lock. Locking, however, has several drawbacks, including deadlocks (each of two processes waits for a lock currently held by the other), priority inversion (a low-priority process holds a lock needed by a high-priority process, and the low-priority process is preempted by a medium-priority process), and convoying (a descheduled process that holds a lock causes other processes to wait). Locking also limits parallelism: even when operations update disjoint parts of the data structure, they are applied sequentially, one after the other. Finally, lock-based implementations are not fault-tolerant: if a process crashes while holding a lock, other processes can end up waiting forever for the lock.

Wait-free implementations overcome most of the above drawbacks of locking

[Her91, Lam77, Pet83]. A wait-free implementation guarantees that every process completes its operation on the data structure in a bounded number of its steps, regardless of whether other processes are slow, fast, or have crashed. This bound (on the number of steps that a process executes to complete an operation on the data structure) is the *time complexity* (of that operation).

It is well understood that whether wait-free algorithms can be efficiently designed depends crucially on what synchronization instructions are available for the task. As we describe in the next section, the synchronization instructions supported by modern machines are not well suited for the task. The main goal of this thesis is to remedy this situation by implementing more useful synchronization instructions.

1.1 Weaknesses of hardware synchronization instructions

Most modern machines support either a *compare&swap* (CAS) instruction (e.g., UltraSPARC [Spa], Itanium [Ita02]), or *restricted LL/SC* (RLL/RSC) instructions (e.g., POWER4 [Pow01], MIPS [Mip02], Alpha [Sit92]). Neither of these instructions are well suited for the design of shared data structures. To understand why, we must first look at their semantics.

The instruction CAS(X, u, v) checks if location X has value u; if so, it changes the value to v and returns *true*, else it returns *false* and leaves the value unchanged. In practice, CAS is most commonly used as follows. First, we would read some value A from a location X; then, we would perform some computation (which may involve reading other locations) and compute the new value to be stored into X; finally, we would use CAS to attempt to change location X from A to the new value. Most often, our intent is for CAS to succeed only if between the read and

the CAS the location X hasn't been changed. However, it is quite possible that the location X changes from A to some value B, and then back to A again between the read and the CAS; in that case, CAS will succeed, even though our intent was for it to fail. This undesirable behavior is known in the literature as the ABA-problem [Ibm83], and it has greatly complicated the design of shared data structures.

Next, we turn to the instructions RLL/RSC, which are also supported on many modern machines. The RLL and RSC instructions act like read and conditional-write, respectively. More specifically, the RLL(X) instruction by process p returns the value of the location X, while the RSC(X, v) instruction by p checks whether some process updated the location X since p's latest RLL, and if no process has updated X it writes v into X and returns *true*; otherwise, it returns *false* and leaves X unchanged.

Due to their semantics, the RLL/RSC instructions do not suffer from the ABA-problem. However, they impose two severe restrictions on their use [Moi97a]: (1) there is a chance of *RSC* failing spuriously: *RSC* might fail even when it should have succeeded, and (2) a process is not allowed to access any shared variable between its *RLL* and the subsequent *RSC*. Due to these restrictions, it is hard to design algorithms based on this instruction pair.

1.2 Solution: LL/SC instructions

The instructions LL/SC have the same semantics as RLL/RSC, except that they do not impose any restrictions on their use. For this reason, they are very well suited for the design of shared data structures. Some examples of recent LL/SC-based lock-free algorithms are (1) the closed objects construction [CJT98], (2) the

construction of f-arrays [Jay02] and snapshots [Jay05, Jay02], (3) the abortable mutual exclusion algorithm [Jay03], and (4) some universal constructions [ADT95, Bar93, Her93, Moi97b, Moi01, ST95].

However, despite the desirability of LL/SC, no processor supports these instructions in hardware because it is impractical to maintain (in hardware) the state information needed to determine the success or failure of each process's SC operation on each word of memory. Thus, there is a gap between what algorithm designers want (namely, LL/SC) and what multiprocessors actually support (namely, CAS or RLL/RSC). To bridge this gap, we must efficiently emulate LL/SC instructions in software, which gives rise to the following research problem:

Research problem: Design a wait-free algorithm that implements LL/SC memory words from memory words supporting either CAS or RLL/RSC operations.

1.3 Implementing LL/SC from CAS

The most efficient algorithm for implementing LL/SC from CAS is due to Moir [Moi97a] (see Algorithm 1.1). His algorithm runs in constant time and has no space overhead. Below, we briefly describe how this algorithm works.

The algorithm stores a sequence number and a value together in the same machine word X. When a process p wishes to perform an LL operation, it first copies the current value of X into a local variable x_p (Line 1), and then returns the value stored in x_p (Line 2). During an SC(v) operation, p tries to change X from the value x_p it had witnessed in its latest LL operation to a value $(x_p.seq + 1, v)$ (Line 3). If p's CAS succeeds, then p's SC has succeeded and so p returns *true*; otherwise,

Types
xtype = **record** *seq*: 40-bit number; *val*: 24-bit value **end**
Shared variables
X: xtype
Local persistent variables at each $p \in \{0, 1, \ldots, N-1\}$
x_p: xtype

procedure LL($p, \mathcal{O}$) **returns** 24-bit value
1: $x_p = \mathtt{X}$
2: **return** $x_p.val$

procedure SC($p, \mathcal{O}, v$) **returns** boolean
3: **return** CAS(X, x_p, $(x_p.seq + 1, v)$)

Algorithm 1.1: Implementation of the N-process 24-bit LL/SC object $\mathcal{O}$ from a 64-bit CAS object and 64-bit registers, based on Moir's algorithm [Moi97a].

p's SC has failed and so p returns *false*.

It is easy to see that p's SC succeeds if and only if X hasn't changed since p's latest LL, i.e., if no other process performed a successful SC since p's latest LL. The only exception is the case when the sequence number in X is incremented sufficiently many times that it wraps around to the same value $x_p.seq$ that p had witnesses in its latest LL. In that case, p's SC will succeed even though the semantics of LL/SC mandate that it should fail. However, if sufficiently many bits in X are reserved for the sequence number (e.g., 32 to 40 bits), then the wraparound is extremely unlikely to occur in practice. Therefore, for practical purposes, the algorithm is a correct implementation of LL/SC from CAS.

Notice that, after reserving sufficiently many bits in X for the sequence number (e.g., 32 to 40 bits), only few bits are left in X for the value (e.g., 24 to 32 bits). Therefore, the algorithm presented above implements only a *small* (e.g., 24- to 32-bit) LL/SC object, which is inadequate for storing pointers, large integers, and doubles.

In this thesis, we focus on implementing *word-sized* LL/SC objects, i.e., objects

that can hold values of machine-word length (e.g., 64 bits). In order to implement such objects, we must store the value and the sequence number in separate machine words. This separation of value and sequence number makes it hard to ensure atomicity of concurrent operations, but our algorithms meet this challenge.

We also consider implementations of *multiword* LL/SC objects, i.e., objects whose value spans across multiple machine words (e.g., 512 or 1024 bits). Many existing applications [AM95a, CJT98, Jay02, Jay05] require support for such objects. In order to implement such objects, we must internally manage values that span across multiple machine words, using only single-word operations. Thus, these objects are more challenging to implement than word-sized objects.

1.4 Design considerations

In addition to time complexity and space complexity, the following three factors should be considered when designing LL/SC algorithms.

1. **Progress condition.** Wait freedom guarantees that a process completes its operation on the data structure in a bounded number of its steps, regardless of the speeds of other processes. A weaker form of implementation, known as *non-blocking* implementation [Lam77], guarantees that if a process p repeatedly takes steps, then the operation of *some* process (not necessarily p) will eventually complete. Thus, non-blocking implementations guarantee that the system as a whole makes progress, but admit starvation of individual processes. An even weaker form of implementation, known as *obstruction-free* implementation [HLM03a], guarantees that a process completes its operation on the data structure in a bounded number of its steps, provided that no other

process takes steps during that operation. This progress condition therefore allows for a situation where all processes starve. To reduce contention between operations (and thus reduce the chance of starvation), obstruction-free algorithms are often used in conjunction with a *contention manager*. A contention manager may, for example, use schemes such as exponential back-off or queuing to reduce contention [HLM03a].

Although wait freedom offers the strongest progress guarantees, it is generally difficult to design wait-free algorithms. Obstruction-free algorithms, on the other hand, offer the weakest progress guarantees but are much simpler to design. The tradeoff between wait-free algorithms on one side and non-blocking and obstruction-free algorithms on the other side is therefore that of progress guarantee versus simplicity of design. In this thesis, we focus our attention on wait-free algorithms.

2. **Knowing the number of processes in advance.** Most wait-free algorithms require that N—the maximum number of processes participating in the algorithm—is known in advance. They need this information in order to allocate and initialize all of their data structures in advance (i.e., before the algorithm starts). In situations where N cannot be known in advance, a conservative estimate has to be applied, which results in wasted space. It is therefore more desirable to have an algorithm that makes no assumptions on N and instead adapts to the actual number of participating processes.

   This problem of unknown N has been studied before [HLM03b], and there are many algorithms that adapt to the actual number of participating processes [DHLM04, DMMJ02, HLM03b, MH02, Mic04a, Mic04b, MS96]. We clas-

sify all such algorithms into three groups. In the first group are the algorithms whose space utilization at any time t is proportional to the number of processes participating in the algorithm at time t. An example of an algorithm in this group is a non-blocking LL/SC algorithm by Doherty, Herlihy, Luchangco, and Moir [DHLM04].

In the second group are the algorithms whose space utilization at any time t is proportional to the maximum number of processes that have *simultaneously* participated in the algorithm prior to time t. An example of an algorithm in this group is a wait-free LL/SC algorithm by Michael [Mic04b] and our LL/SC algorithm in Section 7.1 of Chapter 7.

In the third group are the algorithms whose space utilization at any time t is proportional to the maximum number of processes that have participated in the algorithm at *any* time prior to time t. An example of an algorithm in this group is our LL/SC algorithm in Section 7.2 of Chapter 7.

3. Unbounded vs. bounded sequence numbers. Most LL/SC algorithms in the literature associate a sequence number with each successful SC operation (e.g., Moir's algorithm [Moi97a] in Section 1.3). The purpose of a sequence number is to enable a process to detect whether an LL/SC object has been changed between its LL and SC operations. Some LL/SC algorithms use sequence numbers that grow without bound (e.g., [Moi97a]), while others use sequence numbers that are drawn from a finite set (e.g., [AM95b]). If sufficiently many bits are reserved for the sequence number (e.g., 32 to 40 bits), unbounded algorithms are just as good in practice as the bounded algorithms. Consider, for example, an unbounded LL/SC algorithm that uses 40-bit sequence numbers.

For this algorithm to behave incorrectly, a process would have to perform 2^{40} successful SC operations during the time interval when some other process executes one LL/SC pair. Unbounded sequence numbers are therefore not a practical concern.

1.5 Main results

We now summarize the main results of this thesis.

- *Word-sized LL/SC.* Our first result is a wait-free implementation of a word-sized LL/SC object from a word-sized CAS object and registers. We present three algorithms: the first algorithm uses unbounded sequence numbers; the other two use bounded sequence numbers. The time complexity of LL and SC in all three algorithms is $O(1)$, and the space complexity of the algorithms is $O(N)$ per implemented object, where N is the number of processes accessing the object. The space complexity of implementing M LL/SC objects is $O(NM)$, and the algorithms require N to be known in advance.

- *Multiword LL/SC.* Our second result is a wait-free implementation of a W-word LL/SC object (from word-sized CAS objects and registers). The time complexity of LL and SC is $O(W)$, and the space complexity of the algorithm is $O(NW)$ per implemented object, where N is the number of processes accessing the object. The space complexity of implementing M LL/SC objects is $O(NMW)$. This algorithm requires N to be known in advance, and it uses unbounded sequence numbers.

- *Multiword LL/SC for large number of objects.* Our third result is a wait-free

implementation of an array of M W-word LL/SC objects (from word-sized CAS objects and registers). This algorithm improves the space complexity of our multiword LL/SC algorithm (see above) when implementing a large number of LL/SC objects (i.e., when $M = \omega(N)$). In particular, the space complexity of the algorithm is $O((N^2 + M)W)$, where N is the number of processes accessing the objects. The time complexity of LL and SC is $O(W)$. The algorithm requires N to be known in advance, and it uses unbounded sequence numbers.

- *LL/SC algorithms for an unknown N.* Our fourth result is a wait-free implementation of an array of M W-word LL/SC objects (from word-sized CAS objects and registers), shared by an unknown number of processes. This algorithm supports two new operations—$Join(p)$ and $Leave(p)$—which allow a process p to join and leave the algorithm at any given time. If K is the maximum number of processes that have simultaneously participated in the algorithm (i.e., have joined the algorithm but have not yet left it), then the space complexity of the algorithm is $O((K^2 + M)W)$. The time complexities of procedures Join, Leave, LL, and SC are $O(K)$, $O(1)$, $O(W)$, and $O(W)$, respectively. The algorithm uses unbounded sequence numbers.

  We also present a wait-free implementation of a word-sized LL/SC object (from a word-sized CAS object and registers) that does not require N to be known in advance. The attractive feature of this algorithm is that the Join procedure runs in $O(1)$ time. This algorithm, however, does not allow processes to leave. The time complexities of LL and SC operations are $O(1)$, and the space complexity of the algorithm is $O(K^2 + KM)$. The algorithm uses

unbounded sequence numbers.

- *Multiword Weak-LL/SC for a large number of objects.* Our fifth result is a wait-free implementation of an array of M W-word *Weak-LL/SC* objects from word-sized CAS objects and registers.[1] The time complexity of WLL and SC is $O(W)$, and the space complexity of the algorithm is $O((N+M)W)$, where N is the number of processes accessing the object. The algorithm requires N to be known in advance, and it uses unbounded sequence numbers.

1.6 Using RLL/RSC instead of CAS

Although the main focus of this thesis is implementing LL/SC objects from CAS objects and registers, we note that most of our algorithms can easily be ported to machines that support RLL/RSC instructions, using Moir's implementation of CAS from RLL/RSC [Moi97a].

1.7 The VL operation

In practice, it is often useful to have a support for the *Validate* (VL) operation, which allows a process p to check whether some location X has been updated since p's latest LL, without modifying X. The VL operation returns *true* if X has not been updated; otherwise, it returns *false*. We include the support for the VL operation in all of our algorithms.

[1] A Weak-LL/SC object is the same as the LL/SC object, except that its LL operation—denoted WLL—is allowed to return to the invoking process p a special failure code (instead of the object's value) if the subsequent SC operation by p is sure to fail.

1.8 Notational conventions

Throughout the thesis, we use the following notation: N is the maximum number of processes for which the algorithm is designed, M is the number of implemented LL/SC objects, and k is the maximum number of outstanding LL operations of a given process (i.e., the maximum number of objects on which a process invoked an LL operation but did not yet invoke a matching SC).

1.9 Organization of the rest of the thesis

The remainder of the thesis is organized as follows. In Chapter 2, we present related work. Chapter 3 introduces the system model. Chapters 4–7 constitute the main body of the thesis, namely, algorithms (1)–(5) described earlier. Chapter 8 offers some concluding comments and future work. Finally, the proofs of all our algorithms are presented in Appendix A.

Chapter 2

Related work

In this chapter, we summarize the related work on implementing LL/SC from CAS. For each algorithm we list below, we describe briefly how it works and then present its main characteristics.

2.1 Implementations of LL/SC from CAS

The earliest wait-free algorithm for implementing an LL/SC object from a CAS object and registers is due to Israeli and Rappoport [IR94]. Their algorithm maintains a central variable X, and it stores N bits (along with the value) together in X. During an LL operation, process p writes 1 into the pth bit in X. Each successful SC operation writes 0s into *all* bits in X. By this strategy, process p can determine whether variable X has been changed (between p's LL and subsequent SC) by simply checking whether the pth bit in X is still 1. This approach, however, has the following two drawbacks: (1) it assumes that N bits can be stored (along with the value) into a single memory word, which may be unrealistic, and (2) since LL and

SC both write into variable X, an LL or an SC operation that fails to write into X must keep retrying until it succeeds. Although the algorithm manages to bound the number of retries to N, its time complexity is still an excessive $O(N)$. The space complexity of the algorithm is $O(N^2 + NM)$, and the knowledge of N is required.

Anderson and Moir [AM95b] present an algorithm that implements a small LL/SC object from a word-sized CAS object and registers. Their algorithm is essentially the same as Moir's algorithm [Moi97a] (see Section 1.3), but with an added mechanism for bounding the sequence numbers. This mechanism works as follows. During an LL operation, process p announces the sequence number that it reads from variable X. In each SC operation, p reads announcements by other processes and always chooses to write into X a sequence number that no other process had announced. It is easy to see that, if p reads a sequence number s from X during its LL operation and finds that X has the same sequence number during its subsequent SC, then X couldn't have been written in the meantime (because any process performing a successful SC would have noticed p's announcement and would therefore not have chosen to write s into X). The crux of the algorithm is the observation that p does not have to read *all* the announcements during a *single* SC operation. Instead, p reads announcements one at a time—in its ith SC operation, p reads (a single) announcement belonging to a process $i \bmod N$. Therefore, over N consecutive SC operations, p is sure to read the announcements of all processes. Moreover, the algorithm keeps announcements read during p's N latest SC operations in a special data structure that allows p to choose a new sequence number (that is different than all of the N announcements) in $O(1)$ time. Hence, using the above two strategies, the algorithm achieves $O(1)$ running time for the SC (in addition to $O(1)$ running time for the LL). The space complexity of the algorithm,

however, is significant: the algorithm must maintain N announcements for each process per each implemented LL/SC object. Hence, the space complexity of the algorithm is $O(N^2M)$. This algorithm requires N to be known in advance.

In a different paper, Anderson and Moir [AM95a] present an algorithm that implements a W-word LL/SC object $\mathcal{O}$ from a small LL/SC object and registers. Their algorithm maintains a central variable X that stores the location of two buffers: buffer A, which holds the current value of $\mathcal{O}$, and buffer B, which holds the previous value of $\mathcal{O}$. During an LL operation, process p first reads X to learn the location of the two buffers, and then reads the values from both buffers. The algorithm cleverly ensures that (1) at most one SC operation writes into the two buffers while p is reading them, and (2) at least one value that p reads from the two buffers will be a correct value for p's LL to return. To ensure these two properties, however, the algorithm must keep at least N buffers at each process (per each implemented variable). Hence, the space complexity of the algorithm is $O(N^2MW)$. The time complexity for LL and SC is $O(W)$, and the knowledge of N is required.

Moir [Moi97a] presents two algorithms that implement small LL/SC objects from word-sized CAS objects and registers. We have already described (in Section 1.3) his first algorithm. Its space complexity is $O(N + M)$, and the time complexity of LL and SC is $O(1)$. Moreover, the algorithm does not require N to be known in advance. The second algorithm is a variation of Anderson and Moir's bounded algorithm in [AM95b]. This algorithm reduces the space complexity of [AM95b] from $O(N^2M)$ to $O(N^2k + NM)$ by exploiting the fact that a process p can have at most k outstanding LL operations at any given time. The algorithm works as follows. Each process p keeps the announcements (of the se-

quence numbers) made during its k latest LL operations. In each SC operation, p reads all the announcements made by all other processes and always chooses to write into X a sequence number that no other process had announced. The benefit of this approach is that each process has to maintain one data structure over *all* M LL/SC objects (whereas in [AM95b], each process had to maintain one data structure per implemented LL/SC object). Hence, the space complexity of the algorithm is $O(N^2k + NM)$. Furthermore, by employing the same mechanism as in [AM95b], a process can avoid reading all the announcements during a single SC operation, thereby achieving a constant time complexity for both LL and SC. This algorithm requires the knowledge of N.

Luchangco, Moir, and Shavit [LMS03] present an algorithm that implements word-sized LL/SC objects from word-sized CAS objects and registers. Their algorithm maintains a central variable X for each implemented LL/SC object $\mathcal{O}$, which holds either (1) the current value of $\mathcal{O}$, or (2) a tag of the process that performed the latest LL on $\mathcal{O}$ (which consists of a sequence number and a process id). If X holds some process p's tag, then the current value of $\mathcal{O}$ is located in the pth location of the array VAL_SAVE. The first bit in X is reserved to distinguish between the above two types of values. For this reason, the algorithm implements an LL/SC object whose value is by 1 bit shorter than the length of the machine word (i.e., 63-bit LL/SC object on a 64-bit machine). The algorithm works as follows. During an LL operation, process p first obtains a valid value of $\mathcal{O}$ (either from X or from the array VAL_SAVE), and then attempts to write its tag into X. In each SC operation, p attempts to change the value in X from the tag it installed in X during the latest LL operation to the new value v. If the attempt succeeds, then p's SC has succeeded; otherwise, p's SC has failed. This approach, however, has the following

two drawbacks: (1) it allows an LL operation by some other process to fail p's SC operation, contrary to the specification of SC, and (2) since LL and SC both write into variable X, an LL or an SC operation that fails to write into X must keep retrying until it succeeds. Hence, the algorithm implements only a weaker form of LL/SC and is only obstruction-free (and not wait-free). The space complexity of the algorithm is $O(Nk + M)$. The algorithm uses unbounded sequence numbers and requires the knowledge of N.

Before we present the next two related works, we take a moment to explain the following simple algorithm that implements W-word LL/SC objects from word-sized CAS objects and registers (see Algorithm 2.1). This algorithm is based on the technique used in [Lea02] and was first described by Doherty, Herlihy, Luchangco, and Moir [DHLM04]. The algorithm maintains a single variable X for each LL/SC object $\mathcal{O}$. At all times, X points to the block containing the current value of $\mathcal{O}$. When a process p wishes to perform an LL operation, it first reads X to obtain the address of the block with the current value of $\mathcal{O}$ (Line 1), and then reads the value from that block (Line 2). During an SC(v) operation, p allocates a new block (Line 3), copies value v into it (Line 4), and then tries to swing the pointer in X from the address x_p it had witnesses in its latest LL operation to the address of the new block (Line 5). If p's CAS succeeds, then p's SC has succeeded and so it returns *true*; otherwise, p's SC has failed and so it returns *false*. It is easy to see that, since every SC uses a new block, p's SC succeeds if and only if X hasn't changed since p's latest LL, i.e., if no other process performed a successful SC since p's latest LL.

The above algorithm, although correct, uses an unbounded amount of memory and is therefore not practical. To bound the memory usage, blocks must be freed

Types
xtype = **pointer to array** $0 \,..\, W-1$ **of** 64-bit value
Shared variables
X: xtype
Local persistent variables at each $p \in \{0, 1, \ldots, N-1\}$
x_p: xtype

procedure LL(p, $\mathcal{O}$, *retval*)
1: $x_p = \mathtt{X}$
2: **copy** $*x_p$ **into** $*$*retval*

procedure SC(p, $\mathcal{O}$, v) **returns** boolean
3: *buf* = malloc(W)
4: **copy** $*v$ **into** $*$*buf*
5: **return** CAS(X, x_p, *buf*)

Algorithm 2.1: Implementation of the N-process W-word LL/SC object $\mathcal{O}$ from a 64-bit CAS object and 64-bit registers, based on the technique used in [Lea02].

from memory when they are no longer needed. Care must be taken, however, not to free a block that has been read by some process in its latest LL operation. To see why, suppose that some process p reads an address of a block B during its latest LL operation. Next, suppose that block B is freed from memory, and then allocated again by some process q during its SC operation. If q's SC succeeds, then the address of block B is again written into X. If p now performs an SC operation, p's SC will succeed even though it should have failed. Hence, blocks that have been read by some process in its latest LL operation must not be freed until that process performs a subsequent SC operation.

Michael [Mic04b] enforces the freeing rule by keeping a *hazard pointer* [Mic04a] at each process. When a process p reads an address of a block B (during its LL operation), it writes that address into its hazard pointer. The idea is that no other process will free block B from memory as long as p's hazard pointer holds B's address. In order to ensure this property, each process reads hazard pointers belonging to all other processes before attempting to free some block C from memory; if any process's hazard pointer holds the address of C, then C is not freed

until later. The algorithm exploits the fact that a process p can have at most k outstanding LL operations at any point in time. Thus, each process keeps k hazard pointers, and all k hazard pointers at each process are read before a block is freed from memory. Similar to [AM95b], a process does not read all the announcements at once. Instead, it uses an amortized approach of reading announcements one at a time. However, unlike [AM95b], which works with sequence numbers, this algorithm works with memory pointers. For this reason, the algorithm cannot ensure the $O(1)$ running time for its SC operation.[1] Instead, each SC operation runs in $O(1)$ *expected amortized time*: the first $(Nk - 1)$ SC operations by each process take $O(1)$ time, and the (Nk)th operation takes $O(Nk)$ expected time. The "expected" running time of the (Nk)th operation comes from the fact that there is hashing involved in the execution of that operation. If all entries are hashed perfectly, then the operation runs in $O(Nk)$ time; if all entries are hashed to the same location (a highly unlikely scenario), the operation runs in $O((Nk)^2)$ time. The space complexity of the algorithm is $O(Nk + (N^2k + M)W)$, and the algorithm does not require N to be known in advance.

Doherty et al. [DHLM04], on the other hand, ensure that a block is not freed from memory (if some process has read it in its latest LL operation) by keeping a counter in each block. When a process p reads an address of a block B (during its LL operation), it increments the counter stored in block B. Likewise, during its subsequent SC operation, p decrements the counter stored in B. By the above strategy, once a block's counter reaches a value zero, it means that no process has read that block in its latest LL operation and thus the block can be freed. The counter management, however, is more complicated than described above, because the fol-

[1] We assume here that $W = 1$.

lowing undesirable scenario must be prevented: (1) p reads an address of a block B and then sleeps for a long time; (2) in the meantime, block B is freed from memory; (3) then, p attempts to increment a counter stored in B. This attempt will result in an illegal memory access, since the memory occupied by B has already been freed. To overcome this problem, the algorithm maintains a static counter at each LL/SC object. During an LL operation, process p increments this static counter *instead of a counter inside the block*. Once a successful SC operation replaces block B with a newer one, it copies the value of the static counter into B's counter and then resets the static counter back to zero. With regards to the space complexity, notice that if there are T outstanding LL operations (over all N processes) at some time t, then there are at most T blocks with counters greater than zero at time t. Hence, the space utilization of the algorithm at time t is $O((T + M)W)$. Since in the worst case T is as high as Nk, it means that the space complexity of the algorithm is $O((Nk + M)W)$. This algorithm, however, is only non-blocking and not wait-free. The algorithm does not require N to be known in advance. Finally, we note that, although the algorithm was originally presented as a word-sized LL/SC implementation, it can be trivially extended to a W-word LL/SC implementation.

Table 2.1 summarizes the above discussion on related work and compares it to the results presented in this thesis. The results are listed in chronological order.

Algorithm	*Size of LL/SC*	*Progress Condition*	*Worst-case Time Complexity*	
			LL	*SC*
1. Israeli and Rappoport [IR94], Fig. 3	small	wait-free	$O(N)$	$O(N)$
2. Anderson and Moir [AM95b], Fig. 1	small	wait-free	$O(1)$	$O(1)$
3. Anderson and Moir [AM95a], Fig. 2	W-word	wait-free	$O(W)$	$O(W)$
4. Moir [Moi97a], Fig. 4	small	wait-free	$O(1)$	$O(1)$
5. Moir [Moi97a], Fig. 7	small	wait-free	$O(1)$	$O(1)$
6. Luchangco et al. [LMS03]†	63-bit	obstr.-free	–	–
7. This thesis, Chapter 4	word-sized	wait-free	$O(1)$	$O(1)$
8. Doherty et al. [DHLM04]	W-word	non-blocking	–	–
9. Michael [Mic04b]	W-word	wait-free	$O(W)$	$O(Nk+W)$‡
10. This thesis, Chapter 5	W-word	wait-free	$O(W)$	$O(W)$
11. This thesis, Chapter 6	W-word	wait-free	$O(W)$	$O(W)$
12. This thesis, Sec. 7.1 of Chapter 7	W-word	wait-free	$O(W)$	$O(W)$
13. This thesis, Sec. 7.2 of Chapter 7	word-sized	wait-free	$O(1)$	$O(1)$

Algorithm	*Space Complexity*	*Knowledge of N*	*Bounded or Unbounded*
1. Israeli and Rappoport [IR94], Fig. 3	$O(N^2+NM)$	required	bounded
2. Anderson and Moir [AM95b], Fig. 1	$O(N^2M)$	required	bounded
3. Anderson and Moir [AM95a], Fig. 2	$O(N^2MW)$	required	bounded
4. Moir [Moi97a], Fig. 4	$O(Nk+M)$	not required	unbounded
5. Moir [Moi97a], Fig. 7	$O(N^2k+NM)$	required	bounded
6. Luchangco et al. [LMS03]	$O(Nk+M)$	required	unbounded
7. This thesis, Chapter 4	$O(NM)$	required	bounded
8. Doherty et al. [DHLM04]	$O((Nk+M)W)$	not required	unbounded
9. Michael [Mic04b]	$O(Nk+(N^2k+M)W)$	not required	unbounded
10. This thesis, Chapter 5	$O(NMW)$	required	unbounded
11. This thesis, Chapter 6	$O(Nk+(N^2+M)W)$	required	unbounded
12. This thesis, Sec. 7.1 of Chapter 7	$O(Nk+(N^2+M)W)$	not required	unbounded
13. This thesis, Sec. 7.2 of Chapter 7	$O(N^2+NM)$	not required	unbounded

Table 2.1: A comparison of algorithms that implement LL/SC from CAS.

†This algorithm implements a weaker form of LL/SC in which an LL operation by a process can cause some other process's SC operation to fail.

‡The *amortized* running time for SC is $O(W)$. We note that this algorithm uses hashing and that the presented running times for SC apply to the best case where all entries hash into different locations. In the worst case where all entries hash into the same location, SC operation runs in $O((Nk)^2+W)$ worst-case and $O(Nk+W)$ amortized time.

2.2 Implementations of Weak-LL/SC from CAS

The Weak-LL/SC object was first defined by Anderson and Moir [AM95a], who also present an algorithm that implements a W-word Weak-LL/SC object $\mathcal{O}$ from single-word LL/SC objects and registers. This algorithm works as follows. Each process p maintains two W-word buffers. One buffer holds the value written into $\mathcal{O}$ by p's latest successful SC, and the other buffer is available for use in p's next SC operation. The central variable X holds a pair (b, q), where q is the id of the process that performed the latest SC operation on $\mathcal{O}$, and b is the index of q's buffer that holds the value written by that SC. To perform an SC operation, p first writes the value into its available buffer, and then attempts to write the pointer to that buffer into X (i.e., it writes a pair (b, p) into X, where b is the index of p's available buffer). During a WLL operation, p (1) reads X to obtain a pointer to the buffer that holds the current value of $\mathcal{O}$, (2) reads that buffer, and (3) checks whether X has been modified since p's latest read. If X hasn't been modified, then p is certain that it had read a valid value from the buffer, and so it simply returns that value signaling the success of its WLL operation. Otherwise, some process must have performed a successful SC during p's WLL. Then, by the definition of WLL, p is not obligated to return a value; instead, it can signal failure and return the id of a process that performed a successful SC during p's WLL. Such an id can be obtained simply by reading X. So, p reads X and returns the id obtained, also signaling the failure of its WLL operation. The space complexity of the above algorithm is $O(NMW)$, and the time complexity for WLL and SC is $O(W)$. The algorithm requires N to be known in advance.

Moir [Moi97a] presents an algorithm that implements an array $\mathcal{O}0\,..\,M$ of M

W-word Weak-LL/SC objects from CAS objects and registers. The algorithm maintains a central array $\mathtt{X}0 \ldots M$ that holds the following information for each object $\mathcal{O}i$: (1) the current value of $\mathcal{O}i$, and (2) the id of the process that wrote that value in $\mathcal{O}i$. To perform an SC operation on some object $\mathcal{O}i$, p first writes the value into its local buffer, and then attempts to write its id into the "id" field of $\mathtt{X}i$. If p's attempt succeeds, then p's SC has succeeded. Before returning from its operation, however, p first copies the value from its buffer into the "value" field of $\mathtt{X}i$. If p is slow in copying, then other processes that execute WLL operations on $\mathcal{O}i$ will help p copy that value into $\mathtt{X}i$. In particular, during an WLL operation on some object $\mathcal{O}i$, p first reads the "id" field of $\mathtt{X}i$ to learn the id q of the process that wrote the current value in $\mathcal{O}i$, and then helps q copy the value from q's buffer into the "value" field of $\mathtt{X}i$. While helping q, p also reads the current value of $\mathcal{O}i$. Periodically, p checks the "id" field of $\mathtt{X}i$ to see whether some process has performed an SC operation on $\mathcal{O}i$. If so, p abandons its helping and returns the id of that process, signaling the failure of its WLL operation. Otherwise, p returns the value it had obtained from $\mathtt{X}i$, also signaling the success of its WLL operation. The space complexity of the algorithm is $O(Nk + (N+M)W)$, and the time complexity for WLL and SC is $O(W)$. The algorithm, however, has the following drawback: in order to facilitate the helping scheme by which a process p helps a process q copy the value from its buffer into $\mathtt{X}i$, the values must be copied word-by-word using a CAS operation. Hence, the algorithm has an excessive *CAS-time-complexity* of $O(W)$ for both WLL and SC. (Since CAS is a costly operation in multiprocessors, it is important to keep the CAS-time-complexity of operations small.) The algorithm uses bounded sequence numbers and requires N to be known in advance.

Table 2.2 summarizes the above discussion on related work and compares it

to the Weak-LL/SC algorithm presented in this thesis. The results are listed in chronological order.

Algorithm	*Size of WLL/SC*	*Progress Condition*	*Worst-case Time Complexity*		*CAS-time Complexity*	
			WLL	*SC*	*WLL*	*SC*
1. Anderson and Moir [AM95a], Fig. 1	W-word	wait-free	$O(W)$	$O(W)$	$O(1)$	$O(1)$
2. Moir [Moi97a], Fig. 6	W-word	wait-free	$O(W)$	$O(W)$	$O(W)$	$O(W)$
3. This thesis, Chapter 6	W-word	wait-free	$O(W)$	$O(W)$	$O(1)$	$O(1)$

Algorithm	*Space Complexity*	*Knowledge of N*	*Bounded or Unbounded*
1. Anderson and Moir [AM95a], Fig. 1	$O(NMW)$	required	bounded
2. Moir [Moi97a], Fig. 6	$O(Nk + (N + M)W)$	required	bounded
3. This thesis, Chapter 6	$O(Nk + (N + M)W)$	required	unbounded

Table 2.2: A comparison of algorithms that implement Weak-LL/SC from CAS.

Chapter 3

System model

In this chapter, we describe the system model that we will use in the rest of the thesis. We use Herlihy's [Her93] *concurrent system* model, defined as follows.

A *concurrent system* consists of a collection of *processes* communicating with each other through shared *objects*. Processes are *asynchronous*—they execute at different speeds and may halt at any given time. A process cannot tell whether another process has halted or is running very slowly. The object is a data structure in memory. Each object has a *type*, which defines a set of possible *values* for the object and a set of *operations* that can be applied on the object. A process interacts with the object by invoking operations on the object and receiving associated responses. Processes are *sequential*—each process invokes a new operation only after receiving a response to the previous invocation.

Each object has a *sequential specification* that specifies how the object behaves when all the operations on the object are applied sequentially. For example, a sequential specification of a read/write register specifies that a *read* operation must return the value written by the latest *write* operation. In a concurrent system, op-

erations from different processes may overlap, making the sequential specification insufficient for understanding the behavior of an object. *Linearizability*, defined by Herlihy and Wing [HW90], is a widely accepted criterion for the correctness of a shared object. An object is *linearizable* if operations applied to the object appear to act instantaneously, even though in reality each operation executes over an interval of time. More precisely, every operation applied to the object appears to take effect at some instant between its invocation and response. This instant (at which an operation appears to take effect) is called the *linearization point* for that operation.

An *implementation* of an object $\mathcal{O}$ from some other objects $O_1, O_2, \ldots, O_n$ is a collection of algorithms—one algorithm per operation on $\mathcal{O}$—defined in terms of the operations of $O_1, O_2, \ldots, O_n$. We call $\mathcal{O}$ a *derived* object and $O_1, O_2, \ldots, O_n$ *base* objects. The *space complexity* of an implementation of $\mathcal{O}$ is the number of base objects used by the implementation. A *step* in the execution of $\mathcal{O}$'s operation corresponds to invoking an operation on the base object and receiving the associated response. An *execution history* of an implementation of $\mathcal{O}$ is a sequence of steps taken by all processes while executing $\mathcal{O}$'s operations. The *time complexity* of $\mathcal{O}$'s operation is the maximum number of steps taken by any process while executing that operation.

Chapter 4

Word-sized LL/SC

In this chapter, we present three algorithms that implement a word-sized LL/SC object from word-sized CAS objects and registers. The first algorithm uses unbounded sequence numbers and is presented in Section 4.1. The other two algorithms use bounded sequence numbers and are presented in Section 4.2.

All three algorithms implement an LL/SC object whose size is as big as the memory word of the underlying architecture. For example, on a 64-bit architecture supporting CAS, our algorithms implement a 64-bit LL/SC object; on a 128-bit architecture, they implement a 128-bit LL/SC object. To make the reading easier, in the rest of the chapter we assume a word size of 64 bits, but note that the algorithms are good for any word size.

4.1 An unbounded 64-bit LL/SC implementation

Algorithm 4.1 implements a 64-bit LL/SC object from a CAS object and registers. We begin by providing an intuitive description of how this algorithm works.

Types

valuetype = 64-bit number
seqnumtype = $(64 - \log N)$-bit number
tagtype = **record** *pid*: $0 \,..\, N-1$; *seqnum*: seqnumtype **end**

Shared variables

X: tagtype (X supports *read* and *CAS* operations)
For each $p \in \{0, \ldots, N-1\}$, we have four single-writer, multi-reader registers:
$\texttt{val}_p0$, $\texttt{val}_p1$, $\texttt{oldval}_p$: valuetype
$\texttt{oldseq}_p$: seqnumtype

Local persistent variables at each $p \in \{0, \ldots, N-1\}$

tag_p: tagtype; seq_p: seqnumtype

Initialization

X = (0, 1); $\texttt{val}_0 1 = v_{init}$, the desired initial value of $\mathcal{O}$
$\texttt{oldseq}_0 = 0$; $seq_0 = 2$
For each $p \in \{1, \ldots, N-1\}$ $seq_p = 1$

procedure LL(p, $\mathcal{O}$) **returns** valuetype

1: $tag_p = \texttt{X}$
Let $(q, k) = (tag_p.pid, tag_p.seqnum)$
2: $v = \texttt{val}_q k \bmod 2$
3: $k' = \texttt{oldseq}_q$
4: **if** $(k' = k-2) \vee (k' = k-1)$ **return** v
5: $v' = \texttt{oldval}_q$
6: **return** v'

procedure SC(p, $\mathcal{O}$, v) **returns** boolean

7: $\texttt{val}_p seq_p \bmod 2 = v$
8: **if** CAS(X, tag_p, (p, seq_p))
9: $\texttt{oldval}_p = \texttt{val}_p(seq_p - 1) \bmod 2$
10: $\texttt{oldseq}_p = seq_p - 1$
11: $seq_p = seq_p + 1$
12: **return** *true*
13: **else return** *false*

procedure VL(p, $\mathcal{O}$) **returns** boolean

14: **return** $\texttt{X} = tag_p$

Algorithm 4.1: An unbounded implementation of the 64-bit LL/SC object $\mathcal{O}$ using a 64-bit CAS object and 64-bit registers

4.1.1 How the algorithm works

The algorithm implements a 64-bit LL/SC object $\mathcal{O}$. Central to the implementation is the variable X that supports *CAS* and *read* operations. In addition, there are four shared registers at each process p—$\texttt{val}_p0$, $\texttt{val}_p1$, $\texttt{oldval}_p$ and $\texttt{oldseq}_p$—that are written to only by p but may be read by any process. The meanings of these variables are described as follows.

The algorithm associates a tag with every successful SC operation on $\mathcal{O}$. A

tag consists of a process id and a sequence number. Specifically, the tag associated with a successful SC operation is (p, k) if it is the kth successful SC operation by process p. The variable X always contains the tag corresponding to the latest successful SC.

Suppose that the current value of X is (p, k) (which means that the last successful SC was performed by p and p performed k successful SC operations so far). The algorithm ensures that the value written by the kth successful SC by p is in $\texttt{val}_p 0$ if k is even, or in $\texttt{val}_p 1$ if k is odd; i.e., the value is made available in $\texttt{val}_p k \bmod 2$. The registers $\texttt{oldval}_p$ and $\texttt{oldseq}_p$ hold an older value and its sequence number, respectively. Specifically, if p has so far performed k successful SC operations, $\texttt{oldseq}_p$ and $\texttt{oldval}_p$ contain, respectively, the number $k-1$ and the value written by the $(k-1)$th successful SC by p.

In addition to the shared variables just described, each process p has two persistent local variables, seq_p and tag_p, described as follows. The value of seq_p is the sequence number of p's next SC operation: if p has performed k successful SC operations so far, seq_p has the value $k+1$. (Thus, sequence numbers in our algorithm are local: p's sequence number is based on the number of successful SC's performed by p, not by the system as a whole.) The value of tag_p is the value of X read by p in its latest LL operation.

Given this representation, the variables are initialized as follows. Let v_{init} denote the desired initial value of the implemented object $\mathcal{O}$. We pretend that process 0 performed an "initializing SC" to write the value v_{init}. Accordingly, X is initialized to (0, 1), $\texttt{val}_0 1$ to v_{init}, $\texttt{oldseq}_0$ to 0, and seq_0 to 2. For each process $p \neq 0$, seq_p is initialized to 1. All other variables are arbitrarily initialized.

We now explain the procedure $\texttt{SC}(p, \mathcal{O}, v)$ that describes how process p per-

forms an SC operation on $\mathcal{O}$ to attempt to change $\mathcal{O}$'s value to v. First, p makes available the value v in $\texttt{val}_p 0$ if the sequence number is even, or in $\texttt{val}_p 1$ if the sequence number is odd (Line 7). Next, p tries to make its SC operation take effect by changing the value in X from the tag that p had witnessed in its latest LL operation to the tag corresponding to its current SC operation (Line 8). If the CAS operation fails, it follows that some other process performed a successful SC after p's latest LL. In this case, p's SC must fail. Therefore, p terminates its SC procedure, returning *false* (Line 13). On the other hand, if CAS succeeds, then p's current SC operation has taken effect. To remain faithful to the previously described meanings of the variables $\texttt{oldval}_p$ and $\texttt{oldseq}_p$, p writes in $\texttt{oldval}_p$ the value written by p's earlier successful SC (Line 9) and writes in $\texttt{oldseq}_p$ the sequence number of that SC (Line 10). (Since the sequence number for p's current successful SC is seq_p, it follows that the sequence number for p's earlier successful SC is $seq_p - 1$, and the value written by that SC is in $\texttt{val}_p(seq_p - 1) \bmod 2$; this justifies the code on Lines 9 and 10.) Next, p increments its sequence number (Line 11) and signals successful completion of the SC by returning *true* (Line 12).

We now turn to the procedure $\texttt{LL}(p, \mathcal{O})$ that describes how process p performs an LL operation on $\mathcal{O}$. In the following, let $\text{SC}_{q,i}$ denote the ith successful SC by process q and $v_{q,i}$ denote the value written in $\mathcal{O}$ by $\text{SC}_{q,i}$. First, p reads $\texttt{X}$ to obtain the tag (q, k) corresponding to the latest successful SC operation, $\text{SC}_{q,k}$ (Line 1). Since $\text{SC}_{q,k}$ wrote $v_{q,k}$ into $\texttt{val}_q k \bmod 2$, and since $\texttt{val}_q k \bmod 2$ is not modified until q initiates an SC operation with $seq_q = k + 2$, it follows that at the instant when p performs Line 1, the variable $\texttt{val}_q k \bmod 2$ holds the value $v_{q,k}$. Furthermore, the value of $\texttt{val}_q k \bmod 2$ is guaranteed to be $v_{q,k}$ until q completes $\text{SC}_{q,k+1}$.

So, in an attempt to learn $v_{q,k}$, p reads $\texttt{val}_q k \bmod 2$ (Line 2). By the observation in the previous paragraph, if p is not too slow and executes Line 2 before q completes $\mathrm{SC}_{q,k+1}$, the value v read on Line 2 will indeed be $v_{q,k}$. Otherwise the value v cannot be trusted. To resolve this ambiguity, p must determine if q has completed $\mathrm{SC}_{q,k+1}$ yet. To make this determination, p reads the sequence number k' in $\texttt{oldseq}_q$ (Line 3). If $k' = k-2$ or $k' = k-1$, it follows that $\mathrm{SC}_{q,k+1}$ has not yet completed even if it had been already initiated (because, by Line 10, $\mathrm{SC}_{q,k+1}$ writes k into $\texttt{oldseq}_q$). It follows that the value v obtained on Line 2 *is* $v_{q,k}$. So, p terminates the LL operation, returning v (Line 4).

If $k' \geq k$, q must have completed $\mathrm{SC}_{q,k+1}$, its $(k+1)$th successful SC. It follows that the value in $\texttt{oldval}_q$ is $v_{q,k}$ or a later value (more precisely, the value in $\texttt{oldval}_q$ is $v_{q,i}$ for some $i \geq k$). Therefore, the value in $\texttt{oldval}_q$ is a legitimate value for p's LL to return. Accordingly, p reads the value v' of $\texttt{oldval}_q$ (Line 5) and returns it (Line 6). Although v' is a recent enough value of $\mathcal{O}$ for p's LL to legitimately return, it is important to note that v' is not the current value of $\mathcal{O}$. This is because the algorithm moves a value into $\texttt{oldval}_q$ only after it is no longer the current value. Since the value v' that p's LL returns on Line 6 is not the current value, p's subsequent SC must fail (by the specification of LL/SC). Our algorithm satisfies this requirement because, when p's subsequent SC performs Line 8, the CAS operation fails since tag_p is (q,k) and the value of X is not (q,k) anymore (the value of X is not (q,k) because, by the first sentence of this paragraph, q has completed its $(k+1)$th successful SC). This completes the description of how LL is implemented.

The VL operation by p is simple to implement: p returns *true* if and only if the tag in $\texttt{X}$ has not changed since p's latest LL operation (Line 14).

In our discussion above, we assumed that the sequence numbers never wrap around. This assumption is not a concern in practice, as we now explain. The 64-bit variable X stores in it a process number and a sequence number. Even if there are as many as 16,000 processes sharing the implementation, only 14 bits are needed for storing the process number, leaving 50 bits for the sequence number. In our algorithm, for seq_p to wrap around, p must perform 2^{50} successful SC operations. If p performs a million successful SC operations each second, it takes 32 years for seq_p to wrap around! Thus, wraparound is not a practical concern.

In fact, as we show in Appendix A.1.1, the algorithm works correctly even in the presence of a wraparound, provided that the following weaker assumption holds:

Assumption A: If s is the number of bits in X reserved for the sequence number, then during the time interval when some process p executes one LL/SC pair, no other process q performs more than $2^s - 3$ successful SC operations.

It is easy to see that for $s = 50$, a process p would have to spend 32 years executing a single LL/SC pair for the algorithm to behave incorrectly. Thus, Assumption A is weak enough not to be a practical concern.

Based on the above discussion, we have the following theorem. Its proof is given in the Appendix A.1.1.

Theorem 1 *Algorithm 4.1 is wait-free and, under Assumption A, implements a linearizable 64-bit LL/SC object from a single 64-bit CAS object and an additional six registers per process. The time complexity of LL, SC, and VL is* $O(1)$.

4.1.2 Remark: using RLL/RSC instead of CAS

Using Moir's idea [Moi97a], it is straightforward to replace the CAS instruction (on Line 8 of the algorithm) with RLL/RSC instructions. Specifically, keep Line 7 and Lines 9-13 of the algorithm as they are and replace Line 8 with the following code fragment. The repeat-until loop in this code fragment handles spurious `RSC` failures and terminates in a single iteration if there are no such failures.

8: *flag* = *false*
repeat
if RLL(X) $\neq$ tag_p **go to** L
flag = RSC(X, (p, seq_p))
until *flag*
L: **if** (*flag*)

4.1.3 Remark: implementing read, write

It is straightforward to extend Algorithm 4.1 to implement read and write operations *in addition to* LL, SC and VL operations. Specifically, the implementation of Write(p, $\mathcal{O}$, v) is the same as the implementation of `SC`(p, $\mathcal{O}$, v) with the following changes: replace the CAS on Line 8 with *write*(`X`, (p, seq_p)) and remove Lines 12 and 13. The implementation of Read(p, $\mathcal{O}$) is the same as the code for LL, except that on Line 1 the value of `X` is read into a local variable, different from tag_p (so that the read operation doesn't affect the success of the subsequent SC). The code for LL, SC and VL operations remains the same as in Algorithm 4.1.

It has been shown elsewhere that it is often algorithmically not easy to incorporate the write operation into LL/SC constructions [Jay98]. Thus, it is an interesting feature of our algorithm that it can be extended effortlessly to support the write operation.

4.2 A bounded 64-bit LL/SC implementation

For the rest of this chapter, the goal is to design a *bounded* algorithm that implements a 64-bit LL/SC object using 64-bit CAS objects and 64-bit registers. We achieve this goal in four steps:

1. Implement a 64-bit LL/SC object from a 64-bit WLL/SC object and 64-bit registers.

2. Implement a 64-bit WLL/SC object from a (1-bit, pid)-LL/SC object (which will be described later).

3. Implement a (1-bit, pid)-LL/SC object from a 64-bit CAS object and registers.

4. This step is trivial: simply compose the implementations of the above three steps. This composition results in an implementation of a 64-bit LL/SC object from 64-bit CAS objects and 64-bit registers.

As we will show in the next three sections, the implementations of the first three steps have $O(1)$ time complexity and $O(1)$ space overhead per process. As a result, the 64-bit LL/SC implementation obtained in the fourth step also has $O(1)$ time complexity and $O(1)$ space overhead per process, as desired.

4.3 Implementing 64-bit LL/SC from 64-bit WLL/SC

Recall that a WLL operation, unlike LL, is not always required to return the value of the object: if the subsequent SC operation is sure to fail, the WLL may simply return the identity of a process whose successful SC took effect during the execution of that WLL. Thus, the return value of WLL is of the form (flag, v), where either (1) $\mathit{flag} = \mathit{success}$ and v is the value of the object $\mathcal{O}$, or (2) $\mathit{flag} = \mathit{failure}$ and v is the id of a process whose SC took effect during the WLL.

Algorithm 4.2 implements a 64-bit LL/SC object $\mathcal{O}$. The algorithm uses a single 64-bit WLL/SC variable `X` and, for each process p, a single 64-bit atomic register $\texttt{lastVal}_p$. Note that the invocation of SC at Line 11 (in the implementation of SC) is not a recursive call to the SC procedure, but is merely an invocation of the SC operation on the underlying WLL/SC object `X`. Likewise, Line 12 is not a recursive call to the VL procedure.

We explain how the algorithm works in Section 4.3.2, but fist draw the reader's attention to two conditions that any implementation of an LL operation must satisfy in order to ensure correctness.

4.3.1 Two obligations of LL

Consider an execution of the LL procedure by a process p. Suppose that v is the value of $\mathcal{O}$ when p invokes the LL procedure and suppose that k successful SC's take effect during the execution of this procedure, changing $\mathcal{O}$'s value from v to v_1, v_1 to v_2, ..., v_{k-1} to v_k. We call each of the values $v, v_1, \ldots, v_k$ a *valid* value (with respect to this LL execution by p), since it would be legitimate for p's LL to return any one of these values. Also, we call the value v_k *current* (with respect to

Types

valuetype = 64-bit number

Shared variables

X: valuetype (X supports *WLL*, *SC* and *VL* operations)

For each $p \in \{0, \ldots, N-1\}$, we have one single-writer, multi-reader register:

$\texttt{lastVal}_p$: valuetype

Initialization

$\texttt{X} = v_{init}$, the desired initial value of $\mathcal{O}$

```
    procedure LL(p, O) returns valuetype
1:  (flag, v) = WLL(p, X)
2:  if (flag = success)
3:      lastVal_p = v
4:      return v
5:  val = lastVal_v
6:  (flag, v) = WLL(p, X)
7:  if (flag = success)
8:      lastVal_p = v
9:      return v
10: return val
```

```
    procedure SC(p, O, v) returns boolean
11: return SC(p, X, v)

    procedure VL(p, O) returns boolean
12: return VL(p, X)
```

Algorithm 4.2: Implementation of the 64-bit LL/SC object $\mathcal{O}$ using a 64-bit WLL/SC object and 64-bit registers

this LL execution by p).

Although p's LL operation could legitimately return any valid value, there is a significant difference between returning the current value v_k versus returning an older valid value from $v, v_1, \ldots, v_{k-1}$: assuming that no successful SC operation takes effect between p's LL and p's subsequent SC, the specification of LL/SC operations requires p's subsequent SC to succeed in the former case and fail in the latter case. Thus, p's LL procedure, besides returning a valid value, has the additional obligation of ensuring the success or failure of p's subsequent SC (or VL) based on whether or not its return value is current.

In our algorithm, the SC procedure includes exactly one SC operation on the variable X (Line 11) and the procedure succeeds if and only if the operation suc-

ceeds. Therefore, we can restate the two obligations on p's LL procedure as follows: ($O1$) it must return a valid value u, and ($O2$) if no successful SC is performed after p's LL, p's subsequent SC (or VL) on `X` must succeed if and only if the return value u is current.

4.3.2 How the algorithm works

Algorithm 4.2 is based on two key ideas: (A1) the current value of $\mathcal{O}$ is held in `X`, and (A2) whenever a process p performs LL on $\mathcal{O}$ and obtains a value v, it writes v in $\texttt{lastVal}_p$ unless p is certain that its subsequent SC on $\mathcal{O}$ will fail. With this in mind, consider the procedure $\texttt{LL}(p, \mathcal{O})$ that p executes to perform an LL operation on $\mathcal{O}$. First, p tries to obtain $\mathcal{O}$'s current value by performing a WLL on `X` (Line 1). There are two possibilities: either WLL returns the current value v in `X`, or it fails, returning the id v of a process that performed a successful SC during the WLL. In the first case, p writes v in $\texttt{lastVal}_p$ (to ensure A2) and then returns v (Lines 3 and 4). In the second case, let t be the instant during p's WLL when process v performs a successful SC, and v' be $\mathcal{O}$'s value immediately prior to t (that is, just before v's successful SC). Then, v' is a valid value for p's LL procedure to return. Furthermore, by A2, $\texttt{lastVal}_v$ contains v' at time t. So, when p reads $\texttt{lastVal}_v$ and obtains *val* (Line 5), it knows that *val* must be either v' or some later value of $\mathcal{O}$. This means that *val* is a valid value for p's LL procedure to return. However, p cannot return *val* because its subsequent SC is sure to fail (due to the failure of WLL in Line 1) and, therefore, p must ensure that *val* is not the latest value of $\mathcal{O}$. So, p performs another WLL (Line 6). If this WLL succeeds and returns v, then as before p writes v in $\texttt{lastVal}_p$ and returns v (Lines 8 and 9). Otherwise, p knows that some successful SC occurred during

its execution of WLL in Line 6. At this point, p is certain that *val* is no longer the latest value of $\mathcal{O}$. Furthermore, p knows that its subsequent SC will fail (due to the failure of WLL in Line 6). Thus, returning *val* fulfills both Obligations O1 and O2, justifying Line 10.

The VL operation by p is simple to implement: p returns *true* if and only if the `VL` on `X` returns true (Line 12).

Based on the above discussion, we have the following theorem. Its proof is given in Appendix A.1.2.

Theorem 2 *Algorithm 4.2 is wait-free and implements a linearizable 64-bit LL/SC object from a single 64-bit WLL/SC object and one additional 64-bit register per process. The time complexity of LL, SC, and VL is* $O(1)$.

4.4 Implementing 64-bit WLL/SC from (1-bit, pid)-LL/SC

A *(1-bit, pid)-LL/SC object* is the same as a 1-bit LL/SC object except that its LL operation, which we call *BitPid_LL*, returns not only the 1-bit value written by the latest successful SC, but also the name of the process that performed that SC. Algorithm 4.3 implements a 64-bit WLL/SC object from a (1-bit, pid)-LL/SC object and 64-bit registers. This algorithm is nearly identical to Anderson and Moir's algorithm [AM95a] that implements a multi-word WLL/SC object from a word-sized CAS object and atomic registers. In the following, we describe the intuition underlying the algorithm.

Types

valuetype = 64-bit number

returntype = **record** *flag*: boolean; (*val*: valuetype **or** *val*: $0 \,..\, N-1$) **end**

Shared variables

X: {0, 1} (X supports *BitPid_read*, *BitPid_LL*, *SC* and *VL* operations)

For each $p \in \{0, \ldots, N-1\}$, we have two single-writer, multi-reader registers:

$\texttt{val}_p0, \texttt{val}_p1$: valuetype

Local persistent variables at each $p \in \{0, \ldots, N-1\}$

$index_p$: {0, 1}

Initialization

$\texttt{X} = 1$ (written by process 0)

$\texttt{val}_0 1 = v_{init}$, the desired initial value of $\mathcal{O}$

$index_0 = 0$

For each $p \in \{1, \ldots, N-1\}$: $index_p = 1$

procedure $\texttt{WLL}(p, \mathcal{O})$ **returns** returntype

1: $(b, q) = \texttt{BitPid\_LL}(p, \texttt{X})$
2: $v = \texttt{val}_q b$
3: **if** $\texttt{VL}(p, \texttt{X})$ **return** (*success*, v)
4: $(b, q) = \texttt{BitPid\_read}(p, \texttt{X})$
5: **return** (*failure*, q)

procedure $\texttt{SC}(p, \mathcal{O}, v)$ **returns** boolean

6: $\texttt{val}_p index_p = v$
7: **if** $\neg\texttt{SC}(p, \texttt{X}, index_p)$ **return** *false*
8: $index_p = 1 - index_p$
9: **return** *true*

procedure $\texttt{VL}(p, \mathcal{O})$ **returns** boolean

10: **return** $\texttt{VL}(p, \texttt{X})$

Algorithm 4.3: Implementation of the 64-bit WLL/SC object $\mathcal{O}$ using a (1-bit, pid)-LL/SC object and 64-bit registers, based on Anderson and Moir's algorithm [AM95a]

4.4.1 How the algorithm works

Let $\mathcal{O}$ denote the 64-bit WLL/SC object implemented by the algorithm. Our implementation uses two registers per process p—$\texttt{val}_p0$ and $\texttt{val}_p1$—which only p may write into, but any process may read. One of the two registers holds the value written into $\mathcal{O}$ by p's latest successful SC; the other register is available for use in p's next SC operation (p's local variable $index_p$ stores the index of the available register). Thus, over the N processes, there are a total of $2N$ val registers. Exactly which one of these contains the current value of $\mathcal{O}$ (i.e., the value written by the latest successful SC) is revealed by the (1-bit, pid)-LL/SC object X. Specifically, if

(b, q) is the value of X, then the algorithm ensures that $\texttt{val}_q b$ contains the current value of $\mathcal{O}$.

We now explain how process p performs an SC(v) operation on $\mathcal{O}$. First, p writes the value v into its available register (Line 6). Next, p tries to make its SC operation take effect by "pointing" X to this location (Line 7). If this effort fails, it means that some process performed a successful SC since p's latest WLL. In this case, p terminates its SC operation returning *false* (Line 7). Otherwise, p's SC operation has succeeded. So, $\texttt{val}_p index_p$ now holds the value written by p's latest successful SC. Therefore, to remain faithful to the representation described above, the index of the register available for p's next SC operation is updated to be $1 - index_p$ (Line 8). Finally, p returns *true* to reflect the success of its SC operation (Line 9).

We now turn to the procedure $\texttt{WLL}(p, \mathcal{O})$ that describes how process p performs a WLL operation on $\mathcal{O}$. First, p performs an BitPid_LL operation on X to obtain a value (b, q) (Line 1). By our representation, at the instant when p performs Line 1, $\texttt{val}_q b$ holds the current value v of $\mathcal{O}$. So, in an attempt to learn the value v, p reads $\texttt{val}_q b$ (Line 2). Then, it validates X. If the validate succeeds, p is certain that the value read in Line 2 is indeed v and so it returns v and signals *success* (Line 3). Otherwise, some process must have performed a successful SC after p had executed Line 1. Then, by the definition of WLL, p is not obligated to return a value; instead, it can signal failure and return the id of a process that performed a successful SC during p's WLL. Such an id can be obtained simply by reading X. So, p reads X and returns the id obtained, also signaling *failure* (Lines 4 and 5).

Based on the above discussion, we have the following theorem. Its proof is

given in Appendix A.1.3.

Theorem 3 *Algorithm 4.3 is wait-free and implements a 64-bit WLL/SC object from a single (1-bit, pid)-LL/SC object and an additional three 64-bit registers per process. The time complexity of LL, SC, and VL is* $O(1)$.

4.5 Implementing (1-bit, pid)-LL/SC from 64-bit CAS

Algorithm 4.4 implements a (1-bit, pid)-LL/SC object from 64-bit CAS objects and registers. This algorithm uses a procedure called `select`. As we will explain, the algorithm works correctly provided that `select` satisfies a certain property. This algorithm is inspired by, and is nearly identical to, Anderson and Moir's algorithm in Figure 1 of [AM95b]. The implementation of `select`, however, is novel and is crucial to obtaining constant space overhead per process.

Below we provide an intuitive description of how the algorithm works. Later we present two different implementations of the `select` procedure that offer different tradeoffs.

4.5.1 How the algorithm works

Let $\mathcal{O}$ denote the (1-bit, pid)-LL/SC object implemented by the algorithm. The variables used in the implementation are described as follows.

- The variable `X` supports *read* and *CAS* operations, and contains a value of the form (seq, pid, val), where seq is a sequence number, pid is a process id, and val is a 1-bit value. The first two entries (sequence number and process id) constitute the tag, and the last two entries (process id and 1-bit value) constitute the value of $\mathcal{O}$.

Types

seqnumtype = $(63 - \log N)$-bit number

rettype = **record** *val*: {0, 1}; *pid*: $0 \,..\, N-1$ **end**

entrytype = **record** *seq*: seqnumtype; *pid*: $0 \,..\, N-1$; *val*: {0, 1} **end**

Shared variables

X: entrytype (X supports *read* and *CAS* operations)

A: **array** $0 \,..\, N-1$ **of** entrytype

Local persistent variables at each $p \in \{0, \ldots, N-1\}$

old_p, chk_p: entrytype

seq_p: seqnumtype

Initialization

X = $(-1, p_{init}, v_{init})$, where (p_{init}, v_{init}) is the desired initial value of $\mathcal{O}$

For each $p \in \{0, \ldots, N-1\}$:

Ap = $(0, -1, 0)$

$seq_p = 0$

procedure BitPid_LL($p, \mathcal{O}$) **returns** rettype

1: $old_p = \mathtt{X}$
2: $\mathtt{A}p = (old_p.seq, old_p.pid, 0)$
3: $chk_p = \mathtt{X}$
4: **return** $(old_p.val, old_p.pid)$

procedure SC($p, \mathcal{O}, v$) **returns** boolean

5: **if** $(old_p \neq chk_p)$ **return** *false*
6: **if** ¬CAS(X,old_p,(seq_p,p,v)) **return** *false*
7: seq_p = select(p)
8: **return** *true*

procedure VL($p, \mathcal{O}$) **returns** boolean

9: **return** $(old_p = chk_p = \mathtt{X})$

procedure BitPid_read($p, \mathcal{O}$) **returns** rettype

10: $tmp = \mathtt{X}$
11: **return** $(tmp.val, tmp.pid)$

Algorithm 4.4: A bounded implementation of the 1-bit "pid" LL/SC object using a 64-bit CAS object and 64-bit registers, based on Anderson and Moir's algorithm [AM95b]

- The variable A is an array with one entry per process. A process p announces in Ap the tag that it reads from X in its latest BitPid_LL operation. This array is used by the select procedure, and it is crucial for ensuring Property 1 stated below.

- The variable seq_p is process p's local variable. It holds the value of the next sequence number that p can use in a tag.

We now explain the procedure `BitPid_LL`$(p, \mathcal{O})$ that describes how process p performs a BitPid_LL operation on $\mathcal{O}$. First, p reads `X` to obtain the current tag and value (Line 1). Next, p announces this tag in the array `A` (Line 2). Then, p reads `X` again (Line 3). There are two cases, based on whether the return values of the two reads are the same or not. If they are not the same, we linearize BitPid_LL at the instant when p performs the first read, and let p return that value at Line 4. In this case, since the value of $\mathcal{O}$ has changed after p's LL operation, we must ensure that p's subsequent SC operation will fail. This condition is indeed ensured by Line 5 of the algorithm. In the other case where the reads on Lines 1 and 3 return the same value, we linearize BitPid_LL at the instant when p performs the second read and let the LL operation return that value (at Line 4).

The implementation of the SC operation assumes that the `select` procedure satisfies the following property:

Property 1 *Let* OP *and* OP$'$ *be any two consecutive BitPid_LL operations by some process p. If p reads (s, q, v) from* `X` *in both Lines 1 and 3 of* OP*, then process q does not write $(s, q, *)$ into* `X` *after p executes Line 3 of* OP *and before it invokes* OP$'$.

We now describe how a process p performs an SC operation on $\mathcal{O}$. In the following, let OP denote p's latest execution of the BitPid_LL operation on $\mathcal{O}$. First, p compares the two values that it read from `X` during OP (Line 5). If these values are different then, as already explained, p's SC operation must fail, and Line 5 ensures this outcome. To understand Line 6, observe that by Property 1 the value of `X` is still old_p if and only if no process wrote into `X` after the point where p's latest BitPid_LL operation took effect (at Line 3 of OP). It follows that p's current

SC operation should succeed if and only if the CAS on Line 6 succeeds. Accordingly, if the CAS fails, p terminates the SC operation returning *false* (Line 6). On the other hand, if the CAS succeeds, p obtains a new sequence number to be used in p's next SC operation (Line 7), and completes the SC operation returning *true* (Line 8).

The implementation of the VL operation (Line 9) has the same justification as the SC operation. Finally, the implementation of the BitPid_read operation (Lines 10 and 11) is immediate from our representation.

Based on the above discussion, we have the following theorem. Its proof is given in Appendix A.1.4.

Theorem 4 *Algorithm 4.4 is wait-free and, if the* `select` *procedure satisfies Property 1, implements a linearizable (1-bit, pid)-LL/SC object from 64-bit CAS objects and 64-bit registers. If* τ *is the time complexity of* `select`*, then the time complexity of BitPid_LL, SC, VL, and BitPid_read operations are* $O(1)$*,* $O(1)+\tau$*,* $O(1)$*, and* $O(1)$*, respectively. If* s *is the per-process space overhead of* `select`*, then the per-process space overhead of the algorithm is* $4+s$*.*

4.5.2 Why `X` is read twice

To execute a BitPid_LL operation, notice that a process p reads `X`, announces the tag obtained in $A.p$, and reads `X` once more (Lines 1–3). As we will see in the next section, this double reading of `X`, with the tag announced between the reads, is crucial to our ability to implement the `select` procedure. To see why, suppose that p reads the same tag t at Lines 1 and 3. When subsequently executing an SC operation, p determines the success or failure of its SC based on whether the tag

in X is still t or not. Clearly, such a strategy goes wrong if X has been modified several times (between p's LL and SC) and the tag t has simply reappeared in X because of reuse of that tag. Fortunately, this undesirable scenario is preventable because p publishes the tag t in Ap (at Line 2) even before it reads that tag at Line 3, where p's LL operation takes effect. So, we can prevent the undesirable scenario by requiring processes not to reuse the tags published in the array A (this requirement will be enforced by the implementation of `select`).

4.5.3 An implementation of `select`

In this section, we design an algorithm that implements the `select` procedure. This design is challenging because it must guarantee several properties: the select procedure must satisfy Property 1, be wait-free, and have constant time complexity and constant per-process space overhead. Algorithm 4.5, presented in this section, guarantees all of these properties, but only works for at most $2^{15} = 32{,}768$ processes. A more complex algorithm, presented in the next section, can handle a maximum of $2^{19} = 524{,}288$ processes. To explain our algorithms, we introduce the notion of a sequence number being safe for a process.

Let s be a sequence number, q be a process, and t be a point in time. We say s *is safe for* q *at time* t if the following statement holds for every possible execution E from time t: the first writing into X after t (if any) of a value of the form $(s, q, *)$ does not violate Property 1.

A sequence number s is *unsafe* for q at time t if s is not safe for q at t. Notice that if s is safe for q at time t, it remains safe until q writes $(s, q, *)$ in X for the first time after t. An *interval is safe for* q *at time* t if every sequence number in the interval is safe for q at time t.

In both of our algorithms, the main idea is as follows. At all times, each process p maintains a current safe interval of size Δ; initially, this interval is 0, Δ). Each call to `select` by p returns a sequence number from p's current safe interval. By the time all numbers in p's current safe interval have been returned (which won't happen until p calls `select` Δ times), p determines a new safe interval of size Δ and makes that interval its current safe interval. Since p's current safe interval is not exhausted until p calls `select` Δ times, our algorithms use an amortized approach to finding the next safe interval: the work involved in identifying the next safe interval is distributed evenly over the Δ calls to `select`. Together with an appropriate choice of Δ, this strategy helps achieve constant time complexity for `select`.

The manner in which the next safe interval is determined is different in our two algorithms. The main idea in the first algorithm is as follows. Let $k, k + \Delta)$ be p's current safe interval. Then, $k + \Delta, k + 2\Delta)$ is the first interval that p tests for safety. If there is evidence that this interval is not safe, then the next Δ-sized interval, namely, $k+2\Delta, k+3\Delta)$ is tested for safety. The above steps are repeated until a safe interval is found. It remains to be explained how p tests whether a particular interval I is safe. To perform this test, p reads each element α of array `A` (recall that an element of `A` contains both a process id and a sequence number). If $\alpha = (s, p, *)$, then it is possible that some process read $(s, p, *)$ at Lines 1 and 3 of its latest BitPid_LL operation, thereby making s potentially unsafe for p. Therefore, in our algorithm, p deems an interval I to be safe if and only if it reads no element α such that $\alpha.pid = p$ and $\alpha.seq \in I$. To ensure $O(1)$ time complexity for the `select` procedure, p reads `A` in an incremental manner: it reads only one element of `A` in each invocation of `select`.

Types
seqnumtype = $(63 - \log N)$-bit number
Local persistent variables at each $p \in \{0, \ldots, N-1\}$
val_p, $nextStart_p$: seqnumtype
$procNum_p$: $0 \,..\, N$
Constants
$\Delta = (2N+1)N$
$K = (2N+2)\Delta$
Initialization
$val_p = 0$
$nextStart_p = \Delta$
$procNum_p = 0$

procedure `select`$(p, \mathcal{O})$ **returns** seqnumtype
12: $a = \mathtt{A}procNum_p$
13: **if** $((a.pid = p) \wedge (a.seq \in nextStart_p, nextStart_p \oplus_K \Delta)))$
14: $nextStart_p = nextStart_p \oplus_K \Delta$
15: $procNum_p = 0$
16: **else** $procNum_p = procNum_p + 1$
17: **if** $(procNum_p < N)$
18: $val_p = val_p \oplus_K 1$
19: **else** $val_p = nextStart_p$
20: $nextStart_p = nextStart_p \oplus_K \Delta$
21: $procNum_p = 0$
22: **return** val_p

Algorithm 4.5: A simple selection algorithm

In the following, we explain how the above high level ideas are implemented in our algorithm and why these ideas work.

How the algorithm works

Algorithm 4.5 implements the `select` procedure. Let $TestInt_p$ denote the interval that p is currently testing for safety. The algorithm uses three persistent local variables, described as follows:

- val_p is the sequence number from p's current safe interval that was returned by p's most recent invocation of `select`.

- $nextStart_p$ is the start of the interval $TestInt_p$. Thus, $TestInt_p$ is the interval $nextStart_p, nextStart_p + \Delta)$.

- $procNum_p$ indicates how far the test of safety of $TestInt_p$ has progressed. Specifically, if $procNum_p = k$, it means that the array entries belonging to processes $0, 1, \ldots, k-1$ (namely, $\texttt{A}0, \texttt{A}1, \ldots, \texttt{A}k-1$) have not presented any evidence that $TestInt_p$ is unsafe.

The $\oplus_K$ operator at Lines 18 and 20 denotes addition modulo K. The value for K is chosen to be large enough to ensure that all the intervals that p tests for safety (before it finds the next safe interval) are disjoint (i.e., no wraparound occurs).

Given the above definitions, the algorithm works as follows. First, p reads the next element a of the array `A` (Line 12). If the process id in a is p and the sequence number in a belongs to the interval $TestInt_p$, then the interval is potentially unsafe. Therefore, if the condition at Line 13 holds, p abandons the interval $TestInt_p$ as unsafe. At this point, the Δ-sized interval immediately following $TestInt_p$ becomes the new interval to be tested for safety. To this end, p updates $nextStart_p$ to the beginning of this interval (Line 14) and resets $procNum_p$ to 0 (Line 15). On the other hand, if the condition on Line 13 does not hold, it means that $\texttt{A}procNum_p$ (namely, a) presents no evidence that $TestInt_p$ is unsafe. To reflect this fact, p increments $procNum_p$ (Line 16).

At Line 17, if $procNum_p$ is N, it follows from the meaning of $procNum_p$ that p read the entire array `A` and found no evidence that the interval $TestInt_p$ is unsafe. In this case, p performs the following actions. It switches to $TestInt_p$ as its current safe interval and lets `select` return the first sequence number in this interval (Lines 19 and 22). The Δ-sized interval immediately following this new current safe interval

becomes the new interval to be tested for safety. To this end, p updates $nextStart_p$ to the beginning of this interval (Line 20) and resets $procNum_p$ to 0 (Line 21).

At Line 17, if $procNum_p$ is not yet N, p is not sure yet that $TestInt_p$ is a safe interval. Therefore, it keeps the current safe interval as it is and simply returns the next value from that interval (Lines 18 and 22).

Notice that after p adopts an interval I to be its current safe interval at some time t, p's calls to `select` return successive sequence numbers starting from the first number in I. Therefore, if p makes at most $k \leq \Delta$ calls to `select` before adopting a new interval I' as its current safe interval, then all numbers returned (by the k calls to `select`) are from I and no number is returned more than once. Since I was safe for p at time t, it follows that the numbers returned by the k calls to `select` do not lead to a violation of Property 1. By the above discussion, the correctness of the algorithm rests on the following two claims:

- After a process p adopts an interval I to be its current safe interval, p makes at most Δ calls to `select` before adopting a new interval I' as its current safe interval.
- At the time that p adopts I' to be its current safe interval, I' is indeed safe for p.

The above claims are justified in the next two subsections.

A new interval is identified quickly

Suppose that at time t a process p adopts an interval I to be its current safe interval. Let $t' > t$ be the earliest time when p adopts a new interval I' as its current safe interval. Then, we claim:

Claim A: During the time interval t, t'), process p makes at most Δ calls to `select` *(which return distinct sequence numbers from the interval I). Furthermore, I and I' are disjoint.*

To prove the claim, we make a crucial subclaim which states that, when a process p searches for a next safe interval, no process q can cause p to abandon more than two intervals as unsafe.

Subclaim: If $I_1, I_2, \ldots, I_m$ are the successive intervals that p tests for safety during t, t'), then at most two of $I_1, I_2, \ldots, I_m$ are abandoned by p as unsafe on the basis of the values read in `A`q*, for any q.*

First we argue that the subclaim implies the claim. By the subclaim, during t, t'), p abandons at most $2N$ intervals as unsafe. Notice that, in the worst case, p invokes `select` N times before abandoning any interval as unsafe (the worst case arises if none of `A`0, `A`1, ..., `A`$N-2$ provides any evidence of unsafety, and `A`$N-1$ does). It follows from the above two facts that, during t, t'), p invokes `select` at most $2N \cdot N$ times before it begins testing an interval that is sure to pass the test. Since the testing of this final interval occurs over N calls to `select`, it follows that, during t, t'), p invokes `select` at most $2N^2 + N = (2N+1)N$ times before it identifies the next safe interval I'. Since we fixed Δ to be $(2N+1)N$, we have the first part of the claim.

For the second part, notice that (1) by the subclaim, p abandons at most $2N$ intervals as unsafe, and so $m \leq 2N$, and (2) I' is the interval that p tests for safety after abandoning I_m as unsafe. Furthermore, by the algorithm, each interval in $I, I_1, I_2, \ldots I_m, I'$ is of size Δ and begins immediately after the previous one ends. Since we fixed K to be $(2N+2)\Delta$ and since we perform the arithmetic modulo K, it follows that all of $I, I_1, I_2, \ldots I_m, I'$ are disjoint intervals. Hence, we have the

second part of the claim.

Next, we prove the subclaim. By the algorithm, the testing of I_1 for safety begins only after p writes the first sequence number from I in the variable X. Let $\tau \in t, t')$ be the time when this writing happens. For a contradiction, suppose that the subclaim is false and τ' is the earliest time when the subclaim is violated. More precisely, let τ' be the earliest time in $t, t')$ such that, for some $q \in \{0, 1, \ldots, N-1\}$, p abandons three intervals as unsafe on the basis of the values that it read in Aq. Let I_j, I_k, and I_l denote these three intervals, and let $s_j \in I_j$, $s_k \in I_k$, and $s_l \in I_l$ be the sequence numbers that p read in Aq which caused p to abandon the three intervals. We make a number of observations:

(O1). In the time interval t, τ', p abandons at most $2N + 1$ intervals as unsafe.

Proof: This observation is immediate from the definition of τ'.

(O2). In the time interval t, τ', p calls `select` at most $(2N + 1)N$ times.

Proof: This observation follows from Observation O1 and an earlier observation that, in the worst case, p invokes `select` N times before abandoning any interval as unsafe.

(O3). In the time interval t, τ', all of p's calls to `select` return distinct sequence numbers from I.

Proof: Notice that after p adopts I as its current safe interval at time t, p's calls to `select` return successive sequence numbers starting from the first number in I. Since, by Observation O2, p makes at most $\Delta = (2N + 1)N$ calls to `select` during $t, \tau')$, all numbers returned by these calls are distinct numbers from I.

(O4). In the time interval τ, τ', if X contains a value of the form $(s, p, *)$, then $s \in I$.

Proof: By definition of τ, p writes in X at time τ a value of the form $(s, p, *)$, where $s \in I$. By Observation O3, all values that p subsequently writes in X during τ, τ' are from I. Hence, we have the observation.

(O5). The intervals I, I_j, I_k and I_l are all disjoint (and, therefore, s_j, s_k and s_l are distinct and are not in I).

Proof: Recall that $I_1, I_2, \ldots I_l$ are the intervals that p abandons during t, τ') as unsafe. By Observation O1, $l \leq 2N + 1$. Furthermore, by the algorithm, each of the intervals $I, I_1, I_2, \ldots I_l$ is of size Δ and begins immediately after the previous one ends. Since $K = (2N + 2)\Delta$ and since we perform the arithmetic modulo K, it follows that all of $I, I_1, I_2, \ldots I_l$ are disjoint intervals. Then, the observation follows from the fact that I, I_j, I_k, and I_l are members of $\{I, I_1, I_2, \ldots I_l\}$.

(O6). Recall that p abandons the interval I_l at time τ' because it reads at τ' the value $(s_l, p, *)$ in Aq, where $s_l \in I_l$. Let σ' be the latest time before τ' when q writes $(s_l, p, *)$ in Aq (at Line 2). By the algorithm, this writing must be preceded by q's reading of the value $(s_l, p, *)$ from the variable X (Line 1). Let σ be the latest time before σ' when q reads $(s_l, p, *)$ from X. Then, we claim that $\tau < \sigma < \tau'$.

Proof: By definition of s_j, s_k and s_l, we know that p reads from Aq the values $(s_j, p, *)$, $(s_k, p, *)$ and $(s_l, p, *)$ (in that order) in the time interval τ, τ'. It follows that q's writing of $(s_k, p, *)$ and $(s_l, p, *)$ in Aq occur (in that order) in

the time interval τ, τ'. Since q's reading of $(s_l, p, *)$ in X must occur between the above two writes, it follows that the time σ at which this reading occurs lies in the time interval τ, τ'.

By Observation O6, q reads $(s_l, p, *)$ from X during τ, τ'. Therefore, by Observation O4, we have $s_l \in I$. This conclusion contradicts Observation O5, which states that $s_l \notin I$. Hence, we have the subclaim.

The new interval is safe

In this section, we argue that the rule by which the algorithm determines the safety of an interval works correctly. More precisely, let t be the time when process p adopts an interval I to be its current safe interval, and t' be the earliest time after t when p switches its current safe interval from I to a new interval I'. Then, we claim:

Claim B: The interval I' is safe for process p at time t'.

Suppose that the claim is false and I' is not safe for p at time t'. Then, by the definition of safety, there exists a sequence number $s' \in I'$, a process q, and a time η such that the following scenario, which violates Property 1, is possible:

- η is the first time after t' when p writes $(s', p, *)$ in the variable X (at Line 6 of Algorithm 4.4).
- q's BitPid_LL operation, which is the latest with respect to time η, completes Line 3 before η, and at both Lines 1 and 3 this operation reads from X a value of the form $(s', p, *)$. In the following, let OP denote this BitPid_LL operation by q.

By the algorithm, the testing of I' for safety begins only after p writes the first sequence number s from I in the variable X. Let $\tau \in t, t')$ be the time when this writing happens. We make a few simple observations: (1) at time τ, the sequence number in X is not s' (because X has the sequence number $s \in I$ at τ, $s' \in I'$ and, by Claim A of the previous subsection, the intervals I and I' are disjoint), (2) during the time interval $\tau, t')$, any sequence number that p writes in X is from I (by Claim A) and, hence, is different from s', and (3) during the time interval $t', \eta)$, any sequence number that p writes in X is different from s' (by the definition of η). From the above observations, the value of X is not of the form $(s', p, *)$ at any point during $\tau, \eta)$. Therefore, q must have executed Line 3 of OP before τ. So, q's execution of Line 2 of OP is also before τ. Since q read the same value $(s', p, *)$ at both Lines 1 and 3 of OP, it follows that q writes $(s', p, *)$ in Aq at Line 2 of OP. This value remains in Aq at least until η because OP is q's *latest* BitPid_LL operation with respect to η. Therefore, Aq holds the value $(s', p, *)$ all through the time $\tau, t')$ when p tests different intervals for safety. In particular, when p tests I' for safety, it would find $(s', p, *)$ in Aq and, since $s' \in I'$, it would abandon I' as unsafe. This contradicts the fact that p switches its current safe interval from I to I'.

Based on the above discussion, we have the following lemma. Its proof is given in Appendix A.1.5.

Lemma 1 *Algorithm 4.5 satisfies Property 1. The time complexity of the implementation is* $O(1)$*, and the per-process space overhead is* 3.

A remark about sequence numbers

In our algorithm, the operation $\oplus_K$ is performed modulo $K = (2N+2)\Delta$. Hence, the space of all sequence numbers must be at least K. Since we store a sequence number, a process id, and a 1-bit value in the same memory word `X`, the number of bits we have available for a sequence number is $63 - \lg N$. Hence, K can be at most $2^{63-\lg N} = 2^{63}/N$. Since $K = (2N+2)\Delta$ and $\Delta = (2N+1)N$, the above constraint translates into $(2N+2)(2N+1)N^2 \leq 2^{63}$. It is easy to verify that for $N \leq 2^{15} = 32{,}768$ this inequality holds. Our algorithm is therefore correct if the number of processes that execute it is not more than 32,768. We believe that this restriction is not of practical concern. Furthermore, our second selection algorithm in Section 4.5.4 relaxes this restriction to $N \leq 2^{19} = 524{,}288$, at the expense of performing one additional CAS per `select` operation.

4.5.4 An alternative selection algorithm

In this section, we present an algorithm that supports a larger number of processes than our previous selection algorithm. More specifically, the new algorithm can handle a maximum of $2^{19} = 524{,}288$ processes, whereas the previous algorithm works for a maximum of $2^{15} = 32{,}768$ processes.

The main idea of the algorithm is the same as in the first algorithm: at all times, each process p maintains a current safe interval of size Δ. Each call to `select` by p returns a sequence number from p's current safe interval. By the time all numbers in p's current safe interval are returned (which won't happen until p calls `select` Δ times), p determines a new safe interval of size Δ and makes that interval its current safe interval.

The way the next safe interval is located is different from the first algorithm. In the first algorithm, process p searched for the next safe interval in a linear fashion: first, p tested whether the interval right next to the current safe interval was safe; if there was evidence that this interval was not safe, p selected the next Δ-sized interval to test for safety, and repeated this process until a safe interval was found. In the new algorithm, process p employs a more efficient strategy, based on binary search, for locating the next safe interval.

Our algorithm consists of two stages—the *marking stage*, and the *search stage*. During the marking stage, process p reads each entry $\mathtt{A}k$ of the array $\mathtt{A}$. If $\mathtt{A}k = (s, p, *)$, then it is possible that some process read $(s, p, *)$ at Lines 1 and 3 of its latest BitPid_LL operation, thereby making s potentially unsafe for p. In this case, p puts a mark on $\mathtt{A}k$ to indicate that it contains a sequence number that is potentially unsafe for p. If, on the other hand, $\mathtt{A}k$ is not of the form $(*, p, *)$, then p leaves $\mathtt{A}k$ unchanged.

After the marking stage completes, p initializes a local variable named I_p to some large interval of size $(N + 1)\Delta$, and begins the *search stage*. The search stage consists of many iterations or *passes*, each of which takes place over many invocations of `select`. In each pass, the interval I_p is halved. Ultimately, after all the passes, I_p is reduced to a size of Δ. At that point, p regards the interval I_p safe, and starts using it as its current safe interval. Below we explain this stage in more detail.

Let $C = k, k+\Delta)$ be p's current safe interval, and let $I_p = k+\Delta\,, k+(N+2)\Delta)$ be the interval immediately after C. The search stage consists of a sequence of $\lg(N + 1)$ passes, where each pass involves exactly N consecutive invocations of `select` by p. Each pass consists of two phases:

- *Counting phase:* p goes through all the marked entries in the array `A`, and counts how many sequence numbers fall within the first half of I_p, and how many fall within the second half of I_p.

- *Halving step:* p discards the half of I_p with a higher count, and sets I_p to be the remaining half.

In the following, we assume that $N + 1$ is a power of two.[1] Then, since after each pass the size of I_p halves, at the end of all $\lg(N + 1)$ passes the size of I_p becomes Δ. Further, p regards this interval as safe, and starts using it as its current safe interval.

We now intuitively explain why the above method yields a safe interval. First, observe that the number of marked entries in `A` that contain a sequence number from I_p halves after each pass (since we discard the half of I_p with a higher count). Next, observe that initially there are at most N marked entries (since the size of `A` is N). By the above two observations, it follows that after $\lg(N + 1)$ passes, no marked entry in `A` contains a sequence number in I_p. Hence, at the end of $\lg(N + 1)$ passes, I_p is indeed safe for p.

In the following, we explain how the above high level ideas are implemented in our algorithm and why these ideas work.

How the algorithm works

Algorithm 4.6 implements the `select` procedure. The algorithm uses four persistent local variables, described as follows:

[1] If $N + 1$ is not a power of two, then let $N' + 1$ be the smallest power of two that is greater than $N + 1$, and imagine processes $N, N + 1, \ldots, N' - 1$ to be dummy processes. Since $N' \leq 2N$, the asymptotic time and space complexities are unaffected by this change. Thus, the assumption that $N + 1$ is a power of two is introduced only for convenience, and is not really needed.

Types
seqnumtype = $(63 - \log N)$-bit number
intervaltype = **record** $start, end$: seqnumtype **end**
Local persistent variables at each $p \in \{0, \ldots, N-1\}$
I_p: intervaltype; val_p: seqnumtype
$passNum_p$: $0 \,..\, \lg(N+1)$; $procNum_p$: $0 \,..\, N-1$
Constants
$\Delta = N(\lg(N+1)+1)$; $K = (N+2)\Delta$
Initialization
$passNum_p = 0$; $val_p = 0$; $procNum_p = 0$; $I_p = \Delta, (N+2)\Delta)$

procedure `select`(p) **returns** seqnumtype
12: **if** $(passNum_p = 0)$
13: $a = \mathtt{A}procNum_p$
14: **if** $(a.pid = p)$ `CAS`$(\mathtt{A}procNum_p, a, (a.seq, a.pid, 1))$
15: **if** $(procNum_p < N-1)$
16: $procNum_p$++
17: **else** $procNum_p = 0$
18: $passNum_p$++
19: $val_p = val_p \oplus_K 1$
20: **else** $a = \mathtt{A}procNum_p$
21: **if** $((a.pid = p) \wedge (a.val = 1) \wedge (a.seq \in I_p))$
22: Increase the counter of the half of I_p that contains $a.seq$;
23: **if** $(procNum_p < N-1)$
24: $procNum_p$++
25: $val_p = val_p \oplus_K 1$
26: **else** Set I_p to be the half of I_p with a smaller counter; Reset counters;
27: $procNum_p = 0$
28: **if** $(passNum_p < \lg(N+1))$
29: $val_p = val_p \oplus_K 1$
30: $passNum_p$++
31: **else** $passNum_p = 0$
32: $val_p = I_p.start$
33: $I_p = I_p.end, I_p.end \oplus_K (N+1)\Delta)$
34: **return** val_p

Algorithm 4.6: Another selection algorithm

- I_p is the interval which p halves in the search stage of the algorithm.
- val_p is a sequence number from p's current safe interval which was returned by p's most recent invocation of `select`.
- $passNum_p$ represents the current pass of process p's algorithm. If we consider the marking stage to be a pass zero, then the $passNum_p$ variable takes on values from the range $0 \,..\, \lg(N+1)$.
- $procNum_p$ indicates how far process p's reading of the entries in `A` has progressed. Specifically, if $procNum_p = k$, it means that the array entries belonging to processes $0, 1, \ldots, k-1$ (namely, `A`$0, $`A`$1, \ldots,$ `A`$k-1$) have so far been read.

The algorithm works as follows. First, p reads the variable $passNum_p$ to determine which pass of the algorithm it is currently executing (Line 12). If the value of $passNum_p$ is 0, it means that p is still in the marking stage. So, p reads the next element a of the array `A` (Line 13). If the process id in a is p, p puts a mark on the entry in `A` it just read (Line 14). Otherwise, it leaves the entry unchanged. Next, p checks whether it has gone through all the entries in `A` (i.e., whether it has reached the end of the marking stage) by reading $procNum_p$ (Line 15). If not, p simply increments $procNum_p$ (Line 16) and returns the next value from the current safe interval (Lines 19 and 34). Otherwise, p has reached the end of the marking stage, and so it resets $procNum_p$ to 0 (Line 17) and increments the $passNum_p$ variable (Line 18). Finally, p returns the next value from the current safe interval (Lines 19 and 34).

On the other hand, if the value of $passNum_p$ is not 0, it means that p is in the search stage. So, p reads the next element a of the array A (Line 20). If the process id in a is p and a has the mark, then the sequence number in a is potentially unsafe for p and hence should be counted (Line 21). So, p tests whether the sequence number in a belongs to the first or the second half of I_p, and increments the appropriate counter (Line 22). Next, p checks whether it has counted all the entries in A (i.e., whether it has reached the end of the counting phase), by reading $procNum_p$ (Line 23). If not, p simply increments $procNum_p$ (Line 24), and returns the next value from the current safe interval (Lines 25 and 34). On the other hand, if p has counted all the entries in A, it has all the information it needs to halve the interval I_p appropriately (i.e., to perform the halving step). To this end, p discards the half of I_p with a higher count, and resets the counters (Line 26). Since it has reached the end of a pass, p also resets $procNum_p$ to zero (Line 27). Next, p reads $passNum_p$ to determine whether it has performed all of the $\lg(N+1)$ passes (Line 28). If it hasn't, p simply increments $passNum_p$ and returns the next value from the current safe interval (Lines 29, 30, and 34). On the other hand, if p has reached the end of all the passes, it means that the interval I_p should be p's next safe interval. So, p selects I_p as its current safe interval (Line 32) and resets variables $passNum_p$ and I_p to begin searching for the next safe interval (Lines 31 and 33). Finally, p returns the next (i.e., the first) value from its new current safe interval (Line 34).

Notice that, similar to our first selection algorithm, the correctness of the above algorithm depends on the following two claims:

- After a process p adopts an interval I to be its current safe interval, p makes

at most Δ calls to `select` before adopting a new interval I' as its current safe interval.

- At the time that p adopts I' to be its current safe interval, I' is of size Δ and is indeed safe for p.

We justify the above two claims in the next two subsections.

A new interval is identified quickly

Suppose that at time t a process p adopts an interval I to be its current safe interval. Let $t' > t$ be the earliest time when p adopts a new interval I' as its current safe interval. Then, we claim:

Claim C: During the time interval t, t'), process p makes at most Δ calls to `select` *(which return distinct sequence numbers from the interval I). Furthermore, I and I' are disjoint.*

The first part of the claim trivially holds since (1) during t, t'), p executes exactly $\lg(N+1)+1$ passes, and (2) in each pass p makes exactly N calls to `select`. Hence, during the time interval t, t'), process p makes exactly $N(\lg(N+1)+1) = \Delta$ calls to `select`.

For the second part, notice that soon after t, p initializes I_p to be the interval I'', where I'' is the interval of size $(N+1)\Delta$ immediately after I. Furthermore, notice that I' is the subinterval of I''. Since $K = (N+2)\Delta$ and since we perform the arithmetic modulo K, it follows that the intervals I and I'' are disjoint, and so the intervals I and I' are disjoint as well. Hence, we have the second part of the claim.

The new interval is safe

In this section, we argue that (1) the interval I' is of size Δ, and (2) the rule by which the algorithm determines the safety of an interval works correctly. More precisely, let t be the time when process p adopts an interval I to be its current safe interval and t' be the earliest time after t when p switches its current safe interval from I to a new interval I'. Then, we claim:

Claim D: The interval I' is of size Δ, and is safe for process p at time t'.

To prove this claim, we make a crucial subclaim which states that, at time t', there are no marked entries in A holding a sequence number from the interval I'.

Subclaim: The interval I' is of size Δ, and at time t', no entry in A is of the form $(s, p, 1)$, where $s \in I'$.

We first argue that the above subclaim implies the claim. Suppose that the claim is false and I' is not safe for p at time t'. Then, by the definition of safety, there exists a sequence number $s' \in I'$, a process q, and a time η such that the following scenario, which violates Property 1, is possible:

- η is the first time after t' when p writes $(s', p, *)$ in the variable X (at Line 6 of Algorithm 4.4).
- q's BitPid_LL operation, which is the latest with respect to time η, completes Line 3 before η, and at both Lines 1 and 3 this operation reads from X a value of the form $(s', p, *)$. In the following, let OP denote this BitPid_LL operation by q.

By the algorithm, the testing of I' for safety begins only after p writes the first sequence number s from I in the variable X. Let $\tau \in t, t')$ be the time when

this writing happens. Then, if we use the same argument we used in the proof of Claim B, we conclude that (1) q does not write into $\mathtt{A}q$ during the time interval τ, t', and (2) the latest value that q writes into $\mathtt{A}q$ prior to time τ is $(s', p, *)$. Furthermore, as long as the value in $\mathtt{A}q$ stays of the form $(*, p, *)$, no other process will attempt to put their mark on $\mathtt{A}q$. By the above observations, we conclude that at some time $\tau' \in \tau, t'$ during p's 0th pass, p succeeds in putting its mark on $\mathtt{A}q$. Hence, at all times during τ', t', $\mathtt{A}q$ holds the value $(s', p, 1)$. In particular, at time t', $\mathtt{A}q$ holds the value $(s', p, 1)$, which contradicts the subclaim. Hence, the claim holds.

Next we prove the above subclaim. Let $t'' \in t, t'$ be the time when p completes its 0th pass. For all $k \in \{0, 1, \ldots, \lg(N+1)\}$, let

- I_k denote the value of the interval I_p at the end of p's kth pass, and
- s_k denote the number of marked entries in $\mathtt{A}$ that, at the end of the kth pass, hold a sequence number from I_k. (More specifically, s_k is the number of entries in $\mathtt{A}$ that, at the end of the kth pass, hold a value of the form $(s, p, 1)$, for some $s \in I_k$.)

We make a number of observations:

(O1). The size of the interval I_k is $((N+1)/2^k)\Delta$.

Proof (by induction): For the base case (i.e., $k = 0$), the observation trivially holds, since at the end of the 0th pass I_p is initialized to be of size $(N+1)\Delta$. Hence, the size of I_0 is $(N+1)\Delta$. The induction hypothesis states that the size of I_j is $((N+1)/2^j)\Delta$, for all $j \leq k$. We now show that the size of I_{k+1} is $((N+1)/2^{k+1})\Delta$. By the algorithm, I_{k+1} is a half of I_k. Moreover, we made

an assumption earlier that $N + 1$ is a power of two. Hence, the size of I_{k+1} is exactly $(((N + 1)/2^k)/2)\Delta = ((N + 1)/2^{k+1})\Delta$.

(O2). If an entry $\mathtt{A}q$ holds the value $(s, p, 1)$ at any time $\eta' \in t'', t'$, then $\mathtt{A}q$ holds the value $(s, p, 1)$ at all times during t'', η'.

Proof: Suppose not. Then, at some time $\eta'' \in t'', \eta'$, $\mathtt{A}q$ does not hold the value $(s, p, 1)$. Therefore, at some point during η'', η', the value $(s, p, 1)$ is written into $\mathtt{A}q$. Since by the time η'' process p is already done marking all the entries in $\mathtt{A}$, some process other than p must have written $(s, p, 1)$ into $\mathtt{A}q$, which is impossible. Hence, the observation holds.

(O3). The value of s_k is at most $(N + 1)/2^k - 1$.

Proof (by induction): For the base case (i.e., $k = 0$), the observation trivially holds, since $\mathtt{A}$ can hold at most $N = (N + 1) - 1$ entries. Hence, the value of s_0 is at most N. The induction hypothesis states that the value of s_j is at most $(N + 1)/2^j - 1$, for all $j \leq k$. We now show that the value of s_{k+1} is at most $(N + 1)/2^{k+1} - 1$. Since s_k is the number of entries in $\mathtt{A}$ that, at the end of the kth pass, hold a sequence number from I_k, it follows by Observation O2 that p can count at most s_k sequence numbers during the $(k + 1)$st pass. Moreover, since I_{k+1} is a half of I_k with a smaller count, it follows that at most $\lfloor s_k/2 \rfloor$ of the counted sequence numbers fall within I_{k+1}. Hence, by Observation O2, at the end of the $(k + 1)$st pass, at most $\lfloor s_k/2 \rfloor$ marked entries in $\mathtt{A}$ hold a sequence number from I_{k+1}. Therefore, we have $s_{k+1} = \lfloor s_k/2 \rfloor = \lfloor ((N + 1)/2^k - 1)/2 \rfloor$. Since $N + 1$ is a power of two, it means that $s_{k+1} = (N + 1)/2^{k+1} - 1$. Hence, the observation holds.

By the above observations, at the end of the $\lg(N+1)$st pass, we know that (1) the size of I' is Δ (by Observation O1), and (2) the number of marked entries in A that hold a sequence number from I' is 0 (by Observation O3). Hence, we have the subclaim.

Based on the above discussion, we have the following lemma. Its proof is given in the Appendix A.1.6.

Lemma 2 *Algorithm 4.6 satisfies Property 1. The time complexity of the implementation is* $O(1)$*, and the per-process space overhead is* 4*.*

A remark about sequence numbers

In our algorithm, the operation $\oplus_K$ is performed modulo $K = (N+2)\Delta$. Hence, the space of all sequence numbers must be at least K. Since we store a sequence number, a process id, and a 1-bit value in the same memory word X, the number of bits we have available for a sequence number is $63 - \lg N$. Hence, K can be at most $2^{63-\lg N} = 2^{63}/N$. Since $K = (N+2)\Delta$ and $\Delta = N(\lg(N+1)+1)$, the above constraint translates into $(N+2)N^2(\lg(N+1)+1) \leq 2^{63}$. It is easy to verify that for $N \leq 2^{19} = 524{,}288$, this inequality holds. Our algorithm is therefore correct if the number of processes that execute it is not more than 524,288. This limit is large enough that it is not of any practical concern.

Chapter 5

Multiword LL/SC

All of the algorithms in the previous chapter implement word-sized LL/SC objects. However, many existing applications [AM95a, CJT98, Jay02, Jay05] need *multiword* LL/SC objects, i.e., LL/SC objects whose value does not fit in a single machine word. In this chapter, we present an algorithm that implements a W-word LL/SC object from word-sized CAS objects and registers. The algorithm is designed in two steps:

1. Implement an array of $3N + 1$ small LL/SC objects from word-sized CAS objects and registers.

2. Implement a W-word LL/SC object from an array of $3N + 1$ small LL/SC objects and registers.

The implementation in the first step follows immediately from the following theorem by Moir [Moi97a]. For the rest of this chapter, therefore, we focus on the implementation in the second step.

Theorem 5 ([Moi97a]) *There exists a linearizable N-process wait-free implementation of an array $\mathcal{O}0..M-1$ of M small LL/SC objects from word-sized CAS objects and registers. The space complexity of the implementation is $O(Nk+M)$, and the time complexity of LL, SC, VL, read, and write operations on $\mathcal{O}$ is $O(1)$. Furthermore, the CAS-time-complexity of LL, SC, VL, read, and write operations on $\mathcal{O}$ are 0, 1, 0, 0, and 0, respectively.*

5.1 Implementing a W-word LL/SC Object

Algorithm 5.1 implements a W-word LL/SC object $\mathcal{O}$. We now informally describe how the algorithm works.

5.1.1 The variables used

We begin by describing the variables used in the algorithm. `BUF`$0..3N-1$ is an array of $3N$ W-word buffers. Of these, $2N$ buffers hold the $2N$ most recent values of $\mathcal{O}$ and the remaining N buffers are "owned" by processes, one buffer by each process. Process p's local variable, $mybuf_p$, holds the index of the buffer currently owned by p. `X` holds the tag associated with the current value of $\mathcal{O}$ and consists of two fields: the index of the buffer that holds $\mathcal{O}$'s current value and the sequence number associated with $\mathcal{O}$'s current value. The sequence number increases by 1 (modulo $2N$) with each successful SC on $\mathcal{O}$. The buffer holding $\mathcal{O}$'s current value is not reused until $2N$ more successful SC's are performed. Thus, at any point, the $2N$ most recent values of $\mathcal{O}$ are available and may be accessed as follows. If the current sequence number is k, the sequence numbers of the $2N$ most recent successful SC's (in the order of their recentness) are $k, k-1, \ldots, 0, 2N-1, 2N-$

Types

valuetype = **array** $0\,..\,W-1$ **of** 64-bit word

xtype = **record** *buf*: $0\,..\,3N-1$; *seq*: $0\,..\,2N-1$ **end**

helptype = **record** *helpme*: {0, 1}; *buf*: $0\,..\,3N-1$ **end**

Shared variables

X: xtype; Bank: **array** $0\,..\,2N-1$ **of** $0\,..\,3N-1$

Help: **array** $0\,..\,N-1$ **of** helptype; BUF: **array** $0\,..\,3N-1$ **of** valuetype

Local persistent variables at each $p \in \{0, 1, \ldots, N-1\}$

$mybuf_p$: $0\,..\,3N-1$; x_p: xtype

Initialization

X $= (0, 0)$; BUF0 = the initial value of $\mathcal{O}$; Bank$k = k$, for all $k \in \{0, 1, \ldots, 2N-1\}$;

$mybuf_p = 2N + p$, for all $p \in \{0, 1, \ldots, N-1\}$;

Help$p = (0, *)$, for all $p \in \{0, 1, \ldots, N-1\}$

procedure LL(p, $\mathcal{O}$, *retval*)

1: Help$p = (1, mybuf_p)$
2: $x_p =$ LL(X)
3: **copy** BUFx_p.*buf* **into** $*$*retval*
4: **if** LL(Help$p$) $\equiv (0, b)$
5: $\quad x_p =$ LL(X)
6: $\quad$ **copy** BUFx_p.*buf* **into** $*$*retval*
7: $\quad$ **if** $\neg$VL(X) **copy** BUFb **into** $*$*retval*
8: **if** LL(Help$p$) $\equiv (1, c)$
9: $\quad$ SC(Help$p$, $(0, c)$)
10: $mybuf_p =$ Help$p$.*buf*
11: **copy** $*$*retval* **into** BUF$mybuf_p$

procedure SC(p, $\mathcal{O}$, v) **returns** boolean

12: **if** (LL(Bankx_p.*seq*) $\neq x_p$.*buf*) $\wedge$ VL(X)
13: $\quad$ SC(Bankx_p.*seq*, x_p.*buf*)
14: **if** (LL(Helpx_p.*seq* mod N) $\equiv (1, d)$) $\wedge$ VL(X)
15: $\quad$ **if** SC(Helpx_p.*seq* mod N, $(0, mybuf_p)$)
16: $\quad\quad mybuf_p = d$
17: **copy** $*v$ **into** BUF$mybuf_p$
18: $e =$ Bank$(x_p.seq + 1)$ mod $2N$
19: **if** SC(X, $(mybuf_p, (x_p.seq + 1)$ mod $2N))$
20: $\quad mybuf_p = e$
21: $\quad$ **return** *true*
22: **return** *false*

procedure VL(p, $\mathcal{O}$) **returns** boolean

23: **return** VL(X)

Algorithm 5.1: An implementation of the N-process W-word LL/SC object $\mathcal{O}$ using $3N + 1$ small LL/SC objects and registers

$2, \ldots, k+1$; and Bank$j$ is the index of the buffer that holds the value written to $\mathcal{O}$ by the most recent successful SC with sequence number j. Finally, it turns out that a process p might need the help of other processes in completing its LL operation on $\mathcal{O}$. The variable Help$p$ facilitates coordination between p and the helpers of p.

5.1.2 The helping mechanism

The crux of our algorithm lies in its helping mechanism by which SC operations help LL operations. Specifically, a process p begins its LL operation by announcing its operation to other processes. It then attempts to read the buffer containing $\mathcal{O}$'s current value. This reading has two possible outcomes: either p correctly obtains the value in the buffer or p obtains an inconsistent value because the buffer is overwritten while p reads it. In the latter case, the key property of our algorithm is that p is helped (and informed that it is helped) before the completion of its reading of the buffer. Thus, in either case, p has a valid value: either p reads a valid value in the buffer (former case) or it is handed a valid value by a helper process (latter case). The implementation of such a helping scheme is sketched in the following paragraph.

Consider any process p that performs an LL operation on $\mathcal{O}$ and obtains a value V associated with sequence number s (i.e., the latest SC before p's LL wrote V in $\mathcal{O}$ and had the sequence number s). Following its LL, suppose that p invokes an SC operation. Before attempting to make this SC operation (of sequence number $(s+1) \bmod 2N$) succeed, our algorithm requires p to check if the process $s \bmod N$ has an ongoing LL operation that requires help (thus, the decision of which process to help is based on sequence number). If so, p hands over the buffer it owns containing the value V to the process $s \bmod N$. If several processes try to help, only one will succeed. Thus, the process numbered $s \bmod N$ is helped (if necessary) every time the sequence number changes from s to $(s+1) \bmod 2N$. Since the sequence number increases by 1 with each successful SC, it follows that every process is examined twice for possible help in a span of $2N$ successful SC

operations. Recall further the earlier stated property that the buffer holding $\mathcal{O}$'s current value is not reused until $2N$ more successful SC's are performed. As a consequence of the above facts, if a process p begins reading the buffer that holds $\mathcal{O}$'s current value and the buffer happens to be reused while p still reads it (because $2N$ successful SC's have since taken place), some process is sure to have helped p by handing it a valid value of $\mathcal{O}$.

5.1.3 The role of `Help`p

The variable `Help`p plays an important role in the helping scheme. It has two fields: a binary value (that indicates whether p needs help) and a buffer index. When p initiates an LL operation, it seeks the help of other processes by writing $(1, b)$ into `Help`p, where b is the index of the buffer that p owns (see Line 1). If a process q helps p, it does so handing over its buffer c containing a valid value of $\mathcal{O}$ to p by writing $(0, c)$. (This writing is performed with a SC operation to ensure that at most one process succeeds in helping p.) Once q writes $(0, c)$ in `Help`p, p and q exchange the ownership of their buffers: p becomes the owner of the buffer indexed by c and q becomes the owner of the buffer indexed by b. (This buffer management scheme is the same as in Herlihy's universal construction [Her93].)

The above ideas are implemented as follows. Before p returns from its LL operation, it withdraws its request for help by executing the code at Lines 8–10. First, p reads `Help`p (Line 8). If p was already helped (i.e., `Help`$p \equiv (0, *)$), p updates $mybuf_p$ to reflect that p's ownership has changed to the buffer in which the helper process had left a valid value (Line 10). If p was not yet helped, p attempts to withdraw its request for help by writing 0 into the first field of `Help`p (Line 9). If p does not succeed, some process must have helped p while p was between

Lines 8 and 9; in this case, p assumes the ownership of the buffer handed by that helper (Line 10). If p succeeds in writing 0, then the second field of `Help`p still contains the index of p's own buffer, and so p reclaims the ownership of its own buffer (Line 10).

5.1.4 Two obligations of LL

In Section 4.3 of Chapter 4, we stated the two conditions that any implementation of an LL operation must satisfy in order to ensure correctness. We restate these two conditions below.

Consider an execution of the LL procedure by a process p. Suppose that V is the value of $\mathcal{O}$ when p invokes the LL procedure and suppose that k successful SC's take effect during the execution of this procedure, changing $\mathcal{O}$'s value from V to V_1, V_1 to V_2, ..., V_{k-1} to V_k. We call each of the values $V, V_1, \ldots, V_k$ a *valid* value (with respect to this LL execution by p), since it would be legitimate for p's LL to return any one of these values. Also, we call the value V_k *current* (with respect to this LL execution by p).

Although p's LL operation could legitimately return any valid value, there is a significant difference between returning the current value V_k versus returning an older valid value from $V, V_1, \ldots, V_{k-1}$: assuming that no successful SC operation takes effect between p's LL and p's subsequent SC, the specification of LL/SC operations requires p's subsequent SC to succeed in the former case and fail in the latter case. Thus, p's LL procedure, besides returning a valid value, has the additional obligation of ensuring the success or failure of p's subsequent SC (or VL) based on whether or not its return value is current.

In our algorithm, the SC procedure (Lines 12–22) includes exactly one SC op-

eration on the variable X (Line 19), and the procedure succeeds if and only if the operation succeeds. Therefore, we can restate the two obligations on p's LL procedure as follows: ($O1$) it must return a valid value U, and ($O2$) if no successful SC is performed after p's LL, p's subsequent SC (or VL) on X must succeed if and only if the return value U is current.

5.1.5 Code for LL

A process p performs an LL operation on $\mathcal{O}$ by executing the procedure LL($p, \mathcal{O}$, *retval*), where *retval* is a pointer to a block of W-words in which to place the return value. First, p announces its operation to inform others that it may need their help (Line 1). It then attempts to obtain the current value of $\mathcal{O}$ by performing the following steps. First, p reads X to determine the buffer holding $\mathcal{O}$'s current value (Line 2), and then reads that buffer (Line 3). While p reads the buffer at Line 3, the value of $\mathcal{O}$ might change because of successful SC's by other processes. Specifically, there are three possibilities for what happens while p executes Line 3: (1) no successful SC is performed by any process, (2) fewer than $2N - 1$ successful SC's are performed, or (3) at least $2N$ successful SC's are performed. In the first case, it is obvious that p reads a valid value at Line 3. Interestingly, in the second case too, the value read at Line 3 is a valid value. This is because, as remarked earlier, our algorithm does not reuse a buffer until $2N$ more successful SC's have taken place. In the third case, p cannot rely on the value read at Line 3. However, by the helping mechanism described earlier, a helper process would have made available a valid value in a buffer and written the index of that buffer in Help$p$. Thus, in each of the three cases, p has access to a valid value. Further, as we now explain, p can also determine which of the three cases actually

holds. To do this, p reads `Help`p to check if it has been helped (Line 4). If it has not been helped yet, Case (1) or (2) must hold, which implies that *retval* has a valid value of $\mathcal{O}$. Hence, returning this value meets the obligation $O1$. It meets obligation $O2$ as well because the value in *retval* is the current value of $\mathcal{O}$ at the moment when p read `X` (Line 2); hence, p's subsequent SC (or VL) on `X` will succeed if and only if `X` does not change, i.e., if and only if the value in *retval* is still current. So, p returns from the LL operation after withdrawing its request for help (Lines 8–10) and storing the return value into p's own buffer (Line 11) (p will use this buffer in the subsequent SC operation to help another process complete its LL operation, if necessary).

If upon reading `Help`p (Line 4), p finds out that it has been helped, p knows that a helper process must have written in `Help`p the index of a buffer containing a valid value U of $\mathcal{O}$. However, p is unsure whether this valid value U is current or old. If U is current, it is incorrect to return U: the return of U will fail to meet the obligation $O2$. This is because p's subsequent SC on `X` will fail, contrary to $O2$ (it will fail because `X` has changed since p read it at Line 2). For this reason, although p has access to a valid value handed to it by the helper, it does not return it. Instead, p attempts once more to obtain the current value of $\mathcal{O}$ (Lines 5–7). To do this, p again reads `X` to determine the buffer holding $\mathcal{O}$'s current value (Line 5), and then reads that buffer (Line 6). Next, p validates `X` (Line 7). If this validation succeeds, it is clear that *retval* has a valid value and, by returning this value, the LL operation meets both its obligations (O1 and O2). If the validation fails, $\mathcal{O}$'s value must have changed while p was between Lines 5 and 7. This implies that the value handed by the helper (which had been around even before p executed Line 5) is surely not current. Furthermore, the failure of VL (at Line 7) implies

that p's subsequent SC on X will fail. Thus, returning the value handed by the helper satisfies both obligations, $O1$ and $O2$. So, p copies the value handed by the helper into *retval* (Line 7), withdraws its request for help (Lines 8–10), and stores the return value into p's own buffer (Line 11), to be used in p's subsequent SC operation.

5.1.6 Code for SC

A process p performs an SC operation on $\mathcal{O}$ by executing the procedure SC(p, $\mathcal{O}$, v), where v is the pointer to a block of W words which contain the value to write to $\mathcal{O}$ if SC succeeds. On the assumption that X hasn't changed since p read it in its latest LL, i.e., X still contains the buffer index *bindex* and the sequence number s associated with the latest successful SC, p reads the buffer index b in Bank$s$ (Line 12). The reason for this step is the possibility that Bank$s$ has not yet been updated to hold *bindex*, in which case p should update it. So, p checks whether there is a need to update Bank$s$, by comparing b with *bindex* (Line 12). If there is a need to update, p first validates X (Line 12) to confirm its earlier assumption that X still contains the buffer index *bindex* and the sequence number s. If this validation fails, it means that the values that p read from X have become stale, and hence p abandons the updating. (Notice that, in this case, p's SC operation also fails.) If the validation succeeds, p attempts to update Bank$s$ (Line 13). This attempt will fail if and only if some process did the updating while p executed Lines 12–13. Hence, by the end of this step, Bank$s$ is sure to hold the value *bindex*.

Next, p tries to determine whether some process needs help with its LL operation. Since p's SC is attempting to change the sequence number from s to $s + 1$, the process to help is $q = s \bmod N$. So, p reads Help$q$ to check whether q needs

help (Line 14). If it does, p first validates X (Line 15) to make sure that X still contains the buffer index *bindex* and the sequence number s. If this validation fails, it means that the values that p read from X have become stale, and hence p abandons the helping. (Notice that, in this case, p's SC operation also fails.) If the validation succeeds, p attempts to help q by handing it p's buffer which, by Line 11, contains a valid value of $\mathcal{O}$ (Line 15). If p succeeds in helping q, p gives up its buffer to q and assumes ownership of q's buffer (Line 16). (Notice that p's SC at Line 15 fails if and only if, while p executed Lines 14–15, either another process already helped q or q withdrew its request for help.)

Next, p copies the value v into its buffer (Line 17). Then, p reads the index e of the buffer that holds $\mathcal{O}$'s old value associated with the next sequence number, namely, $(s+1) \bmod 2N$ (Line 18). Finally, p attempts its SC operation (Line 19) by trying to write in X the index of its buffer and the next sequence number $(s+1) \bmod 2N$. This SC will succeed if and only if no successful SC was performed since p's latest LL. Accordingly, the procedure returns *true* if and only if the SC at Line 19 succeeds (Lines 21–22). In the event that SC is successful, p gives up ownership of its buffer, which now holds $\mathcal{O}$'s current value, and becomes the owner of BUFe, the buffer holding $\mathcal{O}$'s old value with sequence number s', which can now be safely reused (Line 20).

The procedure VL is self-explanatory (Line 23). Based on the above discussion, we have the following theorem. Its proof is given in Appendix A.2.

Theorem 6 *Algorithm 5.1 is wait-free and implements a linearizable N-process W-word LL/SC object $\mathcal{O}$ from small LL/SC objects and registers. The time complexity of LL, SC, and VL operations on $\mathcal{O}$ are $O(W)$, $O(W)$ and $O(1)$, respec-*

tively. The implementation requires $O(NW)$ registers and $3N + 1$ small LL/SC objects.

Since each process can have at most two outstanding LL operations in the algorithm (i.e., $k = 2$), by combining the above theorem with Theorem 5 we obtain the following result.

Theorem 7 *There exists an N-process wait-free implementation of a W-word LL/SC object $\mathcal{O}$ from word-sized CAS objects and registers. The space complexity of the implementation is $O(NW)$, and the time complexity of LL, SC, and VL operations on $\mathcal{O}$ are $O(W)$, $O(W)$, and $O(1)$, respectively.*

Chapter 6

LL/SC for large number of objects

The algorithm in the previous chapter requires $O(NW)$ space to implement a W-word LL/SC object. Although these space requirements are modest when a single LL/SC object is implemented, the algorithm does not scale well when the number of LL/SC objects to be supported is large. In particular, in order to implement M W-word LL/SC objects, the algorithm requires $O(NMW)$ space. In this chapter, we show how to remove this multiplicative factor (i.e., NM) from the space complexity, while still maintaining the optimal running times for LL and SC. More precisely, we present the following two results.

- An algorithm with $O(Nk+(N+M)W)$ space complexity that implements M W-word Weak-LL/SC objects from word-sized CAS objects and registers.
- An algorithm with $O(Nk+(N^2+M)W)$ space complexity that implements M W-word LL/SC objects from word-sized CAS objects and registers.

When constructing a large number of LL/SC objects ($M \gg N$), our second algorithm is the first in the literature to be simultaneously (1) wait-free, (2) time optimal, and (3) space efficient. Sections 6.1 and 6.2 discuss the above two algorithms in detail.

6.1 Implementing an array of M W-word Weak-LL/SC objects

Recall that a Weak-LL/SC object is the same as the LL/SC object, except that its LL operation—denoted WLL—is not always required to return the value of the object: if the subsequent SC operation is sure to fail, then WLL may simply return the identity of a process whose successful SC took effect during the execution of that WLL. Thus, the return value of WLL is of the form (flag, v), where either (1) $\mathit{flag} = \mathit{success}$ and v is the value of the object $\mathcal{O}$, or (2) $\mathit{flag} = \mathit{failure}$ and v is the id of a process whose SC took effect during the WLL. We slightly modify the above semantics to account for the fact that we are now implementing a *multiword* LL/SC object. More precisely, we require that an empty buffer, which will be used by the WLL operation to store the return value, be passed to the WLL operation as an argument. Hence, WLL returns either *success* (in which case the buffer contains the return value of WLL) or ($\mathit{failure}, v$), where v is the id of a process whose SC took effect during the WLL (in which case the buffer contains an arbitrary value).

Our algorithm is designed in two steps:

1. Implement an array of M small LL/SC objects from word-sized CAS objects and registers.

2. Implement an array of M W-word Weak-LL/SC object from an array of M small LL/SC objects and registers.

The implementation of the first step follows immediately from the algorithm by Moir [Moi97a] (see Theorem 5 in the previous chapter). For the rest of this section, we focus on the implementation of the second step.

Algorithm 6.1 implements an array $\mathcal{O}0\,..\,M-1$ of M W-word Weak-LL/SC objects from small LL/SC objects and registers. We begin by describing the variables used in the algorithm. $\texttt{BUF}0\,..\,M+N-1$ is an array of $M+N$ W-word buffers. Of these, M buffers hold the current values of the M implemented objects (i.e., $\mathcal{O}0\,..\,M-1$) and the remaining N buffers are "owned" by processes, one buffer by each process. Process p's local variable, $mybuf_p$, is the index of the buffer currently owned by p. $\texttt{X}i$ holds the tag associated with the current value of $\mathcal{O}i$ and consists of two fields: the identity of the last process to perform a successful SC on $\mathcal{O}i$ and the index of the buffer that holds the current value of $\mathcal{O}i$.

We now describe procedure LL(p, i, *retval*) that enables process p to read the current value of an object $\mathcal{O}i$ into *retval*. First, p performs an LL operation on variable $\texttt{X}i$ in order to obtain the tag x associated with the most recent successful SC operation on $\mathcal{O}i$ (Line 1). The *buf* field of x tells p which of the $M+N$ buffers has $\mathcal{O}i$'s current value. So, p copies the value of that buffer into *retval* (Line 2), and then checks whether $\texttt{X}i$ has been modified (Line 3). If $\texttt{X}i$ has not been modified, it means that no process has performed a successful SC on $\mathcal{O}i$ while p was executing Line 3. Therefore, the value of $\mathcal{O}i$ that p read at Line 3 is correct. So, p's WLL returns, signaling success (Line 3). If, on the other hand, VL returns *false*, one or more successful SC's must have been performed on $\mathcal{O}i$ while p was executing

Types

xtype = **record** *pid*: $0\,..\,N-1$; *buf*: $0\,..\,M+N-1$ **end**

Shared variables

X: **array** $0\,..\,M-1$ of xtype

BUF: **array** $0\,..\,M+N-1\,..\,W-1$ **of** 64-bit word

Local persistent variables at each $p \in \{0, 1, \ldots, N-1\}$

$mybuf_p$: $0\,..\,M+N-1$

Initialization

Xj = $(*, j)$, for all $j \in \{0, 1, \ldots, M-1\}$

BUFj = the desired initial value of $\mathcal{O}j$, for all $j \in \{0, 1, \ldots, M-1\}$

For all $p \in \{0, 1, \ldots, N-1\}$

$mybuf_p = M + p$

```
    procedure WLL(p, i, retval)
1:  x = LL(Xi)
2:  copy BUFx.buf into *retval
3:  if VL(Xi) return success
4:  y = read(Xi)
5:  return (failure, y.pid)

    procedure VL(p, i) returns boolean
6:  return VL(Xi)

    procedure SC(p, i, v) returns boolean
7:  x = read(Xi)
8:  copy *v into BUFmybuf_p
9:  if SC(Xi, (p, mybuf_p))
10:     mybuf_p = x.buf
11:     return true
12: return false
```

Algorithm 6.1: An implementation of $\mathcal{O}0\,..\,M-1$: an array of M N-process W-word Weak-LL/SC objects, using small LL/SC objects and registers

Line 3. The *pid* field of Xi holds the name of such a process. So, p reads Xi (Line 4) and returns its *pid* field (Line 5).

We now describe procedure SC(p, i, v) that enables process p to write a value v into an object $\mathcal{O}i$. First, p reads Xi to obtain the tag x associated with the most recent successful SC operation on $\mathcal{O}i$ (Line 7). (Notice that this value is junk if some process performed a successful SC since p's latest WLL, but in this case the SC will fail at Line 9, resulting in no harm.) Next, p copies the value v into the buffer that it currently owns (Line 8), and then executes the SC operation on Xi in an attempt to modify Xi to point to the new value (Line 9). This operation succeeds if and only if no process performed a successful SC on $\mathcal{O}i$ since p's latest

WLL. Consequently, p returns from the SC procedure signaling success at Line 11 (respectively, failure at Line 12) if it succeeds (respectively, fails) in modifying $\mathtt{X}i$ at Line 9. Furthermore, if p succeeds in modifying $\mathtt{X}i$, then p's buffer has $\mathcal{O}i$'s current value, whereas the buffer that held $\mathcal{O}i$'s previous value becomes free. To reflect this change, p gives up its buffer and assumes ownership of the buffer just released (Line 10). (This buffer management scheme is the same as in Herlihy's universal construction [Her93].)

The VL operation returns success if and only if $\mathtt{X}i$ has not changed since p's most recent WLL (Line 6). Based on the above discussion, we have the following theorem. Its proof is given in Appendix A.3.1.

Theorem 8 *Algorithm 6.1 is wait-free and implements an array $\mathcal{O}0\,..\,M-1$ of M N-process W-word Weak-LL/SC objects. The time complexity of WLL, SC, and VL operations on $\mathcal{O}$ are $O(W)$, $O(W)$ and $O(1)$, respectively. The implementation requires $O((N+M)W)$ registers and M small LL/SC objects.*

By combining Theorem 8 with Theorem 5, we obtain the following result.

Theorem 9 *There exists an N-process wait-free implementation of an array $\mathcal{O}0\,..\,M-1$ of M W-word Weak-LL/SC objects from word-sized CAS objects and registers. The time complexity of WLL, SC, and VL operations on $\mathcal{O}$ are $O(W)$, $O(W)$ and $O(1)$, respectively. The space complexity of the implementation is $O(Nk+(N+M)W)$, where k is the maximum number of outstanding WLL operations of a given process The CAS-time-complexity of WLL, SC, and VL are* 0, 1 *and* 0, *respectively.*

6.2 Implementing an array of M W-word LL/SC objects

Algorithm 6.2 implements an array $\mathcal{O}0\,..\,M-1$ of M W-word LL/SC object from CAS objects and registers. We now describe how the algorithm works.

Types

valuetype = **array** $0\,..\,W$ **of** 64-bit value

xtype = **record** *seq*: $(64 - \lg(M + (N+1)N))$-bit number;
buf: $0\,..\,M + (N+1)N - 1$ **end**

helptype = **record** *seq*: $(63 - \lg(M + (N+1)N))$-bit number; *helpme*: $\{0, 1\}$;
buf: $0\,..\,M + (N+1)N - 1$ **end**

Shared variables

X: **array** $0\,..\,M-1$ of xtype; Announce: **array** $0\,..\,N-1$ **of** $0\,..\,M-1$

Help: **array** $0\,..\,N-1$ **of** helptype

BUF: **array** $0\,..\,M + (N+1)N - 1$ **of** $*$valuetype

Local persistent variables at each $p \in \{0, 1, \ldots, N-1\}$

$mybuf_p$: $0\,..\,M + (N+1)N - 1$; Q_p: Single-process queue; x_p: xtype

$lseq_p$: $(63 - \lg(M + (N+1)N))$-bit number; $index_p$: $0\,..\,N-1$

Initialization

X$k = (0, k)$, for all $k \in \{0, 1, \ldots, M-1\}$

BUFk = the desired initial value of $\mathcal{O}k$, for all $k \in \{0, 1, \ldots, M-1\}$

For all $p \in \{0, 1, \ldots, N-1\}$

enqueue$(Q_p, M + (N+1)p + k)$, for all $k \in \{0, 1, \ldots, N-1\}$

$mybuf_p = M + (N+1)p + N$; Help$p = (0, 0, *)$; $index_p = 0$; $lseq_p = 0$

---

procedure LL$(p, i, retval)$

1: Announce$p = i$
2: Help$p = (++lseq_p, 1, mybuf_p)$
3: $x_p =$ Xi
4: **copy** $*$BUF$x_p.buf$ **into** $*retval$
5: **if** $\neg$CAS(Help$p$, $(lseq_p, 1, mybuf_p)$, $(lseq_p, 0, mybuf_p)$)
6: $mybuf_p =$ Help$p.buf$
7: $x_p =$ BUF$mybuf_p W$
8: **copy** $*$BUF$mybuf_p$ **into** $*retval$
9: **return**

procedure VL(p, i) **returns** boolean

10: **return** $(\text{X}i = x_p)$

procedure SC(p, i, v) **returns** boolean

11: **copy** $*v$ **into** $*$BUF$mybuf_p$
12: **if** $\neg$CAS(Xi, x_p, $(x_p.seq + 1, mybuf_p)$)
13: **return** *false*
14: *enqueue*$(Q_p, x_p.buf)$
15: $mybuf_p = dequeue(Q_p)$
16: **if** $(\text{Help}index_p \equiv (s, 1, b))$
17: $j =$ Announce$index_p$
18: $x =$ Xj
19: **copy** $*$BUF$x.buf$ **into** $*$BUF$mybuf_p$
20: BUF$mybuf_p W = x$
21: **if** CAS(Help$index_p$, $(s, 1, b)$, $(s, 0, mybuf_p)$)
22: $mybuf_p = b$
23: $index_p = (index_p + 1) \bmod N$
24: **return** *true*

Algorithm 6.2: Implementation of $\mathcal{O}0\,..\,M-1$: an array of M N-process W-word LL/SC objects

6.2.1 The variables used

We begin by describing the variables used in the algorithm. BUF$0\,..\,M + (N + 1)N - 1$ is an array of $M + (N+1)N$ buffers. Of these, M buffers hold the current values of objects $\mathcal{O}0, \mathcal{O}1, \ldots, \mathcal{O}M - 1$, while the remaining $(N+1)N$ buffers are "owned" by processes, $N + 1$ buffer by each process. Each process p, however, uses only one of its $N + 1$ buffers at any given time. The index of the buffer that p is currently using is stored in the local variable $mybuf_p$, and the indices of the remaining N buffers are stored in p's local queue Q_p. Array X$0\,..\,M - 1$ holds the tags associated with the current values of objects $\mathcal{O}0, \mathcal{O}1, \ldots, \mathcal{O}M - 1$. A tag in X$i$ consists of two fields: (1) the index of the buffer that holds $\mathcal{O}i$'s current value, and (2) the sequence number associated with $\mathcal{O}i$'s current value. The sequence number increases by 1 with each successful SC on $\mathcal{O}i$, and the buffer holding $\mathcal{O}i$'s current value is not reused until some process performs at least N more successful SC's (on any $\mathcal{O}j$). Process p's local variable x_p maintains the tag corresponding to the value returned by p's most recent LL operation; p will use this tag during the subsequent SC operation to check whether the object still holds the same value (i.e., whether it has been modified). Finally, it turns out that a process p might need the help of other processes in completing its LL operation on $\mathcal{O}$. The shared variables Help$p$ and Announce$p$, as well as p's local variables $lseq_p$ and $index_p$, are used to facilitate this helping scheme. Additionally, an extra word is kept in each buffer along with the value. Hence, all the buffers in the algorithm are of length $W + 1$.[1]

[1] The only exception are the buffers passed as an argument to procedures LL and SC, which are of length W.

6.2.2 The helping mechanism

The crux of our algorithm lies in its helping mechanism by which SC operations help LL operations. This helping mechanism is similar to that of Algorithm 5.1, but whereas the mechanism of Algorithm 5.1 requires $O(NMW)$ space, the new mechanism requires only $O(Nk + (N^2 + M)W)$ space. Below, we describe this mechanism in detail.

A process p begins its LL operation on some object $\mathcal{O}i$ by announcing its operation to other processes. It then attempts to read the buffer containing $\mathcal{O}i$'s current value. This reading has two possible outcomes: either p correctly obtains the value in the buffer or p obtains an inconsistent value because the buffer is overwritten while p reads it. In the latter case, the key property of our algorithm is that p is helped (and informed that it is helped) before the completion of its reading of the buffer. Thus, in either case, p has a valid value: either p reads a valid value in the buffer (former case) or it is handed a valid value by a helper process (latter case). The implementation of such a helping scheme is sketched in the following paragraph.

Consider any process p that performs a successful SC operation. During that SC, p checks whether a single process—say, q—has an ongoing LL operation that requires help. If so, p helps q by passing it a valid value and a tag associated with that value. (We will see later how p obtains that value.) If several processes try to help, only one will succeed. Process p makes a decision on which process to help by consulting its variable $index_p$: if $index_p$ holds value j, then p helps process j. The algorithm ensures that $index_p$ is incremented by 1 modulo N after every successful SC operation by p. Hence, during the course of N successful

SC operations, process p examines all N processes for possible help. Recall the earlier stated property that the buffer holding an $\mathcal{O}i$'s current value is not reused until some process performs at least N more successful SC's (on any $\mathcal{O}j$). As a consequence of the above facts, if a process q begins reading the buffer that holds $\mathcal{O}i$'s current value and the buffer happens to be reused while q still reads it (because some process p has since performed N successful SC's), then p is sure to have helped q by handing it a valid value of $\mathcal{O}i$ and a tag associated with that value.

6.2.3 The roles of `Help`p and `Announce`p

The variables `Help`p and `Announce`p play important roles in the helping scheme. `Help`p has three fields: (1) a binary value (that indicates if p needs help), (2) a buffer index, and (3) a sequence number (independent from the sequence numbers in tags). `Announce`p has only one field: an index in the range $0\,..\,M-1$. When p initiates an LL operation on some object $\mathcal{O}i$, it first announces the index of that object by writing i into `Announce`p (see Line 1), and then seeks the help of other processes by writing $(s, 1, b)$ into `Help`p, where b is the index of the buffer that p owns (see Line 2) and s is p's local sequence number incremented by one. If a process q helps p, it does so handing over its buffer c containing a valid value of $\mathcal{O}i$ to p by writing $(s, 0, c)$ into `Help`p. (This writing is performed with a CAS operation to ensure that at most one process succeeds in helping p.) Once q writes $(s, 0, c)$ in `Help`p, p and q exchange the ownership of their buffers: p becomes the owner of the buffer indexed by c and q becomes the owner of the buffer indexed by b. (This buffer management scheme is the same as in Herlihy's universal construction [Her93].) Before q hands over buffer c to process p, it also writes a

tag associated with that value into the Wth location of the buffer. Hence, all the buffers used by the algorithm are of size $W + 1$.

6.2.4 Obtaining a valid value

We now explain the mechanism by which a process p obtains a valid value to help some other process q with. Suppose that process p wishes to help process q complete its LL operation on some object $\mathcal{O}i$. To obtain a valid value to help q with, p first attempts to read the buffer containing $\mathcal{O}i$'s current value. This reading has two possible outcomes: either p correctly obtains the value in the buffer or p obtains an inconsistent value because the buffer is overwritten while p reads it. In the latter case, by an earlier stated property, p knows that there exists some process r that has performed at least N successful SC operations (on any $\mathcal{O}j$). Therefore, r must have already helped q, in which case p's attempt to help q will surely fail. Hence, it does not matter that p obtained an inconsistent value of $\mathcal{O}i$ because p will anyway fail in giving that value to q. As a result, if p helps q complete its LL operation on some object $\mathcal{O}i$, it does so with a valid value of $\mathcal{O}i$.

6.2.5 Code for LL

A process p performs an LL operation on some object $\mathcal{O}i$ by executing the procedure LL($p, i,$ *retval*), where *retval* is a pointer to a block of W words in which to place the return value. First, p announces its operation to inform others that it needs their help (Lines 1 and 2). It then attempts to obtain the current value of $\mathcal{O}i$ by performing the following steps. First, p reads the tag stored in $\mathtt{X}i$ to determine the buffer holding $\mathcal{O}i$'s current value (Line 3), and then reads that buffer (Line 4). While p reads the buffer at Line 4, the value of $\mathcal{O}i$ might change because

of successful SC's by other processes. Specifically, there are three possibilities for what happens while p executes Line 4: (1) no process performs a successful SC, (2) no process performs more than $N - 1$ successful SC's, or (3) some process performs N or more successful SC's. In the first case, it is obvious that p reads a valid value at Line 4. Interestingly, in the second case too, the value read at Line 4 is a valid value. This is because, as remarked earlier, our algorithm does not reuse a buffer until some process performs at least N successful SC's. In the third case, p cannot rely on the value read at Line 4. However, by the helping mechanism described earlier, a helper process would have made available a valid value (and a tag associated with that value) in a buffer and written the index of that buffer in `Help`p. Thus, in each of the three cases, p has access to a valid value as well as a tag associated with that value. Further, as we now explain, p can also determine which of the three cases actually holds. To do this, p performs a CAS on `Help`p to try to revoke its request for help (Line 5). If p's CAS succeeds, it means that p has not been helped yet. Therefore, Case (1) or (2) must hold, which implies that *retval* has a valid value of $\mathcal{O}$. Hence, p returns from the LL operation at Line 9.

If p's CAS on `Help`p fails, p knows that it has been helped and that a helper process has written in `Help`p the index of a buffer containing a valid value U of $\mathcal{O}i$ (as well as a tag associated with U). So, p reads U and its associated tag (Lines 7 and 8), and takes ownership of the buffer it was helped with (Line 6). Finally, p returns from the LL operation at Line 9.

6.2.6 Code for SC

A process p performs an SC operation on some object $\mathcal{O}i$ by executing the procedure SC(p, i, v), where v is the pointer to a block of W words which contain the

value to write to $\mathcal{O}i$ if SC succeeds. First, p writes the value v into its local buffer (Line 11), and then tries to make its SC operation take effect by changing the value in $\mathtt{X}i$ from the tag it had witnessed in its latest LL operation to a new tag consisting of (1) the index of p's local buffer and (2) a sequence number (of the previous tag) incremented by one (Line 12). If the CAS operation fails, it follows that some other process performed a successful SC after p's latest LL, and hence p's SC must fail. Therefore, p terminates its SC procedure, returning *false* (Line 13). On the other hand, if CAS succeeds, then p's current SC operation has taken effect. In that case, p gives up ownership of its local buffer, which now holds $\mathcal{O}i$'s current value, and becomes the owner of the buffer B holding $\mathcal{O}i$'s old value. To remain true to the promise that the buffer that held $\mathcal{O}i$'s current value (B, in this case) is not reused until some process performs at least N successful SC's, p enqueues the index of buffer B into its local queue (Line 14), and then dequeues some other buffer index from the queue (Line 15). Notice that, since p's local queue contains N buffer indices when p inserts B's index into it, p will not reuse buffer B until it performs at least N successful SC's.

Next, p tries to determine whether some process needs help with its LL operation. As we stated earlier, the process to help is $q = index_p$. So, p reads $\mathtt{Help}q$ to check whether q needs help (Line 16). If it does, p consults variable $\mathtt{Announce}q$ to learn the index j of the object that q needs help with (Line 17). Next, p reads the tag stored in $\mathtt{X}j$ to determine the buffer holding $\mathcal{O}j$'s current value (Line 18), and then copies the value from that buffer into its own buffer (Line 19). Then, p writes into the Wth location of the buffer the tag that it read from $\mathtt{X}j$ (Line 20). Finally, p attempts to help q by handing it p's buffer (Line 21). If p succeeds in helping q, then, by an earlier argument, the buffer that p handed over to q contains

a valid value of $\mathcal{O}j$. Hence, p gives up its buffer to q and assumes ownership of q's buffer (Line 22). (Notice that p's CAS at Line 21 fails if and only if, while p executed Lines 16–21, either another process already helped q or q withdrew its request for help.) Regardless of whether process q needed help or not, p increments the $index_p$ variable by 1 modulo N (Line 23) to ensure that in the next successful SC operation it helps some other process (Line 23), and then terminates its SC procedure by returning *true* (Line 24).

The procedure VL is self-explanatory (Line 10). Based on the above discussion, we have the following theorem. Its proof is given in Appendix A.3.2.

Theorem 10 *Algorithm 6.2 is wait-free and implements an array $\mathcal{O}0..M-1$ of M linearizable N-process W-word LL/SC objects. The time complexity of LL, SC and VL operations on $\mathcal{O}$ are $O(W)$, $O(W)$ and $O(1)$, respectively. The space complexity of the implementation is $O(Nk+(M+N^2)W)$, where k is the maximum number of outstanding LL operations of a given process.*

6.2.7 Remarks

Sequence number wrap-around

Each 64-bit variable $\mathtt{X}i$ stores in it a buffer index and an unbounded sequence number. The algorithm relies on the assumption that during the time interval when some process p executes one LL/SC pair, the sequence number stored in $\mathtt{X}i$ does not cycle through all possible values. If we reserve 32 bits for the buffer index (which allows the implementation of up to 2^{31} LL/SC objects, shared by up to $2^{15} = 32{,}768$ processes), we still will have 32 bits for the sequence number, which is large enough that sequence number wraparound is not a concern in practice.

The number of outstanding LL operations

Modifying the code in Algorithm 6.2 to handle multiple outstanding LL/SC operations is straightforward. Simply require that each LL operation, in addition to returning a value, also returns the associated tag, which the caller must pass to the matching SC.

Chapter 7

LL/SC for unknown N

Notice that the LL/SC algorithm in the previous chapter requires that N—the maximum number of processes accessing the algorithm—is known in advance. This knowledge of N is used in three places in the algorithm: (1) to initialize arrays `Help`, `Announce`, and `BUF` to sizes N, N, and $M + N(N + 1)$, respectively; (2) to initialize a local queue at each process to size N; and (3) to implement the helping scheme by which a process helps all other processes in a span of N successful SC operations.

The drawback to the above requirement is that, in cases where N cannot be known in advance, a conservative estimate has to be applied which can result in space wastage. For example, if N is estimated too conservatively, arrays `Help`, `Announce`, and `BUF` would occupy much more space than is actually needed. It is therefore more desirable to have an algorithm that makes no assumptions on N, but instead adapts to the actual number of participating processes.

In this chapter, we present such an algorithm. In particular, we design an algorithm that supports two new operations—*Join*(p) and *Leave*(p)—which allow a

process p to join and leave the algorithm at any given time. If K is the maximum number of processes that have simultaneously participated in the algorithm (i.e., that have joined the algorithm but have not yet left it), then the space complexity of the algorithm is $O(Kk + (K^2 + M)W)$. The time complexities of procedures Join, Leave, LL, SC, and VL are $O(K)$, $O(1)$, $O(W)$, $O(W)$, and $O(1)$, respectively. This algorithm is a modified version of Algorithm 6.2 (see Chapter 6), and is described in detail in Section 7.1.

We also present a modified version of Algorithm 4.1 (see Chapter 4) that does not require N to be known in advance. The attractive feature of this algorithm is that the Join procedure runs in $O(1)$ time. This algorithm, however, does not allow processes to leave. The time complexities of LL, SC, and VL operations are $O(1)$, and the space complexity of the algorithm is $O(K^2 + KM)$. Section 7.2 discusses this algorithm in detail.

7.1 Implementing an array of M W-word LL/SC objects for an unknown number of processes

In this section, we present an algorithm that implements an array $\mathcal{O}0\,..\,M-1$ of M W-word LL/SC objects shared by an unknown number of processes. The algorithm is given in three steps. First, we introduce an important building block of the algorithm, namely, an implementation of a dynamic array that supports constant-time read and write operations (with some restrictions). Next, we restate Algorithm 6.2, but with small modifications that will make it easier to remove the assumption of N. Finally, we present our main result, namely, an algorithm that implements an array of M W-word LL/SC objects shared by an unknown number of processes.

These three steps are described in detail in Sections 7.1.1, 7.1.2, and 7.1.3.

7.1.1 Dynamic arrays

A dynamic array is just like a regular array except that it places no bound on the highest location that can be written. In particular, a process can write into the ith location of the dynamic array for any natural number i. At all times, the size of the array must stay proportional to the highest location written so far. Furthermore, all reads and writes in the array must complete in $O(1)$ time. In this thesis, we consider only a weaker version of dynamic array that has the following restrictions: (1) all writes into the same location write the same value, (2) a write into a location i must precede a read on that location, and (3) a write into a location i must precede a write into location $i + 1$. We capture the above restrictions in an object that we call a *DynamicArray* object. This object is formally defined as follows.

A DynamicArray object supports two operations: $write(i, v)$ and $read(i)$. The $write(i, v)$ operation writes value v into the ith location of the array, while the $read(i)$ operation returns the value stored in the ith location of the array. The following restrictions are placed on the usage of *read* and *write* operations:

- Before $write(k + 1, *)$ is invoked, at least one $write(k, *)$ must complete.
- Before $read(k)$ is invoked, at least one $write(k, *)$ must complete.
- If $write(k, v)$ and $write(k, v')$ are invoked, then $v = v'$.

Algorithm 7.1 implements a DynamicArray object from a CAS object and registers. In the following, we first describe the main idea behind the algorithm and then describe the algorithm in detail.

The main idea

The main idea of the algorithm is as follows. We maintain two (static) arrays at all times: array A of length k, and array B of length $2k$. (Initially, A is of length 1, and B is of length 2.) When a process writes a value into some array location j, for $j \geq k/2$, it writes that value into both Aj and Bj. Additionally, it copies the array location $Aj - k/2$ into $Bj - k/2$. By this mechanism, when the array A fills up (i.e., when the location $Ak - 1$ is written), all the locations of array A have been copied into array B. Therefore, B contains the same values as A, and can hence be used in place of A. A new array of length $4k$ is then allocated (and used in place of B), and the algorithm proceeds the same way as before.

The algorithm

Central to the algorithm is variable `D`, which stores a pointer to the block containing three fields: (1) a pointer to array A, (2) a pointer to array B, and (3) the length of array A.

We now explain the `write`$(p, i, v, \mathcal{D})$ procedure that describes how a process p writes a value v into the ith location of DynamicArray $\mathcal{D}$. In the following, let $c'(i)$ denote the largest power of 2 smaller than or equal to i.[1] First, p reads `D` to obtain a pointer to the block containing arrays A and B (Line 1). Next, p checks whether the length of array A is greater than i (Line 2). If it is, then p writes v into Ai (Line 3). To help with the amortized copying of array A into B, p writes v into Bi (Line 4) and copies the location $Ai - c'(i)$ into $Bi - c'(i)$ (Line 5).

Notice that, by initialization, the lengths of A and B are powers of 2 at all times.

[1] If $i = 0$, then $c'(i) = 0$.

Types
arraytype = **array of** 64-bit value
dtype = **record** *size*: 64-bit number; A, B: $*$arraytype **end**
Shared variables
D: $*$dtype
Initialization
D = malloc(sizeof dtype)
D$\rightarrow$*size* = 1;
D$\rightarrow A$ = malloc(1);
D$\rightarrow B$ = malloc(2);

procedure write($p, i, v, \mathcal{D}$)
1: d = D
2: **if** ($d\rightarrow$*size* $> i$)
3: $d\rightarrow Ai = v$
4: $d\rightarrow Bi = v$
5: $d\rightarrow Bi - c'(i) = d\rightarrow Ai - c'(i)$
6: **else** *newD* = malloc(sizeof dtype)
7: *newD*$\rightarrow A = d\rightarrow B$
8: *newD*$\rightarrow B$ = malloc($4 * d\rightarrow$*size*)
9: *newD*$\rightarrow$*size* $= 2 * d\rightarrow$*size*
10: **if** $\neg$CAS(D, d, *newD*) free(*newD*$\rightarrow B$); free(*newD*)
11: write($i, v, \mathcal{D}$)

procedure read($p, i, \mathcal{D}$) **returns** valuetype
12: d = D
13: **return** $d\rightarrow Ai$

Algorithm 7.1: An implementation of a DynamicArray object. Function $c'(i)$ returns the largest power of 2 smaller or equal to i. (If $i = 0$, then $c'(i) = 0$.)

Let k be the length of A when p tries to write v into Ai. Then, if $i \geq k/2$, we have $c'(i) = k/2$ (since k is a power of 2). Hence, p copies the location $Ai - k/2$ into $Bi - k/2$, which is consistent with our main idea presented earlier. If $i < k/2$, then, by definition of $c'(i)$, we have $i - c'(i) \geq 0$. Hence, p copies some location Aj into Bj, for $j \in \{0, 1, \ldots, k/2-1\}$, which causes no harm. Hence, by copying $Ai - c'(i)$ into $Bi - c'(i)$, p remains faithful to the earlier idea of amortized copying of array A into array B.

If the length of array A is equal to i, then p knows that the array A has been

filled up. Furthermore, by an earlier discussion, all the values in A have already been copied into B. So, p prepares a new block *newD* that will hold pointers to the new values for arrays A and B. Next, p sets *newD.A* to B (Line 7), *newD.B* to a newly allocated array twice the size of B (Line 8), and *newD.size* to the size of B (Line 9). Then, p attempts to swing the pointer in `D` from the block that p had witnessed at Line 1, to the new block *newD* (Line 10). If p's CAS is successful, then p has successfully installed the new block *newD* in `D`. Otherwise, some other process must have installed its own block into `D`, and so p frees up the memory occupied by the block *newD* (Line 10). In either case, the length of an array A in the new block is sure to be greater than i. So, p calls the `write` procedure again to complete installing value v into $\mathcal{D}$ (Line 11). Notice that, since the size of the new array A is strictly greater than i, p will not make another recursive call to `write`, thus ensuring a constant running time for the `write` operation.

The `read`$(p, i, \mathcal{D})$ procedure is very simple: a process p simply reads `D` to obtain a pointer to the block containing the most recent values of A and B (Line 12), and then returns the value stored in Ai (Line 13). Notice that, since we require that at least one *write*$(i, *)$ operation completes before *read*(i) starts, the length of the array A is at least $i+1$ when p reads Ai. Furthermore, by the above discussion, location Ai contains the value written by *write*$(i, *)$. Therefore, p returns the correct value.

We now calculate the space complexity of the algorithm at some time t. First, notice that there are only two arrays at time $t = 0$: one array of length 1 and one of length 2. During the first *write*$(1, *)$ operation, a new array of length 4 is allocated. Similarly, during the first *write*$(2, *)$ operation, a new array of length 8 is allocated. In general, during the first *write*$(2^j, *)$ operation, a new array of

length 2^{j+2} is allocated. So, if $write(K, *)$ is the operation with the highest index among all operations invoked prior to time t, then at time t the largest allocated array is of length $2^{\lfloor \lg K \rfloor + 2}$. Hence, the lengths of all allocated arrays at time t are $1, 2, 4, 8, \ldots, 2^{\lfloor \lg K \rfloor + 1}$, and $2^{\lfloor \lg K \rfloor + 2}$. Consequently, the space occupied by the arrays at time t is $2^{\lfloor \lg K \rfloor + 3} - 1$, and the space occupied by the blocks at time t is $\lfloor \lg K \rfloor + 1$. Therefore, the space permanently used by the algorithm at time t is $O(K)$. However, we also have to count the space occupied by the blocks and arrays allocated at Lines 6 and 8 that were not successfully installed in D but have not yet been freed from memory (at Line 10). The number of such blocks and arrays at time t is at most n, where n is the number of processes executing the algorithm at time t. Since the largest allocated array is of length at most $2^{\lfloor \lg K \rfloor + 2}$, the space used by the blocks and arrays at time t is $O(nK)$. Therefore, the space used by the algorithm at time t is $O(nK)$.

Based on the above discussion, we have the following theorem. Its proof is given in Appendix A.4.1.

Theorem 11 *Algorithm 7.1 is wait-free and implements a DynamicArray object $\mathcal{D}$ from a word-sized CAS object and registers. The time complexity of read and write operations on $\mathcal{D}$ is $O(1)$. The space used by the algorithm at any time t is $O(nK)$, where n is the number of processes executing the algorithm at time t, and K is the highest location written in $\mathcal{D}$ prior to time t.*

7.1.2 Restatement of Algorithm 6.2

We now restate Algorithm 6.2, which implements an array of M W-word LL/SC object shared by a fixed number of processes N. We introduce some modifications

to this algorithm that will make it easier to remove the assumption of N. Algorithm 7.2 shows the result of these modifications. In the following, we will refer to Algorithms 6.2 and 7.2 by the names A and B, respectively.

The main difference between algorithms A and B lies in the way the variables are organized. Below, we summarize the differences between the two organizations.

1. In algorithm A, process p's shared variables `Help`p and `Announce`p are located in shared arrays `Help` and `Announce`, respectively. Hence, if a process q wishes to access p's shared variables, it can do so by simply reading `Help`p or `Announce`p. In algorithm B, on the other hand, process p's shared variables are stored in p's own block of memory. Furthermore, an array `NameArray` holds pointers to memory blocks of all processes. Hence, if a process q wishes to access p's shared variables, it must first read `NameArray`p to obtain the address l of p's memory block, and then read the variables $l \rightarrow$`Help` and $l \rightarrow$`Announce`.

2. In algorithm A, there is a single array `BUF` of length $M + N(N+1)$ which holds all the buffers used by the algorithm. The index of a buffer is therefore a number in the range $0\,..\,M + N(N+1) - 1$. Hence, if a process wishes to access a buffer with index b, it simply reads the location `BUF`b. In algorithm B, on the other hand, array `BUF` is divided into $N+1$ smaller arrays: (1) a central array of length M, and (2) N arrays of length $N+1$ each, which are kept at processes' memory blocks (one array per process). The index of a buffer is therefore either a pair $(0, i)$, where i is the index into the central array, or a tuple $(1, p, i)$, where i is the index into the array located at pro-

Types

valuetype = **array** $0\,..\,W$ **of** 64-bit value

indextype = **record** *type*: {0, 1}; **if** (*type* = 0) (*bindex*: 31-bit number)
else (*name*: 15-bit number; *bindex*: 15-bit number) **end**

xtype = **record** *seq*: 32-bit number; *buf*: indextype **end**

helptype = **record** *seq*: 31-bit number; *helpme*: {0, 1};
buf: indextype **end**

blocktype = **record** Announce: 31-bit number; Help: helptype; Q: Single-process queue;
BUF: **array** $0\,..\,N$ **of** $*$valuetype; *index*: 15-bit number; *mybuf*: indextype;
lseq: 31-bit number; x: xtype **end**

Shared variables

X: **array** $0\,..\,M-1$ of xtype; BUF: **array** $0\,..\,M-1$ **of** $*$valuetype

NameArray: **array** $0\,..\,N-1$ **of** $*$blocktype

Local persistent variables at each p

loc_p: blocktype

Initialization

X$k = (0, k)$, for all $k \in \{0, 1, \ldots, M-1\}$

BUFk = the desired initial value of $\mathcal{O}k$, for all $k \in \{0, 1, \ldots, M-1\}$

For all $p \in \{0, 1, \ldots, N-1\}$

NameArray$p = \&loc_p$

enqueue($loc_p.Q$, (1, p, k)), for all $k \in \{0, 1, \ldots, N-1\}$

$loc_p.mybuf = (1, p, N)$; loc_p.Help $= (0, 0, *)$; $loc_p.index = 0$; $loc_p.lseq = 0$

procedure LL(p, i, *retval*)

1: loc_p.Announce $= i$
2: loc_p.Help $=$ (++$loc_p.lseq$, 1, $loc_p.mybuf$)
3: $loc_p.x =$ Xi
4: **copy** $*$GetBuf($loc_p.x.buf$) **into** $*$*retval*
5: **if** $\neg$CAS(loc_p.Help,
($loc_p.lseq$, 1, $loc_p.mybuf$),
($loc_p.lseq$, 0, $loc_p.mybuf$))
6: $loc_p.mybuf = loc_p$.Help.*buf*
7: $b =$ GetBuf($loc_p.mybuf$)
8: $loc_p.x = bW$
9: **copy** $*b$ **into** $*$*retval*
10: **return**

procedure GetBuf(b) **returns** $*$valuetype

11: **if** (b.*type* $= 0$) **return** BUFb.*bindex*
12: $l =$ NameArray$b$.*name*
13: **return** $l\rightarrow$BUFb.*bindex*

procedure VL(p, i) **returns** boolean

14: **return** (X$i = loc_p.x$)

procedure SC(p, i, v) **returns** boolean

15: **copy** $*v$ **into** $*$GetBuf($loc_p.mybuf$)
16: **if** $\neg$CAS(Xi, $loc_p.x$,
($loc_p.x.seq + 1$, $loc_p.mybuf$))
17: **return** *false*
18: *enqueue*($loc_p.Q$, $loc_p.x.buf$)
19: $loc_p.mybuf =$ *dequeue*($loc_p.Q$)
20: $l =$ NameArray$loc_p.index$
21: **if** ($l\rightarrow$Help $\equiv (s, 1, c)$)
22: $j = l\rightarrow$Announce
23: $x =$ Xj
24: $d =$ GetBuf($loc_p.mybuf$)
25: **copy** $*$GetBuf($x.buf$) **into** $*d$
26: $dW = x$
27: **if** CAS($l\rightarrow$Help, $(s, 1, c)$,
$(s, 0, loc_p.mybuf)$)
28: $loc_p.mybuf = c$
29: $loc_p.index = (loc_p.index + 1) \bmod N$
30: **return** *true*

Algorithm 7.2: Slightly modified version of Algorithm 6.2

cess p's memory block. Hence, if a process wishes to access a buffer with index $b = (0, i)$, it simply reads the location `BUF`i. If, on the other hand, a process wishes to access a buffer with index $b = (1, p, i)$, it must first read `NameArray`p to obtain the address l of p's memory block, and then read the location l.`BUF`i. This method (for accessing a buffer given its index) is captured by the procedure `GetBuf` (see Algorithm 7.2).

As in algorithm A, we will need to store a sequence number and a buffer index together in a single machine word. From the previous paragraph, the buffer index consists of one bit (to distinguish between the central array and an array stored at process's memory block) and $\lg(\max(M, N(N+1)))$ bits to describe either (1) an index within the central array, or (2) an index within a process's array and that process's name. Assuming 64 bits per machine word, this leaves $64 - 1 - \lg(\max(M, N(N+1)))$ bits for the sequence number. Rather than using these long expressions, in the rest of the section we assume the values 2^{31} and 2^{15} for M and N, respectively, which then leaves 32 bits for the sequence number.

3. In algorithm A, each process p maintains persistent local variables $mybuf_p$, $lseq_p$, $index_p$, x_p, and Q_p. In algorithm B, on the other hand, all of the above variables are located in p's memory block and p maintains the address of that memory block in its persistent local variable loc_p.

Given the above discussion, the code for Algorithm 7.2 is self-explanatory.

7.1.3 The algorithm

We now present the main result, namely, Algorithm 7.3 that implements an array $\mathcal{O}0\,..\,M-1$ of M W-word LL/SC objects where the number of processes accessing the array is not known in advance. The code for Algorithm 7.3 is presented in two figures: Figure 7.1, which gives the code for LL, SC, and VL operations, and Figure 7.2, which gives the code for Join and Leave operations. We start by describing the code in Figure 7.1.

Implementation of LL, SC, and VL

Figure 7.1 presents the code for procedures LL, SC, and VL of Algorithm 7.3. The statements given in rectangular boxes represent the differences with Algorithm 7.2. Operations `da_read` and `da_write` denote read and write operations on the dynamic array. We now describe the changes that were made to the original algorithm.

The array `NameArray`, as well as the arrays `BUF` located at process' memory blocks, are now dynamic arrays. Variable `N` holds the maximum number of processes that have simultaneously participated in the algorithm so far. Each process maintains its own estimate N of `N` which it periodically updates to match `N`. At all times, the algorithm ensures that the length of process's queue Q is at least N, and that the length of process's array `BUF` is at least $N+1$. Process' memory blocks are no longer allocated in advance. Instead, when a process joins the algorithm (by executing the Join procedure), it will either (1) allocate a new memory block, or (2) get a memory block from another process that has already left the algorithm. In either case, the implementation of Join guarantees that each process p participating

Types

valuetype = **array** $0\,..\,W$ **of** 64-bit value

indextype = **record** *type*: {0, 1}; **if** (*type* = 0) (*bindex*: 31-bit number)
else (*name*: 15-bit number; *bindex*: 15-bit number) **end**

xtype = **record** *seq*: 32-bit number; *buf*: indextype **end**

helptype = **record** *seq*: 31-bit number; *helpme*: {0, 1};
buf: indextype **end**

blocktype = **record** `Announce`: 31-bit number; `Help`: helptype; Q: Single-process queue;
`BUF`: **dynamic array of** $*$valuetype; *index*: 15-bit number; *mybuf*: indextype;
lseq: 31-bit number; x: xtype
name: 15-bit number; N: 15-bit number **end**

Shared variables

`X`: **array** $0\,..\,M-1$ of xtype; `BUF`: **array** $0\,..\,M-1$ **of** $*$valuetype

`NameArray`: **dynamic array of** $*$blocktype; `N`: $0\,..\,N$

Local persistent variables at each p

loc_p: blocktype

Initialization

`X`$k = (0, (0, k))$, for all $k \in \{0, 1, \ldots, M-1\}$;

`BUF`k = the desired initial value of $\mathcal{O}k$, for all $k \in \{0, 1, \ldots, M-1\}$; `N` $= 0$

procedure `LL`$(p, i, retval)$

1: loc_p.`Announce` $= i$
2: loc_p.`Help` $=$ (++loc_p.*lseq*, 1, loc_p.*mybuf*)
3: $loc_p.x =$ `X`i
4: **copy** $*$`GetBuf`(loc_p.*x.buf*) **into** $*$*retval*
5: **if** ¬`CAS`(loc_p.`Help`,
(loc_p.*lseq*, 1, loc_p.*mybuf*),
(loc_p.*lseq*, 0, loc_p.*mybuf*))
6: loc_p.*mybuf* $=$ loc_p.`Help`.*buf*
7: $b =$ `GetBuf`(loc_p.*mybuf*)
8: $loc_p.x = bW$
9: **copy** $*b$ **into** $*$*retval*
10: **return**

procedure `GetBuf`(b) **returns** $*$valuetype

11: **if** $(b.type = 0)$ **return** `BUF`$b.bindex$
12: $l =$ `da_read`(`NameArray`, $b.name$)
13: **return** `da_read`(l→`BUF`, $b.bindex$)

procedure `VL`(p, i) **returns** boolean

14: **return** (`X`$i = loc_p.x$)

procedure `SC`(p, i, v) **returns** boolean

15: **copy** $*v$ **into** $*$`GetBuf`(loc_p.*mybuf*)
16: **if** ¬`CAS`(`X`i, $loc_p.x$,
($loc_p.x.seq + 1$, loc_p.*mybuf*))
17: **return** *false*
18: *enqueue*($loc_p.Q$, $loc_p.x$.*buf*)
19: **if** ($loc_p.N <$ `N`)
20: $loc_p.N$++
21: `da_write`(loc_p.`BUF`, $loc_p.N$,
`malloc`$(W+1)$)
22: loc_p.*mybuf* $=$ (1, loc_p.*name*, $loc_p.N$)
23: **else** loc_p.*mybuf* $=$ *dequeue*($loc_p.Q$)
24: $l =$ `da_read`(`NameArray`, loc_p.*index*)
25: **if** (l→`Help` ≡ $(s, 1, c)$)
26: $j = l$→`Announce`
27: $x =$ `X`j
28: $d =$ `GetBuf`(loc_p.*mybuf*)
29: **copy** $*$`GetBuf`(x.*buf*) **into** $*d$
30: $dW = x$
31: **if** `CAS`(l→`Help`, $(s, 1, c)$,
$(s, 0, loc_p$.*mybuf*))
32: loc_p.*mybuf* $= c$
33: loc_p.*index* $= (loc_p$.*index* $+ 1)$ mod $loc_p.N$
34: **return** *true*

Figure 7.1: Code for procedures LL, SC, and VL of Algorithm 7.3

in the algorithm has a unique memory block. The algorithm also assigns (during the Join procedure) a unique name to each participating process. This name is guaranteed to be small: if K processes are currently participating in the algorithm, then a new process joining the algorithm will be assigned a name in the range $0..K$. (Hence, a process' name is sure the be smaller than `N`.) A process stores this name in a variable *name* located at that process's memory block. If a process p has a name n, then the nth location in (dynamic) array `NameArray` holds a pointer to the memory block owned by p. When p leaves the algorithm, it leaves its memory block in the nth location of `NameArray`; this block will be used later by another process that obtains name n.

We now explain the code at Lines 19–23. After a process p inserts the index of the previous current buffer into its local queue (Line 18), it checks whether its estimate N matches the actual value `N` (Line 19). If it doesn't, then p increments its estimate N by one (Line 20). Next, p allocates a new buffer, and then writes that buffer into the Nth location of its `BUF` array (Line 21). By doing so, p implicitly increments the length of array `BUF` to $N + 1$, thus maintaining the earlier stated invariant on the length of `BUF`. Then, p takes the buffer it had allocated and uses it as its own local buffer (Line 22). Notice that, in this case, p does not dequeue an index from its local queue; hence, p implicitly increases the length of its queue to N, thus maintaining the earlier stated invariant on the size of Q.

If, on the other hand, N does match the value of `N`, then p withdraws a new buffer index from its local queue and uses that buffer as its own local buffer (Line 23). The only other major change is at Line 33, where p increments its variable *index* by 1 modulo its local estimate N (versus a fixed N in the original algorithm). The changes at Lines 12, 13, and 24 are due to the fact that array

`NameArray`, as well as the arrays `BUF` located at process' memory blocks, are now dynamic arrays.

Recall that in the original algorithm where N was fixed, each process p made a promise not to reuse the buffer B that held some $\mathcal{O}i$'s current value until p performed at least N successful SC operations. (Process p kept its promise by enqueueing B's index into its local queue, which was of length at least N at all times.) This promise gave p enough time to help all other processes interested in B to obtain valid values for their LL operations, before overwriting B. To ensure that all N processes are helped during this time, p would help a process with name $j = \mathit{index}$ during an SC operation, and then increment *index* by 1 modulo N. Since N is not fixed in the new algorithm, and since each process increments its *index* variable modulo *its local estimate* N, it is not clear that the above property still holds. We now show that it does.

Suppose that some process q reads (at Line 3 of its LL) the tag of a buffer B that holds the current value of an object $\mathcal{O}i$. Suppose further that after q performs that read, some process p performs a successful SC on $\mathcal{O}i$. Then, we will show that p does not reuse B before it checks whether q needs help, thereby ensuring that the above property holds. In the following, we let n and j denote, respectively, the values of p's estimate N and p's variable *index*, at the time t when p inserts B's index into its local queue Q (Line 18).

Notice that, by the algorithm, there are n items already in the local queue when B's index is inserted at time t. Hence, B is not written until p performs at least $n + 1$ dequeues on its local queue. Notice further that, each time p satisfies the condition at Line 19, the following holds: (1) p does not dequeue an element from its local queue, and (2) the values of N and *index* both increase by one. Moreover,

each time p does not satisfy the condition at Line 19, the following holds: (1) p dequeues an element from its local queue, and (2) the value of N remains the same and *index* increases by one (modulo N). As a result, the value of *index* wraps around to 0 after p dequeues exactly $n - j$ elements from its local queue. Let $t' > t$ be the first time after t when *index* wraps around to 0, and let n' be the value of N at time t'. Then, p dequeues at most j elements from the local queue before *index* again reaches value j. Consequently, at the moment when p performs the $(n+1)$st dequeue from its local queue (which returns the index of B), variable *index* has gone through the values $j, j+1, \ldots, n'-1, 0, 1, \ldots j-1, j$, and processes with names $j, j+1, \ldots, n'-1, 0, 1, \ldots, j-1$ have been helped by p. Since q has obtained a name prior to time t, it follows that q's name is certainly less than n' Therefore, p would have checked whether q needs help before reusing B, which proves the above property.

We now turn to Figure 7.2 which gives the code for procedures Join and Leave of Algorithm 7.3.

Implementation of Join and Leave

As we stated earlier, the Join operation must (1) give each process a unique name, (2) give each process a unique memory block, (3) ensure that if K processes are participating in an algorithm then a new process obtains a name in the range $0 \,..\, K$, (4) guarantee that if a process obtains a name n, then a pointer to its memory block has been written into the nth location of the array `NameArray`, and (5) ensure that variable `N` holds the maximum number of processes that have simultaneously participated in the algorithm so far. We now explain how the implementation in Figure 7.2 ensures these properties.

Types

nodetype = **record** `owned`: boolean; `loc`: blocktype; `next`: ∗nodetype **end**

Shared variables

`Head`: ∗nodetype

Local persistent variables at each process p

$node_p$: ∗nodetype

Initialization

```
Head = malloc(sizeof nodetype); Head→owned = false; Head→next = ⊥
Head→loc = malloc(sizeof blocktype); Head→loc→Help = (0, 0, ∗)
Init(∗(Head→loc), 0)
```

```
    procedure Join(p)
35: mynode = malloc(sizeof nodetype)
36: mynode→owned = true
37: mynode→next = ⊥
38: mynode→loc = malloc(sizeof blocktype)
39: mynode→loc→Help = (0, 0, ∗)
40: name = 0
41: cur = Head
42: while (true)
43:     if CAS(cur→owned, false, true)
44:         free(mynode→loc)
45:         free(mynode)
46:         break
47:     da_write(NameArray, name, cur→loc)
48:     name++
49:     if (cur→next = ⊥)
50:         if CAS(cur→next, ⊥, mynode)
51:             cur = mynode
52:             break
53:     cur = cur→next
54: da_write(NameArray, name, cur→loc)
55: while ((n = N) < name + 1)
56:     CAS(N, n, name + 1)
57: loc_p = ∗(cur→loc)
58: node_p = cur
59: if (cur = mynode)
60:     Init(loc_p, name)
```

```
    procedure Leave(p)
61: node_p→owned = false

    procedure Init(loc, name)
62: loc.N = 1
63: da_write(loc.BUF, 0,
                malloc(W + 1))
64: da_write(loc.BUF, 1,
                malloc(W + 1))
65: loc.mybuf = (1, name, 0)
66: enqueue(loc.Q, (1, name, 1))
67: loc.index = 0
68: loc.name = name
```

Figure 7.2: Code for procedures Join and Leave of Algorithm 7.3, based on the renaming algorithm of Herlihy et al. [HLM03b] and the algorithm for allocating hazard-pointer records by Michael [Mic04a]

The algorithm for Join and Leave is essentially the same as the renaming algorithm of Herlihy et al. [HLM03b] and the algorithm for allocating new hazard-pointer records of Michael [Mic04a]. The algorithm maintains a linked list of nodes, with variable `Head` pointing to the head of the list. Each node in the list has a boolean field `owned`, which indicates whether the node is owned by some process or not. A node can be owned by at most one process at any given time. If a process p captures ownership of the kth node in the list, then its also captures ownership of the name k.[2] Each node in the list also has a field `loc` which holds the pointer to a memory block. The idea is that when a process p captures ownership of some node it also captures ownership of the memory block at that node, and will use that memory block in the LL/SC algorithm. Each node in the list has a field `next` which holds the pointer to the next node in the list. Finally, process p's local persistent variable $node_p$ holds the pointer to the node owned by p.

We now explain how the algorithm works. When a process p wishes to join the algorithm, it first prepares a new node that it will attempt to insert into the linked list (Lines 35–39). Next, it initializes its local variable *name* to 0, and, starting at the head of the list, tries to capture the first available node in the list (Lines 41–53). As we stated earlier, if p succeeds in capturing the kth node in the list, then it has also captured ownership of the name k, as well as the memory block stored at that node. While traversing through the list, process p also makes sure that array `NameArray` matches the contents of the linked list, i.e., that the jth location in the array holds the pointer to the memory block stored at the jth node in the list.

In order to capture a node, p performs the CAS operation on the `owned` field of that node (Line 43), trying to change its value from *false* (indicating that no

[2] We assume that the list starts with the 0th node.

process owns the node), to *true* (indicating that the node is owned by some process). If p's CAS succeeds, it means that p has successfully captured the node and so p terminates the loop and frees up the node that it had previously allocated (Lines 44–46). If p fails to in capturing a node (because a node was already owned by some other process, or because some other process's CAS succeeded before p's), p increments its variable *name* (Line 48), and then writes the memory block at that node into array `NameArray` (Line 47). Next, p checks whether it is at the last node in the list (Line 49), and if so, it tries to insert its own node at the back of the list (Line 50). If p's CAS succeeds, it means that p has successfully installed its node at the end of the list. Furthermore, since p had already set the `owned` field of that node to *true* (at Line 36), it means that p has ownership of that node. Hence, p terminates its loop at Line 52. If, on the other hand, p's CAS fails, it means that some other process must have inserted its own node into the list. In that case, the node that p was currently visiting is no longer the last node in the list. So, p moves on to the next node in the list (Line 53) and repeats the above steps.

By the above algorithm, at the moment when p exits the loop, its variable *name* holds the position of the node in the list that p had captured (which is the same as p's new name). Since p had not previously written that node into array `NameArray`, p does so at Line 54. Notice that, if p captures the kth node in the list (i.e., if p's name is k), it means that p must have found the first k nodes to be owned by other processes. (Recall that the list starts with the 0th node.) Hence, the number of processes participating in the algorithm when p captures the kth node is $k + 1$ or more [HLM03b]. To ensure that variable `N`, which holds the maximum number of processes participating in the algorithm so far, is up to date, process p performs the following steps. First, p reads `N` (Line 55). If the value

of N is smaller than $k+1$, then p tries to write $k+1$ into N (Line 56). There are two possibilities: either p's CAS succeeds or it fails. In the former case, N has been correctly updated; furthermore, the next time next time p tests the condition at Line 55, it will break out of the loop. In the latter case, some other process must have written into N and p may have to repeat the loop. However, since N is increased by at least one with each write, p will repeat the loop at most $k+1$ times. Consequently, after p's last iteration of the loop, N will hold a value greater than or equal to $k+1$.

Next, p sets its two persistent variables $node_p$ and loc_p to point to, respectively, the node in the list that p had captured and the memory block stored at that node (Lines 57 and 58). Finally, p checks whether it captured the same node that it had allocated at the beginning of the Join operation (Line 59). If so, it initializes the memory block stored at that node (Line 60). If p had captured some other node, then the memory block at that node has already been initialized (by a process who inserted that node into the list), and so there is no need for p to initialize that memory block.

The initialization of a block proceeds as follows. First, p sets the estimate of N to 1 (Line 62). Next, it allocated two new buffers and writes them at locations 0 and 1 of the array BUF (Lines 63 and 64). Then, p takes one of the two buffers to be its local buffer (Line 65) and enqueues the index of the other buffer into the local queue Q (Line 66). Finally, p sets its variable *index* to 0 and its variable *name* to *name* (Lines 67 and 68).

Operation Leave is extremely simple: p simply releases the ownership of the node it had previously captured during its Join operation (Line 61)

Based on the above discussion, we have the following theorem. Its proof is

given in Appendix A.4.2.

Theorem 12 *Algorithm 7.3 is wait-free and implements an array $\mathcal{O}0\,..\,M-1$ of M linearizable W-word LL/SC objects. The time complexity of LL, SC and VL operations on $\mathcal{O}$ are $O(W)$, $O(W)$ and $O(1)$, respectively. The time complexity of Join and Leave operations is $O(K)$ and $O(1)$, respectively, where K is the maximum number of processes that have simultaneously participated in the algorithm. The space complexity of the implementation is $O(Kk + (K^2 + M)W)$, where k is the maximum number of outstanding LL operations of a given process.*

7.2 Implementing a word-sized LL/SC object for an unknown number of processes

We now present Algorithm 7.4 that implements a word-sized LL/SC object shared by any number of processes in which the Join procedure runs in $O(1)$ time. The time complexities of LL, SC, and VL operations are $O(1)$, and the space complexity of the algorithm is $O(K^2 + KM)$. This algorithm is a modified version of Algorithm 4.1 (see Chapter 4). Below, we describe how the algorithm works.

Recall that Algorithm 4.1 stores the process id and a sequence number together in a central variable `X`. The assumption is that, once a process p reads a process id q from `X`, it can immediately locate all the shared variables owned by q, namely, $\texttt{oldseq}_q$, $\texttt{oldval}_q$, $\texttt{val}_q0$, and $\texttt{val}_q1$. Although the exact mechanism as to how p learns the locations of q's shared variables is not described in the algorithm, it is easy to see that the following approach will do: maintain an array `A` of length N, with one entry for each process, and store in each entry $\texttt{A}r$ the address of the

Types

valuetype = 64-bit value

xtype = **record** *type*: {0, 1}; **if** (*type* = 0) (*name*: 20-bit number; *seqnum*: 43-bit number)
else (*ptr*: ∗blocktype) **end**

blocktype = **record** `name`: 20-bit number; `val0`, `val1`, `oldval`: valuetype;
`oldseq`: 43-bit number **end**

Shared variables

`X`: xtype; $\mathcal{D}$: **dynamic array of** ∗blocktype; `N`: 20-bit number

Local persistent variables at each p

x_p: xtype; seq_p: 43-bit number; loc_p: blocktype; $first_p$: boolean

Initialization

loc = `malloc(sizeof` ∗blocktype); loc→`oldseq` = 0;
loc→`val1` = v_{init},the desired initial value of $\mathcal{O}$; loc→`name` = 0;
`X` = (1, loc); `N` = 0

procedure `LL`(p, $\mathcal{O}$) **returns** valuetype

1: x_p = `X`
2: **if** (x_p.*type* = 1)
3: $l = x_p.ptr$
4: $k = 1$
5: `da_write`($\mathcal{D}$, l→`name`, l)
6: `CAS`(`N`, l→`name`, l→`name` + 1)
7: **else** l = `da_read`($\mathcal{D}$, x_p.*name*)
8: $k = x_p.seqnum$
9: $v = l$→`val`k mod 2
10: $k' = l$→`oldseq`
11: **if** ($k' = k-2$) ∨ ($k' = k-1$) **return** v
12: $v' = l$→`oldval`
13: **return** v'

procedure `Join`(p)

14: $seq_p = 1$
15: loc_p.`oldseq` = 0
16: $first_p$ = *true*

procedure `SC`(p, $\mathcal{O}$, v) **returns** boolean

17: loc_p.`val`seq_p mod 2 = v
18: **if** ($first_p$)
19: loc_p.`name` = `N`
20: **if** ($succ$ = `CAS`(`X`, x_p, (1, &loc_p)))
21: $first_p$ = *false*
22: **else** $succ$ = `CAS`(`X`, x_p, (0, loc_p.`name`, seq_p))
23: **if** ($succ$)
24: loc_p.`oldval` = loc_p.`val`($seq_p - 1$) mod 2
25: loc_p.`oldseq` = $seq_p - 1$
26: $seq_p = seq_p + 1$
27: **return** *true*
28: **return** *false*

procedure `VL`(p, $\mathcal{O}$) **returns** boolean

29: **return** (`X` = x_p)

Algorithm 7.4: An unbounded implementation of the 64-bit LL/SC object $\mathcal{O}$ shared by an unknown number of processes

block containing r's shared variables. To lean the location of q's shared variables, p simply reads the address stored in $\texttt{A}q$.

To simulate the above approach in the new algorithm (where N is not known in advance), we keep a dynamic array $\mathcal{D}$ in place of the static array A. This array will grow in size as more processes keep joining the algorithm. When a process p joins the algorithm, it first obtains a name—say, i—which will be its unique index into the array $\mathcal{D}$. Next, p inserts the address of the block that contains p's shared variables into location i of array $\mathcal{D}$. From this point onwards, p uses its name i in place of its process id, i.e., it writes i into X instead of p. It is easy to see that if some process q reads i from X, it can learn the location of p's shared variables by simply consulting the ith location in array $\mathcal{D}$.

The main challenge in implementing the above scheme is to allow concurrent processes to obtain unique names in $O(1)$ time. Below, we explain how the algorithm addresses this issue. We start by describing the variables used by the algorithm.

Each process p maintains a block of memory where it keeps its shared variables, namely, oldseq, oldval, val0, and val1 (which have the same meaning as in Algorithm 4.1), as well as a new variable name which holds p's name. The address of this memory block is kept in p's local variable loc_p. In addition to loc_p, each process p maintains the following three (local) variables: (1) seq_p, which stores p's sequence number, (2) x_p, which stores the value of X that p had read in its latest LL operation, and (3) $first_p$, which holds value *true* if p hasn't yet performed a successful SC operation, and *false* otherwise.

In addition to the variables stored at each process, there are two global shared variables, namely, X and N. Variable N stores an unbounded integer, and is used by

processes to acquire names. Variable X stores the following information: if q is the latest process to perform a successful SC, then X holds either (1) a pair $(1, b)$, where b is a pointer to q's block of memory, or (2) a tuple $(0, k, s)$, where k is q's name and s is a sequence number.

We now explain the procedure SC$(p, \mathcal{O}, v)$ that describes how a process p performs an SC operation on $\mathcal{O}$ to attempt to change $\mathcal{O}$'s value to v. First, p makes available the value v in loc_p.val0 if the sequence number is even, or in loc_p.val1 if the sequence number is odd (Line 17). Next, p checks whether it had previously performed at least one successful SC (Line 18). If it hasn't, then p reads variable N to obtain a name, and saves that name in loc_p.name (Line 19). (Multiple processes reading N at the same time may get the same name; however, only one process will actually keep that name, as we explain below.) Next, p tries to make its SC operation take effect by changing the value in X from the value that p had witnessed in its latest LL operation to a value $(1, \&loc_p)$ (Line 20). If the CAS operation succeeds, then p's SC is successful. Since it is p's *first* successful SC, p updates variable $first_p$ to *false* (Line 21). Furthermore, p keeps the name it had read from N. If, on the other hand, p's CAS fails, then p's SC has failed and so p terminates its SC procedure by returning *false* (Line 28). Furthermore, p discards the name it had read from N. Therefore, of all processes that had read N at the same time, only one process (namely, the process that performed a successful CAS on X) actually keeps that name; all other processes abandon it, and attempt to capture a name again during their subsequent SC operations.

If p had previously performed a successful SC (Line 18), p attempts to change the value in X to a value $(0, i, s)$, where i is p's name and s is a sequence number (Line 22). Again, if the CAS operation succeeds, then p's SC is successful.

Otherwise, p's SC has failed, and p terminates its SC procedure by returning *false* (Line 28).

If p's SC is successful (for the first time or not), p performs the same steps as in Algorithm 4.1. Namely, it (1) writes into loc_p.`oldval` the value written by p's earlier successful SC (Line 24), (2) writes into loc_p.`oldval` the sequence number of that SC (Line 25), and (3) increments its sequence number (Line 26). Finally, p signals successful completion of the SC by returning *true* (Line 27).

We now turn to the procedure `LL`$(p, \mathcal{O})$ that describes how a process p performs an LL operation on $\mathcal{O}$. In the following, let $\mathrm{SC}_{q,i}$ denote the ith successful SC by process q, and $v_{q,i}$ denote the value written in $\mathcal{O}$ by $\mathrm{SC}_{q,i}$. First, p reads the current value x of variable `X` (Line 1). Suppose that $\mathrm{SC}_{q,k}$ is the latest successful SC operation to write into `X` before p reads `X`. Then, if $x = (1, l)$ (Line 2), we have $k = 1$ (Line 4). Furthermore, l is the address of the memory block containing q's shared variables. Since it is possible that l has not yet been inserted into the DynamicArray $\mathcal{D}$, p inserts l into $\mathcal{D}$ (Line 5) and increments `N` to a value that is by one greater than the value of q's name (Line 6). By doing so, q ensures that before another process obtains a new name, the following holds: (1) the address of p's memory block has been inserted into $\mathcal{D}$, and (2) variable `N` is by one greater than p's name. As a result, each process obtains a name that is unique, and all entries in array $\mathcal{D}$ are written (for the first time) in sequential order: entry j is written before entry $j + 1$, for all $j \geq 0$.

If $x = (0, i, k)$ (Line 2), then we have $k > 1$. Furthermore, by the above argument, the address l of q's memory block has already been written into location i of array $\mathcal{D}$. So, p simply reads that location to obtain l (Line 7).

Notice that, in both of the above cases, p is able to obtain address l of q's

memory block: either directly from X (Line 3), or indirectly from $\mathcal{D}$ (Line 7). From this point onwards, p proceeds in the same way as in Algorithm 4.1. In particular, p first reads $l\rightarrow$ `val`k mod 2 to try to learn $v_{q,k}$ (Line 9). Next, p reads the sequence number k' in $l \rightarrow$ `oldseq` (Line 10). If $k' = k-2$ or $k' = k-1$, then $\mathrm{SC}_{q,k+1}$ has not yet completed, and the value v obtained on Line 9 *is* $v_{q,k}$. So, p terminates the LL operation, returning v (Line 11). If $k' \geq k$, q must have completed $\mathrm{SC}_{q,k+1}$. Hence, the value in $l\rightarrow$ `oldval` is $v_{q,k}$ or a later value (more precisely, the value in $l\rightarrow$ `oldval` is $v_{q,i}$ for some $i \geq k$). Therefore, the value in $l\rightarrow$ `oldval` is not too old for p's LL to return. Accordingly, p reads the value v' of $l\rightarrow$ `oldval` (Line 12) and returns it (Line 13).

The VL procedure is self-explanatory. Based on the above, we have the following theorem. Its proof is given in Appendix A.4.3.

Theorem 13 *Algorithm 7.4 is wait-free and implements a linearizable* 64*-bit LL/SC object from* 64*-bit CAS objects and registers. The time complexity of Join, LL, SC, and VL is* $O(1)$*. The space complexity of the algorithm is* $O(K^2 + KM)$*, where* K *is the total number of processes that have joined the algorithm and* M *is the number of implemented LL/SC objects.*

Chapter 8

Conclusion and future work

The goal of this thesis was to bridge the gap that exists between the instructions that are available on most modern multiprocessors (namely, CAS or RLL/RSC), and the instructions that are of interest to algorithm designers (namely, LL/SC). To this effect, we have presented a series of time-optimal, wait-free algorithms that implement LL/SC objects from CAS objects and registers. We started with Algorithm 4.1 that implements a single word-sized LL/SC object shared by a fixed number of processes, and finished with Algorithm 7.3 that implements a large number of multiword LL/SC objects shared by an unknown number of processes. When constructing a large number of LL/SC objects, the latter is the first algorithm in the literature to be simultaneously (1) wait-free, (2) time optimal, and (3) space efficient. The next section summarizes the results of the thesis in more detail.

8.1 Summary of the results

This thesis presents a total of eight algorithms that implement LL/SC objects from CAS objects and registers. Below, we summarize these results.

1. *Word-sized LL/SC.* In Chapter 4, we presented a wait-free implementation of a word-sized LL/SC object from a word-sized CAS object and registers. We have given three algorithms: the first algorithm uses unbounded sequence numbers; the other two use bounded sequence numbers. The time complexity of LL and SC in all three algorithms is $O(1)$, and the space complexity of the algorithms is $O(N)$ per implemented object, where N is the number of processes accessing the object. The space complexity of implementing M LL/SC objects is $O(NM)$, and the algorithms require N to be known in advance.

2. *Multiword LL/SC.* In Chapter 5, we presented a wait-free implementation of a W-word LL/SC object (from word-sized CAS objects and registers). The time complexity of LL and SC is $O(W)$, and the space complexity of the algorithm is $O(NW)$ per implemented object, where N is the number of processes accessing the object. The space complexity of implementing M LL/SC objects is $O(NMW)$. This algorithm requires N to be known in advance, and it uses unbounded sequence numbers.

3. *Multiword LL/SC for large number of objects.* In Chapter 6, we presented a wait-free implementation of an array of M W-word LL/SC objects (from word-sized CAS objects and registers). This algorithm improves the space complexity of the algorithm in Chapter 5 when implementing a large number of LL/SC objects (i.e., when $M \gg N$). In particular, the space complexity of

the algorithm is $O((N^2 + M)W)$, where N is the number of processes accessing the objects. The time complexity of LL and SC is $O(W)$. The algorithm requires N to be known in advance, and it uses unbounded sequence numbers.

We also presented a wait-free implementation of an array of M W-word *Weak-LL/SC* objects from word-sized CAS objects and registers. The time complexity of WLL and SC is $O(W)$, and the space complexity of the algorithm is $O((N + M)W)$, where N is the number of processes accessing the object. The CAS-time-complexity of WLL and SC is $O(1)$. The algorithm requires N to be known in advance, and it uses unbounded sequence numbers.

4. *LL/SC algorithms for an unknown N*. In Chapter 7, we presented a wait-free implementation of an array of M W-word LL/SC objects (from word-sized CAS objects and registers), shared by an unknown number of processes. This algorithm supports two new operations—*Join*(p) and *Leave*(p)—which allow a process p to join and leave the algorithm at any given time. If K is the maximum number of processes that have simultaneously participated in the algorithm (i.e., have joined the algorithm but have not yet left it), then the space complexity of the algorithm is $O((K^2 + M)W)$. The time complexities of procedures Join, Leave, LL, and SC are $O(K)$, $O(1)$, $O(W)$, and $O(W)$, respectively.

   We also presented a wait-free implementation of a word-sized LL/SC object (from a word-sized CAS object and registers) that does not require N to be known in advance. The attractive feature of this algorithm is that the Join procedure runs in $O(1)$ time. This algorithm, however, does not allow processes to leave. The time complexities of LL and SC operations are $O(1)$, and the

space complexity of the algorithm is $O(K^2 + KM)$.

8.2 Future work

We would like to extend the results in this thesis in the following ways.

- The following problem is still open: design a time-optimal, wait-free algorithm that implements M word-sized LL/SC objects from CAS objects and registers, using only $O(N + M)$ space. This problem is important because, if solved, it would present a way to simulate the memory supporting LL and SC instructions on any modern hardware with virtually no time and space overhead. We would like to tackle this problem by either (1) designing an algorithm for it, or (2) show that no such algorithm exists. If latter is the case, it would be interesting to see whether the problem becomes solvable if some other hardware instructions are also available for the task (e.g., *fetch&Add*, *fetch&Store*), or if we are willing to sacrifice the time-optimality requirement and seek, for example, an $O(\lg N)$-time solution.

- It would be interesting to measure the performance of our LL/SC algorithms on real multiprocessor systems and compare it with other LL/SC algorithms in the literature. This exercise would help us understand better all the factors that figure in the performance of an LL/SC algorithm in practice. Also, we would like to test the performance of our algorithms on the emerging multi-core systems and see what the differences are compared to the multiprocessor systems. We also plan to make the code for LL/SC available to the public.

- We plan to use LL/SC instructions to design efficient wait-free algorithms for shared objects such as such as queues, stacks, adaptive snapshots etc. We have some encouraging preliminary results [JP05, Pet05].

Bibliography

[ADT95] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 538–547, 1995.

[AM95a] J. Anderson and M. Moir. Universal constructions for large objects. In *Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 168–182, September 1995.

[AM95b] 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.

[Bar93] G. Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.

[CJT98] T.D. Chandra, P. Jayanti, and K. Y. Tan. A polylog time wait-free construction for closed objects. In *Proceedings of the 17th Annual Symposium on Principles of Distributed Computing*, pages 287–296, June 1998.

[DHLM04] S. Doherty, M. Herlihy, V. Luchangco, and M. Moir. Bringing practical lock-free synchronization to 64-bit applications. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 31–39, 2004.

[DMMJ02] D. Detlefs, P. Martin, M. Moir, and G. Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

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

[Her93] M. Herlihy. A methodology for implementing highly concurrent data structures. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[HLM03a] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[HLM03b] M. Herlihy, V. Luchangco, and M. Moir. Space- and time-adaptive nonblocking data structures. In *Proceedings of Computing: Australasian Theory Symposium*, 2003.

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

[Ibm83] IBM T.J Watson Research Center. *System/370 Principles of Operation*, 1983. Order Number GA22-7000.

[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.

[Ita02] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual Volume 1: Application Architecture*, 2002. Revision 2.1.

[Jay98] P. Jayanti. A complete and constant time wait-free implementation of CAS from LL/SC and vice versa. In *Proceedings of the 12th International Symposium on Distributed Computing*, pages 216–230, September 1998.

[Jay02] P. Jayanti. f-arrays: implementation and applications. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, pages 270 – 279, 2002.

[Jay03] P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, July 2003.

[Jay05] P. Jayanti. An optimal multi-writer snapshot algorithm. In *Proceedings of the 37th ACM Symposium on Theory of Computing*, 2005.

[JP05] P. Jayanti and S. Petrovic. Logarithmic-time single-deleter multiple-inserter wait-free queues and stacks. Unpublished manuscript, May 2005.

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

[Lea02] D. Lea. Java specification request for concurrent utilities (JSR166), 2002. http://jcp.org.

[LMS03] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 314–323, 2003.

[MH02] M. Moir M. Herlihy, V. Luchangco. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. *Lecture Notes in Computer Science*, 2508:339–353, 2002.

[Mic04a] M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[Mic04b] M. Michael. Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In *Proceedings of the 18th Annual Conference on Distributed Computing*, pages 144–158, 2004.

[Mip02] MIPS Computer Systems. *MIPS64™ Architecture For Programmers Volume II: The MIPS64™ Instruction Set*, 2002. Revision 1.00.

[Moi97a] 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.

[Moi97b] M. Moir. Transparent support for wait-free transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319, September 1997.

[Moi01] M. Moir. Laziness pays! Using lazy synchronization mechanisms to improve non-blocking constructions. *Distributed Computing*, 14(4):193–204, 2001.

[MS96] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 267–276, May 1996.

[Pet83] G. L. Peterson. Concurrent reading while writing. *ACM TOPLAS*, 5(1):56–65, 1983.

[Pet05] S. Petrovic. Efficient algorithms for the adaptive collect object. Unpublished manuscript, May 2005.

[Pow01] IBM Server Group. *IBM e server POWER4 System Microarchitecture*, 2001.

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

[Spa] SPARC International. *The SPARC Architecture Manual*. Version 9.

[ST95] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, August 1995.

Appendix A

Proofs

Throughout the appendix, we let LP(op) denote the linearization point of the operation op. We also use the term *LP* as a shorthand for *linearization point*.

A.1 Proof of the algorithms in Chapter 4

A.1.1 Proof of Algorithm 4.1

In the following, let $\text{SC}_{p,i}$ denote the ith *successful* SC by process p and $v_{p,i}$ denote the value written in $\mathcal{O}$ by $\text{SC}_{p,i}$. The operations are linearized according to the following rules. We linearize each SC operation at Line 8 and each VL at Line 14. Let OP be any execution of the LL operation by p. The linearization point of OP is determined by two cases. If OP returns at Line 4, then we linearize OP at Line 1. Otherwise, let $\text{SC}_{q,k}$ be the latest successful SC operation to execute Line 8 prior to Line 1 of OP, and let v' be the value that p reads at Line 5 of OP. We show that there exists some $i \geq k$ such that (1) at some time t during the execution of OP, $\text{SC}_{q,i}$ was the latest successful SC operation to execute Line 8, and (2) $v' = v_{q,i}$.

We then linearize OP at time t.

In the rest of this section, let s denote the number of bits in X reserved for the sequence number. We show that Algorithm 4.1 is correct under the following assumption:

Assumption A: During the time interval when some process p executes one LL/SC pair, no other process q performs more than $2^s - 3$ successful SC operations.

In the following, we assume that the initializing step was performed by process 0 during its first successful SC operation.

Lemma 3 *At the beginning of* $SC_{p,i}$, seq_p *holds the value* $i \bmod 2^s$.

Proof. Prior to $\text{SC}_{p,i}$, exactly $i - 1$ successful SC operations are performed by p. Therefore, variable seq_p was incremented exactly $i - 1$ times prior to $\text{SC}_{p,i}$. Since seq_p is initialized to 1, it follows that at the beginning of $\text{SC}_{p,i}$, seq_p holds the value $i \bmod 2^s$. □

Lemma 4 *During* $SC_{p,i}$, *p writes* $(p, i \bmod 2^s)$ *into* X *at Line 8 and* $(i-1) \bmod 2^s$ *into* $\texttt{oldseq}_p$ *at Line 10.*

Proof. According to Lemma 3, at the beginning of $\text{SC}_{p,i}$, seq_p holds the value $i \bmod 2^s$. Therefore, p writes $(p, i \bmod 2^s)$ into X at Line 8 and $(i - 1) \bmod 2^s$ into $\texttt{oldseq}_p$ at Line 10. □

Lemma 5 *From the moment p performs Line 7 of* $SC_{p,i}$, *until p completes the* $SC_{p,i+1}$ *operation,* $\texttt{val}_p i \bmod 2$ *holds the value* $v_{p,i}$.

Proof. According to Lemma 3, at the beginning of $\mathrm{SC}_{p,i}$, seq_p holds the value $i \bmod 2^s$. Since $(i \bmod 2^s) \bmod 2 = i \bmod 2$, it follows that p writes $v_{p,i}$ into $\mathtt{val}_p i \bmod 2$ at Line 7 of $\mathrm{SC}_{p,i}$. Furthermore, since p increments seq_p at Line 11 of $\mathrm{SC}_{p,i}$, the value $v_{p,i}$ (in $\mathtt{val}_p i \bmod 2$) will not be overwritten until seq_p reaches the value $(i+2) \bmod 2^s$, which in turn will not happen until p executes Line 11 of $\mathrm{SC}_{p,i+1}$. Therefore, variable $\mathtt{val}_p i \bmod 2$ holds the value $v_{p,i}$ from the moment p performs Line 7 of $\mathrm{SC}_{p,i}$, until p completes $\mathrm{SC}_{p,i+1}$. □

Lemma 6 *During $SC_{p,i}$, p writes $v_{p,i-1}$ into $\mathtt{oldval}_p$ at Line 9.*

Proof. By Lemma 5, $\mathtt{val}_p(i-1) \bmod 2$ holds the value $v_{p,i-1}$ at all times during $\mathrm{SC}_{p,i}$. As a result, p writes $v_{p,i-1}$ into $\mathtt{oldval}_p$ at Line 9 of $\mathrm{SC}_{p,i}$. □

Lemma 7 *Let* OP *be an LL operation by process p, and $SC_{q,k}$ the latest successful SC operation to execute Line 8 prior to Line 1 of* OP. *If* OP *terminates at Line 4, then it returns the value $v_{q,k}$.*

Proof. Let $\mathcal{I}$ be the time interval starting from the moment q performs Line 7 of $\mathrm{SC}_{q,k}$ until q completes $\mathrm{SC}_{q,k+1}$. According to Lemma 5, variable $\mathtt{val}_q k \bmod 2$ holds the value $v_{q,k}$ at all times during $\mathcal{I}$. Furthermore, by Lemma 4, q writes $(q, k \bmod 2^s)$ into X at Line 8 of $\mathrm{SC}_{q,k}$. Therefore, p reads $(q, k \bmod 2^s)$ from X at Line 1 of OP. Since $(k \bmod 2^s) \bmod 2 = k \bmod 2$, it follows that p reads $\mathtt{val}_q k \bmod 2$ at Line 2 of OP. Our goal is to show that p executes Line 2 of OP during $\mathcal{I}$, and therefore reads $v_{q,k}$ from $\mathtt{val}_q k \bmod 2$.

At the moment when p reads $(q, k \bmod 2^s)$ from X at Line 1 of OP, q must have already executed Line 7 of $\mathrm{SC}_{q,k}$ and not yet executed Line 8 of $\mathrm{SC}_{q,k+1}$.

Hence, p executes Line 1 during $\mathcal{I}$. From the fact that OP terminates at Line 4, it follows that p satisfies the condition at Line 4. Therefore, the value that p reads from $\texttt{oldseq}_q$ at Line 3 of OP is either $(k-2) \bmod 2^s$ or $(k-1) \bmod 2^s$. Hence, by Lemma 4 and Assumption A, it follows that when p performs Line 3, q did not yet complete $\text{SC}_{q,k+1}$. So, p executes Line 3 during $\mathcal{I}$. Since p performed both Lines 1 and 3 during $\mathcal{I}$, it follows that p performs Line 2 during $\mathcal{I}$ as well. Therefore, by Lemma 5, p reads $v_{q,k}$ from $\texttt{val}_q k \bmod 2$ at Line 2, and the value that OP returns is $v_{q,k}$. □

Lemma 8 *Let* OP *be an LL operation by process p, and $\text{SC}_{q,k}$ be the latest successful SC operation to execute Line 8 prior to Line 1 of* OP. *If* OP *terminates at Line 6, let v' be the value that p reads at Line 5 of* OP. *Then, there exists some $i \geq k$ such that (1) at some time t during the execution of* OP, *$\text{SC}_{q,i}$ is the latest successful SC operation to execute Line 8, (2) $\text{SC}_{q,i+1}$ executes Line 8 during* OP, *and (3) $v' = v_{q,i}$.*

Proof. By Lemma 4, q writes $(q, k \bmod 2^s)$ into X at Line 8 of $\text{SC}_{q,k}$. Therefore, p reads $(q, k \bmod 2^s)$ from X at Line 1 of OP. Furthermore, since OP terminates at Line 6, the condition at Line 4 of OP doesn't hold. Therefore, p reads a value different than $(k-1) \bmod 2^s$ and $(k-2) \bmod 2^s$ from $\texttt{oldseq}_q$ at Line 3 of OP. Then, by Lemma 4, q completes Line 10 of $\text{SC}_{q,k+1}$ before p performs Line 3 of OP. Consequently, q completes Line 9 of $\text{SC}_{q,k+1}$ before p performs Line 5 of OP. As a result, the value v' that p reads at Line 5 of OP was written by q (into $\texttt{oldval}_q$ at Line 9) in either $\text{SC}_{q,k+1}$ or a later SC operation by q. We examine two cases: either v' was written by $\text{SC}_{q,k+1}$, or it was written by $\text{SC}_{q,i}$, for some $i \geq k+2$.

In the first case, by Lemma 6, we have $v' = v_{q,k}$. Furthermore, by definition of $\mathrm{SC}_{q,k}$, at the time when p executes Line 1 of OP, $\mathrm{SC}_{q,k}$ is the latest successful SC to execute Line 8. Finally, by definition of $\mathrm{SC}_{q,k}$ and an earlier argument, $\mathrm{SC}_{q,k+1}$ executes Line 8 between Lines 1 and 5 of OP. Hence, the lemma holds in this case.

In the second case, by Lemma 6, we have $v' = v_{q,i-1}$. Since, at the time when p executes Line 1 of OP, $\mathrm{SC}_{q,k}$ is the latest successful SC to execute Line 8, it means that $\mathrm{SC}_{q,i-1}$ and $\mathrm{SC}_{q,i}$ did not execute Line 8 prior to Line 1 of OP. Furthermore, since $\mathrm{SC}_{q,i}$ had executed Line 9 prior to Line 5 of OP, it follows that $\mathrm{SC}_{q,i-1}$ and $\mathrm{SC}_{q,i}$ had executed Line 8 prior to Line 5 of OP. Consequently, $\mathrm{SC}_{q,i-1}$ and $\mathrm{SC}_{q,i}$ had executed Line 8 during OP, which proves the lemma. □

Lemma 9 (Correctness of LL) *Let* OP *be any LL operation, and* OP′ *be the latest successful SC operation such that LP(*OP′*) < LP(*OP*). Then,* OP *returns the value written by* OP′.

Proof. Let p be the process executing OP. Let $\mathrm{SC}_{q,k}$ be the latest successful SC operation to execute Line 8 prior to Line 1 of OP. We examine the following two cases: (1) OP returned at Line 4, and (2) OP returned at Line 6. In the first case, since all SC operations are linearized at Line 8 and since OP is linearized at Line 1, we have $\mathrm{SC}_{q,k} = \mathrm{OP}'$. Furthermore, by Lemma 7, OP returns the value written by $\mathrm{SC}_{q,k}$. Therefore, the lemma holds. In the second case, by Lemma 8, there exists some $i \geq k$ such that (1) at some time t during the execution of OP, $\mathrm{SC}_{q,i}$ is the latest successful SC operation to execute Line 8, and (2) OP returns $v_{q,i}$. Since all SC operations are linearized at Line 8 and since OP is linearized at time t, it follows that $\mathrm{SC}_{q,i} = \mathrm{OP}'$. Therefore, the lemma holds in this case as well. □

Lemma 10 *Let* OP *be an LL operation by process p, and $SC_{q,k}$ be the latest successful SC operation to execute Line 8 prior to Line 1 of* OP. *If* X *does not change during* OP, *then* OP *terminates at Line 4.*

Proof. By Lemma 4, q writes $(q, k \bmod 2^s)$ into X at Line 8 of $SC_{q,k}$. Therefore, p reads $(q, k \bmod 2^s)$ from X at Line 1 of OP. Since X does not change during OP, q does not execute Line 8 of $SC_{q,k+1}$ during OP. Consequently, q does not execute Line 10 of $SC_{q,k+1}$, or any later SC operation, during OP. Therefore, by Lemma 6, p reads either $(k-1) \bmod 2^s$ or $(k-2) \bmod 2^s$ from $\texttt{oldseq}_q$ at Line 3 of OP. Hence, p terminates at Line 4. □

Lemma 11 (Correctness of SC) *Let* OP *be any SC operation by some process p, and* OP′ *be the latest LL operation by p that precedes* OP. *Then,* OP *succeeds if and only if there does not exist any successful SC operation* OP″ *such that $LP(\text{OP}') < LP(\text{OP}'') < LP(\text{OP})$.*

Proof. Let $SC_{q,k}$ be the latest successful SC operation to execute Line 8 prior to Line 1 of OP′. By Lemma 4, q writes $(q, k \bmod 2^s)$ into X at Line 8 of $SC_{q,k}$. Therefore, p reads $(q, k \bmod 2^s)$ from X at Line 1 of OP′. If OP returned *false*, then clearly the CAS at Line 8 of OP failed, which means that the value in X was different than $(q, k \bmod 2^s)$. We examine the following two cases: (1) OP′ returned at Line 4, and (2) OP′ returned at Line 6. In the first case, from the fact that X was different than $(q, k \bmod 2^s)$ at Line 8 of OP, it follows that some successful SC operation OP″ executed Line 8 between the time p executed Line 1 of OP′ and the time p executed Line 8 of OP. Since all SC operations are linearized at Line 8 and since OP′ is linearized at Line 1, it follows that $LP(\text{OP}') < LP(\text{OP}'') < LP(\text{OP})$.

Hence, OP is correct to return *false*.

In the second case, by Lemma 8, there exists some $i \geq k$ such that (1) at some time t during the execution of OP′, $\mathrm{SC}_{q,i}$ is the latest successful SC operation to execute Line 8, (2) $\mathrm{SC}_{q,i+1}$ executes Line 8 during OP′, and (3) $v' = v_{q,i}$. Since OP′ is linearized at time t and since all SC operations are linearized at Line 8, we have $LP(\mathrm{OP}') < LP(SC_{q,i+1}) < LP(\mathrm{OP})$. Hence, OP is correct to return *false*.

If OP returned *true*, then the CAS at Line 8 of OP succeeded. Hence, the value in X was equal to $(q, k \bmod 2^s)$. Then, by Assumption A, from the moment p reads $(q, k \bmod 2^s)$ at Line 1 of OP′, until p executes Line 8 of OP, X does not change. Hence, by Lemma 10, OP′ terminates at Line 4 and is linearized at Line 1. Furthermore, no successful SC operation executes Line 8 between Line 1 of OP′ and Line 8 of OP. Since all SC operations are linearized at Line 8 and since OP′ is linearized at Line 1, it follows that no successful SC is linearized between OP′ and OP. Hence, OP is correct to return *true*. □

Lemma 12 (Correctness of VL) *Let* OP *be any VL operation by some process p, and* OP′ *be the latest LL operation by p that precedes* OP. *Then,* OP *returns true if and only if there does not exist any successful SC operation* OP″ *such that* $LP(\mathrm{OP}') < LP(\mathrm{OP}'') < LP(\mathrm{OP})$.

Proof. Similar to the proof of Lemma 11. □

Theorem 1 *Algorithm 4.1 is wait-free and, under Assumption A, implements a linearizable* 64*-bit LL/SC object from a single* 64*-bit CAS object and an additional six registers per process. The time complexity of LL, SC, and VL is* $O(1)$.

Proof. The theorem follows immediately from Lemmas 9, 11, and 12. □

A.1.2 Proof of Algorithm 4.2

We start the proof by giving a formal specification of the WLL/SC object:

Definition 1 *Let $\mathcal{O}$ be a WLL/SC object. In every execution history H on object $\mathcal{O}$, the following is true:*

- *each operation* OP *takes effect at some instant $LP(\text{OP})$ during its execution interval.*
- *if an WLL operation* OP *performed by process p returns $(failure, q)$, then there exists a successful SC operation* OP$'$ *performed by process q such that:*
  1. *$LP(\text{OP}')$ lies in the execution interval of* OP*,*
  2. *$LP(\text{OP}) < LP(\text{OP}')$.*
- *the responses of all successful WLL operations and all VL and SC operations, when ordered according to their LP times, are consistent with the sequential specifications of LL, SC and VL.*

Let $\mathcal{O}$ be the LL/SC object implemented by Algorithm 4.2. Let OP be any LL operation, OP$'$ be any SC operation, and OP$''$ be any VL operation on $\mathcal{O}$. We let OP(1), OP(2), OP$'(1)$, and OP$''(1)$ denote, respectively, the `WLL` operation at Line 1 of OP, the `WLL` operation at Line 6 of OP, the `SC` operation at Line 11 of OP$'$, and the `VL` operation at Line 12 of OP$''$.

The operations on $\mathcal{O}$ are linearized according to the following rules: OP$'$ is linearized at $LP(\text{OP}'(1))$ and OP$''$ is linearized at $LP(\text{OP}''(1))$. The linearization point of OP is determined by three cases. If OP(1) succeeds, we linearize OP at $LP(\text{OP}(1))$. If OP(1) fails and OP(2) succeeds, we linearize OP at $LP(\text{OP}(2))$.

Otherwise, we linearize OP as follows. Let $(failure, q)$ be the value returned by OP(1). Then, by Definition 1, there exists some successful SC operation SC_q by process q such that (1) $LP(\mathrm{SC}_q(1))$ lies in the execution interval of OP(1), and (2) $LP(\mathrm{OP}(1)) < LP(\mathrm{SC}_q(1))$. Let LL_q be the latest LL operation by q to write into $\texttt{lastVal}_q$ prior to Line 5 of OP. Then, if LL_q is executed before SC_q, we linearize OP to the point just prior to $LP(\mathrm{SC}_q)$. Otherwise, we linearize OP to the point just prior to $LP(\mathrm{LL}_q)$. (Notice that $LP(\mathrm{LL}_q)$ is either at $LP(\mathrm{LL}_q(1))$ or $LP(\mathrm{LL}_q(2))$, and so the linearization point of OP is well defined.)

We start by showing that the linearization point for the LL operation always falls within the execution interval of that operation. (This property trivially holds for the SC and VL operations.)

Lemma 13 *Let* OP *be any LL operation. Then,* $LP(\mathrm{OP})$ *falls within the execution interval of* OP.

Proof. If either one of OP(1) and OP(2) succeeds, the lemma trivially holds. Suppose that both OP(1) and OP(2) fail. Let $(failure, q)$ be the value returned by OP(1). Then, by Definition 1, there exists some successful SC operation SC_q by q such that (1) $LP(\mathrm{SC}_q(1))$ lies in the execution interval of OP(1), and (2) $LP(\mathrm{OP}(1)) < LP(\mathrm{SC}_q(1))$. Let LL_q be the latest LL operation by q to write into $\texttt{lastVal}_q$ prior to Line 5 of OP. We examine the following two cases: (1) LL_q is executed before SC_q, and (2) LL_q is executed after SC_q.

In the first case, OP is linearized to the point just prior to $LP(\mathrm{SC}_q)$. Since $LP(\mathrm{SC}_q) = LP(\mathrm{SC}_q(1))$, and since $LP(\mathrm{SC}_q(1))$ lies within the execution interval of OP(1), it follows that the linearization point for OP lies within the execution interval of OP.

In the second case, OP is linearized to the point just prior to $LP(\mathrm{LL}_q)$. Notice that, by definition of LL_q, $LP(\mathrm{LL}_q)$ lies before Line 5 of OP. Furthermore, since LL_q is executed after SC_q and since $LP(\mathrm{SC}_q(1))$ lies in the execution interval of OP(1), $LP(\mathrm{LL}_q)$ lies after $LP(\mathrm{OP}(1))$. As a result, $LP(\mathrm{LL}_q)$ lies within the execution interval of OP. Consequently, the linearization point for OP lies within the execution interval of OP, which proves the lemma. □

Lemma 14 (Correctness of LL) *Let* OP *be any LL operation, and* OP′ *be the latest successful SC operation such that LP(*OP′*) < LP(*OP*). Then,* OP *returns the value written by* OP′.

Proof. We examine the following three cases: (1) OP returns at Line 4, (2) OP returns at Line 9, and (3) OP returns at Line 10. In the first case, OP(1) succeeds and OP is linearized at $LP(\mathrm{OP}(1))$. Since OP′ is the latest successful SC operation such that $LP(\mathrm{OP}') < LP(\mathrm{OP})$, it follows that OP′(1) is the latests successful SC operation on X such that $LP(\mathrm{OP}'(1)) < LP(\mathrm{OP}(1))$. Let v be the value that OP′ writes in $\mathcal{O}$. Then, OP(1) returns (*success*, v). Hence, OP returns v and the lemma holds in this case. The proof for the second case is identical, and is therefore omitted.

In the third case, let p be the process executing OP. Since OP returns at Line 10, it follows that both OP(1) and OP(2) failed. Let (*failure*, q) be the value returned by OP(1). Then, by Definition 1, there exists some successful SC operation SC_q by q such that (1) $LP(\mathrm{SC}_q(1))$ lies in the execution interval of OP(1), and (2) $LP(\mathrm{OP}(1)) < LP(\mathrm{SC}_q(1))$. Let LL_q be the latest LL operation by q to write into $\mathtt{lastVal}_q$ before Line 5 of OP. We examine the following two cases: (1) LL_q is executed before SC_q, and (2) LL_q is executed after SC_q.

In the first case, let OP″ be the latest LL operation by q to precede SC_q. Then, we show that OP″ = LL_q. Notice that, since $LP(\mathrm{SC}_q(1))$ succeeds, one of OP″(1) and OP″(2) must have succeeded. Hence, OP″ writes into $\texttt{lastVal}_q$. As a result, OP″ is the latest LL operation by q to write into $\texttt{lastVal}_q$ prior to SC_q. Since $LP(\mathrm{SC}_q(1))$ lies in the execution interval of OP(1), it follows that $LP(\mathrm{SC}_q(1))$ takes place before Line 5 of OP. Furthermore, since LL_q is executed before SC_q, it follows that LL_q is the latest LL by q to write into $\texttt{lastVal}_q$ prior to SC_q. Hence, we have OP″ = LL_q.

Notice that, since OP is linearized just prior to $LP(\mathrm{SC}_q)$, it means that OP′ is the latest successful SC operation such that $LP(\text{OP}') < LP(\mathrm{SC}_q)$. Consequently, OP′(1) is the latest successful SC operation on X such that $LP(\text{OP}'(1)) < LP(\mathrm{SC}_q(1))$. Let v be the value that OP′ writes in $\mathcal{O}$. Then, since $\mathrm{SC}_q(1)$ succeeds, it means that either $\mathrm{LL}_q(1)$ or $\mathrm{LL}_q(2)$ returns (*success*, v). Therefore, LL_q writes v into $\texttt{lastVal}_q$. Since LL_q is the latest LL operation by q to write into $\texttt{lastVal}_q$ prior to Line 5 of OP, it means that OP returns v. Hence, the lemma holds in this case.

In the second case (when LL_q is executed after SC_q), OP is linearized just prior to $LP(\mathrm{LL}_q)$. Hence, OP′ is the latest successful SC operation such that $LP(\text{OP}') < LP(\mathrm{LL}_q)$. Since LL_q writes into $\texttt{lastVal}_q$, it follows that LL_q is linearized at either $LP(\mathrm{LL}_q(1))$ or $LP(\mathrm{LL}_q(2))$. Consequently, OP′(1) is the latest successful SC operation on X such that $LP(\text{OP}'(1)) < LP(\mathrm{LL}_q(1))$ or $LP(\text{OP}'(1)) < LP(\mathrm{LL}_q(2))$. Let v be the value that OP′ writes in $\mathcal{O}$. Then, $\mathrm{LL}_q(1)$ or $\mathrm{LL}_q(2)$ returns (*success*, v), and LL_q writes v into $\texttt{lastVal}_q$. Since LL_q is the latest LL operation to write into $\texttt{lastVal}_q$ prior to Line 5 of OP, it means that OP returns v. Hence, the lemma holds in this case as well. □

Lemma 15 (Correctness of SC) *Let* OP *be any SC operation by some process p, and* OP′ *be the latest LL operation by p that precedes* OP. *Then,* OP *succeeds if and only if there does not exist any successful SC operation* OP″ *such that* $LP(\text{OP}') < LP(\text{OP}'') < LP(\text{OP})$.

Proof. If OP returns *false*, we examine the following three cases: (1) OP′(1) succeeds, (2) OP′(1) fails and OP′(2) succeeds, and (3) both OP′(1) and OP′(2) fail. In the first case, since OP returns *false*, there exists some successful SC operation OP″ such that $LP(\text{OP}'(1)) < LP(\text{OP}''(1)) < LP(\text{OP}(1))$. Since by our linearization, OP′ is linearized at $LP(\text{OP}'(1))$, OP″ is linearized at $LP(\text{OP}''(1))$, and OP is linearized at $LP(\text{OP}(1))$, we have $LP(\text{OP}') < LP(\text{OP}'') < LP(\text{OP})$. Hence, OP is correct to return *false*. The proof for the second case is identical, and is therefore omitted.

In the third case, let $(failure, q)$ be the value returned by OP′(1). Then, by Definition 1, there exists some successful SC operation SC_q by q such that (1) $LP(\text{SC}_q(1))$ lies in the execution interval of OP′(1), and (2) $LP(\text{OP}'(1)) < LP(\text{SC}_q(1))$. Let LL_q be the latest LL operation by q to write into $\texttt{lastVal}_q$ prior to Line 5 of OP. We examine the following two cases: (1) LL_q is executed before SC_q, and (2) LL_q is executed after SC_q. In the first case, OP′ is linearized at the point just prior to $LP(\text{SC}_q)$. Since $LP(\text{SC}_q) = LP(\text{SC}_q(1))$, and since $LP(\text{SC}_q(1))$ lies in the execution interval of OP′(1), it follows that $LP(\text{OP}')$ lies before Line 5 of OP′. In the second case, OP′ is linearized at the point just prior to $LP(\text{LL}_q)$. Since, by definition of LL_q, $LP(\text{LL}_q)$ lies before Line 5 of OP′, it follows in this case too that $LP(\text{OP}')$ lies before Line 5 of OP′. Let $(failure, r)$ be the value returned by OP′(2). Then, by Definition 1, there exists some suc-

cessful SC operation SC_r by r such that (1) $LP(\mathrm{SC}_r(1))$ lies in the execution interval of $\mathrm{OP}'(2)$, and (2) $LP(\mathrm{OP}'(2)) < LP(\mathrm{SC}_r(1))$. Consequently, we have $LP(\mathrm{OP}') < LP(\mathrm{SC}_r(1)) < LP(\mathrm{OP})$. Since $LP(\mathrm{SC}_r) = LP(\mathrm{SC}_r(1))$, we have $LP(\mathrm{OP}') < LP(\mathrm{SC}_r) < LP(\mathrm{OP})$. Hence, OP is correct to return *false*.

If OP returns *true*, then clearly $\mathrm{OP}(1)$ succeeds. Hence, either $\mathrm{OP}'(1)$ or $\mathrm{OP}'(2)$ succeeds. Without loss of generality, assume that $\mathrm{OP}'(1)$ succeeds. Then, by Definition 1, there does not exist any successful SC operation SC_q such that $LP(\mathrm{OP}'(1)) < LP(\mathrm{SC}_q(1)) < LP(\mathrm{OP}(1))$. Since OP' is linearized at $LP(\mathrm{OP}'(1))$, OP is linearized at $LP(\mathrm{OP}(1))$, and any other successful SC operation SC_q is linearized at $LP(\mathrm{SC}_q(1))$, it follows that there does not exist any successful SC operation SC_q such that $LP(\mathrm{OP}') < LP(\mathrm{SC}_q) < LP(\mathrm{OP})$. Hence, OP is correct to return *true*. □

Lemma 16 (Correctness of VL) *Let* OP *be any VL operation by some process p, and* OP' *be the latest LL operation by p that precedes* OP. *Then,* OP *returns true if and only if there does not exist any successful SC operation* OP'' *such that* $LP(\mathrm{OP}') < LP(\mathrm{OP}'') < LP(\mathrm{OP})$.

Proof. Similar to the proof of Lemma 15. □

Theorem 2 *Algorithm 4.2 is wait-free and implements a linearizable* 64*-bit LL/SC object from a single* 64*-bit WLL/SC object and one additional* 64*-bit register per process. The time complexity of LL, SC, and VL is* $O(1)$.

Proof. The theorem follows immediately from Lemmas 14, 15, and 16. □

A.1.3 Proof of Algorithm 4.3

Let $\mathcal{O}$ be the WLL/SC object implemented by Algorithm 4.3. Let OP be any WLL operation, OP$'$ be any SC operation, and OP$''$ be any VL operation on $\mathcal{O}$. Then, we set $LP(\text{OP})$ at Line 1, $LP(\text{OP}')$ at Line 7, and $LP(\text{OP}'')$ at Line 10.

In the following, let $\text{SC}_{p,i}$ denote the ith *successful* SC on $\mathcal{O}$ by process p and $v_{p,i}$ denote the value written in $\mathcal{O}$ by $\text{SC}_{p,i}$. We will assume that the initializing step was performed by process 0 during its first successful SC operation.

Lemma 17 *Let $SC_{p,i}$ be a successful SC operation by process p. If i is odd, then p changes the value of $index_p$ from 1 to 0 at Line 8 of $SC_{p,i}$. Otherwise, p changes the value of $index_p$ from 0 to 1 at Line 8 of $SC_{p,i}$.*

Proof. (By induction) For the base case (i.e., $i = 1$), the lemma holds trivially by initialization. The inductive hypothesis states that the lemma holds for some $\text{SC}_{p,k}$, $k \geq 1$. We now show that the lemma holds for $\text{SC}_{p,k+1}$ as well. If $k+1$ is odd, then, by inductive hypothesis, p changes the value of $index_p$ from 0 to 1 at Line 8 of $\text{SC}_{p,k}$. Therefore, at the beginning of $\text{SC}_{p,k+1}$, the value of $index_p$ is 1. Consequently, p changes $index_p$ to $1 - 1 = 0$ at Line 8 of $\text{SC}_{p,k+1}$. Hence, the lemma holds in this case.

If, on the other hand, $k+1$ is even, then, by inductive hypothesis, p changes the value of $index_p$ from 1 to 0 at Line 8 of $\text{SC}_{p,k}$. Therefore, at the beginning of $\text{SC}_{p,k+1}$, the value of $index_p$ is 0. Consequently, p changes $index_p$ to $1 - 0 = 1$ at Line 8 of $\text{SC}_{p,n+1}$. Hence, the lemma holds. □

Corollary 1 *At the beginning of $SC_{p,i}$, $index_p$ holds the value $i \bmod 2$.*

Lemma 18 *From the moment p performs Line 6 of $SC_{p,i}$, until p completes the $SC_{p,i+1}$ operation, variable* $\texttt{val}_p i \bmod 2$ *holds the value $v_{p,i}$.*

Proof. Notice that, by Corollary 1, at the beginning of $\text{SC}_{p,i}$ $index_p$ holds the value $i \bmod 2$. Therefore, p writes $v_{p,i}$ into $\texttt{val}_p i \bmod 2$ at Line 6 of $\text{SC}_{p,i}$. Since, by Lemma 17, p changes $index_p$ at Line 11 of $\text{SC}_{p,i}$ to $1 - (i \bmod 2) \neq i \bmod 2$, $v_{p,i}$ will not be overwritten in $\texttt{val}_p i \bmod 2$ until $index_p$ reaches the value $i \bmod 2$ again. By Lemma 17, $index_p$ does not reach the value $i \bmod 2$ until p changes $index_p$ to $1 - (1 - (i \bmod 2)) = i \bmod 2$ at Line 11 of $\text{SC}_{p,i+1}$. Therefore, $\texttt{val}_p i \bmod 2$ holds the value $v_{p,i}$ from the moment p performs Line 6 of $\text{SC}_{p,i}$ until p completes $\text{SC}_{p,i+1}$. □

Lemma 19 (Correctness of WLL) *Let* OP *be any WLL operation, and* OP′ *the latest successful SC operation such that $LP(\text{OP}') < LP(\text{OP})$. If* OP *returns* $(success, v)$*, then v is the value that* OP′ *writes in $\mathcal{O}$. If* OP *returns* $(failure, q)$*, then there exists a successful SC operation* OP″ *by process q such that: (1) $LP(\text{OP}'')$ lies in the execution interval of* OP*, and (2) $LP(\text{OP}) < LP(\text{OP}'')$.*

Proof. Let p be the process executing OP. If OP returns $(success, v)$, let q be the process that executes OP′. Then, $\text{OP}' = \text{SC}_{q,i}$, for some i. Notice that, by Corollary 1, q writes $i \bmod 2$ into X at Line 7 of $\text{SC}_{q,i}$. Hence, p reads $(i \bmod 2, q)$ from X at Line 2 of OP. Since the VL call at Line 3 of OP succeeds, no process (including q) performs a successful SC operation on X between Lines 1 and 3 of OP. Then, by Lemma 18, $\texttt{val}_q i \bmod 2$ holds the value $v_{q,i}$ at all times between Lines 1 and 3 of OP. Consequently, p reads $v_{q,i}$ at Line 2 of OP and returns the correct value at Line 3 of OP.

If OP returns $(failure, q)$, then the VL call at Line 3 of OP fails. Hence, some other process performs a successful SC on X between Lines 1 and 3 of OP. Consequently, the value (b, q) that p reads from X at Line 4 of OP was written into X by q while p was between Lines 1 and 4 of OP. Therefore, there exists a successful SC operation OP$''$ by q such that q executes Line 7 between Lines 1 and 4 of OP. Since $LP(\text{OP})$ is at Line 1 and $LP(\text{OP}'')$ is at Line 7, it follows that (1) $LP(\text{OP})''$ lies in the execution interval of OP, and (2) $LP(\text{OP}) < LP(\text{OP}'')$. □

Lemma 20 (Correctness of SC) *Let* OP *be any SC operation by process p, and* OP$'$ *be the latest LL operation by p that precedes* OP. *Then,* OP *succeeds if and only if there does not exist any successful SC operation* OP$''$ *such that* $LP(\text{OP}') < LP(\text{OP}'') < LP(\text{OP})$.

Proof. If OP returns *false*, then the SC call at Line 7 of OP fails. We examine the following two cases: (1) OP$'$ returns at Line 3, and (2) OP$'$ returns at Line 5. Notice that, in both cases, there exists some successful SC operation OP$''$ on $\mathcal{O}$ that writes into X (at Line 7) between Line 1 of OP$'$ and Line 7 of OP. Since $LP(\text{OP}')$ is at Line 1, $LP(\text{OP})$ at Line 7, and $LP(\text{OP}'')$ at Line 7, it follows that $LP(\text{OP}') < LP(\text{OP}'') < LP(\text{OP})$. Hence, OP was correct in returning *false*.

If OP returns *true*, then the SC operation on X at Line 7 of OP succeeds. Therefore, there does not exist any successful SC operation OP$''$ on $\mathcal{O}$ such that OP$''$ executes Line 7 between Line 1 of OP$'$ and Line 7 of OP. Since $LP(\text{OP}')$ is at Line 1, $LP(\text{OP})$ at Line 7, and for any successful SC operation OP$''$ $LP(\text{OP}'')$ is at Line 7, it follows that there does not exist any successful SC operation OP$''$ such that $LP(\text{OP}') < LP(\text{OP}'') < LP(\text{OP})$. Hence, OP was correct in returning *true*. □

Lemma 21 (Correctness of VL) *Let* OP *be any VL operation by some process p, and* OP′ *be the latest LL operation by p that precedes* OP. *Then,* OP *returns true if and only if there does not exist any successful SC operation* OP″ *such that* $LP(\text{OP}') < LP(\text{OP}'') < LP(\text{OP})$.

Proof. Similar to the proof of Lemma 20. □

Theorem 3 *Algorithm 4.3 is wait-free and implements a* 64*-bit WLL/SC object from a single (1-bit, pid)-LL/SC object and an additional three* 64*-bit registers per process. The time complexity of LL, SC, and VL is* $O(1)$.

Proof. The theorem follows immediately from Lemmas 19, 20, and 21. □

A.1.4 Proof of Algorithm 4.4

Let $\mathcal{O}$ be the (1-bit, pid)-LL/SC object implemented by Algorithm 4.4. The operations on $\mathcal{O}$ are linearized according to the following rules. Let OP be any SC operation on $\mathcal{O}$. The linearization point of OP is determined by two cases. If the condition at Line 5 of OP holds (i.e., $old_p \neq chk_p$), we linearize OP at any point during Line 5. Otherwise, we linearize OP at Line 6. Let OP′ be any VL operation on $\mathcal{O}$. The linearization point of OP′ is determined by two cases. If the first part of the condition at Line 9 of OP′ does not hold (i.e., $old_p \neq chk_p$), we linearize OP′ at any point during Line 9. Otherwise, we linearize OP′ at the point at Line 9 when X is read. Let OP″ be any LL operation on $\mathcal{O}$. The linearization point of OP″ is determined by two cases. If OP″ reads different values at Lines 1 and 3, we linearize OP″ at Line 1. Otherwise, we linearize OP″ at Line 3.

In the following, we assume that the `select` operation satisfies Property 1. We restate this property below.

Property 1 *Let* OP *and* OP′ *be any two consecutive BitPid_LL operations by some process p. If p reads* (s, q, v) *from* X *in both Lines 1 and 3 of* OP*, then process q does not write* $(s, q, *)$ *into* X *after p executes Line 3 of* OP *and before it invokes* OP′.

Lemma 22 (Correctness of BitPid_LL) *Let* OP *be any BitPid_LL operation and* OP′ *be the latest successful SC operation such that* $LP(\text{OP}') < LP(\text{OP})$. *Let q be the process executing* OP′ *and v be the value that* OP′ *writes in* $\mathcal{O}$. *Then,* OP *returns* (v, q).

Proof. Let p be the process executing OP. We examine the following two cases: (1) the values that p reads at Lines 1 and 3 of OP are different, and (2) the values that p reads at Lines 1 and 3 of OP are the same. In the first case, OP is linearized at Line 1. Since OP′ is the latest successful SC such that $LP(\text{OP}') < LP(\text{OP})$, it means that OP′ is the latest successful SC to write into X before Line 1 of OP. Hence, OP returns (v, q).

In the second case, OP is linearized at Line 3. Since OP′ is the latest successful SC such that $LP(\text{OP}') < LP(\text{OP})$, it means that OP′ is the latest successful SC to write into X before Line 3 of OP. Therefore, p reads (v, q) at Line 3 of OP. Since the values that p reads at Lines 1 and 3 of OP are the same, OP returns (v, q). □

Lemma 23 (Correctness of SC) *Let* OP *be any SC operation by process p, and* OP′ *be the latest BitPid_LL operation by p that precedes* OP. *Then,* OP *succeeds*

if and only if there does not exist any successful SC operation OP″ *such that* $LP(\text{OP}') < LP(\text{OP}'') < LP(\text{OP})$.

Proof. We examine the following three cases: (1) OP returns *false* at Line 5, (2) OP returns *false* at Line 6, and (3) OP returns *true* at Line 8. In the first case, the values that p reads at Lines 1 and 3 of OP′ are different. Hence, there exists some successful SC operation OP″ that writes into X (at Line 6) between Lines 1 and 3 of OP′. Since, by the linearization, OP′ is linearized at Line 1, OP is linearized at some point during Line 5, and OP″ is linearized at Line 6, it means that $LP(\text{OP}') < LP(\text{OP}'') < LP(\text{OP})$. Hence, OP was correct in returning *false*.

In the second case, the values that p reads at Lines 1 and 3 of OP′ are the same and the CAS at Line 6 of OP fails. Hence, there exists some successful SC operation OP″ that writes into X (at Line 6) between Line 3 of OP′ and Line 6 of OP. Since, by the linearization, OP′ is linearized at Line 3, OP is linearized at Line 6, and OP″ is linearized at Line 6, it means that $LP(\text{OP}') < LP(\text{OP}'') < LP(\text{OP})$. Hence, OP was correct in returning *false*.

In the third case, the values that p reads at Lines 1 and 3 of OP′ are the same and the CAS at Line 6 of OP succeeds. Let (s, q, v) be the value that p reads from X at Lines 1 and 3 of OP′. Then, X has the value (s, q, v) when p executes Line 6 of OP. Since, by Property 1, no process writes (s, q, v) into X between Line 3 of OP′ and Line 6 of OP, it means that X doesn't change between Line 3 of OP′ and Line 6 of OP. Consequently, there does not exist any successful SC operation OP″ such that OP″ writes into X (at Line 6) between Line 3 of OP′ and Line 6 of OP. Since, by the linearization, OP′ is linearized at Line 3, OP is linearized at Line 6, and any successful SC operation OP″ is linearized at Line 6, it follows that there does not

exist any successful SC operation OP$''$ such that $LP(\text{OP}') < LP(\text{OP}'') < LP(\text{OP})$. Hence, OP was correct in returning *true*. □

Lemma 24 (Correctness of VL) *Let* OP *be any VL operation by some process p, and* OP$'$ *be the latest LL operation by p that precedes* OP*. Then,* OP *returns true if and only if there does not exist any successful SC operation* OP$''$ *such that* $LP(\text{OP}') < LP(\text{OP}'') < LP(\text{OP})$.

Proof. Similar to the proof of Lemma 23. □

Theorem 4 *Algorithm 4.4 is wait-free and, if the* `select` *procedure satisfies Property 1, implements a linearizable (1-bit, pid)-LL/SC object from* 64*-bit CAS objects and* 64*-bit registers. If* τ *is the time complexity of* `select`*, then the time complexity of BitPid_LL, SC, VL, and BitPid_read operations are* $O(1)$*,* $O(1)+\tau$*,* $O(1)$*, and* $O(1)$*, respectively. If s is the per-process space overhead of* `select`*, then the per-process space overhead of the algorithm is* $4+s$.

Proof. The theorem follows immediately from Lemmas 22, 23, and 24. □

A.1.5 Proof of Algorithm 4.5

We show that Algorithm 4.5 satisfies Property 1. We restate Property 1 below.

Property 1 *Let* OP *and* OP$'$ *be any two consecutive BitPid_LL operations by some process p. If p reads* (s,q,v) *from* X *in both Lines 1 and 3 of* OP*, then process q does not write* $(s,q,*)$ *into* X *after p executes Line 3 of* OP *and before it invokes* OP$'$.

Definition 2 *An 'epoch of p' is the period of time between the two consecutive executions of Line 19 in* `select`(p)*, or a period of time between the end of the initialization phase of the algorithm and the first time Line 19 is executed in* `select`(p).

Definition 3 *Interval* $x, x \oplus_K \Delta)$ *is the 'current interval' of an epoch if* x *is the value of the variable* $nextStart_p$ *at the beginning of that epoch.*

Lemma 25 *Let* E *be an epoch of* p *and* t *an arbitrary point in time during* E. *Let* E_t *be the time interval that spans from the moment* E *starts until time* t. *If the condition at Line 13 of* `select`(p) *holds true at most* $2N$ *times during* E_t, *then all the sequence numbers that* `select`(p) *returns during* E_t *are unique and belong to the current interval of* E.

Proof. We prove the lemma in two steps. First, we prove that the total number of sequence numbers returned by `select`(p) during E_t is at most $(2N + 1)N$. Then, we use this fact to prove that all the sequence numbers that `select`(p) returns during E_t are unique and belong to the current interval of E.

Claim 1 *The total number of sequence numbers returned by* `select`(p) *during* E_t *is at most* $(2N + 1)N$.

Proof. Let t_e be the latest time during E_t that a sequence number is returned by `select`(p). (If there is no such time, then the claim trivially holds.) Then, since $t_e \leq t$, it follows that the condition at Line 13 of `select`(p) holds true at most $2N$ times prior to t_e. Thus, the value of $procNum_p$ was set to 0 at Line 15 at most $2N$ times prior to t_e. Furthermore, since t_e belongs to E, the value of

$procNum_p$ hasn't yet reached N at time t_e (otherwise, the new epoch would have started at Line 19). Since $procNum_p$ has been set to 0 at most $2N$ times prior to t_e, the number of times $procNum_p$ was incremented at Line 16 prior to t_e is at most $(2N+1)(N-1)$ (otherwise, $2N$ resets wouldn't be able to prevent $procNum_p$ from reaching the value N). Therefore, the condition at Line 13 does not hold at most $(2N+1)(N-1)$ times prior to t_e. Hence, the total number of times the condition at Line 13 was tested prior to t_e is at most $(2N+1)(N-1)+2N = 2N^2+N-1$. Then, the total number of sequence numbers returned by `select`(p) during E_t is at most $2N^2+N-1+1 = (2N+1)N$. □

Let t_1 and t_2 in E be the times of two successive executions of Line 22 by p. Since t_1 and t_2 belong to E, p does not execute Line 19 during (t_1, t_2). Hence, p executes Line 18 during (t_1, t_2), incrementing val_p by one. Thus, the value of val_p at time t_2 is by one greater than the value of val_p at time t_1. To show that val_p always stays within the current interval of E, observe the following. At the beginning of an epoch, val_p is set to the first value in the current interval. Furthermore, by Claim 1, Line 22 is executed at most $(2N+1)N$ times during E_t. Consequently, val_p stays within the current interval at all times during E_t, and all the sequence numbers that `select`(p) returns during E_t are unique and belong to the current interval of E. □

Lemma 26 *During an epoch by some process p, the condition at Line 13 holds true at most* $2N$ *times.*

Proof. Suppose not. Let E be an epoch by p during which the condition at Line 13 holds true more than $2N$ times. Then, there exists an entry in `A` for which

the condition at Line 13 is true three or more times. Let $\texttt{A}q$ be the first such entry. Let t_1, t_2 and t_3 in E be the times that the entry $\texttt{A}q$ is read at Line 12 of $\texttt{select}(p)$. Let a_1, a_2, and a_3 be the values that $\texttt{A}q$ holds at times t_1, t_2, and t_3, respectively. Let x_1, x_2, and x_3 be the values that $nextStart_p$ holds at times t_1, t_2, and t_3, respectively. Then, since a_1, a_2, and a_3 satisfy the condition at Line 13 of $\texttt{select}(p)$, they must be of the form $(s_1, p, *)$, $(s_2, p, *)$, and $(s_3, p, *)$, respectively, where s_1, s_2, and s_3 belong to intervals $x_1, x_1 \oplus_K \Delta)$, $x_2, x_2 \oplus_K \Delta)$, and $x_3, x_3 \oplus_K \Delta)$, respectively. Let C be the current interval of E. Let E_{t_3} be the time interval that spans from the moment E starts until time t_3. Then, at the beginning of E, C is set to $x, x \oplus_K \Delta)$ and $nextStart_p$ to $x \oplus_K \Delta$, for some x. Furthermore, each time the condition at Line 13 is true, $nextStart_p$ is incremented by Δ at Line 14. Since $\texttt{A}q$ is the first entry for which the condition at Line 13 is true three or more times, it means that during E_{t_3}, the condition at Line 13 could have been true at most $2N$ times. Hence, during E_{t_3}, $nextStart_p$ could have been incremented at most $2N$ times. Therefore, $nextStart_p$ has a value at most $x \oplus_K (2N+1)\Delta$ at time t_3. Then, since $\oplus_K$ is performed modulo $K = (2N+2)\Delta$, at no point during E_{t_3} does the interval $nextStart_p, nextStart_p \oplus_K \Delta)$ intersect with C. Therefore, the intervals $x_1, x_1 \oplus_K \Delta)$, $x_2, x_2 \oplus_K \Delta)$, and $x_3, x_3 \oplus_K \Delta)$ are disjoint and do not intersect with C. Consequently, the sequence numbers s_1, s_2 and s_3 are distinct and do not belong to C.

Since, by an earlier argument, the condition at Line 13 holds true at most $2N$ times during E_{t_3}, it follows by Lemma 25 that all the sequence numbers returned by $\texttt{select}(p)$ during E_{t_3} belong to C. Furthermore, it follows by the algorithm that at all times during (t_1, t_3), the latest sequence number written into $\texttt{X}$ by p is returned by $\texttt{select}(p)$ during E_{t_3}. Consequently, at all times during (t_1, t_3), if $\texttt{X}$ holds a value of the form $(s, p, *)$, then s belongs to C.

Since Aq holds the value a_1 (respectively, a_2, a_3) at time t_1 (respectively, t_2, t_3), process q must have written values a_2 and a_3 into Aq (at Line 2 of BitPid_LL) at some time during (t_1, t_3). Hence, q must have read a_3 from X at some time during (t_1, t_3). Since, at all times during (t_1, t_3), if X holds a value of the form $(s, p, *)$, then s belongs to C, we have $s_3 \in C$. This is a contradiction to the fact that the intervals $x_3, x_3 \oplus_K \Delta)$ and C are disjoint. Hence, we have the lemma. □

Lemma 27 *All sequence numbers returned by* `select` *during an epoch are unique and belong to that epoch's current interval.*

Proof. Let t be the time at the very end of the current epoch. Then, the lemma holds trivially by Lemmas 25 and 26. □

Lemma 28 *Current intervals of two consecutive epochs are disjoint.*

Proof. Let E be an epoch by some process p, and C be the current interval of E. Then, at the beginning of E, C is set to $x, x \oplus_K \Delta)$ and $nextStart_p$ to $x \oplus_K \Delta$, for some x. Furthermore, each time the condition at Line 13 holds true, $nextStart_p$ is incremented by Δ at Line 14. Since, by Lemma 26, the condition at Line 13 can hold true at most $2N$ times during an epoch, $nextStart_p$ can be at most $x \oplus_K (2N+1)\Delta$ at the end of E. Therefore, the current interval of the next epoch can be at most $x \oplus_K (2N+1)\Delta, x \oplus_K (2N+2)\Delta)$. Since $\oplus_K$ is done modulo $K = (2N+2)\Delta$, intervals $x \oplus_K (2N+1)\Delta, x \oplus_K (2N+2)\Delta)$ and $x, x \oplus_K \Delta)$ are disjoint. Hence, we have the lemma. □

Lemma 1 *Algorithm 4.5 satisfies Property 1. The time complexity of the implementation is $O(1)$, and the per-process space overhead is 3.*

Proof. Suppose that `select` does not satisfy Property 1. Then, there exist two consecutive BitPid_LL operations OP and OP′ by some process p and a successful SC operation OP″ by some process q, such that the following is true: p reads (s, q, v) at both Lines 1 and 3 of OP, yet process q writes $(s, q, *)$ at Line 6 of OP″ before p invokes OP′. Let t be the time when q executes Line 6 of OP″. Let E be process q's epoch at time t, and C the current interval of E. Since, by Lemma 27, all the sequence numbers returned by `select`(q) during E are unique and belong to C, it follows that all the sequence numbers q writes into `X` during E are unique and belong to C. Consequently, $s \in C$.

Let E' be the epoch that precedes E. (If there is no such epoch then the lemma trivially holds, since, by the argument above, all the sequence number that q writes into `X` during E are unique, and there can not be a process p that has read (s, q, v) at Lines 1 and 3 of OP before q writes it at Line 6 of OP″.) Let C' be the current interval of E'. Let t' be the first time that q reads `A`p at Line 12 during E'. By Lemma 27, all the sequence numbers returned by `select`(q) during $E' \cup E$ belong to $C' \cup C$. Moreover, it follows by the algorithm that at all times during (t', t), the latest sequence number written into `X` by q was returned by `select`(q) during $E' \cup E$. Consequently, at all times during (t', t), if `X` holds a value of the form $(s, q, *)$, then s belongs to $C' \cup C$. Since, by Lemma 28, C' and C are disjoint, `X` does not contain the value of the form $(s, q, *)$ during (t', t). Then, p must have read (s, q, v) at Line 3 of OP before t'. Consequently, p wrote $(s, q, 0)$ into `A`p at Line 2 of OP before t'. Since OP is p's latest BitPid_LL at time t, no other BitPid_LL by p wrote into `A`p after Line 2 of OP and before t. Therefore, and at all times during (t', t), `A`p holds the value $(s, q, 0)$.

Observe that, at the end of E', $procNum_p$ has value N. Hence, in the last N executions of `select`(q) during E', $procNum_p$ has been incremented by 1 at Line 16. Thus, in the last N executions of `select`(q) during E', every entry in `A` was read once, and none satisfied the condition at Line 13. Consequently, we have the following: (1) in the execution of `select`(q) where the entry `A`p was read, the condition at Line 13 was not satisfied, and (2) in the last N executions of `select`(q) during E', variable $nextStart_p$ didn't change. Since $C = x, x \oplus_K \Delta)$, where x is the value of $nextStart_p$ at the end of E', during the execution of `select`(q) where the entry `A`p was read, `A`p was not of the form $(s', q, 0)$, for some $s' \in C$. This is a contradiction to the fact that at all times during (t', t), the entry `A`p contains the value $(s, q, 0)$, where $s \in C$. □

A.1.6 Proof of Algorithm 4.6

We show that Algorithm 4.6, satisfies Property 1. We restate Property 1 below.

Property 1 *Let* OP *and* OP′ *be any two consecutive BitPid_LL operations by some process p. If p reads* (s, q, v) *from* `X` *in both Lines 1 and 3 of* OP*, then process q does not write* $(s, q, *)$ *into* `X` *after p executes Line 3 of* OP *and before it invokes* OP′.

Definition 4 *An 'epoch of p' is the period of time between the two consecutive executions of Line 31 in* `select`(p)*, or a period of time between the end of the initialization phase of the algorithm and the first time Line 31 is executed in* `select`(p).

Definition 5 *A 'pass k of an epoch E' is a period of time in E during which the*

variable $passNum_p$ holds the value k.

Definition 6 *Interval C is the 'current interval' of an epoch, if C is the value of the variable I_p at the beginning of that epoch.*

We introduce the following notation. Let E be some epoch. Then, for all $k \in \{0, 1, \ldots, \lg(N+1)\}$, we let I_k^E denote the value of the interval I_p at the end of the kth pass of an epoch E, and s_k^E denote the number of entries in `A` that, at the end of the kth pass of an epoch E, hold the value of the form $(s, p, 1)$, for some $s \in I_k^E$.

In the following, we assume that $(N+1)$ is a power of two.

Lemma 29 *There are $\lg(N+1)+1$ passes in an epoch.*

Proof. At the beginning of an epoch, the value of variable $passNum_p$ is set to 0. Furthermore, an epoch ends when the Line 31 is executed for the first time during that epoch, which happens only when the condition at Line 28 is fails, i.e., when $passNum_p$ reaches value $\lg(N+1)$. Hence, at the end of an epoch, the value of $passNum_p$ is $\lg(N+1)$. Since the variable $passNum_p$ is incremented only at Lines 18 and 30 and is incremented only by one, it follows that during an epoch, $passNum_p$ goes through all the values in the range $0\,..\,\lg(N+1)$. Hence, there are exactly $\lg(N+1)+1$ passes in an epoch. □

Lemma 30 *In any given pass, process p invokes* `select` *exactly N times.*

Proof. At the beginning of any pass, the value of $procNum_p$ is zero (since the pass begins at Lines 18, 30, and 31, and $procNum_p$ is set to zero at Lines 17 and 27; likewise, $procNum_p$ is set to zero at initialization time). Furthermore, a pass ends only

after the variable $procNum_p$ reaches the value $N-1$ (since the variable $passNum_p$ is modified only after the conditions at Lines 15 and 23 are *false*). Hence, during a pass, $procNum_p$ goes through all the values in the range $0\,..\,N-1$. Notice that in every invocation of `select` in which a pass doesn't end, process p executes Lines 16 and 24 (since it doesn't execute Lines 18, 30, and 31). Hence, p increments $procNum_p$ by one in the first $N-1$ invocations of `select` during the pass, and then ends the pass during the Nth invocation. Hence, in any given pass, process p invokes `select` exactly N times. □

Lemma 31 *Let E be an epoch by some process p, and $t \in E$ the time when p completes the 0th pass of E. Then, if some entry $\mathtt{A}q$ holds the value $(s, p, 1)$ at time $t' \in E$, for $t' \geq t$, then $\mathtt{A}q$ holds the value $(s, p, 1)$ at all times during (t, t').*

Proof. Suppose not. Then, at some time $t'' \in (t, t')$, $\mathtt{A}q$ does not hold the value $(s, p, 1)$. Therefore, at some point during (t'', t'), the value $(s, p, 1)$ is written into $\mathtt{A}q$. Since by the time t'', process p has already performed the 0th pass of E, some process other than p must have written $(s, p, 1)$ into $\mathtt{A}q$, which is impossible. Hence, we have the lemma. □

Lemma 32 *If E is an epoch by some process p, then the length of the interval I_k^E is $((N+1)/2^k)\Delta$, for all $k \in \{0, 1, \ldots, \lg(N+1)\}$.*

Proof. (By induction) For the base case (i.e., $k = 0$), the lemma trivially holds, since at the beginning of the 0th pass I_p is initialized to be of length $(N+1)\Delta$. The inductive hypothesis states that the length of I_j^E is $((N+1)/2^j)\Delta$, for all $j \leq k$. We

now show that the length of I^E_{k+1} is $((N+1)/2^{k+1})\Delta$. Notice that, by the algorithm, I^E_{k+1} is a half of I_k. Moreover, we made an assumption earlier that $N+1$ is a power of two. Hence, the length of I^E_{k+1} is exactly $(((N+1)/2^k)/2)\Delta = ((N+1)/2^{k+1})\Delta$. □

Lemma 33 *If E is an epoch by some process p, then the value of s^E_k is at most $(N+1)/2^k - 1$, for all $k \in \{0, 1, \ldots, \lg(N+1)\}$.*

Proof. For the base case (i.e., $k = 0$), the lemma trivially holds, since A can hold at most N entries. The inductive hypothesis states that the value of s^E_j is at most $(N+1)/2^j - 1$, for all $j \leq k$. We now show that the value of s^E_{k+1} is at most $(N+1)/2^{k+1} - 1$. Since s_k is the number of entries in A that, at the end of the kth pass, hold a sequence number from I^E_k, it follows by Lemma 31 that p can count at most s^E_k sequence numbers during the $(k+1)$st pass. Moreover, since I^E_{k+1} is a half of I^E_k with a smaller count, it follows that at most $\lfloor s^E_k/2 \rfloor$ of the counted sequence numbers fall within I^E_{k+1}. Hence, by Lemma 31, at the end of the $(k+1)$st pass, at most $\lfloor s^E_k/2 \rfloor$ entries in A are of the form $(s, p, 1)$, $s \in I^E_{k+1}$. Therefore, we have $s^E_{k+1} = \lfloor s^E_k/2 \rfloor = \lfloor ((N+1)/2^k - 1)/2 \rfloor$. Since $N+1$ is a power of two, it means that $s^E_{k+1} = (N+1)/2^{k+1} - 1$. Hence, the observation holds. □

Lemma 34 *Let E be an epoch by some process p. Let I' be the value of the interval I_p at the end of E. Then, I' contains exactly Δ sequence numbers.*

Proof. The lemma follows immediately by Lemmas 29 and 32. □

Lemma 35 *Let E be an epoch by some process p. Let I' be the value of the interval I_p at the end of E. Then, at the end of E, no entry in* `A` *is of the form* $(s, p, 1)$*, where* $s \in I'$.

Proof. The lemma follows immediately by Lemmas 29 and 33. □

Lemma 36 *Let E be an epoch by some process p. Then, all the sequence numbers that* `select`(p) *returns during E are unique and belong to the current interval of E.*

Proof. Let C be the current interval of E. We prove the lemma in two steps. First, we prove that the total number of sequence numbers returned by `select`(p) during E is at most $N(\lg(N+1)+1)$. Then, we use this fact to prove that all the sequence numbers that `select`(p) returns during E are unique and belong to C.

Claim 2 *The total number of sequence numbers returned by* `select`(p) *during E is at most $N(\lg(N+1)+1)$.*

Proof. During each pass of E, `select`(p) returns exactly N sequence numbers (by Lemma 30). Since an epoch consists of $\lg(N+1)+1$ passes (by Lemma 29), the total number of sequence numbers returned by `select`(p) during E is therefore $N(\lg(N+1)+1)$. □

Let t_1 and t_2 in E be the times of two successive executions of Line 34 by p. Since t_1 and t_2 belong to E, p does not execute Line 31 during (t_1, t_2). Hence, p executes either Line 19, Line 25, or Line 29 during (t_1, t_2) and increments val_p by one. Thus, the value of val_p at time t_2 is by one greater than the value of val_p at

time t_1. To show that val_p always stays within C, observe the following. At the beginning of an epoch, val_p is set to the first value in C. Furthermore, by Lemma 2, Line 34 is executed at most $N(\lg(N+1)+1)$ times during E. Consequently, val_p stays within C at all times during E. Therefore, all the sequence numbers that `select`(p) returns during E are unique and belong to C. □

Lemma 37 *Current intervals of two consecutive epochs are disjoint.*

Proof. Let E and E' be any two consecutive epochs. Let C (respectively, C') be the current interval of E (respectively, E'). Let I be the value of the interval I_p at the start of E. Let I' be the value of the interval I_p after Line 33 of `select`(p) is executed during E. By Lemma 34, I is of the form $x, x \oplus_K \Delta)$, for some x. Therefore, $C = x, x \oplus_K \Delta)$ and $I' = x \oplus_K \Delta, x \oplus_K (N+2)\Delta)$. Since the operation $\oplus_K$ is performed modulo $K = (N+2)\Delta$, intervals C and I' are disjoint. Furthermore, since C' is a subinterval of I', C and C' are disjoint as well. □

Lemma 2 *Algorithm 4.6 satisfies Property 1. The time complexity of the implementation is $O(1)$, and the per-process space overhead is 4.*

Proof. Suppose that `select` does not satisfy Property 1. Then, there exist two consecutive BitPid_LL operations OP and OP$'$ by some process p and a successful SC operation OP$''$ by some process q, such that the following is true: p reads (s, q, v) at both Lines 1 and 3 of OP, yet process q writes $(s, q, *)$ at Line 6 of OP$''$ before p invokes OP$'$. Let t be the time when q executes Line 6 of OP$''$. Let E be the q's epoch at time t. Let C be the current interval of E. Since, by Lemma 36, all the sequence numbers returned by `select`(q) during E are unique and belong to

C, it follows that all the sequence numbers that q writes into X during E are unique and belong to C. Consequently, $s \in C$.

Let E' be the epoch that precedes E. (If there is no such epoch then the lemma trivially holds, since, by the argument above, all the sequence number that q writes to X during E are unique, and there can not be a process p that has read (s, q, v) at Lines 1 and 3 of OP before q writes it at Line 6 of OP$''$.) Let C' be the current interval of E'. Let t' be the time that q reads Ap at Line 13 during E'. By Lemma 36, all the sequence numbers returned by select(q) during $E' \cup E$ belong to $C' \cup C$. Furthermore, it follows by the algorithm that at all times during (t', t), the latest sequence number written into X by q was returned by select(q) during $E' \cup E$. Consequently, at all times during (t', t), if X holds a value of the form $(s', q, *)$, then s' belongs to $C' \cup C$. Since, by Lemma 37, C' and C are disjoint, X does not hold the value (s, q, v) during (t', t). Then, p must have read (s, q, v) at Line 3 of OP before time t'. Consequently, p wrote $(s, q, 0)$ into Ap at Line 2 of OP before time t'. Since OP is p's latest BitPid_LL at time t, it follows that no other BitPid_LL by p writes into Ap after Line 2 of OP and before time t. Consequently, no process $r \neq q$ writes $(*, r, 1)$ into Ap after Line 2 of OP and before t. As a result, (1) Ap holds the value $(s, q, 0)$ at time t', and (2) at all times during (t', t), no process other than q writes into Ap. Let $t'' \in E'$ be the time when p performs the CAS at Line 14. Then, since no other process changes Ap during (t', t), p's CAS at time t'' succeeds, and at all times during (t'', t) Ap holds the value $(s, q, 1)$. Let I be the value of the interval I_p at the end of E'. Then, we have $I = C$. Since $s \in C$, it follows that $s \in I$, which is a contradiction to Lemma 35. □

A.2 Proof of the algorithm in Chapter 5

Let $\mathcal{H}$ be any execution history of Algorithm 5.1. Let OP be some LL operation, OP′ some SC operation, and OP″ some VL operation in $\mathcal{H}$. Then, we define the linearization points for OP, OP′, and OP″ as follows. If the condition at Line 4 of OP fails (i.e., LL(Help$p$) $\not\equiv (0, b)$), we linearize OP at Line 2. If the condition at Line 7 fails (i.e., VL(X) returns *true*), we linearize OP at Line 5. If the condition at Line 7 succeeds, let p be the process executing OP. Then, we show that (1) there exists exactly one SC operation SC_q on $\mathcal{O}$ that writes into Help$p$ during OP, and (2) the VL operation on X at Line 14 of SC_q is executed at some time t during OP; we then linearize OP at time t. We linearize OP′ at Line 19, and OP″ at Line 23.

In the following, we assume that the initializing step was performed by an arbitrary process during its first successful SC operation.

Lemma 38 *Let $SC_0, SC_1, \ldots, SC_K$ be all the successful SC operations in $\mathcal{H}$. Let p_i, for all $i \in \{0, 1, \ldots, K\}$, be the process executing SC_i. Let t_i, for all $i \in \{0, 1, \ldots, K\}$, be the time when p_i executes Line 19 of SC_i. Let LL_i, for all $i \in \{1, 2, \ldots K\}$, be the latest LL operation by p_i prior to SC_i. Let t'_i, for all $i \in \{1, 2, \ldots K\}$, be the latest time during LL_i that p_i performs an LL operation on X. Then, for all $i \in \{1, 2, \ldots K\}$, we have $t'_i > t_{i-1}$.*

Proof. Suppose not. Then, there exists some index j such that $t'_j < t_{j-1}$. (By initialization, we have $j > 1$.) Then, process p_{j-1} performs a successful SC on X (at time t_{j-1}) between p_j's latest LL on X (at time t'_j) and p_j's SC on X (at time t_j). Therefore, p_j's SC on X at time t_j fails, which is a contradiction to the fact that SC_j is successful. □

Lemma 39 *Let $SC_0, SC_1, \ldots, SC_K$ be all the successful SC operations in $\mathcal{H}$. Let p_i, for all $i \in \{0, 1, \ldots, K\}$, be the process executing SC_i. Then, p_i writes the value of the form $(*, i \bmod 2N)$ into* X *at Line 19 of SC_i.*

Proof. Suppose not. Let j be the smallest index such that p_j writes a value different than $(*, j \bmod 2N)$ into X at Line 19 of SC_j. (By initialization, we have $j > 0$.) Let t_{j-1} (respectively, t_j) be the time when p_{j-1} (respectively, p_j) executes Line 19 of SC_{j-1} (respectively, SC_j). Let LL_j be p_j's latest LL operation prior to SC_j, and let t'_j be the latest time during LL_j that p_j performs an LL operation on X. Then, by Lemma 38, we have $t_{j-1} < t'_j < t_j$. Furthermore, by definition of j, p_{j-1} writes $(*, (j-1) \bmod 2N)$ into X at time t_{j-1}. Since X doesn't change during (t_{j-1}, t_j), it follows that p_j reads $(*, (j-1) \bmod 2N)$ from X at time t'_j. Hence, p_j writes $(*, j \bmod 2N)$ into X at Line 19 of SC_j, which is a contradiction. □

Lemma 40 *Let $SC_0, SC_1, \ldots, SC_K$ be all the successful SC operations in $\mathcal{H}$. Let p_i, for all $i \in \{0, 1, \ldots, K\}$, be the process executing SC_i. Let t_i, for all $i \in \{0, 1, \ldots, K\}$, be the time when p_i executes Line 19 of SC_i. Let $(a_i, i \bmod 2N)$, for all $i \in \{0, 1, \ldots, K\}$, be the value that p_i writes into* X *at time t_i. Let t'_i, for all $i \in \{0, 1, \ldots K-1\}$, be the first time during (t_i, t_{i+1}) that some process p that had performed an LL operation on* X *after time t_i begins Line 14. Then, the following holds for all $i \in \{0, 1, \ldots K-1\}$: (1) at all times during (t'_i, t_{i+1}), variable* `Bank`$i \bmod 2N$ *holds value a_i, and (2) variable* `Bank`j*, for $j \neq i \bmod 2N$, is not written during (t_i, t_{i+1}).*

Proof. (By induction) Suppose that the lemma holds for all $i < k$; we now show that the lemma holds for k as well. (During the proof of the inductive step, we will

also prove the base case of $i = 0$.)

Claim 3 *During* (t_k, t_{k+1})*, no process writes into* `Bank`j*, for all* $j \neq k \bmod 2N$.

Proof. Suppose not. Then, there exists some process q and some index $j \neq k \bmod 2N$, such that q writes into `Bank`j during (t_k, t_{k+1}). Let $t \in (t_k, t_{k+1})$ be the time when q performs that write (at Line 13). Let t', t'', and t''' be the latest times prior to t when q performs, respectively, the latest LL on `X` (at Line 2 or 5), the LL on `Bank`j (at Line 12), and the VL on `X` (at Line 12). Notice that, since q writes into `Bank`j, it means that q reads a value $(*, j)$ from `X` at time t'. Since $j \neq k \bmod 2N$, we have $t' \in (t_i, t_{i+1})$ and $j = i \bmod 2N$, for some $i < k$. Moreover, since q satisfies the condition at Line 12, it means that (1) q reads a value different than a_i from `Bank`j at time t'', and (2) q's VL on `X` at time t''' succeeds. Since, by the inductive hypothesis, `Bank`j holds value a_i at all times during (t'_i, t_{i+1}), it follows that $t'' < t'_i$. Furthermore, since q executes Line 13 at time $t \in (t_k, t_{k+1})$, for $k > i$, it follows that $t'_i < t$. Consequently, value a_i is written into `Bank`j during (t'', t), which is a contradiction to the fact that q writes into `Bank`j at time t. □

Claim 4 *During* (t_k, t_{k+1})*, no process writes a value different than* a_k *into* `Bank`$k \bmod 2N$.

Proof. Suppose not. Then, there exists some process q that writes a value different than a_k into `Bank`$k \bmod 2N$ during (t_k, t_{k+1}). Let $t \in (t_k, t_{k+1})$ be the time when q performs that write (at Line 13). Let t', t'', and t''' be the latest times prior to t when q performs, respectively, the latest LL on `X` (at Line 2 or 5), the LL on `Bank`$k \bmod 2N$ (at Line 12), and the VL on `X` (at Line 12). Notice that, since q

writes a value different than a_k into Bank$k$ mod $2N$, it means that q reads a value $(a_i, *)$ from X at time t', for some $i < k$. Therefore, we have $t' \in (t_i, t_{i+1})$, for some $i < k$. Since q satisfies the condition at Line 12, it means that (1) q reads a value different than a_i from Bank$k$ mod $2N$ at time t'', and (2) q's VL on X at time t''' succeeds. Since, by the inductive hypothesis, Bank$k$ mod $2N$ holds value a_i at all times during (t'_i, t_{i+1}), it follows that $t'' < t'_i$. Furthermore, since q executes Line 13 at time $t \in (t_k, t_{k+1})$, for $k > i$, it follows that $t'_i < t$. Consequently, value a_i is written into Bank$k$ mod $2N$ during (t'', t), which is a contradiction to the fact that q writes into Bank$k$ mod $2N$ at time t. □

Claim 5 *At some point during (t_k, t'_k), variable* Bank$k$ mod $2N$ *holds value a_k.*

Proof. Suppose not. Then, at all times during (t_k, t'_k), variable Bank$k$ mod $2N$ holds a value different than a_k. Since, by Lemma 4, no process writes a value different than a_k into Bank$k$ mod $2N$ during (t_k, t_{k+1}), it means that Bank$k$ mod $2N$ doesn't change during (t_k, t'_k). Hence, p reads a value different than a_k from Bank$k$ mod $2N$ at Line 12, its VL operation on X at Line 12 succeeds, and it performs a successful SC on Bank$k$ mod $2N$ at Line 13. Since p reads a_k from X during its latest LL operation (by definition of p), it follows that p writes a_k into Bank$k$ mod $2N$ at Line 13, which is a contradiction. □

The lemma follows immediately from Claims 4 and 5. □

Lemma 41 *Let p be some process, and LL_p some LL operations by p in $\mathcal{H}$. Let t be the time when p executes Line 1 of LL_p, and t' the time just prior to Line 10 of LL_p. Let t'' be either (1) the moment when p executes Line 1 of its first LL*

operation after LL_p, if such operation exists, or (2) the end of $\mathcal{H}$, otherwise. Then, the following statements hold:

(S1) During the time interval (t, t'), exactly one *write into `Helpp` is performed.*

(S2) Any value written into `Helpp` during (t, t'') is of the form $(0, *)$.

(S3) Let $t''' \in (t, t')$ be the time when the write from statement (S1) takes place. Then, during the time interval (t''', t''), no process writes into `Helpp`.

Proof. Statement (S2) follows trivially from the fact that the only two operations that can affect the value of `Helpp` during (t, t'') are (1) the SC at Line 9 of LL_p, and (2) the SC at Line 15 of some other process's SC operation, both of which attempt to write $(0, *)$ into `Helpp`.

We now prove statement (S1). Suppose that (S1) does not hold. Then, during (t, t'), either (1) two or more writes on `Helpp` are performed, or (2) no writes on `Helpp` are performed. In the first case, we know (by an earlier argument) that each write on `Helpp` during (t, t') must have been performed either by the SC at Line 9 of LL_p, or by the SC at Line 15 of some other process's SC operation. Let SC_1 and SC_2 be the first two SC operations on `Helpp` to write into `Helpp` during (t, t'). Let q_1 (respectively, q_2) be the process executing SC_1 (respectively, SC_2). Let LL_1 (respectively, LL_2) be the latest LL operations on `Helpp` by q_1 (respectively, q_2) to precede SC_1 (respectively, SC_2). Then, both LL_1 and LL_2 return a value of the form $(1, *)$. Furthermore, LL_2 takes place after SC_1, or else SC_2 would fail. Since `Helpp` doesn't change between SC_1 and SC_2, it means that LL_2 returns the value of the form $(0, *)$, which is a contradiction.

In the second case (where no writes on `Helpp` take place during (t, t')), the LL operation at Line 8 of LL_p returns a value of the form $(1, *)$. Furthermore,

the SC at Line 9 of LL_p must succeed, which is a contradiction to the fact that no writes into `Helpp` take place during (t, t'). Hence, statement (S1) holds.

We now prove statement (S3). Suppose that (S3) does not hold. Then, at least one write on `Helpp` takes place during (t''', t''). By an earlier argument, any write on `Helpp` during (t''', t'') must have been performed either by the SC at Line 9 of LL_p, or by the SC at Line 15 of some other process's SC operation. Let SC_3 be the first SC operations on `Helpp` to write into `Helpp` during (t''', t''). Let q_3 be the process executing SC_3. Let LL_3 be the latest LL operations on `Helpp` by q_3 to precede SC_3. Then, LL_3 returns a value of the form $(1, *)$. Furthermore, LL_3 must take place after time t''', or else SC_3 would fail. Since `Helpp` doesn't change between time t''' and SC_3, it means that LL_3 returns the value of the form $(0, *)$, which is a contradiction. Hence, we have statement (S3). □

In Figure A.1, we present a number of invariants satisfied by the algorithm. In the following, we let $PC(p)$ denote the value of process p's program counter. For any register r at process p, we let $r(p)$ denote the value of that register. We let $\mathcal{P}$ denote a set of processes such that $p \in \mathcal{P}$ if and only if $PC(p) \in \{1, 11-15, 17-19, 21-23\}$ or $PC(p) \in \{2-9\} \wedge$ `Helpp` $\equiv (1, *)$. We let $\mathcal{P}'$ denote a set of processes such that $p \in \mathcal{P}'$ if and only if $PC(p) \in \{2-10\} \wedge$ `Helpp` $\equiv (0, *)$. We let $\mathcal{P}''$ denote a set of processes such that $p \in \mathcal{P}''$ if and only if $PC(p) = 16$. We let $\mathcal{P}'''$ denote a set of processes such that $p \in \mathcal{P}'''$ if and only if $PC(p) = 20$.

Lemma 42 *The algorithm satisfies the invariants in Figure A.1.*

Proof. (By induction) For the base case, (i.e., $t = 0$), all the invariants hold by initialization. The inductive hypothesis states that the invariants hold at time $t \geq 0$.

1. For any processes $p \in \mathcal{P}$, we have $mybyf_p \in 0\,..\,3N-1$.
2. For any process $p \in \mathcal{P}'$, we have $\texttt{Help}p.buf \in 0\,..\,3N-1$.
3. For any process $p \in \mathcal{P}''$, we have $d(p) \in 0\,..\,3N-1$.
4. For any process $p \in \mathcal{P}'''$, we have $e(p) \in 0\,..\,3N-1$.
5. $\texttt{X}.buf \in 0\,..\,3N-1$.
6. Let $(*, k)$ be the value of X. Then, for all $j \neq k$, $\texttt{Bank}j \in 0\,..\,3N-1$.
7. Let p and q (respectively, p' and q', p'' and q'', p''' and q'''), be any two processes in $\mathcal{P}$ (respectively, $\mathcal{P}'$, $\mathcal{P}''$, $\mathcal{P}'''$). Let $(*, k)$ be the value of X. Let i and j be any two indices different than k. Then, we have $mybuf_p \neq mybuf_q \neq \texttt{Bank}i \neq \texttt{Bank}j \neq \texttt{X}.buf \neq \texttt{Help}p'.buf \neq \texttt{Help}q'.buf \neq d(p'') \neq d(q'') \neq e(p''') \neq e(q''')$.

Figure A.1: The invariants satisfied by Algorithm 5.1

Let t' be the earliest time after t that some process, say p, makes a step. Then, we show that the invariants holds at time t' as well.

First, notice that if $PC(p) = \{1{-}8, 11, 12, 14, 17, 18, 21{-}23\}$, or if $PC(p) = \{9, 13, 15, 19\}$ and p's SC fails, then none of the invariants are affected by p's step and hence they hold at time t' as well.

If $PC(p) = 9$ and p's SC succeeds, then p moves from $\mathcal{P}$ to $\mathcal{P}'$ and writes $mybuf_p$ into $\texttt{Help}p.buf$. Consequently, invariant 2 holds by IH:1 and invariant 7 by IH:7. All other invariants trivially hold.

If $PC(p) = 10$, then, by Lemma 41, p was in $\mathcal{P}'$ at time t. Furthermore, p is in $\mathcal{P}$ at time t'. Since p writes $\texttt{Help}p.buf$ into $mybuf_p$, invariant 1 holds by IH:2 and invariant 7 by IH:7. All other invariants trivially hold.

If $PC(p) = 12$ and p's SC succeeds, let $(*, k)$ be the value of X at time t. Then, by Lemma 40, p's SC writes into variable $\texttt{Bank}k$. Hence, none of the invariants

are affected by p's step, and so they hold at time t' as well.

If $PC(p) = 15$ and p's SC succeeds, then p moves from $\mathcal{P}$ to $\mathcal{P}''$. Let `Help`q be the variable that p writes into during this step. Then, we have $d(p) =$ `Help`q*.buf* at time t, `Help`q*.buf* $=$ *mybuf*$_p$ at time t', and `Help`q changes from value $(1, *)$ at time t to a value $(0, *)$ at time t'. Therefore, by Lemma 41, we have $PC(q) \in \{2 - 10\}$ at times t and t', and `Help`q*.buf* $=$ *mybuf*$_q$. Hence, q moves from $\mathcal{P}$ to $\mathcal{P}'$, and $d(p) =$ *mybuf*$_q$. Consequently, invariant 2 holds by IH:1, invariant 3 by IH:1, and invariant 7 by IH:7. All other invariants trivially hold.

If $PC(p) = 16$, then p moves from $\mathcal{P}''$ to $\mathcal{P}$ and writes $d(p)$ into *mybuf*$_p$. Then, invariant 1 holds by IH:3 and invariant 7 by IH:7. All other invariants trivially hold.

If $PC(p) = 19$ and p's SC succeeds, then p moves from $\mathcal{P}$ to $\mathcal{P}'''$. Let (b, k) be the value of variable `X` at time t. Then, by Lemma 39 and the algorithm, the value of `X` at time t' is (*mybuf*$_p$, $(k + 1) \bmod 2N$). Furthermore, by Lemma 40, variable `Bank`k holds value b at time t'. Finally, by Lemma 40, $e(p)$ has the value that `Bank`$(k + 1) \bmod 2N$ holds at time t. Consequently, invariant 5 holds by IH:1, invariant 6 by IH:5, invariant 4 by IH:6, and invariant 7 by IH:7. All other invariants trivially hold.

If $PC(p) = 20$, then p moves from $\mathcal{P}'''$ to $\mathcal{P}$ and writes $e(p)$ into *mybuf*$_p$. Then, invariant 1 holds by IH:4, and invariant 7 by IH:7. All other invariants trivially hold. □

Lemma 43 *Let p be some process, and SC_p some successful SC operation by p in $\mathcal{H}$. Let v be the value that SC_p writes in $\mathcal{O}$. Let (b, i) be the value that p writes into `X` at Line 19 of SC_p. Then, `BUF`b holds the value v until `X` changes at least*

$2N$ *times.*

Proof. Notice that, by the algorithm, the only places where BUFb can be modified is either at Line 11 of some LL operation, or at Line 17 of some SC operation. Let t be the time when p writes v into BUFb at Line 17 of SC_p. Let t' be the time when p writes (b, i) into X at Line 19 of SC_p. Let t'' be the first time after t that X changes. Let t''' be the $2N$th time after t that X changes. Then, by Invariant 7, no process q can be at Line 11 or 17 with $mybuf_q = b$ during (t, t'). Similarly, no process q can be at Line 11 or 17 with $mybuf_q = b$ during (t', t''). Notice that, by Lemma 40, we have Bank$i = b$ at time t''. Furthermore, variable Bank$i$ holds value b at all times during (t'', t'''). Hence, by Invariant 7, no process q can be at Line 11 or 17 with $mybuf_q = b$ during (t'', t'''). Consequently, no process writes into BUFb during (t, t'''), which proves the lemma. □

Lemma 44 *Let p be some process, and LL_p some LL operation by p in $\mathcal{H}$. Let t be the time when p executes Line 2 of LL_p, and t' the time when p executes Line 4 of LL_p. If the condition at Line 4 of LL_p fails (i.e., LL(Help$p$) $\not\equiv (0, b)$), then X changes at most $2N - 1$ times during (t, t').*

Proof. Suppose not. Then, the condition at Line 4 of LL_p fails and X changes $2N$ or more times during (t, t'). Let $t'' \in (t, t')$ be the $2N$th time after t that X changes. Let (b, i) be the value that p reads from X at time t. Since the condition at Line 4 of LL_p fails, it means that Help$p$ holds the value $(1, a)$ at all times during (t, t'), for some a. Notice that, by Lemma 39, there exist two successful SC operations SC_1 and SC_2 on X (at Line 19) such that (1) SC_1 writes the value of the form $(*, s)$ into X at some time $t_1 \in (t, t'')$, for some $s \bmod 2N = p$, (2) SC_2

writes the value of the form $(*, (s+1) \bmod 2N)$ into X at some time $t_2 \in (t_1, t'')$, and (3) SC_2 is the first SC operation to write into X after t_1. Let p_2 be the process executing SC_2, LL_2 the latest LL operation on X by p_2 prior to SC_2, and SC_{p_2} the SC operation on $\mathcal{O}$ by p_2 during which SC_2 is executed. Then, by Lemma 38, LL_2 is executed during (t_1, t_2) and returns the value of the form $(*, s)$. Hence, at Line 14 of SC_{p_2}, p_2 performs an LL operation on Help$p$. Since Help$p$ holds the value $(1, a)$ at all times during (t, t'), p_2's LL on Help$p$ must return the value $(1, a)$. Furthermore, since SC_2 succeeds, the VL operation at Line 14 of SC_{p_2} succeeds as well. Therefore, p_2 executes the SC operation at Line 15 of SC_{p_2}. Since Help$p$ doesn't change during (t', t''), it also doesn't change between the time p_2 performs the LL of Help$p$ at Line 14 of SC_{p_2}, and the time p_2 performs the SC of Help$p$ at Line 15 of SC_{p_2}. Consequently, p_2's SC at Line 15 succeeds, writing the value of the form $(0, *)$ into Help$p$, which is a contradiction to the fact that Help$p$ doesn't change during (t, t'). □

Lemma 45 *Let p be some process, and LL_p some LL operation by p in $\mathcal{H}$. Let t be the time when p executes Line 2 of LL_p, and t' the time when p executes Line 4 of LL_p. If the condition at Line 4 of LL_p fails (i.e., LL(Help$p$) $\not\equiv (0, b)$), then the value that p writes into* retval *at Line 3 of LL_p is the value of $\mathcal{O}$ at time t.*

Proof. Let (b, i) be the value that p reads from X at time t. Let SC_q be the SC operation on $\mathcal{O}$ that wrote that value into X, and q the process that executed SC_q. Let $t'' < t$ be the time during SC_q when q wrote (b, i) into X, and v the value that SC_q writes in $\mathcal{O}$. Then, by Lemma 43, BUFb will hold the value v until X changes at least $2N$ times after t''. Since X doesn't change during (t'', t), it means that BUFb will hold the value v until X changes at least $2N$ times after t. Notice

that, by Lemma 44, X can change at most $2N - 1$ times during (t, t'). Therefore, BUFb holds the value v at all times during (t, t'), and hence the value that p writes into *retval* at Line 3 of LL_p is the value of $\mathcal{O}$ at time t. □

Lemma 46 *Let p be some process, and LL_p some LL operation by p in $\mathcal{H}$. Let t be the time when p executes Line 5 of LL_p, and t' the time when p executes Line 7 of LL_p. If the condition at Line 7 of LL_p fails (i.e., VL(X) returns* true*), then the value that p writes into* retval *at Line 6 of LL_p is the value of $\mathcal{O}$ at time t.*

Proof. Let (b, i) be the value that p reads from X at time t. Let SC_q be the SC operation on $\mathcal{O}$ that wrote that value into X, and q the process that executed SC_q. Let $t'' < t$ be the time during SC_q when q wrote (b, i) into X, and v the value that SC_q writes in $\mathcal{O}$. Then, by Lemma 43, BUFb will hold the value v until X changes at least $2N$ times after t''. Since X doesn't change during (t'', t), it means that BUFb will hold the value v until X changes at least $2N$ times after t. Because p's VL operation on X at Line 7 of LL_p returns *true* at time t', it means that X doesn't change during (t, t'). Therefore, BUFb holds the value v at all times during (t, t'), and hence the value that p writes into *retval* at Line 6 of LL_p is the value of $\mathcal{O}$ at time t. □

Lemma 47 *Let p be some process, and LL_p some LL operation by p in $\mathcal{H}$. Let t be the time when p executes Line 1 of LL_p, and t' the time when p executes Line 4 of LL_p. If the condition at Line 4 of LL_p succeeds (i.e., LL(Help$p$) $\equiv (0, b)$), then (1) there exists exactly one SC operation SC_q on $\mathcal{O}$ that writes into Help$p$ during (t, t'), and (2) the VL operation on X at Line 14 of SC_q is executed during (t, t').*

Proof. Since the condition at Line 4 of LL_p succeeds, it means that some SC operation SC_q writes the value of the form $(0, *)$ into `Help`p during (t, t'). By Lemma 41, SC_q is the only SC operation that writes into `Help`p during (t, t'). Let $t'' \in (t, t')$ be the time when SC_q writes into `Help`p. Let q be the process executing SC_q. Since q writes into `Help`p at time t'', it means that `Help`p does not change between q's LL at Line 14 of SC_q and t''. Therefore, q's LL at Line 14 of SC_q occurs during the time interval (t, t''). Consequently, q's VL at Line 14 of SC_q occurs during the time interval (t, t'') as well. □

Lemma 48 *Let p be some process, and LL_p some LL operation by p in $\mathcal{H}$. Let t be the time when p executes Line 1 of LL_p, and t' the time when p executes Line 4 of LL_p. If the condition at Line 7 of LL_p succeeds (i.e., `VL`(X) returns* false*), let SC_q be the SC operation on $\mathcal{O}$ that writes into `Help`p during (t, t'), and let $t'' \in (t, t')$ be the time when the VL operation on `X` at Line 14 of SC_q is performed. Then, the value that LL_p returns is the value of $\mathcal{O}$ at time t''.*

Proof. Let q be the process executing SC_q. Let LL_q be q's latest LL operation on $\mathcal{O}$ before SC_q. Since the VL operation on `X` at Line 14 of SC_q succeeds, it means that either the condition at Line 7 of LL_q failed, or that Line 7 of LL_q was never executed. In the first case, let t_q be the time when q executes Line 5 of LL_q. In the second case, let t_q be the time when q executes Line 2 of LL_q. In either case, by Lemmas 45 and 46, LL_q returns the value of $\mathcal{O}$ at time t_q. Let v be the value returned by LL_q. Since the VL operation on `X` at Line 14 of SC_q succeeds, it means that v is the value of $\mathcal{O}$ at time t'' as well.

Let t'_q be the time just before q starts executing Line 11 of LL_q. Let t''_q be the time when q executes the SC operation on `Help`p at Line 15 of SC_q. Let b be the

value of $mybuf_q$ at time t'_q. Notice that, by the algorithm, the only places where BUFb can be modified is either at Line 11 of some LL operation, or at Line 17 of some SC operation. By Invariant 7, we know that during (t'_q, t''_q), no process $r \neq q$ can be at Line 11 or 17 with $mybuf_r = b$. Therefore, BUFb holds the value v at all times during (t'_q, t''_q). Since $mybuf_q$ doesn't change during (t'_q, t''_q) as well, it means that q writes $(0, b)$ into Help$p$ at time $t''_q \in (t, t')$. Because, by Lemma 41, no other process writes into Help$p$ during (t, t'), it means that p reads b at Line 4 of LL_p (at time t'). Let t''' be the time when p executes Line 7 of LL_p. Then, by Invariant 7, we know that during (t''_q, t''') no process r can be at Line 11 or 17 with $mybuf_r = b$. Therefore, BUFb holds the value v at all times during (t''_q, t'''). So, at Line 6 of LL_p, p writes into *retval* the value v, which is the value of $\mathcal{O}$ at time t''.
□

Lemma 49 (Correctness of LL) *Let p be some process, and LL_p some LL operation by p in $\mathcal{H}$. Let* LP(LL_p) *be the linearization point for LL_p. Then, LL_p returns the value of $\mathcal{O}$ at $LP(LL_p)$.*

Proof. This lemma follows immediately from Lemmas 45, 46, and 48. □

Lemma 50 (Correctness of SC) *Let p be some process, and SC_p some SC operation by p in $\mathcal{H}$. Let LL_p the latest LL operation by p to precede SC_p. Then, SC_p succeeds if and only if there does not exist any other successful SC operation SC' such that $LP(LL_p) < LP(SC') < LP(SC_p)$.*

Proof. If SC_p succeeds, then the SC on X at Line 19 of SC_p succeeds. Hence, LP(LL_p) is either at Line 2 of LL_p or at Line 5 of LL_p. In either case, X doesn't

change during between $LP(LL_p)$ and Line 19 of SC_p. Since we linearize all SC operations at Line 19, it follows that there does not exist any successful SC operation SC' such that $LP(LL_p) < LP(SC') < LP(SC_p)$. Hence, SC_p was correct in returning *true*.

If SC_p fails, we examine the following three possibilities: (1) LL_p is linearized at Line 2, (2) LL_p is linearized at Line 5, and (3) LL_p is linearized at some point between Lines 2 and 4 (the third linearization case). In the first case, since SC_p fails, variable X changes between Line 2 of LL_p and Line 19 of SC_p. Since we linearize all SC operations at Line 19, it follows that there exists some successful SC operation SC' such that $LP(LL_p) < LP(SC') < LP(SC_p)$. Hence, SC_p was correct in returning *false*. The proof for the second case is identical, and is therefore omitted.

In the third case, the VL operation at Line 7 of LL_p fails. Hence, variable X changes between Lines 5 and 7 of LL_p. Since we linearize all SC operations at Line 19, it follows that there exists some successful SC operation SC' such that $LP(LL_p) < LP(SC') < LP(SC_p)$. Hence, SC_p was correct in returning *false*. □

Lemma 51 (Correctness of VL) *Let p be some process, and VL_p some VL operation by p in $\mathcal{H}$. Let LL_p the latest LL operation by p to precede VL_p. Then, VL_p succeeds if and only if there does not exist some successful SC operation SC' such that $LP(LL_p) < LP(SC') < LP(VL_p)$.*

Proof. Similar to the proof of Lemma 51. □

Theorem 6 *Algorithm 5.1 is wait-free and implements a linearizable N-process W-word LL/SC object $\mathcal{O}$ from small LL/SC objects and registers. The time complexity of LL, SC, and VL operations on $\mathcal{O}$ are $O(W)$, $O(W)$ and $O(1)$, respectively. The implementation requires $O(NW)$ registers and $3N + 1$ small LL/SC objects.*

Proof. This theorem follows immediately from Lemmas 49, 50, and 51. □

A.3 Proof of the algorithms in Chapter 6

A.3.1 Proof of Algorithm 6.1

Let $\mathcal{H}$ be any execution history of Algorithm 6.1. Let OP be some WLL operation, OP′ some SC operation, and OP″ some VL operation on $\mathcal{O}$ in $\mathcal{H}$. Then, we set the linearization points for OP, OP′, and OP″ at Line 1 (of OP), Line 9 (of OP′), and Line 6 (of OP″), respectively.

In Figure A.2, we present a number of invariants satisfied by the algorithm. In the following, we let $PC(p)$ denote the value of process p's program counter. For any register r at process p, we let $r(p)$ denote the value of that register. We let $\mathcal{P}$ denote a set of processes such that $p \in \mathcal{P}$ if and only if $PC(p) \in \{1-9, 11, 12\}$. We let $\mathcal{P}'$ denote a set of processes such that $p \in \mathcal{P}'$ if and only if $PC(p) = 10$.

Lemma 52 *The algorithm satisfies the invariants in Figure A.2.*

Proof. (By induction) For the base case, (i.e., $t = 0$), all the invariants hold by initialization. The inductive hypothesis states that the invariants hold at time $t \geq 0$.

1. For any processes $p \in \mathcal{P}$, we have $mybuf_p \in 0\,..\,M+N-1$.
2. For any process $p \in \mathcal{P}'$, we have $x(p).buf \in 0\,..\,M+N-1$.
3. For any index $i \in 0\,..\,M-1$, we have $\mathtt{X}i.buf \in 0\,..\,M+N-1$.
4. Let p and q (respectively, p' and q'), be any two processes in $\mathcal{P}$ (respectively, $\mathcal{P}'$). Let i and j be any two indices in $0\,..\,M-1$. Then, we have $mybuf_p \neq mybuf_q \neq \mathtt{X}i.buf \neq \mathtt{X}j.buf \neq x(p').buf \neq x(q').buf$.

Figure A.2: The invariants satisfied by Algorithm 6.1

Let t' be the earliest time after t that some process, say p, makes a step. Then, we show that the invariants holds at time t' as well.

First, notice that if $PC(p) = \{1-8, 11, 12\}$, or if $PC(p) = 9$ and p's SC fails, then none of the invariants are affected by p's step and hence they hold at time t' as well.

If $PC(p) = 9$ and p's SC succeeds, then p moves from $\mathcal{P}$ to $\mathcal{P}'$. Let $\mathtt{X}i$ be the variable that p writes to. Then, since p's SC is successful, we have $\mathtt{X}i.buf = x(p).buf$ at time t, and $\mathtt{X}i.buf = mybuf_p$ at time t'. Consequently, invariant 3 holds by IH:1, invariant 2 by IH:3, and invariant 4 by IH:4. All other invariants trivially hold.

If $PC(p) = 10$, then p moves from $\mathcal{P}'$ to $\mathcal{P}$ and writes $x(p).buf$ into $mybuf_p$. Then, invariant 1 holds by IH:2 and invariant 4 by IH:4. All other invariants trivially hold. □

Lemma 53 *Let $\mathcal{O}i$ be some Weak-LL/SC object in the array $\mathcal{O}0\,..\,M-1$. Let* OP *be some WLL operation on $\mathcal{O}i$, and* OP′ *be the latest successful SC operation on*

$\mathcal{O}i$ to execute Line 9 prior to Line 1 of OP. *If the VL operation at Line 3 of* OP *returns true, then at the end of* OP*, retval holds the value written by* OP′.

Proof. Let v be the value that OP′ writes in $\mathcal{O}i$. Let p be the process executing OP, and q be the process executing OP′. Let t_1 be the time when p executes Line 1 of OP, t_2 the time when p starts executing Line 2 of OP, and t_3 the time when p completes executing Line 2 of OP. Since OP′ is the latest successful SC operation on $\mathcal{O}i$ to execute Line 9 prior to Line 1 of OP, it follows that p reads from Xi at time t_1 the value that q writes in Xi at Line 9 of OP′. Therefore, p reads during (t_2, t_3) the same buffer B that q wrote v into at Line 8 of OP′. Let t_4 be the time when q starts writing into B at Line 8 of OP′, t_5 the time when q completes writing into B at Line 8 of OP′, and t_6 the time when q writes into Xi at Line 9 of OP′. Then, the following claim holds.

Claim 6 *During (t_4, t_5), no process other than q writes into B. During (t_5, t_3), no process writes into B.*

Proof. Suppose not. Then, either some process other than q writes into B during (t_4, t_5), or some process writes into B during (t_5, t_3). In the first case, let r_1 be the process that writes into B during (t_4, t_5). Then, at some point during (t_4, t_5), we have $mybuf_{r_1} = mybuf_q$, which is a contradiction to Invariant 4. In the second case, let r_2 be the first process to start writing into B at some time $\tau_1 \in (t_5, t_3)$ and k be the index of buffer B. Then, by the earlier argument, $\tau_1 \notin (t_5, t_6)$. Furthermore, by Invariant 4, r_2 does not write into B as long as Xi holds value $(*, k)$. Since Xi holds value $(*, k)$ at time t_6 and doesn't change during (t_6, t_1) nor during (t_1, t_3), it means that $\tau_1 > t_3$. This, however, is a contradiction to the fact that $\tau_1 \in (t_5, t_3)$. Hence, we have the claim. □

The above claim shows that (1) during (t_4, t_5), no process other than q writes into B, and (2) during (t_5, t_3), no process writes into B. Consequently, p reads v from B during (t_2, t_3), which proves the lemma. □

Lemma 54 (Correctness of WLL) *Let $\mathcal{O}i$ be any Weak-LL/SC object in the array $\mathcal{O}0\,..\,M-1$. Let* OP *be any WLL operation on $\mathcal{O}i$, and* OP′ *be the latest successful SC operation on $\mathcal{O}i$ such that LP(*OP′*) < LP(*OP*). If* OP *returns "success", then retval contains the value written by* OP′*. If* OP *returns* $(failure, q)$*, then there exists a successful SC operation* OP″ *by process q such that: (1) LP(*OP″*) lies in the execution interval of* OP*, and (2) LP(*OP*) < LP(*OP″*).*

Proof. Let p be the process executing OP. If OP returns *success* (at Line 3), let SC_q be the latest successful SC operation on $\mathcal{O}i$ to execute Line 9 prior to Line 1 of OP, and v_q be the value that SC_q writes in $\mathcal{O}i$. Since all SC operations are linearized at Line 9 and since OP is linearized at Line 1, we have $\mathrm{SC}_q = \mathrm{OP}'$. Furthermore, by Lemma 53, *retval* contains value v_q. Therefore, the lemma holds in this case.

If OP returns $(failure, q)$ (at Line 5), then p reads $(q, *)$ from Xi at Line 4 of OP. Let SC'_q be the successful SC operation by q that wrote that value into Xi. Since the VL at Line 3 of OP fails, it means that Xi changes between Lines 1 and 3 of OP. Therefore, SC'_q must have written $(q, *)$ into Xi after Line 1 of OP (and before Line 4 of OP). Since SC'_q is linearized at Line 9 and since OP is linearized at Line 1, it follows that (1) $\mathrm{LP}(\mathrm{SC}'_q)$ lies in the execution interval of OP, and (2) $\mathrm{LP}(\mathrm{OP}) < \mathrm{LP}(\mathrm{SC}'_q)$, which proves the lemma. □

Lemma 55 (Correctness of SC) *Let $\mathcal{O}i$ be any Weak-LL/SC object in the array $\mathcal{O}0\,..\,M-1$. Let* OP *be any SC operation on $\mathcal{O}i$ by some process p, and* OP′ *be*

the latest WLL operation on $\mathcal{O}i$ by p prior to OP. *Then,* OP *succeeds if and only if there does not exist any successful SC operation* OP$''$ *on $\mathcal{O}i$ such that LP(*OP$'$*) < LP(*OP$''$*) < LP(*OP*).*

Proof. If OP succeeds, then between Line 1 of OP$'$ and Line 9 of OP, variable Xi does not change. Since all SC operations are linearized at Line 9 and since OP$'$ is linearized at Line 1, it follows that there does not exist any successful SC operation OP$''$ on $\mathcal{O}i$ such that LP(OP$'$) < LP(OP$''$) < LP(OP).

If OP fails, then variable Xi changes between Line 1 of OP$'$ and Line 9 of OP. Since all SC operations are linearized at Line 9 and since OP$'$ is linearized at Line 1, it follows that there exists some successful SC operation OP$''$ on $\mathcal{O}i$ such that LP(OP$'$) < LP(OP$''$) < LP(OP). Therefore, we have the lemma. □

Lemma 56 (Correctness of VL) *Let $\mathcal{O}i$ be any Weak-LL/SC object in the array $\mathcal{O}0\,..\,M-1$. Let* OP *be any VL operation on $\mathcal{O}i$ by some process p, and* OP$'$ *be the latest WLL operation on $\mathcal{O}i$ by p that precedes* OP. *Then,* OP *returns true if and only if there does not exist some successful SC operation* OP$''$ *on $\mathcal{O}i$ such that LP(*OP$'$*) < LP(*OP$''$*) < LP(*OP*).*

Proof. Similar to the proof of Lemma 55. □

Theorem 8 *Algorithm 6.1 is wait-free and implements an array $\mathcal{O}0\,..\,M-1$ of M N-process W-word Weak-LL/SC objects. The time complexity of LL, SC, and VL operations on $\mathcal{O}$ are $O(W)$, $O(W)$ and $O(1)$, respectively. The implementation requires $O((N+M)W)$ registers and M small LL/SC objects.*

Proof. This theorem follows immediately from Lemmas 54, 55, and 56. □

A.3.2 Proof of Algorithm 6.2

Let $\mathcal{H}$ be any execution history of Algorithm 6.2. Let OP be some LL operation, OP$'$ some SC operation, and OP$''$ some VL operation on $\mathcal{O}i$ in $\mathcal{H}$, for some i. Then, we define the linearization points for OP, OP$'$, and OP$''$ as follows. If the CAS at Line 5 of OP succeeds, then LP(OP) is Line 3 of OP. Otherwise, let t be the time when OP executes Line 2, and t' be the time when OP performs the CAS at Line 5. Let v be the value that OP reads from `BUF` at Line 8 of OP. Then, we show that there exists a successful SC operation SC_q on $\mathcal{O}i$ such that (1) at some point t'' during (t, t'), SC_q is the latest successful SC on $\mathcal{O}i$ to execute Line 12, and (2) SC_q writes v into $\mathcal{O}i$. We then set LP(OP) to time t''. We set LP(OP$'$) to Line 12 of OP$'$, and LP(OP$''$) to Line 10 of OP$''$.

Lemma 57 *Let p be some process, and LL_p some LL operation by p in $\mathcal{H}$. Let t and t' be the times when p executes Line 2 and Line 5 of LL_p, respectively. Let t'' be either (1) the time when p executes Line 2 of its first LL operation after LL_p, if such operation exists, or (2) the end of $\mathcal{H}$, otherwise. Then, the following statements hold:*

(S1) During the time interval $(t, t'$, exactly one *write into* `Help`p *is performed.*

(S2) Any value written into `Help`p *during (t, t'') is of the form $(*, 0, *)$.*

(S3) Let $t''' \in (t, t'$ be the time when the write from statement (S1) takes place. Then, during the time interval (t''', t''), no process writes into `Help`p.

Proof. Statement (S2) follows trivially from the fact that the only two operations that can affect the value of `Help`p during (t, t'') are (1) the CAS at Line 5 of LL_p,

and (2) the CAS at Line 21 of some other process' SC operation, both of which attempt to write $(*, 0, *)$ into `Help`p.

We now prove statement (S1). Suppose that (S1) does not hold. Then, during $(t, t'$, either (1) two or more writes on `Help`p are performed, or (2) no writes on `Help`p are performed. In the first case, we know (by an earlier argument) that each write on `Help`p during $(t, t'$ is performed either by the CAS at Line 5 of LL_p, or by the CAS at Line 21 of some other process' SC operation. Let CAS_1 and CAS_2 be the first two CAS operations on `Help`p to write into `Help`p during $(t, t'$. Then, by the algorithm, both CAS_1 and CAS_2 are of the form CAS(`Help`p, $(*, 1, *)$, $(*, 0, *)$). Since CAS_1 succeeds and `Help`p doesn't change between CAS_1 and CAS_2, it follows that CAS_2 fails, which is a contradiction.

In the second case (where no writes on `Help`p take place during $(t, t'$), `Help`p doesn't change throughout $(t, t'$. Therefore, p's CAS at Line 5 of LL_p succeeds, which is a contradiction to the fact that no writes on `Help`p take place during $(t, t'$. Hence, statement (S1) holds.

We now prove statement (S3). Suppose that (S3) does not hold. Then, at least one write on `Help`p takes place during (t''', t''). By an earlier argument, any write on `Help`p during (t''', t'') is performed either by the CAS at Line 5 of LL_p, or by the CAS at Line 21 of some other process' SC operation. Let CAS_3 be the first CAS operation on `Help`p to write into `Help`p during (t''', t''). Then, by the algorithm, CAS_3 is of the form CAS(`Help`p, $(*, 1, *)$, $(*, 0, *)$). Since `Help`p holds the value $(*, 0, *)$ at time t''' (by (S2)), and since `Help`p doesn't change between time t''' and CAS_3, it follows that CAS_3 fails, which is a contradiction. Hence, we have statement (S3). □

1. For any process p, we have $|Q_p| \geq N$.
2. For any process p such that $PC(p) = 15$, we have $|Q_p| \geq N + 1$.
3. For any process p and any value b in Q_p, we have $b \in 0\,..\,M+(N+1)N-1$.
4. For any processes $p \in \mathcal{P}$, we have $mybuf_p \in 0\,..\,M + (N + 1)N - 1$.
5. For any process $p \in \mathcal{P}'$, we have $\texttt{Help}p.buf \in 0\,..\,M + (N + 1)N - 1$.
6. For any process $p \in \mathcal{P}''$, we have $x_p.buf \in 0\,..\,M + (N + 1)N - 1$.
7. For any process $p \in \mathcal{P}'''$, we have $b(p) \in 0\,..\,M + (N + 1)N - 1$.
8. For any index $i \in 0\,..\,M - 1$, we have $\texttt{X}i.buf \in 0\,..\,M + (N + 1)N - 1$.
9. Let p and q (respectively, p' and q', p'' and q'', p''' and q'''), be any two processes in $\mathcal{P}$ (respectively, $\mathcal{P}'$, $\mathcal{P}''$, $\mathcal{P}'''$). Let r be any process and b_1 and b_2 any two values in Q_r. Let i and j be any two indices in $0\,..\,M - 1$. Then, we have $mybuf_p \neq mybuf_q \neq b_1 \neq b_2 \neq \texttt{X}i.buf \neq \texttt{X}j.buf \neq \texttt{Help}p'.buf \neq \texttt{Help}q'.buf \neq x_{p''}.buf \neq x_{q''}.buf \neq b(p''') \neq b(q''')$.

Figure A.3: The invariants satisfied by Algorithm 6.2

In Figure A.3, we present a number of invariants satisfied by the algorithm. In the following, we let $PC(p)$ denote the value of process p's program counter. For any register r at process p, we let $r(p)$ denote the value of that register. We let $\mathcal{P}$ denote a set of processes such that $p \in \mathcal{P}$ if and only if $PC(p) \in \{1, 2, 7-13, 16-21, 23, 24\}$ or $PC(p) \in \{3 - 5\} \wedge \texttt{Help}p \equiv (*, 1, *)$. We let $\mathcal{P}'$ denote a set of processes such that $p \in \mathcal{P}'$ if and only if $PC(p) \in \{3 - 6\} \wedge \texttt{Help}p \equiv (*, 0, *)$. We let $\mathcal{P}''$ denote a set of processes such that $p \in \mathcal{P}''$ if and only if $PC(p) = 14$. We let $\mathcal{P}'''$ denote a set of processes such that $p \in \mathcal{P}'''$ if and only if $PC(p) = 22$. Finally, we let $|Q_p|$ denote the length of process p's local queue Q_p.

Lemma 58 *The algorithm satisfies the invariants in Figure A.3.*

Proof. (By induction) For the base case, (i.e., $t = 0$), all the invariants hold by initialization. The inductive hypothesis states that the invariants hold at time $t \geq 0$. Let t' be the earliest time after t that some process, say p, makes a step. Then, we show that the invariants holds at time t' as well.

First, notice that if $PC(p) = \{1-4, 7-11, 13, 16-20, 23, 24\}$, or if $PC(p) = \{5, 12, 21\}$ and p's CAS fails, then none of the invariants are affected by p's step and hence they hold at time t' as well.

If $PC(p) = 5$ and p's CAS succeeds, then p moves from $\mathcal{P}$ to $\mathcal{P}'$ and writes $mybuf_p$ into `Help`p.*buf*. Consequently, invariant 5 holds by IH:4 and invariant 9 by IH:9. All other invariants trivially hold.

If $PC(p) = 6$, then, by Lemma 57, p was in $\mathcal{P}'$ at time t. Furthermore, p is in $\mathcal{P}$ at time t'. Since p writes `Help`p.*buf* into $mybuf_p$, invariant 4 holds by IH:5 and invariant 9 by IH:9. All other invariants trivially hold.

If $PC(p) = 12$ and p's CAS succeeds, then p moves from $\mathcal{P}$ to $\mathcal{P}''$. Let `X`i be the variable that p writes to. Then, since p's CAS is successful, we have `X`i.*buf* $= x_p$.*buf* at time t, and `X`i.*buf* $= mybuf_p$ at time t'. Consequently, invariant 8 holds by IH:4, invariant 6 by IH:8, and invariant 9 by IH:9. All other invariants trivially hold.

If $PC(p) = 14$, then p leaves $\mathcal{P}''$. Furthermore, p enqueues x_p.*buf* into the queue Q_p. Consequently, invariant 1 holds by IH:1, invariant 2 by IH:1, invariant 3 by IH:6, and invariant 9 by IH:9. All other invariants trivially hold.

If $PC(p) = 15$, then p joins $\mathcal{P}$. Furthermore, p reads and dequeues the front element of the queue Q_p. Consequently, invariant 4 holds by IH:3, invariant 1 by IH:2, and invariant 9 by IH:9. All other invariants trivially hold.

If $PC(p) = 21$ and p's CAS succeeds, then p moves from $\mathcal{P}$ to $\mathcal{P}'''$. Let `Helpq` be the variable that p writes into during this step. Then, we have $b(p) =$ `Helpq.buf` at time t, `Helpq.buf` $= mybuf_p$ at time t', and `Helpq` changes from value $(*, 1, *)$ at time t to a value $(*, 0, *)$ at time t'. Therefore, by Lemma 57, we have $PC(q) \in \{3 - 5\}$ at times t and t', and `Helpq.buf` $= mybuf_q$. Hence, q moves from $\mathcal{P}$ to $\mathcal{P}'$, and $b(p) = mybuf_q$. Consequently, invariant 5 holds by IH:4, invariant 7 by IH:4, and invariant 9 by IH:9. All other invariants trivially hold.

If $PC(p) = 22$, then p moves from $\mathcal{P}'''$ to $\mathcal{P}$. Furthermore, p writes $b(p)$ into $mybuf_p$. Consequently, invariant 4 holds by IH:7 and invariant 9 by IH:9. All other invariants trivially hold. □

Lemma 59 *Let $t_0 < t_1 < \ldots < t_K$ be all the times in $\mathcal{H}$ when some variable `Xi` is written to (by a successful CAS at Line 12). Then, for all $j \in \{0, 1, \ldots, K\}$, the value written into `Xi` at time t_j is of the form $(j, *)$.*

Proof. Suppose not. Let j be the smallest index such that, at time t_j, a value $k \neq j$ is written into `Xi` by some process p. (By initialization, we have $j \geq 1$.) Then, by the algorithm, p's CAS at time t_j is of the form `CAS(Xi`, $(k - 1, *)$, $(k, *)$`)`. Since `Xi` holds value $j - 1$ at time t_j, and since $k \neq j$, it follows that p's CAS fails, which is a contradiction to the fact that p writes into `Xi` at time t_j. □

Lemma 60 *Let $\mathcal{O}i$ be an LL/SC object. Let t be the time when some process p reads `Xi` (at Line 3 or 18), and $t' > t$ the first time after t that p completes Line 4 or Line 19. Let* OP *be the latest successful SC operation on $\mathcal{O}i$ to execute Line 12 prior to time t, and v the value that* OP *writes in $\mathcal{O}i$. If there exists some process q*

such that `Helpq` *holds value* $(*, 1, *)$ *throughout* (t, t') *and doesn't change, then* p *reads value* v *from* `BUF` *at Line 4 or Line 19 (during* (t, t')*).*

Proof. Let r be the process executing OP. Since OP is the latest successful SC operation on $\mathcal{O}i$ to execute Line 12 prior to time t, it follows that p reads from Xi at time t the value that r writes in Xi at Line 12 of OP. Therefore, p reads during (t, t') the same buffer B that r wrote v into at Line 11 of OP. Let t_1 be the time when r starts writing into B at Line 11 of OP, t_2 the time when r completes writing into B at Line 11 of OP, t_3 the time when r writes into Xi at Line 12 of OP, and t'' the time when p starts reading B during (t, t'). Then, the following claim holds.

Claim 7 *During* (t_1, t_2)*, no process other than* r *writes into* B*. During* (t_2, t')*, no process writes into* B*.*

Proof. Suppose not. Then, either some process other than r writes into B during (t_1, t_2), or some process writes into B during (t_2, t'). In the first case, let r_1 be the process that writes into B during (t_1, t_2). Then, at some point during (t_1, t_2), we have $mybuf_{r_1} = mybuf_r$, which is a contradiction to Invariant 9. In the second case, let r_2 be the first process to start writing into B at some time $\tau_1 \in (t_2, t')$, and k be the index of buffer B. Then, by an earlier argument, $\tau_1 \notin (t_2, t_3)$. Furthermore, by Invariant 9 , r_2 does not write into B as long as Xi holds value $(*, k)$. Therefore, Xi changes during (t_3, τ_1).

Since Xi doesn't change during (t_3, t), it means that (1) $\tau_1 > t$ and (2) some process writes into Xi during (t, τ_1). Let r_3 be the first such process, $\tau_2 \in (t, \tau_1)$ the time when r_3 writes into Xi, and SC_{r_3} the SC operation during which r_3 performs that write. Let τ_3 be the time when r_3 executes Line 14 of SC_{r_3}. Then, at time

τ_3, r_3 enqueues k into Q_{r_3}. Furthermore, by Invariant 9, r_2 does not write into B during (τ_2, τ_3), nor does it write into B during the time Q_{r_3} contains value k. Therefore, we have $\tau_3 \in (\tau_2, \tau_1)$. Finally, we know that k is dequeued from Q_{r_3} during (τ_3, τ_1).

Let τ_4 be the first time after τ_3 that k is dequeued from Q_{r_3}. (Notice that, by the above argument, $\tau_4 \in (\tau_3, \tau_1)$.) Then, by Invariant 1, r_3 executes Lines 16–23 N times during (τ_3, τ_4). Since during each execution of Lines 16–23 r_3 increments variable $index_{r_3}$ by 1 modulo N, there exists an execution E of Lines 16–23 during which $index_{r_3} = q$. Because `Help`q holds value $(*, 1, *)$ throughout (t, t') and doesn't change, it follows that (1) r_3 satisfies the condition at Line 16 of E, and (2) r_3's CAS at Line 21 of E succeeds. This, however, is a contradiction to the fact that `Help`$q = (*, 1, *)$ throughout (t, t'). Hence, we have the claim. □

The above claim shows that (1) during (t_1, t_2), no process other than r writes into B, and (2) during (t_2, t'), no process writes into B. Consequently, p reads v from B during (t, t'), which proves the lemma. □

Lemma 61 *Let $\mathcal{O}i$ be an LL/SC object and* OP *some LL operation on $\mathcal{O}i$. Let SC_q be the latest successful SC operation on $\mathcal{O}i$ to execute Line 12 prior to Line 3 of* OP*, and v_q the value that SC_q writes in $\mathcal{O}i$. If the CAS at Line 5 of* OP *succeeds, then* OP *returns value v_q.*

Proof. Let p be the process executing OP. Let t time when p executes Line 3 of OP, and $t' > t$ be the time when p completes Line 4 of OP. Since the CAS at Line 5 of OP succeeds, it follows by Lemma 57 that `Help`p holds value $(*, 1, *)$ throughout (t, t') and doesn't change during that time. Therefore, by Lemma 60,

p reads v_q from BUF at Line 4 of OP, which proves the lemma. □

Lemma 62 *Let $\mathcal{O}i$ be an LL/SC object, and* OP *an LL operation on $\mathcal{O}i$ such that the CAS at Line 5 of* OP *fails. Let p be the process executing* OP. *Let t and t' be the times, respectively, when p executes Lines 2 and 5 of* OP. *Let x and v be the values that p reads from* BUF *at Lines 7 and 8 of* OP, *respectively. Then, there exists a successful SC operation SC_q on $\mathcal{O}i$ such that (1) at some point during (t, t'), SC_q is the latest successful SC on $\mathcal{O}i$ to execute Line 12, and (2) SC_q writes x into* Xi *and v into $\mathcal{O}i$.*

Proof. Since p's CAS at time t' fails, it means that Help$p = (s, 0, b)$ just prior to t'. Then, by Lemma 57, there exists a single process r that writes into Help$p$ during (t, t') (at Line 21). Let $t_1 \in (t, t')$ be the time when r performs that write, and E be r's execution of Lines 16–22 during which r performs that write. Then, r's CAS at Line 21 of E (at time t_1) is of the form CAS(Help$p$, $(s, 1, *)$, $(s, 0, *)$), for some s. Therefore, at time t_1, Help$p$ has value $(s, 1, *)$. Hence, by Lemma 57, p writes $(s, 1, *)$ into Help$p$ at Line 2 of OP (at time t). Since a value of the form $(s, *, *)$ is written into Help$p$ for the first time at time t, it follows that r reads $(s, 1, *)$ from Help$p$ at Line 16 of E at some time $t_2 \in (t, t_1)$. Consequently, r reads variable Announce$p$ at Line 17 of E at some time $t_3 \in (t_2, t_1)$. Since p writes i into Announce$p$ at Line 1 of OP, it follows that r reads i from Announce$p$ at time t_2. Hence, r reads Xi at Line 18 of E.

Let t_4 be the time when r reads Xi at Line 18 of E, t_5 the time when r starts Line 19 of E, t_6 the time when r completes Line 19 of E, and t_7 the time when r executes Line 20 of E. Let SC_q be the latest successful SC operation on $\mathcal{O}i$ to execute Line 12 prior to time t_4, x_q the value that SC_q writes in Xi, and v_q the value

that SC_q writes in $\mathcal{O}i$. Then, at time t_4, r reads x_q from $\mathtt{X}i$. Furthermore, since t_1 is the first (and only) time that $\mathtt{Help}p$ is written during (t, t'), it follows that $\mathtt{Help}p$ holds value $(*, 1, *)$ at all times during (t_4, t_6) and doesn't change during that time. Therefore, by Lemma 60, r reads v_q from $\mathtt{BUF}$ during (t_5, t_6).

Let B be the buffer that r writes v_q into during (t_5, t_6). Then, at time t_7, r writes x_q into BW. Furthermore, since r writes the index of buffer B into $\mathtt{Help}p$ at Line 21 of E (at time t_1), it follows that p reads buffer B at Lines 7 and 8 of OP. Let t_8 be the time when p reads BW at Line 7 of OP, t_9 the time when p starts reading B at Line 8 of OP, and t_{10} the time when p completes reading B at Line 8 of OP. Then, we show that the following claim holds.

Claim 8 *During (t_5, t_6), no process other than r writes into B, and during (t_6, t_{10}), no process writes into B.*

Proof. Suppose not. Then, either some process other than r writes into B during (t_5, t_6), or some process writes into B during $(t_6, t_1 0)$. In the former case, let r_1 be the process that writes into B during (t_5, t_6). Then, at some point during (t_5, t_6), we have $mybuf_{r_1} = mybuf_r$, which is a contradiction to Invariant 9. In the latter case, let r_2 be the first process to write into B at some time $\tau_1 \in (t_6, t_{10})$. Then, by an earlier argument, we know that $\tau_1 \notin (t_6, t_1)$. We now show that $\tau_1 \notin (t_1, t_{10})$.

Let b be the index of buffer B. We know by Invariant 9 that r_2 does not write into B as long as (1) $\mathtt{Help}p = (s, 0, b)$, and (2) p is between Lines 2 and 6 of OP. Furthermore, since p sets $mybuf_p$ to b at Line 6 of OP, r_2 does not write into B after p executes Line 6 of OP and before it completes OP. Therefore, throughout (t_1, t_{10}), r_2 does not write into B. Hence, $\tau_1 \notin (t_1, t_{10})$. Since, by an earlier argument, $\tau_1 \notin (t_6, t_1)$, it follows that $\tau_1 \notin (t_6, t_{10})$. This, however, is a contradiction to the

fact that r_2 writes into B during (t_6, t_{10}). □

The above claim shows that (1) during (t_5, t_6), no process other than r writes into B, and (2) during (t_6, t_{10}), no process writes into B. Consequently, p reads x_q from BW at time t_8 and v_q from B during (t_9, t_{10}). Since SC_q is the latest successful SC operation on $\mathcal{O}i$ to execute Line 12 prior to time t_4, and since $t_4 \in (t, t')$, we have the lemma. □

Lemma 63 (Correctness of LL) *Let $\mathcal{O}i$ be some LL/SC object. Let* OP *be any LL operation on $\mathcal{O}i$, and* OP′ *be the latest successful SC operation on $\mathcal{O}i$ such that LP(*OP′*) < LP(*OP*). Then,* OP *returns the value written by* OP′.

Proof. Let p be the process executing OP. We examine the following two cases: (1) the CAS at Line 5 of OP succeeds, and (2) the CAS at Line 5 of OP fails. In the first case, let SC_q be the latest successful SC operation on $\mathcal{O}i$ to execute Line 12 prior to Line 3 of OP, and v_q be the value that SC_q writes in $\mathcal{O}i$. Since all SC operations are linearized at Line 12 and since OP is linearized at Line 3, we have $\mathrm{SC}_q = \mathrm{OP}'$. Furthermore, by Lemma 61, OP returns value v_q. Therefore, the lemma holds in this case.

In the second case, let t and t' be the times, respectively, when p executes Lines 2 and 5 of OP. Let v be the value that p reads from `BUF` at Line 8 of OP. Then, by Lemma 62, there exists a successful SC operation SC_r on $\mathcal{O}i$ such that (1) at some time $t'' \in (t, t')$, SC_r is the latest successful SC on $\mathcal{O}i$ to execute Line 12, and (2) SC_r writes v into $\mathcal{O}i$. Since all SC operations are linearized at Line 12 and since OP is linearized at time t'', we have $\mathrm{SC}_r = \mathrm{OP}'$. Therefore, the lemma holds. □

Lemma 64 (Correctness of SC) *Let $\mathcal{O}i$ be some LL/SC object. Let* OP *be any SC operation on $\mathcal{O}i$ by some process p, and* OP′ *be the latest LL operation on $\mathcal{O}i$ by p prior to* OP. *Then,* OP *succeeds if and only if there does not exist any successful SC operation* OP″ *on $\mathcal{O}i$ such that LP(*OP′*) < LP(*OP″*) < LP(*OP*).*

Proof. We examine the following two cases: (1) the CAS at Line 5 of OP′ succeeds, and (2) the CAS at Line 5 of OP′ fails. In the first case, let t_1 be the time when p executes Line 3 of OP′, and t_2 be the time when p executes Line 12 of OP. Then, we show that the following claim holds.

Claim 9 *Process p's CAS at time t_2 succeeds if and only if there does not exist some other SC operation on $\mathcal{O}i$ that performs a successful CAS at Line 12 during (t_1, t_2).*

Proof. Suppose that no other SC operation on $\mathcal{O}i$ performs a successful CAS at Line 12 during (t_1, t_2). Then, Xi doesn't change during (t_1, t_2), and hence p's CAS at time t_2 succeeds.

Suppose that some SC operation SC_q on $\mathcal{O}i$ does perform a successful CAS at Line 12 during (t_1, t_2). Then, by Lemma 59, Xi holds different values at times t_1 and t_2. Hence, p's CAS at time t_2 fails, which proves the claim. □

Since all SC operations are linearized at Line 12 and since OP′ is linearized at time t_1, it follows from the above claim that OP succeeds if and only if there does not exist some successful SC operation OP″ on $\mathcal{O}i$ such that LP(OP′) < LP(OP″) < LP(OP). Hence, the lemma holds in this case.

In the second case (when the CAS at Line 5 of OP′ fails), let t and t' be the times when p executes Lines 2 and 5 of OP′, respectively. Let x and v be the values that

p reads from BUF at Lines 7 and 8 of OP′, respectively. Then, by Lemma 62, there exists a successful SC operation SC_r on $\mathcal{O}i$ such that (1) at some time $t'' \in (t, t')$, SC_r is the latest successful SC on $\mathcal{O}i$ to execute Line 12, and (2) SC_r writes x into Xi and v into $\mathcal{O}i$. Therefore, at Line 7 of OP′, p reads the value that variable Xi holds at time t''. We now prove the following claim.

Claim 10 *Process p's CAS at time t_2 succeeds if and only if there does not exist some other SC operation on $\mathcal{O}i$ that performs a successful CAS at Line 12 during (t'', t_2).*

Proof. Suppose that no other SC operation on $\mathcal{O}i$ performs a successful CAS at Line 12 during (t'', t_2). Then, Xi doesn't change during (t'', t_2), and hence p's CAS at time t_2 succeeds.

Suppose that some SC operation SC_q on $\mathcal{O}i$ does perform a successful CAS at Line 12 during (t'', t_2). Then, by Lemma 59, Xi holds different values at times t'' and t_2. Hence, p's CAS at time t_2 fails, which proves the claim. □

Since all SC operations are linearized at Line 12 and since OP′ is linearized at time t_1, it follows from the above claim that OP succeeds if and only if there does not exist some successful SC operation OP″ on $\mathcal{O}i$ such that LP(OP′) < LP(OP″) < LP(OP). Hence, the lemma holds. □

Lemma 65 (Correctness of VL) *Let $\mathcal{O}i$ be some LL/SC object. Let* OP *be any VL operation on $\mathcal{O}i$ by some process p, and* OP′ *be the latest LL operation on $\mathcal{O}i$ by p that precedes* OP. *Then,* OP *returns true if and only if there does not exist some successful SC operation* OP″ *on $\mathcal{O}i$ such that LP(*OP″*) ∈ (LP(*OP′*), LP(*OP*)).*

Proof. Similar to the proof of Lemma 64. □

Theorem 10 *Algorithm 6.2 is wait-free and implements an array $\mathcal{O}0..M-1$ of M linearizable N-process W-word LL/SC objects. The time complexity of LL, SC and VL operations on $\mathcal{O}$ are $O(W)$, $O(W)$ and $O(1)$, respectively. The space complexity of the implementation is $O(Nk+(M+N^2)W)$, where k is the maximum number of outstanding LL operations of a given process.*

Proof. The theorem follows immediately from Lemmas 63, 64, and 65. □

A.4 Proof of the algorithms in Chapter 7

A.4.1 Proof of Algorithm 7.1

Let $\mathcal{H}$ be the execution history of Algorithm 7.1. Then, we show that the following lemmas hold.

Lemma 66 *Let $t_1, t_2, \ldots, t_m$ be all the times in $\mathcal{H}$ that variable `D` is written. Let d_i, for all $i \in \{1, 2, \ldots, m\}$, be the value written into `D` at time t_i. Let d_0 be the initializing value for `D`. Then, we have $d_0 \neq d_1 \neq \ldots \neq d_m$. Furthermore, for all $i \in \{0, 1, \ldots, m\}$, we have: (1) $d_i \rightarrow size = 2^i$, (2) $d_i \rightarrow A$ is an array of size 2^i, (3) $d_i \rightarrow B$ is an array of size 2^{i+1}, and (4) $d_i \rightarrow A = d_{i-1} \rightarrow B$ for $i > 0$.*

Proof. Suppose that the first part of the claim doesn't hold. Then, there exist some indices i and j in $\{0, 1, \ldots, m\}$ such that $d_i = d_j$. Let t'_i (respectively, t'_j) be the latest time prior to t_i (respectively, t_j) that d_i was returned by a `malloc` at Line 6 (or during initialization). Then, by the algorithm, d_i (respectively, d_j) is not freed

after time t_i (respectively, t_j). Hence, by the uniqueness of allocated addresses, we have $t'_j < t'_i$ and $t'_i < t'_j$, which is a contradiction. Therefore, we have the first part of the claim.

We prove the second part of the claim by induction. Suppose that that the claim holds for all $i < j$; we show that the claim holds for j as well. Let p be the process that writes d_j into D at time t_j. Let t be the latest time prior to t_j that p reads D (at Line 1). Then, since p's CAS at time t_j succeeds, p reads d_{j-1} from D at time t. Furthermore, since, by the first part of the claim, we have $d_0 \neq d_1 \neq \ldots \neq d_{j-1} \neq d_j$, it follows that $t \in (t_{j-1}, t_j)$. By inductive hypothesis, we have (1) $d_{j-1} \rightarrow size = 2^{j-1}$, (2) $d_{j-1} \rightarrow A$ is of size 2^{j-1}, and (3) $d_{j-1} \rightarrow B$ is of size 2^j. Therefore, during time (t, t_j), p (1) writes 2^j into $d_j \rightarrow size$ (Line 9), (2) sets $d_j \rightarrow B$ to be a new array of size 2^{j+1} (Line 8), and (3) sets $d_j \rightarrow A$ to be $d_{j-1} \rightarrow B$ (Line 7). Hence, we have the claim. □

In the following, let K be the maximum i such that write$(i, *)$ is invoked in $\mathcal{H}$. Let $\mathcal{E}_i$, for all $i \in \{0, 1, \ldots, K\}$, be the collection of all executions of write$(i, *)$ in $\mathcal{H}$. Let t_i, for all $i \in \{0, 1, \ldots, K\}$, be the earliest time some execution in $\mathcal{E}_i$ is invoked. Let t'_i, for all $i \in \{0, 1, \ldots, K\}$, be the earliest time some execution in $\mathcal{E}_i$ completes. (Notice that, by the definition of DynamicArray, $t_i > t'_{i-1}$ for all $i \in \{1, 2, \ldots, K\}$.) Let v_i, for all $i \in \{0, 1, \ldots, K\}$, be the value written by an execution in $\mathcal{E}_i$. Let $c(i)$, for all $i \in \{1, 2, \ldots, K\}$, be the smallest power of 2 *greater or equal* to i, and $c'(i)$ be the largest power of 2 *smaller or equal* to i. (If $i = 0$, then $c'(i) = 0$.) If E is an execution of procedure write in $\mathcal{H}$, then, in the following, we slightly abuse notation, and say that "E executes Line 4", instead of "process p executing E executes Line 4."

Lemma 67 *For all $i \in \{0, 1, \ldots, K\}$, we have the following.*

If i is not a power of 2:

(1) $\mathcal{E}_i \neq \emptyset$.

(2) Let d be the value of variable `D` *at time t_i. Then, we have $d{\rightarrow}size = c(i)$.*

(3) For all $E \in \mathcal{E}_i$, E does not execute Lines 6–11. (Notice that this implies that E does not invoke the `write` *procedure at Line 11; therefore, we can say, for example, "Line 4 of E," without being ambiguous.)*

(4) Variable `D` *doesn't change during (t_i, t_i').*

(5) Let E be any execution in $\mathcal{E}_i$, and d be the value that E reads from `D` *at Line 1. Then, E writes v_i into $d{\rightarrow}Ai$ at Line 3.*

(6) Let E be any execution in $\mathcal{E}_i$, and d be the value that E reads from `D` *at Line 1. Then, E writes v_i into $d{\rightarrow}Bi$ at Line 4.*

(7) Let E be any execution in $\mathcal{E}_i$, and d be the value that E reads from `D` *at Line 1. Then, E copies $d{\rightarrow}Ai - c'(i)$ into $d{\rightarrow}Bi - c'(i)$ at Line 5.*

If i is a power of 2:

(8) $\mathcal{E}_i \neq \emptyset$.

(9) Let d be the value of variable `D` *at time t_i. Then, we have $d{\rightarrow}size = i$.*

(10) For all $E \in \mathcal{E}_i$, E executes `write` *at Line 11 at most once. (In the following, let $\mathcal{E}_i' \subset \mathcal{E}_i$ be the collection of all executions in $\mathcal{E}_i$ that execute* `write` *at Line 11, and $\mathcal{E}_i'' \subset \mathcal{E}_i$ be the collection of all executions in $\mathcal{E}_i$ that do not*

execute `write` *at Line 11. Let* $E(1)$ *and* $E(2)$*, for all* $E \in \mathcal{E}'_i$*, denote* E*'s first and second execution of Lines 1–11, respectively.)*

(11) Exactly one execution $E \in \mathcal{E}_i$ *performs a successful CAS at Line 10. Furthermore,* E *performs this CAS at some time* t''_i *during* (t_i, t'_i)*.*

(12) Variable `D` *changes exactly once during* (t_i, t'_i)*, at time* t''_i*.*

(13) Let E *(respectively,* E'*) be any execution in* $\mathcal{E}'_i$ *(respectively,* $\mathcal{E}''_i$*). Then,* $E(2)$ *(respectively,* E'*) executes Line 1 after time* t''_i*.*

(14) Let E *(respectively,* E'*) be any execution in* $\mathcal{E}'_i$ *(respectively,* $\mathcal{E}''_i$*), and* d *be the value that* $E(2)$ *(respectively,* E'*) reads from* `D` *at Line 1. Then,* $E(2)$ *(respectively,* E'*) writes* v_i *into* $d \rightarrow Ai$ *at Line 3.*

(15) Let E *(respectively,* E'*) be any execution in* $\mathcal{E}'_i$ *(respectively,* $\mathcal{E}''_i$*), and* d *be the value that* $E(2)$ *(respectively,* E'*) reads from* `D` *at Line 1. Then,* $E(2)$ *(respectively,* E'*) writes* v_i *into* $d \rightarrow Bi$ *at Line 3.*

(16) Let E *(respectively,* E'*) be any execution in* $\mathcal{E}'_i$ *(respectively,* $\mathcal{E}''_i$*), and* d *be the value that* $E(2)$ *(respectively,* E'*) reads from* `D` *at Line 1. Then,* $E(2)$ *(respectively,* E'*) copies* $d \rightarrow A0$ *into* $d \rightarrow B0$ *at Line 5.*

Proof. (By induction) We assume that the lemma holds for all $i < j$, and show that it also holds for j. (During the proof of the inductive step, we will also prove the base case of $i = 0$.) We first prove the case when j is not a power of 2.

Let d be the value of variable `D` at time t_j. By inductive hypothesis, `D` has changed exactly $\lg(c(j))$ times prior to time t_j (by Statements (3) and (11)). Then, by Lemma 66, we have (1) $d \rightarrow size = c(j)$, (2) $d \rightarrow A$ is an array of size $c(j)$,

and (3) $d \to B$ is an array of size $2c(j)$. Hence, Statement (2) holds. Since, by Lemma 66, the value of $\texttt{D} \to \textit{size}$ only increases after time t_j, it follows that all executions in $\mathcal{E}_j$ satisfy the condition at Line 2. Therefore, no execution in $\mathcal{E}_j$ executes Line 6–11, which proves Statement (3). Statements (5), (6), and (7) follow directly from the algorithm. Statement (1) follows trivially by the definition of DynamicArray. Statement (4) follows directly from the following claim.

Claim 11 *During (t_j, t'_j), variable D doesn't change.*

Proof. The claim follows immediately from the fact that (1) during (t_j, t'_j), no execution in $\mathcal{E}_j$ writes into D (by Statement (3)), and (2) during (t_j, t'_j), no execution in $\mathcal{E}_i$, for all $i < j$, writes into D (by inductive hypothesis for Statements (3) and (11)). □

We now prove the case when j is a power of 2. Let d be the value of variable D at time t_j. By inductive hypothesis, D has changed exactly $\lg j$ times prior to time t_j (by Statements (3) and (11)). Then, by Lemma 66, we have (1) $d \to \textit{size} = j$, (2) $d \to A$ is an array of size j, and (3) $d \to B$ is an array of size $2j$. Hence, Statement (9) holds. We now prove the following claims.

Claim 12 *Let t be either (1) the earliest time during (t_j, t'_j) that some execution $E \in \mathcal{E}_j$ performs a CAS at Line 10, or (2) t_j, if there is no such execution. Then, throughout (t_j, t), variable D holds value d.*

Proof. The claim follows immediately from the fact that (1) during (t_j, t), no execution in $\mathcal{E}_j$ writes into D (by definition of t), (2) during (t_j, t), no execution in $\mathcal{E}_i$, for all $i < j$, writes into D (by inductive hypothesis for Statements (3) and (11)), and (3) D holds value d at time t_j. □

Claim 13 *At least one execution $E \in \mathcal{E}_j$ performs a CAS at Line 10 during (t_j, t'_j).*

Proof. Suppose not. Then, during (t_j, t'_j), no execution in $\mathcal{E}_j$ performs a CAS at Line 10. Consequently, by Claim 12, variable D holds value d throughout (t_j, t'_j). Let $E \in \mathcal{E}_j$ be the execution that completes at time t'_j. Let $E(1)$ be the first execution of Lines 1–11 by E. Then, $E(1)$ reads d from D at Line 1. Therefore, $E(1)$ does not satisfy the condition at Line 2 and hence executes Line 6–11, which is a contradiction to the fact that no execution in $\mathcal{E}_j$ performs a CAS at Line 10 during (t_j, t'_j). □

Claim 14 *Let t be the earliest time (by Claim 13) during (t_j, t'_j) that some execution $E \in \mathcal{E}_j$ performs a CAS at Line 10. Then, E's CAS at time t succeeds.*

Proof. Notice that, by Claim 12, variable D holds value d throughout (t_j, t). Therefore, E reads d from D at Line 1, and E's CAS at time t succeeds, which proves the claim. □

Claim 15 *Let t be the earliest time (by Claim 13) during (t_j, t'_j) that some execution in $\mathcal{E}_j$ performs a CAS at Line 10. Let E be any execution in $\mathcal{E}_j$, and $E(l)$ any execution of Lines 1–11 of E. Then, if $E(l)$ executes Line 1 prior to time t, it executes Lines 6–11. Otherwise, it executes Lines 3–5.*

Proof. By Claim 12, we know that variable D holds value d throughout (t_j, t). Therefore, if $E(l)$ executes Line 1 before time t, it will find d in variable D. Consequently, $E(l)$ will not satisfy the condition at Line 2, and will hence execute Lines 6–11.

Suppose that $E(l)$ executes Line 1 after time t. Let d' be the value that $E(l)$ reads from `D` at Line 1. Then, by Lemma 66 and Claim 14, we have $d' \rightarrow size \geq 2j$. Therefore, $E(l)$ satisfies the condition at Line 2, and hence executes Lines 3–5. □

Claim 16 *At most one execution $E \in \mathcal{E}_j$ performs a successful CAS at Line 10.*

Proof. Suppose not. Then, two or more executions in $\mathcal{E}_j$ perform a successful CAS at Line 10. Let t and t' be the earliest two times that some execution in $\mathcal{E}_j$ performs a successful CAS at Line 10. Then, by Claim 14, t is the earliest time during (t_j, t'_j) that any execution in $\mathcal{E}_j$ performs a CAS at Line 10.

Let E be the execution in $\mathcal{E}_j$ that performs a successful CAS at time t'. Let $E(l)$ be the execution of Lines 1–11 of E during which E performs that CAS. Let t'' be the time when $E(l)$ executes Line 1, and d' be the value that $E(l)$ reads from `D` at time t''. Then, since $E(l)$'s CAS at time t' succeeds, it follows that $t'' \in (t, t')$ (by Lemma 66). Therefore, by Claim 15, $E(l)$ executes Lines 3–5, which is a contradiction to the fact that $E(l)$ performs a successful CAS at Line 10. □

Notice that, by Claim 15, if an execution $E \in \mathcal{E}_j$ executes `write` at Line 11, it will not execute Lines 6–11 again. Therefore, we have Statement (10). Statement (11) follows directly from Claims 14 and 16. Statement (13) follows directly from Claim 15. Statements (14), (15), and (16) follow directly by the algorithm. Statement (8) follows trivially by the definition of DynamicArray. Statement (12) follows directly from the following claim.

Claim 17 *During (t_j, t'_j), variable `D` changes exactly once, at time t''_i.*

Proof. The claim follows immediately from the fact that (1) during (t_j, t'_j), no execution in $\mathcal{E}_i$, for all $i < j$, writes into `D` (by inductive hypothesis for State-

ments (3) and (11)), and (2) during (t_j, t'_j), exactly one execution in $\mathcal{E}_j$ writes into D (by Statement (11)). □

□

Lemma 68 *No execution E in $\mathcal{H}$ writes or reads an unallocated memory region at Lines 3, 4, or 5.*

Proof. Let E be an execution in $\mathcal{H}$, and $\mathcal{E}_s$ the collection that E belongs to, for some $s \in \{1, 2, \ldots, K\}$. If s is not a power of 2, let d be the value of variable D that E reads at Line 1. Then, by Statement (2) of Lemma 67 and by Lemma 66, we have $d \rightarrow size \geq c(s)$. Furthermore, arrays $d \rightarrow A$ and $d \rightarrow B$ are of sizes at least $c(s)$ and $2c(s)$, respectively. Therefore, E writes into a valid array location at Lines 3 and 4. Since $s > c'(s)$, E reads and writes a valid array location at Line 5 as well.

If s is not a power of 2, we examine two possibilities: either $E \in \mathcal{E}'_j$ or $E \in \mathcal{E}''_j$. In the first case, let d be the value of variable D that $E(2)$ reads at Line 1. Then, by Statement (13) of Lemma 67 and by Lemma 66, we have $d \rightarrow size \geq 2s$. Furthermore, arrays $d \rightarrow A$ and $d \rightarrow B$ are of sizes at least $2s$ and $4s$, respectively. Therefore, $E(2)$ writes into a valid array location at Lines 3 and 4. Since $s = c'(s)$, E reads and writes a valid array location at Line 5 as well. (The argument for the second case is identical, and is therefore omitted.) □

Lemma 69 *Let R be any read$(i, *)$ operation in $\mathcal{H}$, for some $i \in \{0, 1, \ldots, K\}$. Then, R returns v_i.*

Proof. (In the following, let $D(t)$ to denote the value of variable D at time t.) Let t_d and t_r be the times when R executes Lines 12 and 13, respectively. Then, we show that the following claim holds:

Claim 18 *The length of the array $D(t_d) \rightarrow A$ is at least $i + 1$.*

Proof. Notice that, by the definition of DynamicArray, at least one execution $E \in \mathcal{E}_i$ completes before R starts. Then, the claim follows immediately by Statements (2), (9), and (12) of Lemma 67 and by Lemma 66. □

Suppose that the lemma doesn't hold. Then, the value that R reads from $D(t_d) \rightarrow Ai$ at time t_r is different than v_i. By Lemma 67, we know that at least one execution in $\mathcal{E}_i$ writes v_i into both $D(t_i) \rightarrow Ai$ and $d' \rightarrow Bi$ during (t_i, t_i'). Furthermore, D doesn't change during (t_i, t_i') and no other execution in $\mathcal{E}_j$, for all $j < i$, writes into $D(t_i) \rightarrow Ai$ and $D(t_i) \rightarrow Bi$ during (t_i, t_i'). Therefore, we have $D(t_i') \rightarrow Ai = v_i$ and $D(t_i') \rightarrow Bi = v_i$ at time t_i'.

Let t be any time in (t_i', t_r). Let $t^1, t^2, \ldots, t^{m(t)}$ be all the times during (t_i', t) when D changes. Let A_j, for all $j \in \{1, 2, \ldots, m(t)\}$, be the array in $D(t^i) \rightarrow A$. Let B_j, for all $j \in \{1, 2, \ldots, m(t)\}$, be the array in $D(t^i) \rightarrow B$. Let A_0 be the array in $D(t_i') \rightarrow A$, B_0 the array in $D(t_i') \rightarrow B$, and $t^0 = t_i'$. Then, we prove the following claim.

Claim 19 *If no execution writes a value different than v_i at index i (of some array) during (t_i', t), then at all times during (t^j, t) and for all $j \in \{0, 1, 2, \ldots, m(t)\}$, we have $A_j i = v_i$.*

Proof. Suppose not. Then, let t' be the earliest time that the following occurs: for some k, $A_k i \neq v_i$ at time $t' \in (t^k, t)$. Notice that, since $A_0 i = v_i$ at time t'_i, and since no execution writes a value different than v_i at index i during (t'_i, t), it follows that $k \neq 0$. Similarly, since $B_0 i = v_i$ at time t'_i, and since no execution writes a value different than v_i at index i during (t'_i, t), it follows that $A_1 i = v_i$ at time t^1 and $A_1 i = v_i$ throughout (t^1, t). Therefore, we have $k \geq 2$.

It follows by Lemma 67 that some execution in $\mathcal{E}_{2^{k-2}c(i)}$ writes into D at time t^{k-1}. Furthermore, some execution in $\mathcal{E}_{2^{k-1}c(i)}$ writes into D at time t^k. Finally, during (t^{k-1}, t^k), some execution $E \in \mathcal{E}_{2^{k-2}c(i)+i}$ reads the value in $A_{k-1} i$ (at Line 5) and writes that value into $B_{k-1} i$ (at Line 5). Then, by definition of t', E reads v_i from $A_{k-1} i$. Therefore, E writes v_i into $B_{k-1} i$. Consequently, since no execution writes a value different than v_i at index i during (t'_i, t), it follows that $A_k i = v_i$ at time t^k and $A_k i = v_i$ throughout (t^k, t), which is a contradiction to the fact that $A_k i \neq v_i$ at time t'. □

By Claim 19, some execution writes a value different than v_i at index i (of some array) during (t'_i, t_r) (because R reads a value different than v_i at time t_r). Let t_e be the earliest time that some execution E writes a value different than v_i at index i (of some array) during (t'_i, t_r). Then, by Lemma 67, (1) E writes into one of the arrays $A_0, A_1, \ldots, A_{m(t_r)}$ or $B_0, B_1, \ldots, B_{m(t_r)}$, (2) $E \in \mathcal{E}_j$, for some $j > i$, and (3) E's write at time t_e takes place at Line 5 of E. Therefore, E writes into $B_k i$ at time t_e, for some $k \in 0, 1, \ldots, m(t_r)$.

Notice that E reads $A_k i$ at Line 5 prior to time t_e. Since t_e is the first time during (t'_i, t_r) that some execution writes a value different than v_i at index i (of some array), it follows that during (t'_i, t_e), no execution writes a value different

than v_i at index i (of some array). Therefore, by Claim 19, we have $A_k i = v_i$ throughout (t_i', t_e). Consequently, E reads v_i from $A_k i$ at Line 5, and therefore writes v_i into $B_k i$ at time t_e, which is a contradiction. Hence, we have the lemma. □

Theorem 11 *Algorithm 7.1 is wait-free and implements a DynamicArray object $\mathcal{D}$ from a word-sized CAS object and registers. The time complexity of read and write operations on $\mathcal{D}$ is $O(1)$. The space used by the algorithm at any time t is $O(nK)$, where n is the number of processes executing the algorithm at time t, and K is the highest location written in $\mathcal{D}$ prior to time t.*

Proof. The theorem follows immediately from Lemma 69. □

A.4.2 Proof of Algorithm 7.3

Lemmas associated with the code in Figure 7.2

We say that node n is *allocated* at time t if there exists a call to `malloc` at time t (at Line 35) that returns n. Node n is *released* at time t if some process executes a `free` operation at Line 45 at time t with the argument n. Node n is *installed* at time t if some process executes a successful CAS at Line 50 at time t with the third argument n.

We say that node n is *alive* at time t if it has been allocated prior to time t but hasn't been released, or if n is the initial dummy node and n hasn't been released. Node n is *active* at time t if it has been installed prior to time t or if it is the initial dummy node. We let *Alive* denote the set of all nodes that are alive. We let L denote the sequence of nodes that are active, arranged in the order of their installation.

1. $|L| \geq 1$.
2. For any node $n \in L$, we have $n \in \mathit{Alive}$.
3. For any two nodes n_i and n_j such that $i \neq j$, we have $n_i \neq n_j$.
4. $\texttt{Head} = n_0$.
5. $*(n_i.\texttt{next}) = n_{i+1}$, for all $i \in \{0, 1, \ldots, |L| - 2\}$.
6. $n_{|L|-1}.\texttt{next} = \perp$.
7. For all $p \in \mathcal{P}'$, we have $\mathit{mynode}(p) \in \mathit{Alive}$.
8. For all $p \in \mathcal{P}'$ and all $i \in \{0, 1, \ldots, |L| - 1\}$, we have $*\mathit{mynode}(p) \neq n_i$.
9. For all p and q in $\mathcal{P}'$ such that $p \neq q$, we have $\mathit{mynode}(p) \neq \mathit{mynode}(q)$.
10. For all $p \in \mathcal{P}''$, we have $\mathit{mynode}(p) \rightarrow \texttt{next} = \perp$.
11. For all $p \in \mathcal{P}'''$, we have $*\mathit{cur}(p) = n_j$, for some $j \in \{0, 1, \ldots, |L| - 1\}$.
12. For any $p \in \mathcal{P}'$ such that $PC(p) = 53$, we have $\mathit{cur}(p) \rightarrow \texttt{next} \neq \perp$.

Figure A.4: The invariants satisfied by Algorithm 7.3

We let $PC(p)$ denote the value of process p's program counter at time t. For any register r at process p, we let $r(p)$ denote the value of that register at time t. We let $\mathcal{P}'$ denote the set of processes such that $p \in \mathcal{P}'$ if and only if $PC(p) \in \{36 - 45, 47 - 50, 53\}$. We let $\mathcal{P}''$ denote the set of processes such that $p \in \mathcal{P}''$ if and only if $PC(p) \in \{38 - 45, 47 - 50, 53\}$. We let $\mathcal{P}'''$ denote the set of processes such that $p \in \mathcal{P}'''$ if and only if $PC(p) \in \{42 - 45, 47 - 50, 53\}$. We let $|L|$ denote the length of L. We let n_i denote the ith element of L, for all $i \in \{0, 1, \ldots, |L| - 1\}$. Then, Algorithm 7.3 satisfies the following invariants.

Lemma 70 *Algorithm 7.3 satisfies the invariants in Figure A.4.*

Proof. (By induction) For the base case (i.e., $t = 0$), the lemma holds trivially

by initialization. The inductive hypothesis states that the lemma holds at all times prior to $t \geq 0$. Let t' be the earliest time after t that some process, say p, makes a step. Then, we show that the lemma holds at time t' as well.

Notice that, if $PC(p) \in \{36, 38-40, 42-44, 46-48, 51, 52, 54-68\}$, then none of the invariants are affected by p's step and hence they hold at time t' as well.

If $PC(p) = 35$, then p joins $\mathcal{P}'$ and writes into $mybuf(p)$ a pointer to a newly allocated node. Consequently, invariant 8 holds by IH:2, invariant 9 by IH:7, and invariant 7 by definition of *Alive*. All other invariants trivially hold.

If $PC(p) = 37$, then p joins $\mathcal{P}''$ and writes $\perp$ into $mynode(p)\rightarrow$`next`. Hence, we have invariant 10. Furthermore, invariant 5 holds by IH:8. All other invariants trivially hold.

If $PC(p) = 41$, then p joins $\mathcal{P}'''$ and writes `Head` into $cur(p)$. Consequently, invariant 11 holds by IH:4. All other invariants trivially hold.

If $PC(p) = 45$, then p leaves $\mathcal{P}'$, $\mathcal{P}''$, and $\mathcal{P}'''$, and frees up the node $*⇳†\wr⌈(p)$. Consequently, invariant 2 holds by IH:8 and invariant 7 by IH:9. All other invariants trivially hold.

If $PC(p) = 49$, or if $PC(p) = 50$ and p's CAS fails, then we have $cur(p)\rightarrow$ `next` $\neq \perp$ at both times t and t'. Consequently, invariant 12 holds. All other invariants trivially hold.

If $PC(p) = 50$ and p's CAS succeeds, then (1) p leaves $\mathcal{P}'$, $\mathcal{P}''$, and $\mathcal{P}'''$, (2) $*mynode(p)$ joins L, (3) $*cur(p)$.`next` $= \perp$ at time t, and (4) $*cur(p)$.`next` $=$ $mynode(p)$ at time t'. Let l be the length of L at time t. Then, by IH:11, IH:5, and IH:6, it follows that $*cur(p) = n_{l-1}$. Therefore, invariant 5 holds. Furthermore, invariant 1 holds by IH:1, invariant 2 by IH:7, invariant 3 by IH:8, invariant 6 by

IH:10, invariant 8 by IH:9. All other invariants trivially hold.

If $PC(p) = 53$, then p writes $cur(p) \rightarrow \texttt{next}$ into $cur(p)$. Consequently, invariant 11 holds by IH:12, IH:5, and IH:6. All other invariants trivially hold. □

Lemma 71 *Let $n \neq n_0$ be any node in L and t be the time when n is installed in L. Let p be the process that installs n in L. Let t_1 (respectively, t_2, t_3, t_4, t_5) be the time when p executes Line 35 (respectively, Line 36, 37, 38, 39). Let B be the memory block that p allocates at time t_4. Then, we have the following: (1) p allocates n at time t_1, (2) $n.\texttt{owned}$ holds value* true *at all times during (t_2, t), (3) $n.\texttt{next}$ holds value $\perp$ at all times during (t_3, t), (4) $n.\texttt{loc}$ holds a pointer to B at all times during (t_4, t), (5) B is not released during (t_4, t), and (6) $B.\texttt{Help}$ holds value $(0, 0, *)$ at all times during (t_5, t).*

Proof. The lemma follows immediately by Invariant 9. □

Lemma 72 *Let n be any node in L and t be the time when n is installed in L. If $n \neq n_0$, let p be the process that installs n in L and $t' < t$ be the latest time prior to t when p executes Line 38. If $n = n_0$, let $t' = 0$. Let B be the memory block allocated at time t'. Then, (1) B is not released (at Line 44) after time t', and (2) $n.\texttt{loc}$ points to B at all times after t'.*

Proof. If $n = n_0$, then the lemma holds immediately by Invariant 8. We now show that the lemma also holds for $n \neq n_0$. Notice that, by Lemma 71, it follows that (1) p allocates n at Line 35 at some time $t'' < t'$, (2) $n.\texttt{loc}$ holds a pointer to B at all times during (t', t), and (3) B is not released during (t', t). Furthermore, by Invariant 8, no process writes into $n.\texttt{loc}$ after time t, and no process releases B after time t. Therefore, we have the lemma. □

Lemma 73 *For any two nodes n_i and n_j in L such that $i \neq j$, we have $n_i.\texttt{loc} \neq n_j.\texttt{loc}$.*

Proof. If $i \neq 0$ (respectively, $j \neq 0$), let p_i (respectively, p_j) be the process that installs n_i (respectively, n_j) in L, t_i (respectively, t_j) be the time when p_i (respectively, p_j) executes Line 38, and B_i (respectively, B_j) be the block that p_i (respectively, p_j) allocates at time t_i (respectively, t_j). If $i = 0$ (respectively, $j = 0$), let $t_i = 0$ (respectively, $t_j = 0$). Then, by Lemma 72, $n_i.\texttt{loc}$ (respectively, $n_j.\texttt{loc}$) has value B_i (respectively, B_j), at all times after t_i (respectively, t_j), and B_i (respectively, B_j) is not released after time t_i (respectively, t_j). Without loss of generality, let $t_i < t_j$. Then, by the uniqueness of allocated addresses, we have $B_j \neq B_i$, which proves the lemma. □

In the following, we let B_i, for all $i \in \{0, 1, \ldots, |L| - 1\}$, denote the memory block pointed by $n_i.\texttt{loc}$.

Lemma 74 *At the time when some process p starts its kth execution of the loop at Line 42 we have (1) $|L| \geq k$, (2) $\text{cur}(p) = n_{k-1}$, and (3) $\text{name}(p) = k - 1$.*

Proof. (By induction) For the base case (i.e., $k = 1$), notice that by Invariant 1, we have $|L| \geq 1$. Furthermore, by Invariant 4, we have $cur(p) = n_0$. Finally, by Line 40, we have $name(p) = 0$. Therefore, the lemma holds for the base case. The inductive hypothesis states that the lemma holds for some $k \geq 1$. We now show that the lemma holds for $k + 1$ as well.

By inductive hypothesis, we have $cur(p) = n_{k-1}$ and $name(p) = k - 1$ when p starts its kth iteration of the loop. Since p increments $name(p)$ at Line 48 during that iteration, it follows that $name(p) = k$ when p starts its $k + 1$st iteration of

the loop. Furthermore, by Invariants 12 and 5, it follows that $L \geq k+1$ and that $cur(p) = n_k$ when p starts its $k+1$st iteration of the loop. Hence, we have the lemma. □

Definition 7 *If at some time a process p either (1) performs a successful CAS at Line 43 with* $\mathrm{cur}(p) = n$ *or (2) performs a successful CAS at Line 50 with* $\mathrm{mynode}(p) = n$*, then we say that p acquires ownership of a node $n \in L$. If i is the index of n in L (i.e., $n = n_i$), then we also say that p acquires ownership of name i and memory block B_i.*

Lemma 75 *If a process p exits the loop at Line 46 during the kth iteration of the loop, then we have (1) $|L| \geq k$, (2) p has ownership of n_{k-1}, (3)* $\mathrm{name}(p) = k-1$*, (4)* $*\mathrm{cur}(p) = n_{k-1}$*, and (5)* $*\mathrm{mynode}(p) \neq n_{k-1}$.

Proof. Claims 1, 2, 3, and 4 follow immediately from Lemma 74. Claim 5 follows immediately by Invariant 8. □

Lemma 76 *If a process p exits the loop at Line 52 during the kth iteration of the loop, then we have (1) $|L| \geq k+1$, (2) p has ownership of n_k, (3)* $\mathrm{name}(p) = k$*, (4)* $*\mathrm{cur}(p) = n_k$*, and (5)* $*\mathrm{mynode}(p) = n_k$.

Proof. Notice that, by Lemma 74, when p begins its kth iteration of the loop, we have $|L| \geq k$, $name(p) = k-1$, and $*cur(p) = n_{k-1}$. Since p's CAS at Line 50 is successful, it follows that $n_{k-1}.\texttt{next} = \perp$ just before that CAS. Hence, by Invariant 6, $|L| = k$ just before p executes Line 50. Consequently, p installs $*mynode(p)$ into the kth position in L, and so we have $*mynode(p) = n_k$ and

$|L| = k + 1$ after p's CAS. Therefore, Claims 1, 2, and 5 hold. Furthermore, Claim 3 holds by Line 48 and Claim 4 by Line 51. □

Lemma 77 *If a process p captures a node $n \in L$, then p subsequently satisfies the condition at Line 59 if and only if p captured n at Line 52.*

Proof. The lemma follows immediately by Lemmas 75 and 76. □

Lemma 78 *If a process p acquires ownership of some node $n \in L$, then $*\text{node}_p = n$ at the time when p subsequently executes Line 61.*

Proof. The lemma follows immediately by Lemmas 75 and 76, and Line 58. □

Definition 8 *Let t be the time when p acquires ownership of some node $n \in L$, $t' > t$ be the first time after t when p executes Line 61, and i be the position of n in L (i.e., $n = n_i$). Then, we say that p releases ownership of node n (respectively, name i, memory block B_i) at time t', and that p owns node n (respectively, name i, memory block B_i) at all times during (t, t').*

Lemma 79 *For any node $n \in L$, at most one process owns n.*

Proof. Suppose not. Then, there exists some time such that two or more processes own some node in L. Let t be the earliest such time and n the node in L owned by two processes. Let p and q be those two processes. Without loss of generality, assume that p acquired ownership of n first, at some time $t' < t$. (Notice that, by definition of t, q acquires ownership of n at time t.) Then, by Invariant 3, q acquires ownership of n at Line 43. We examine two possibilities: either p

acquires ownership if n at Line 43 or at Line 50. In the first case, p writes *true* into $n.\texttt{owned}$ at time t'. Furthermore, since t is the earliest time that two or more processes own the same node, it follows that during (t', t) no process writes *false* into $n.\texttt{owned}$. Therefore, $n.\texttt{owned} = \mathit{true}$ at time t, and so q's CAS at time t fails. This, however, is a contradiction to the fact that q acquires ownership of n at time t.

In the second case (where p acquires ownership at Line 50), it follows by Lemma 71 that $n.\texttt{owned} = \mathit{true}$ at time t'. By the same argument as above, $n.\texttt{owned} = \mathit{true}$ at time t as well. Therefore, q's CAS at time t fails, which is a contradiction to the fact that q acquires ownership of n at time t. □

Lemma 80 *Let t be the time when some process p starts its kth execution of the loop at Line 44, and $t' < t$ be the latest time prior to t when p executes Line 42. Then, there exists some time $t'' \in (t', t)$ such that the number of nodes in $n_1, n_2, \ldots, n_k$ that are owned by some process at time t'' plus the number of processes (including p) that do not own any nodes but are in their jth execution of the loop at Line 44 at time t'', for $j \leq k$, is at least k.*

Proof. The proof is identical to the proof of Lemma A.4 in [HLM03b]. □

Definition 9 *A memory block B_i, $i \in \{0, 1, \ldots, |L| - 1$, becomes* active *the first time some process that captures B_i (at Line 52) completes its Join procedure.*

Lemma 81 *At the moment when a memory block B_i, $i \in \{0, 1, \ldots, |L| - 1$, becomes active, we have (1) $B_i.N = 1$, (2) $|B_i.\mathit{BUF}| = 2$, (3) B_i.mybuf $= (1, i, 0)$, (4) $|B_i.Q| = 1$, (5) the value in $B_i.Q$ is $(1, i, 1)$, (6) B_i.index $= 0$, and (7) B_i.name $= i$.*

Proof. Let p be the process that first captures B_i (at Line 52). Then, by Lemma 77, p subsequently satisfies the condition at Line 59. Hence, p executes steps at Lines 62–68. Since, by Lemma 77, no other process ever writes into B_i at Lines 62–68, the lemma follows trivially by Lines 62–68. □

Lemma 82 *After a memory block B_i, $i \in \{0, 1, \ldots, |L| - 1$, becomes active, no process writes into B_i at Lines 62–68.*

Proof. The lemma follows immediately by Lemma 77. □

Lemma 83 *Any process that executes Line 47 during the ith iteration of the loop (at Line 42) writes a pointer to B_{i-1} into the $i-1$st location of* `NameArray`.

Proof. The lemma follows immediately by Lemma 74. □

Lemma 84 *If a process p exits the loop at Line 43 during the ith iteration of the loop, then p writes B_{i-1} into the $i-1$st location of* `NameArray` *at Line 54. If a process p exits the loop at Line 52 during the ith iteration of the loop, then p writes B_i into the ith location of* `NameArray` *at Line 54.*

Proof. The lemma follows immediately by Lemmas 75 and 76. □

Lemma 85 *Algorithm 7.3 writes into array* `NameArray` *in accordance with the specification of the DynamicArray object.*

Proof. Notice that, by Lemma 83, any process that executes Line 47 during the ith iteration of the loop writes the same value (namely, a pointer to B_{i-1}) into the

$i-1$st location of `NameArray`. Furthermore, by Lemma 84, if a process exits the loop at Line 43 (respectively, Line 52) during the ith iteration of the loop, then p writes the same value, namely, B_{i-1} (respectively, B_i), into the $i-1$st (respectively, ith) location of `NameArray` at Line 54. Therefore, all values written into the same location are the same, and each process writes into the locations of `NameArray` in order. Hence, we have the lemma. □

Lemma 86 *The value of* `N` *increases with each write into* `N`*.*

Proof. Suppose not. Let t be the first time that `N` is written and its value does not increase. Let p be the process that performs that write. Then, p's CAS at Line 56 must be of the form `CAS`(N, i, j), where $j \leq i$. This, however, is a contradiction to the fact that p had satisfied the condition at Line 55 prior to this CAS. □

Lemma 87 *At the moment when a process p exits the loop at Line 55, we have* name$(p) <$ `N`*.*

Proof. The lemma follows immediately by Line 55. □

Lemma 88 *Any process p completes the loop at Line 55 after at most* name$(p)+1$ *iterations.*

Proof. Suppose not. Then, p executes the loop at Line 55 for at least $name(p)+2$ iterations. Notice that, during the first $name(p)+1$ iterations, p does not perform a successful CAS at Line 56 (because, by Lemma 86, p would exit the loop right after that CAS). Therefore, the value in `N` changes at least $name(p)+1$ times during those $name(p)+1$ iterations. Then, by Lemma 86, it follows that after p

executes $name(p) + 1$ iterations of the loop the value of N is at least $name(p) + 1$. Therefore, p does not execute the $(name(p) + 2)$nd iteration of the loop, which is a contradiction. □

Lemma 89 *Location i in* `NameArray` *holds value B_i at all times, for all $i < N$.*

Proof. Let j be the value of N. If $j = 0$, then the lemma trivially holds. Otherwise, let p be the process that first wrote j into N, and t be the time when p performed that write. Then, we have $name(p) = j - 1$ at time t. Hence, p had executed Line 48 exactly $j - 1$ times prior to t. Consequently, by Lemma 83, p had written pointers to memory blocks $B_0, B_1, \ldots, B_{j-2}$ into locations $0, 1, \ldots, j - 2$ of `NameArray`, respectively. Furthermore, by Lemma 84, p had written a pointer to B_{j-1} into `NameArray` at Line 54. Thus, we have the lemma. □

Lemma 90 *For any memory block B_i, $i \in \{0, 1, \ldots, |L| - 1\}$, if B_i is active then $i < N$.*

Proof. Let p be the process that first captures B_i (at Line 52). Then, by Lemma 76, we have $name(p) = i$ when p exits the loop (at Line 42). Consequently, by Lemma 87, at the moment when p exits the loop at Line 55, we have $i <$ N. Since, by Lemma 86, the value in N never decreases, we have the lemma. □

Corollary 2 *For any memory block B_i, $i \in \{0, 1, \ldots, |L| - 1\}$, if B_i is active then location i in* `NameArray` *holds value B_i.*

We let $\mathcal{P}$ denote a set of processes such that $p \in \mathcal{P}$ if and only if $PC(p) \in 1 \ldots 34$.

Lemma 91 *For any process $p \in \mathcal{P}$, the following holds:*

1. $\text{loc}_p = B_i$, *for some* $i \in 0, 1, \ldots, |L| - 1$.

2. *For all* $q \in \mathcal{P}$ *such that* $p \neq q$, *we have* $\text{loc}_p \neq \text{loc}_q$.

Proof. The first claim follows immediately by Lemmas 75 and 76. The second claim follows immediately by the first claim and Lemma 73. □

Lemmas associated with the code in Figure 7.1

Let $\mathcal{H}$ be any finite execution history of Algorithm 7.3. Let OP be some LL operation, OP′ some SC operation, and OP″ some VL operation on $\mathcal{O}i$ in E, for some i. Then, we define the linearization points for OP, OP′, and OP″ as follows. If the CAS at Line 5 of OP succeeds, then LP(OP) is Line 3 of OP. Otherwise, let t be the time when OP executes Line 2, and t' be the time when OP performs the CAS at Line 5. Let v be the value that OP reads at Line 9 of OP. Then, we show that there exists a successful SC operation SC_q on $\mathcal{O}i$ such that (1) at some point t'' during (t, t'), SC_q is the latest successful SC on $\mathcal{O}i$ to execute Line 16, and (2) SC_q writes v into $\mathcal{O}i$. We then set LP(OP) to time t''. We set LP(OP′) to Line 16 of OP′, and LP(OP″) to Line 14 of OP″.

Lemma 92 *Let B_i, for $i \in \{0, 1, \ldots, |L| - 1$ be any memory block. Let t be the time when B_i is installed in L. Let t' be the end of $\mathcal{H}$. Let $LL_1, LL_2, \ldots, LL_m$ be the sequence of all LL operations in $\mathcal{H}$ such that, if a process p is executing LL_j, then $\text{loc}_p = B_i$ during LL_j, for all $j \in \{1, 2, \ldots, m\}$. Let t_j and t'_j, for all $j \in \{1, 2, \ldots, m\}$, be the times when Lines 2 and 5 of LL_j are executed. Then, the following statements hold:*

(S1) During the time interval $(t_j, t'_j$*, for all* $j \in \{1, 2, \ldots, m\}$*,* exactly one *write into* B_i.Help *is performed.*

(S2) Any value written into B_i.Help *during* $(t_j, t'_j$*, for all* $j \in \{1, 2, \ldots, m\}$*, is of the form* $(*, 0, *)$.

(S3) During the time intervals $(t, t_1), (t'_1, t_2), (t'_2, t_3), \ldots, (t'_{m-1}, t_m), (t'_m, t')$*, the value in* B_i.Help *is of the form* $(*, 0, *)$ *and doesn't change.*

Proof. Let j be any index in $\{1, 2, \ldots, m\}$. Statement (S2) follows trivially from the fact that the only two operations that can affect the value of B_i.Help during (t_j, t'_j) are (1) the CAS at Line 5 of LL_j, and (2) the CAS at Line 31 of some other process' SC operation, both of which attempt to write $(*, 0, *)$ into B_i.Help.

We now prove statement (S1). Suppose that (S1) does not hold. Then, during $(t_j, t'_j$, either (1) two or more writes on B_i.Help are performed, or (2) no writes on B_i.Help are performed. In the first case, we know (by an earlier argument) that each write on B_i.Help during $(t_j, t'_j$ is performed either by the CAS at Line 5 of LL_j, or by the CAS at Line 31 of some other process' SC operation. Let CAS_1 and CAS_2 be the first two CAS operations on B_i.Help to write into B_i.Help during $(t_j, t'_j$. Then, by the algorithm, both CAS_1 and CAS_2 are of the form CAS(B_i.Help, $(*, 1, *)$, $(*, 0, *)$). Since CAS_1 succeeds and B_i.Help doesn't change between CAS_1 and CAS_2, it follows that CAS_2 fails, which is a contradiction.

In the second case (where no writes on B_i.Help take place during (t_j, t'_j), B_i.Help doesn't change throughout $(t_j, t'_j$. Therefore, the CAS at Line 5 of LL_j succeeds, which is a contradiction to the fact that no writes on B_i.Help take place during $(t_j, t'_j$. Hence, statement (S1) holds.

We now prove statement (S3). Suppose that (S3) statement doesn't hold. Let (t'', t''') be any of the intervals (t, t_1), (t'_1, t_2), (t'_2, t_3), $\ldots$, (t'_{m-1}, t_m), (t'_m, t') during which the statement doesn't hold. Notice that, by Lemma 71, $B_i.\texttt{Help} = (*, 0, *)$ at time t. Furthermore, by statements (S1) and (S2), $B_i.\texttt{Help} = (*, 0, *)$ at times $t'_1, t'_2, \ldots, t'_m$. Hence, $B_i.\texttt{Help} = (*, 0, *)$ at time t''. Let CAS_3 be the first CAS operation on $B_i.\texttt{Help}$ to write into $B_i.\texttt{Help}$ during (t'', t'''). Then, by the algorithm, CAS_3 is of the form CAS($B_i.\texttt{Help}$, $(*, 1, *)$, $(*, 0, *)$). Since $B_i.\texttt{Help}$ doesn't change between time t'' and CAS_3, it follows that CAS_3 fails, which is a contradiction. Hence, we have statement (S3). □

In Figure A.5, we present a number of invariants that the algorithm satisfies. In the following, we let $PC(p)$ denote the value of process p's program counter. Without loss of generality, we assume that when a process completes any of the procedures its program counter immediately jumps to the start of the next procedure it wishes to execute. For any register r at process p, we let $r(p)$ denote the value of that register. We let $\mathcal{P}_1$ denote a set of processes such that $p \in \mathcal{P}_1$ if and only if $PC(p) \in \{1, 2, 7-17, 24-31, 33, 34\}$ or $PC(p) \in \{3-5\} \wedge loc_p.\texttt{Help} \equiv (*, 1, *)$. We let $\mathcal{P}_2$ denote a set of processes such that $p \in \mathcal{P}_2$ if and only if $PC(p) \in \{3-6\} \wedge loc_p.\texttt{Help} \equiv (*, 0, *)$. We let $\mathcal{P}_3$ denote a set of processes such that $p \in \mathcal{P}_3$ if and only if $PC(p) = 18$. We let $\mathcal{P}_4$ denote a set of processes such that $p \in \mathcal{P}_4$ if and only if $PC(p) = 32$. We let $\mathcal{B}_1$ (respectively, $\mathcal{B}_2$, $\mathcal{B}_3$) denote a set of memory blocks such that $B \in \mathcal{B}_1$ (respectively, $\mathcal{B}_2$, $\mathcal{B}_3$) if and only if (1) $B \in \mathcal{B}$, and (2) there exists some process p such that $loc_p = B$ and $p \in \mathcal{P}_1$ (respectively, $\mathcal{P}_2$, $\mathcal{P}_3$). We let $\mathcal{B}_0$ denote a set of memory blocks such that $B \in \mathcal{B}_0$ if and only if (1) $B \in \mathcal{B}$, and (2) there does not exist any process $p \in \mathcal{P}$ such that

$loc_p = B$. Finally, for any buffer $B \in \mathcal{B}$, we let $|B.Q|$ denote the length of the queue $B.Q$, and $|B.\texttt{BUF}|$ denote the length of the array $B.\texttt{BUF}$.

Lemma 93 *Algorithm 7.3 satisfies the invariants in Figure A.5.*

Proof. For the base case, (i.e., $t = 0$), all the invariants hold by initialization. The inductive hypothesis states that the invariants hold at time $t \geq 0$. Let t' be the earliest time after t that some process, say p, makes a step. Then, we show that the invariants holds at time t' as well.

First, notice that if $PC(p) = \{1{-}4, 7{-}9, 11{-}13, 15, 17, 24{-}30, 35{-}58, 60{-}67\}$, or if $PC(p) = \{5, 16, 31\}$ and p's CAS fails, then none of the invariants are affected by p's step and hence they hold at time t' as well. In the following, let $B = loc_p$.

If $PC(p) = 5$ and p's CAS succeeds, then B moves from $\mathcal{B}_1$ to $\mathcal{B}_2$ and p writes $B.mybuf$ into $B.\texttt{Help}.buf$. Consequently, invariant 5 holds by IH:5 and invariant 6 by IH:6. All other invariants trivially hold.

If $PC(p) = 6$, then, by Lemma 92, B was in $\mathcal{B}_2$ at time t. Furthermore, B is in $\mathcal{B}_1$ at time t'. Since p writes $B.\texttt{Help}.buf$ into $B.mybuf$, invariant 5 holds by IH:5 and invariant 6 by IH:6. All other invariants trivially hold.

If $PC(p) \in \{10, 14, 17, 34\}$, then B moves either from $\mathcal{B}_1$ to $\mathcal{B}_1$ (if the next operation that p executes is LL, VL, or SC), or from $\mathcal{B}_1$ to $\mathcal{B}_0$ (if the next operation that p executes is Leave). In either case, invariant 5 holds by IH:5 and invariant 6 by IH:6. All other invariants trivially hold.

If $PC(p) = 16$ and p's CAS succeeds, then B moves from $\mathcal{B}_1$ to $\mathcal{B}_3$. Let $\texttt{X}i$ be the variable that p writes to. Then, since p's CAS is successful, we have $\texttt{X}i.buf = B.x.buf$ at time t, and $\texttt{X}i.buf = B.mybuf$ at time t'. Consequently,

1. For any memory block $B \in \mathcal{B}$, we have $|B.Q| \geq B.N$.

2. For any memory block $B \in \mathcal{B}$, we have $|B.\texttt{BUF}| \leq B.N + 1$.

3. For any process $p \in \mathcal{P}$ such that $PC(p) \in \{19, 20, 23\}$, we have $|loc_p.Q| \geq loc_p.N + 1$.

4. For any process $p \in \mathcal{P}$ such that $PC(p) = 21$, we have $|loc_p.\texttt{BUF}| \leq loc_p.N$.

5. Let B^0 (respectively, B^1, B^2, B^3) be any memory block in $\mathcal{B}_0$ (respectively, $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$). Let B be any memory block in $\mathcal{B}$, and b be any value in $B.Q$. Let p_4 be any process in $\mathcal{P}_4$. Let i be any index in $0\,..\,M-1$. If b (respectively, $B^0.mybuf$, $B^1.mybuf$, $B^2.\texttt{Help}.buf$, $B^3.x.buf$, $c(p_4)$, $\texttt{X}j.buf$) is of the form $(0, j)$, then we have $j \in 0\,..\,M-1$. If b (respectively, $B^0.mybuf$, $B^1.mybuf$, $B^2.\texttt{Help}.buf$, $B^3.x.buf$, $c(p_4)$, $\texttt{X}j.buf$) is of the form $(1, n, j)$, then we have (1) $B_n \in \mathcal{B}$, (2) the jth location in array $B_n.\texttt{BUF}$ holds a pointer to a buffer of length $W + 1$, and (3) there does not exist any process $p \in \mathcal{P}$ such that $loc_p = B_n$, $PC(p) = 22$, and $loc_p.N = j$.

6. Let B^0 and C^0 (respectively, B^1 and C^1, B^2 and C^2, B^3 and C^3) be any two memory blocks in $\mathcal{B}_0$ (respectively, $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$). Let B be any memory block in $\mathcal{B}$, and b_1 and b_2 any two values in $B.Q$. Let p_4 and q_4 be any two processes in $\mathcal{P}_4$. Let i and j be any two indices in $0\,..\,M-1$. Then, we have $B^0.mybuf \neq C^0.mybuf \neq B^1.mybuf \neq C^1.mybuf \neq b_1 \neq b_2 \neq \texttt{X}i.buf \neq \texttt{X}j.buf \neq B^2.\texttt{Help}.buf \neq C^2.\texttt{Help}.buf \neq B^3.x.buf \neq C^3.x.buf \neq c(p_4) \neq c(q_4)$.

7. For any memory block $B \in \mathcal{B}$, we have $0 < B.N \leq \texttt{N}$.

8. For any process $p \in \mathcal{P}$ such that $PC(p) = 20$, we have $0 < loc_p.N < \texttt{N}$.

9. For any memory block $B \in \mathcal{B}$, we have $B.index < B.N$.

Figure A.5: The invariants satisfied by Algorithm 7.3

invariant 5 holds by IH:5 and invariant 6 by IH:6. All other invariants trivially hold.

If $PC(p) = 18$, then B leaves $\mathcal{B}_3$. Furthermore, p enqueues $B.x.buf$ into the queue $B.Q$. Consequently, invariant 1 holds by IH:1, invariant 3 by IH:1, invariant 5 by IH:5, and invariant 6 by IH:6. The other two invariants trivially hold.

If $PC(p) = 19$, then we have $B.N < \mathtt{N}$. Consequently, invariant 8 holds by IH:7. All other invariants trivially hold.

If $PC(p) = 20$, then p increments $B.N$ by one. Consequently, invariant 1 holds by IH:3, invariant 4 by IH:2, and invariant 7 by IH:8. All other invariants trivially hold.

If $PC(p) = 21$, then the length of $B.\mathtt{BUF}$ increases to at least $B.N + 1$. Consequently, invariant 2 holds by IH:4. Notice that, by IH:4, the length of $B.\mathtt{BUF}$ was at most $B.N$ prior to this step. Therefore, location $B.N$ does not hold a pointer to a buffer of length $W + 1$. Consequently, by IH:5, none of the values appearing in the inequality of invariant 6 were of the form $(1, B.name, B.N)$ prior to this step. Hence, invariant 5 holds. All other invariants trivially hold.

If $PC(p) = 22$, then B joins $\mathcal{B}_1$. Furthermore, p writes $(1, B.name, B.N)$ into $B.mybuf$. Notice that, by IH:5, none of the values appearing in the inequality of invariant 6 were of the form $(1, B.name, B.N)$ prior to this step. Consequently, invariant 6 holds. Invariant 5 holds by the fact that p had written a buffer of length $W + 1$ into location $B.N$ of $B.\mathtt{BUF}$ at Line 21. All other invariants trivially hold.

If $PC(p) = 23$, then B joins $\mathcal{B}_1$. Furthermore, p reads and dequeues the front element of the queue $B.Q$. Consequently, invariant 1 holds by IH:3, invariant 5 by IH:5, and invariant 6 by IH:6. All other invariants trivially hold.

If $PC(p) = 31$ and p's CAS succeeds, then B moves from $\mathcal{B}_1$ to $\mathcal{B}_4$. Let B' be the memory block pointed to by $l(p)$. Then, we have $c(p) = B'.\texttt{Help}.buf$ at time t, $B'.\texttt{Help}.buf = B.mybuf$ at time t', $B'.\texttt{Help}$ changes from value $(*, 1, *)$ at time t to a value $(*, 0, *)$ at time t'. Therefore, by Lemma 92, there exists some process q such that (1) $loc_q = B'$ and $PC(q) \in \{3 - 5\}$ at times t and t', and (2) $B'.\texttt{Help}.buf = B'.mybuf$ at time t. Hence, B' moves from $\mathcal{B}_1$ to $\mathcal{B}_2$, and $c(p) = B'.mybuf$. Consequently, invariant 5 holds by IH:5 and invariant 6 by IH:6. All other invariants trivially hold.

If $PC(p) = 32$, then B moves from $\mathcal{B}_4$ to $\mathcal{B}_1$. Furthermore, p writes $c(p)$ into $B.mybuf$. Consequently, invariant 5 holds by IH:5 and invariant 6 by IH:6. All other invariants trivially hold. □

If $PC(p) = 59$ and p does not satisfy the condition at Line 59, then, by Lemma 77, B moves either from $\mathcal{B}_0$ to $\mathcal{B}_1$ (if the next operation that p executes is LL), or from $\mathcal{B}_0$ to $\mathcal{B}_0$ (if the next operation that p executes is Leave). In the latter case, none of the invariants are affected by p's step and hence they hold at time t'. In the former case, invariant 5 holds by IH:5 and invariant 6 by IH:6. All other invariants trivially hold.

If $PC(p) = 59$ and p satisfies the condition at Line 59, then none of the invariants are affected by p's step and hence they hold at time t' as well.

If $PC(p) = 68$, then, by Lemma 77, B joins $\mathcal{B}$. Furthermore, B either joins $\mathcal{B}_1$ (if the next operation that p executes is LL), or $\mathcal{B}_0$ (if the next operation that p executes is Leave). Let i be the name that p captures during the Join procedure. Then, we have $B = B_i$. Notice that, by Lemma 81, we have invariants 1, 2, 5, and 9. Furthermore, by IH:5, all the values in the inequality of invariant 6 are either of the form $(0, *)$ or $(1, n, *)$, for some $B_n \in \mathcal{B}$. Since B has just joined

$\mathcal{B}$ and since all memory blocks in $\mathcal{B}$ are different (by Lemma 73), it follows that $n \neq i$. Hence, by Lemma 81, we have invariant 6. Invariant 7 follows immediately by Lemmas 81 and 90. All other invariants trivially hold. □

Lemma 94 *Algorithm 7.3 reads array* `NameArray` *in accordance with the specification of the DynamicArray object.*

Proof.

Let p be any process. Then, the only two places where p reads `NameArray` is at Lines 12 and 24 of the algorithm. If p reads `NameArray` at Line 12, let i be the location in `NameArray` that p reads. Then, by Invariant 5, it follows that $B_i \in \mathcal{B}$. Consequently, by Corollary 2, location i of `NameArray` had already been written, which means that the lemma holds.

If p reads `NameArray` at Line 24, let j be the location in `NameArray` that p reads. Then, by Invariants 9 and 7, it follows that $i <$ `N`. Consequently, by Lemma 89, location i of `NameArray` had already been written, which means that the lemma holds. □

Lemma 95 *Let B_i, for $i \in \{0, 1, \ldots, |L| - 1\}$ be any memory block. Then, Algorithm 7.3 reads and writes into the array B_i.*`BUF` *in accordance with the specification of the DynamicArray object.*

Proof. Let p be the first process that captures ownership of B_i. Then, by Lemma 77, p is the only process that writes into B_i.`BUF` at Lines 63 and 64, and that p writes into locations 0 and 1 of B_i.`BUF`. Notice that, by Lemma 81, $B_i.N = 1$ when B_i becomes active. Since B_i.`BUF` is written at Line 21 only after

$B_i.N$ is incremented at Line 20, it follows that locations 2, 3, ... are written in order. Hence, the writes into B_i.BUF are in accordance with the specification of the DynamicArray object.

Let q be any process that reads some location j of array B_i.BUF at Line 13. Then, by Invariant 5, we have (1) $B_i \in \mathcal{B}$ and (2) the jth location in B_i.BUF had already been written. Consequently, the lemma holds. □

Lemma 96 *Let $t_0 < t_1 < \ldots < t_K$ be all the times in $\mathcal{H}$ when some variable X_i is written to (by a successful CAS at Line 16). Then, for all $j \in \{0, 1, \ldots, K\}$, the value written into X_i at time t_j is of the form $(j, *)$.*

Proof. Suppose not. Let j be the smallest index such that, at time t_j, a value $k \neq j$ is written into X_i by some process p. (By initialization, we have $j \geq 1$.) Then, by the algorithm, p's CAS at time t_j is of the form CAS(X_i, $(k-1, *)$, $(k, *)$). Since X_i holds value $j-1$ at time t_j, and since $k \neq j$, it follows that p's CAS fails, which is a contradiction to the fact that p writes into X_i at time t_j. □

Lemma 97 *Let $\mathcal{O}i$ be an LL/SC object. Let t be the time when some process p reads X_i (at Line 3 or 27), and $t' > t$ the first time after t that p completes Line 4 or Line 29. Let* OP *be the latest successful SC operation on $\mathcal{O}i$ to execute Line 16 prior to time t, and v the value that* OP *writes in $\mathcal{O}i$. If there exists some process $q \in \mathcal{P}$ such that* loc_q.Help *holds value $(*, 1, *)$ throughout (t, t') and doesn't change, then p reads value v at Line 4 or Line 29 (during (t, t')).*

Proof. Let r be the process executing OP. Let B be the buffer that q owns (i.e., $B = loc_q$). Since OP is the latest successful SC operation on $\mathcal{O}i$ to execute Line 16

prior to time t, it follows that p reads from Xi at time t the value that r writes in Xi at Line 16 of OP. Therefore, p reads during (t, t') the same buffer b that r wrote v into at Line 15 of OP. Let t_1 be the time when r starts writing into b at Line 15 of OP, t_2 the time when r completes writing into b at Line 15 of OP, t_3 the time when r writes into Xi at Line 16 of OP, and t'' the time when p starts reading b during (t, t'). Then, the following claim holds.

Claim 20 *During (t_1, t_2), no process other than r writes into b. During (t_2, t'), no process writes into b.*

Proof. Suppose not. Then, either some process other than r writes into b during (t_1, t_2), or some process writes into b during (t_2, t'). In the first case, let r_1 be the process that writes into b during (t_1, t_2). Then, at some point during (t_1, t_2), we have $loc_{r_1}.mybuf = loc_r.mybuf$, which is a contradiction to Invariant 6. In the second case, let r_2 be the first process to start writing into b at some time $\tau_1 \in (t_2, t')$, and k be the index of buffer b. Then, by an earlier argument, $\tau_1 \notin (t_2, t_3)$. Furthermore, by Invariant 6 , r_2 does not write into b as long as Xi holds value $(*, k)$. Therefore, Xi changes during (t_3, τ_1).

Since Xi doesn't change during (t_3, t), it means that (1) $\tau_1 > t$ and (2) some process writes into Xi during (t, τ_1). Let r_3 be the first such process, $\tau_2 \in (t, \tau_1)$ the time when r_3 writes into Xi, SC_{r_3} the SC operation during which r_3 performs that write, and B' the memory block that r_3 owns during SC_{r_3} (i.e., $B' = loc_{r_3}$). Let τ_3 be the time when r_3 executes Line 18 of SC_{r_3}. Then, at time τ_3, r_3 enqueues k into $B'.Q$. Furthermore, by Invariant 6, r_2 does not write into b during (τ_2, τ_3), nor does it write into b during the time $B.Q$ contains value k. Therefore, we have $\tau_3 \in (\tau_2, \tau_1)$. Finally, we know that k is dequeued from $B'.Q$ during (τ_3, τ_1).

Let τ_4 be the first time after τ_3 that k is dequeued from $B'.Q$. (Notice that, by the above argument, $\tau_4 \in (\tau_3, \tau_1)$.) Then, we have the following subclaim.

Subclaim 1 *There exists some process r_5 and an execution E of Lines 24–32 by r_5 such that (1) E takes place during (τ_3, τ_4), (2) $\mathrm{loc}_{r_5} = B'$ during E, and (3) $B'.\mathrm{index} = B.\mathrm{name}$ during E.*

Proof. Let n, m, and j denote the values of $B'.N$, shared variable N, and $B'.index$, respectively, at time τ_3. Then, by Invariant 1, there are n items already in $B'.Q$ before b's index is inserted at time τ_3. So, b is not dequeued until at least $n + 1$ dequeues are performed on $B'.Q$. Notice that, each time some process $r_6 \in \mathcal{P}$, such that $loc_{r_6} = B'$, satisfies the condition at Line 19, the following holds: (1) r_6 does not dequeue an element from $B'.Q$, and (2) the values of $B'.N$ and $B'.index$ both increase by one (at Lines 20 and 33). Moreover, each time r_6 does not satisfy the condition at Line 19, the following holds: (1) r_6 dequeues an element from $B'.Q$, and (2) the value of $B'.N$ remains the same and $B'.index$ increases by one modulo $B'.N$ (at Line 33). As a result of the above two facts, the value of $B'.index$ wraps around to 0 (at Line 33) after exactly $n - j$ elements are dequeued from $B'.Q$. Let $\tau_5 > \tau_3$ be the first time after τ_3 when $B'index$ wraps around to 0, and let n' be the value of $B'.N$ at time τ_5. Notice that, since b is not dequeued until at least $n + 1$ dequeues are performed on $B'.Q$, we have $\tau_5 \in (\tau_3, \tau_4)$. By the same argument as above, at most j elements are dequeued from $B'.Q$ before $B'.index$ again reaches value j (at Line 33). Therefore, during (τ_3, τ_5), variable $B'.index$ has gone through the values $j, j+1, \ldots, n'-1, 0, 1, \ldots j-1, j$. Since B has become active prior to time τ_3, it follows by Lemma 90 that $B.name < n'$. Therefore, there exists some process r_5 and an execution E of Lines 24–32 by r_5 such that (1) E

takes place during (τ_3, τ_5), (2) $loc_{r_5} = B'$ during E, and (3) $B'.index = B.name$ during E. Hence, we have the subclaim. □

Since B.`Help` holds value $(*, 1, *)$ throughout (t, t') and doesn't change, it follows that (1) r_5 reads B at Line 24 of E (by Lemma 2), (2) r_5 satisfies the condition at Line 25 of E, and (3) r_5's CAS at Line 31 of E succeeds. This, however, is a contradiction to the fact that B.`Help` $= (*, 1, *)$ throughout (t, t'). Hence, we have the claim. □

The above claim shows that (1) during (t_1, t_2), no process other than r writes into b, and (2) during (t_2, t'), no process writes into b. Consequently, p reads v from b during (t, t'), which proves the lemma. □

Lemma 98 *Let $\mathcal{O}i$ be an LL/SC object and* OP *some LL operation on $\mathcal{O}i$. Let SC_q be the latest successful SC operation on $\mathcal{O}i$ to execute Line 16 prior to Line 3 of* OP*, and v_q the value that SC_q writes in $\mathcal{O}i$. If the CAS at Line 5 of* OP *succeeds, then* OP *returns value v_q.*

Proof. Let p be the process executing OP. Let t time when p executes Line 3 of OP, and $t' > t$ be the time when p completes Line 4 of OP. Since the CAS at Line 5 of OP succeeds, it follows by Lemma 92 that loc_p.`Help` holds value $(*, 1, *)$ throughout (t, t') and doesn't change during that time. Therefore, by Lemma 97, p reads v_q at Line 4 of OP, which proves the lemma. □

Lemma 99 *Let $\mathcal{O}i$ be an LL/SC object, and* OP *an LL operation on $\mathcal{O}i$ such that the CAS at Line 5 of* OP *fails. Let p be the process executing* OP*. Let t and t' be the times, respectively, when p executes Lines 2 and 5 of* OP*. Let x and v be the values*

that p reads at Lines 8 and 9 of OP*, respectively. Then, there exists a successful SC operation SC_q on $\mathcal{O}i$ such that (1) at some point during (t, t'), SC_q is the latest successful SC on $\mathcal{O}i$ to execute Line 16, and (2) SC_q writes x into $\mathtt{X}i$ and v into $\mathcal{O}i$.*

Proof.

Since p's CAS at time t' fails, it means that $loc_p.\mathtt{Help} = (s, 0, k)$ just prior to t'. Then, by Lemma 92, there exists a single process r that writes into $loc_p.\mathtt{Help}$ during (t, t') (at Line 31). Let $t_1 \in (t, t')$ be the time when r performs that write, and E be r's execution of Lines 24–32 during which r performs that write. Then, r's CAS at Line 31 of E (at time t_1) is of the form CAS($loc_p.\mathtt{Help}, (s, 1, *), (s, 0, *)$), for some $s > 1$. Therefore, at time t_1, $loc_p.\mathtt{Help}$ has value $(s, 1, *)$. Hence, by Lemma 92, p writes $(s, 1, *)$ into $loc_p.\mathtt{Help}$ at Line 2 of OP (at time t). Since a value of the form $(s, *, *)$ is written into $loc_p.\mathtt{Help}$ for the first time at time t, it follows that r reads $(s, 1, *)$ from $loc_p.\mathtt{Help}$ at Line 25 of E at some time $t_2 \in (t, t_1)$. Consequently, r reads variable $loc_p.\mathtt{Announce}$ at Line 26 of E at some time $t_3 \in (t_2, t_1)$. Since p writes i into $loc_p.\mathtt{Announce}$ at Line 1 of OP, it follows that r reads i from $loc_p.\mathtt{Announce}$ at time t_2. Hence, r reads $\mathtt{X}i$ at Line 27 of E.

Let t_4 be the time when r reads $\mathtt{X}i$ at Line 27 of E, t_5 the time when r starts Line 29 of E, t_6 the time when r completes Line 29 of E, and t_7 the time when r executes Line 30 of E. Let SC_q be the latest successful SC operation on $\mathcal{O}i$ to execute Line 16 prior to time t_4, x_q the value that SC_q writes in $\mathtt{X}i$, and v_q the value that SC_q writes in $\mathcal{O}i$. Then, at time t_4, r reads x_q from $\mathtt{X}i$. Furthermore, since t_1 is the first (and only) time that $loc_p.\mathtt{Help}$ is written during (t, t'), it follows

that loc_p.Help holds value $(*, 1, *)$ at all times during (t_4, t_6) and doesn't change during that time. Therefore, by Lemma 97, r reads v_q at Line 29.

Let d be the buffer that r writes v_q into during (t_5, t_6). Then, at time t_7, r writes x_q into dW. Furthermore, since r writes the index of buffer d into loc_p.Help at Line 31 of E (at time t_1), it follows that p reads buffer d at Lines 8 and 9 of OP. Let t_8 be the time when p reads dW at Line 8 of OP, t_9 the time when p starts reading d at Line 8 of OP, and t_{10} the time when p completes reading d at Line 8 of OP. Then, we show that the following claim holds.

Claim 21 *During (t_5, t_6), no process other than r writes into d, and during (t_6, t_{10}), no process writes into d.*

Proof. Suppose not. Then, either some process other than r writes into d during (t_5, t_6), or some process writes into d during (t_6, t_10). In the former case, let r_1 be the process that writes into d during (t_5, t_6). Then, at some point during (t_5, t_6), we have $loc_{r_1}.mybuf = loc_r.mybuf$, which is a contradiction to Invariant 6. In the latter case, let r_2 be the first process to write into d at some time $\tau_1 \in (t_6, t_{10})$. Then, by an earlier argument, we know that $\tau_1 \notin (t_6, t_1)$. We now show that $\tau_1 \notin (t_1, t_{10})$.

Let k' be the index of buffer d. We know by Invariant 6 that r_2 does not write into d as long as (1) loc_p.Help $= (s, 0, k')$, and (2) p is between Lines 2 and 6 of OP. Furthermore, since p sets $loc_p.mybuf$ to k' at Line 6 of OP, r_2 does not write into d after p executes Line 6 of OP and before it completes OP. Therefore, throughout (t_1, t_{10}), r_2 does not write into d. Hence, $\tau_1 \notin (t_1, t_{10})$. Since, by an earlier argument, $\tau_1 \notin (t_6, t_1)$, it follows that $\tau_1 \notin (t_6, t_{10})$. This, however, is a contradiction to the fact that r_2 writes into d during (t_6, t_{10}). □

The above claim shows that (1) during (t_5, t_6), no process other than r writes into d, and (2) during (t_6, t_{10}), no process writes into d. Consequently, p reads x_q from dW at time t_8 and v_q from d during (t_9, t_{10}). Since SC_q is the latest successful SC operation on $\mathcal{O}i$ to execute Line 16 prior to time t_4, and since $t_4 \in (t, t')$, we have the lemma. □

Lemma 100 (Correctness of LL) *Let $\mathcal{O}i$ be some LL/SC object. Let* OP *be any LL operation on $\mathcal{O}i$, and* OP′ *be the latest successful SC operation on $\mathcal{O}i$ such that LP(*OP′*) < LP(*OP*). Then,* OP *returns the value written by* OP′.

Proof. Let p be the process executing OP. We examine the following two cases: (1) the CAS at Line 5 of OP succeeds, and (2) the CAS at Line 5 of OP fails. In the first case, let SC_q be the latest successful SC operation on $\mathcal{O}i$ to execute Line 16 prior to Line 3 of OP, and v_q be the value that SC_q writes in $\mathcal{O}i$. Since all SC operations are linearized at Line 16 and since OP is linearized at Line 3, we have $\text{SC}_q = \text{OP}'$. Furthermore, by Lemma 98, OP returns value v_q. Therefore, the lemma holds in this case.

In the second case, let t and t' be the times, respectively, when p executes Lines 2 and 5 of OP. Let v be the value that p reads at Line 9 of OP. Then, by Lemma 99, there exists a successful SC operation SC_r on $\mathcal{O}i$ such that (1) at some time $t'' \in (t, t')$, SC_r is the latest successful SC on $\mathcal{O}i$ to execute Line 16, and (2) SC_r writes v into $\mathcal{O}i$. Since all SC operations are linearized at Line 16 and since OP is linearized at time t'', we have $\text{SC}_r = \text{OP}'$. Therefore, the lemma holds. □

Lemma 101 (Correctness of SC) *Let $\mathcal{O}i$ be some LL/SC object. Let* OP *be any SC operation on $\mathcal{O}i$ by some process p, and* OP′ *be the latest LL operation on*

$\mathcal{O}i$ by p prior to OP. *Then,* OP *succeeds if and only if there does not exist any successful SC operation* OP$''$ *on $\mathcal{O}i$ such that LP(*OP$'$*) < LP(*OP$''$*) < LP(*OP*).*

Proof. We examine the following two cases: (1) the CAS at Line 5 of OP$'$ succeeds, and (2) the CAS at Line 5 of OP$'$ fails. In the first case, let t_1 be the time when p executes Line 3 of OP$'$, and t_2 be the time when p executes Line 16 of OP. Then, we show that the following claim holds.

Claim 22 *Process p's CAS at time t_2 succeeds if and only if there does not exist some other SC operation on $\mathcal{O}i$ that performs a successful CAS at Line 16 during* (t_1, t_2).

Proof. Suppose that no other SC operation on $\mathcal{O}i$ performs a successful CAS at Line 16 during (t_1, t_2). Then, Xi doesn't change during (t_1, t_2), and hence p's CAS at time t_2 succeeds.

Suppose that some SC operation SC_q on $\mathcal{O}i$ does perform a successful CAS at Line 16 during (t_1, t_2). Then, by Lemma 96, Xi holds different values at times t_1 and t_2. Hence, p's CAS at time t_2 fails, which proves the claim. □

Since all SC operations are linearized at Line 16 and since OP$'$ is linearized at time t_1, it follows from the above claim that OP succeeds if and only if there does not exist some successful SC operation OP$''$ on $\mathcal{O}i$ such that LP(OP$'$) < LP(OP$''$) < LP(OP). Hence, the lemma holds in this case.

In the second case (when the CAS at Line 5 of OP$'$ fails), let t and t' be the times when p executes Lines 2 and 5 of OP$'$, respectively. Let x and v be the values that p reads at Lines 8 and 9 of OP$'$, respectively. Then, by Lemma 99, there exists a successful SC operation SC$_r$ on $\mathcal{O}i$ such that (1) at some time $t'' \in (t, t')$, SC$_r$ is

the latest successful SC on $\mathcal{O}i$ to execute Line 16, and (2) SC_r writes x into $\text{X}i$ and v into $\mathcal{O}i$. Therefore, at Line 8 of OP′, p reads the value that variable $\text{X}i$ holds at time t''. We now prove the following claim.

Claim 23 *Process p's CAS at time t_2 succeeds if and only if there does not exist some other SC operation on $\mathcal{O}i$ that performs a successful CAS at Line 16 during (t'', t_2).*

Proof. Suppose that no other SC operation on $\mathcal{O}i$ performs a successful CAS at Line 16 during (t'', t_2). Then, $\text{X}i$ doesn't change during (t'', t_2), and hence p's CAS at time t_2 succeeds.

Suppose that some SC operation SC_q on $\mathcal{O}i$ does perform a successful CAS at Line 16 during (t'', t_2). Then, by Lemma 96, $\text{X}i$ holds different values at times t'' and t_2. Hence, p's CAS at time t_2 fails, which proves the claim. □

Since all SC operations are linearized at Line 16 and since OP′ is linearized at time t_1, it follows from the above claim that OP succeeds if and only if there does not exist some successful SC operation OP″ on $\mathcal{O}i$ such that $\text{LP}(\text{OP}') < \text{LP}(\text{OP}'') < \text{LP}(\text{OP})$. Hence, the lemma holds. □

Lemma 102 (Correctness of VL) *Let $\mathcal{O}i$ be some LL/SC object. Let* OP *be any VL operation on $\mathcal{O}i$ by some process p, and* OP′ *be the latest LL operation on $\mathcal{O}i$ by p that precedes* OP. *Then,* OP *returns true if and only if there does not exist some successful SC operation* OP″ *on $\mathcal{O}i$ such that LP(*OP″*) ∈ (LP(*OP′*), LP(*OP*)).*

Proof. Similar to the proof of Lemma 101. □

Theorem 12 *Algorithm 7.3 is wait-free and implements an array $\mathcal{O}0..M-1$ of M linearizable W-word LL/SC objects. The time complexity of LL, SC and VL operations on $\mathcal{O}$ are $O(W)$, $O(W)$ and $O(1)$, respectively. The time complexity of Join and Leave operations is $O(K)$ and $O(1)$, respectively, where K is the maximum number of processes that have simultaneously participated in the algorithm. The space complexity of the implementation is $O(Kk + (K^2 + M)W)$, where k is the maximum number of outstanding LL operations of a given process.*

Proof. The theorem follows immediately from Lemmas 100, 101, and 102. □

A.4.3 Proof of Algorithm 7.4

Let $\mathcal{H}$ be any execution history of Algorithm 7.4. We say an SC operation is *successful* if the CAS operation at Line 19 or 20 of that SC succeeds; otherwise, we say an SC operation has *failed*. We say an SC operation by some process p is *first-successful* if it is the first successful SC operation by p in $\mathcal{H}$.

In the rest of this section, let b denote the number of bits in X reserved for the sequence number. We show Algorithm 7.4 is correct, under the following assumption:

Assumption A: During the time interval when some process p executes one LL/SC pair, no other process q performs more than $2^b - 3$ successful SC operations.

Lemma 103 *Let* OP *be an SC operation in $\mathcal{H}$ by some process p. Then, p satisfies the condition at Line 18 of* OP *if and only if p didn't perform a successful SC prior to* OP.

Proof. If p didn't perform a successful SC operation prior to OP, it means that p never satisfied the condition at Line 20 prior to OP. Therefore, $first_p$ has the initializing value of *true* at the start of OP, and hence p satisfies the condition at Line 18 of OP.

If p did perform a successful SC operation prior to OP, let OP′ be the first successful SC operation by p in $\mathcal{H}$. Then, by the above argument, p satisfies the condition at Line 18 OP′. Furthermore, since OP′ is successful, p satisfies the condition at Line 20 as well. Therefore, p writes *false* into $first_p$ at Line 21 of OP′. Since p never writes *true* into $first_p$ (except during initialization), it follows that $first_p$ has value *false* at all times after OP′. Hence, p does not satisfy the condition at Line 18 of OP. □

Corollary 3 *Once a process writes a value of the form* $(1, *)$ *into* X*, it never writes another value of the form* $(1, *)$ *into* X *again.*

Corollary 4 *Let* OP *be a first-successful SC operation in* $\mathcal{H}$ *by some process* p*. Let* k *be the value that* p *reads from* N *at Line 19 (by Lemma 103) of* OP*. Then, at all times after Line 19 of* OP *completes, we have* $\mathrm{loc}_p.\texttt{name} = k$.

Lemma 104 *Let* $t_1 < t_2 < \ldots < t_m$ *be all the times in* $\mathcal{H}$ *that variable* N *is written. Then, for all* $i \in \{1, 2, \ldots, m\}$*, the value written into* N *at time* t_i *is* i.

Proof. Suppose not. Let j be the smallest index such that, at time t_j, a value different than j is written into N. Let k be that value and p the process who wrote it. Then, p performed the write with a CAS at Line 6 and that CAS must have been of the form $\texttt{CAS}(\texttt{N}, k-1, k)$. Since N holds a value $j-1$ at time t_j, and since

$k - 1 \neq j - 1$, it follows that p's CAS fails, which is a contradiction to the fact that p writes into N at time t_j. □

In the following, we assume that the initializing step was performed by some process during its first-successful SC, and that the memory block allocated during the initialization belongs to that process.

Lemma 105 *Let $SC_0, SC_1, \ldots, SC_M$ be the sequence of all first-successful SC operations in $\mathcal{H}$, ordered by the time they write into* X*. Let p_i, for all $i \in \{0, 1, \ldots, M\}$, be the process executing SC_i, t_i be the time when p_i writes into* X *during SC_i (at Line 20), and $t'_i < t_i$ be the time when p_i reads* N *at Line 19 of SC_i. (If $i = 0$, then $t'_i = t_i = 0$.) Let SC'_i, for all $i \in \{0, 1, \ldots, M\}$, be the first successful SC operation to write into* X *after SC_i, and t''_i be the beginning of the execution of SC'_i. Let p'_i, for all $i \in \{0, 1, \ldots, M\}$, be the process executing SC'_i, and LL'_i be the latest LL operation by p'_i prior to SC'_i. Then, for all $i \in \{0, 1, \ldots, M\}$, we have (1) $t''_i > t_i$, (2) LL'_i is executed entirely during the time interval (t_i, t''_i), and (3) during (t'_i, t''_i), variable* N *is written at least once.*

Proof. Let j be any index in $\{0, 1, \ldots, M\}$. Then, by Lemma 103 and initialization, p_j writes $(1, \&loc_{p_j})$ into X at time t_j. Let t be the time when SC'_j writes into X. Since SC'_j is the first successful SC operation to write into X after time t_j, it means that X has value $(1, \&loc_{p_j})$ throughout (t_j, t). Furthermore, since SC'_j is successful, the value that p'_j reads from X at Line 1 of LL'_j is $(1, \&loc_{p_j})$. By Corollary 3, t_j is the only time during $\mathcal{H}$ that value $(1, \&loc_{p_j})$ is written into X. Consequently, p'_j performs a read at Line 1 of LL'_j *after* time t_j, and hence we have (1) $t''_j > t_j$, and (2) LL'_j is executed entirely during the time interval (t_j, t''_j).

We now show that N is written at least once during (t'_j, t''_j). Suppose not. Let c be the value that p_j reads from N at time t'_j. Then, throughout (t'_j, t''_j), N has value c. Notice that, during (t'_j, t_j), p_j writes c into loc_{p_j}.name (at Line 19). Furthermore, by Corollary 4, loc_{p_j}.name has value c at all times after t_j. Since p'_j reads $(1, \&loc_{p_j})$ from X at Line 1 of LL'_j after time t_j, it means that p'_j reads c from loc_{p_j}.name at Line 6 of LL'_j. Therefore, the CAS at Line 6 of LL'_j is of the form CAS(N, c, $c+1$). Since p'_j executes that CAS during (t_j, t''_j), and since N has value c throughout (t_j, t''_j), it follows that p'_j performs a successful CAS at Line 6 of LL'_j, which is a contradiction to the fact that N is not written during (t'_j, t''_j). Hence, we have the lemma. □

Lemma 106 *Let $SC_0, SC_1, \ldots, SC_M$ be the sequence of all first-successful SC operations in $\mathcal{H}$, ordered by the time they write into* X. *Let t_i, for all $i \in \{0, 1, \ldots, M\}$, be the time when SC_i writes into* X. *Let p_i, for all $i \in \{0, 1, \ldots, M\}$, be the process executing SC_i, and c_i be the value that p_i reads from* N *at Line 19 of SC_i. Then, we have $c_1 \neq c_2 \neq \ldots \neq c_M$.*

Proof. Suppose not. Then, j be the smallest index such that, when SC_j writes into X at time t_j, we have $c_j = c_i$, for some $i < j$. Let t'_j be the time when SC_j begins its execution. Let OP be the first successful SC operation to write into X after SC_i, t be the time when OP begins its execution, and t' be the time when OP writes into X. Let $t'_i < t_i$ be the time when p_i reads N at Line 19 of SC_i. (If $i = 0$, then $t'_i = t_i = 0$.) Then, by Lemma 105, we have (1) $t > t_i$, and (2) variable N is written during (t'_i, t). Since we have $c_j = c_i$, it follows by Lemma 104 that (1) $SC_j \neq$OP, and (2) $t'_j < t$. Furthermore, by definition of OP, we have $t' \in (t, t_j)$.

Let LL_j be the latest LL operation by p_j prior to SC_j. Let t''_j be the time

when p_j reads X at Line 1 of LL_j, and v_j the value that it reads from X. Then, by Corollary 3 and the fact that $t' \in (t''_j, t_j)$, we have $v_j \neq (1, *)$. Therefore, it follows that (1) $t''_j < t_i$, and (2) $v_j = (0, k, s)$, for some k and s.

Notice that, by definition of j, if a value $(0, k, *)$ is written into X two or more times during $(0, t_j)$, then all those writes are performed by the same process. Let q be the process that writes v_j into X. Let τ be the latest time prior to t''_j that q writes v_j into X, and OP′ be the SC operation by q during which it performs that write. Then, since SC_j is successful, it means that X holds value v_j at time t_j. Furthermore, since $t_i \in (t''_j, t_j)$, value v_j is written into X at some time $\tau' \in (t_i, t_j)$.

Notice that, at times τ and τ', we have $seq_q = s$. Since, during OP′, q increments seq_q by one (at Line 24), it follows that during (τ, τ'), seq_q is incremented at least 2^b times. Therefore, q performs at least $2^b - 1$ successful SC operations during (τ, t_j). Since OP′ is the latest successful SC operation to write into X prior to time t''_j, it follows that q performs at least $2^b - 1$ successful SC operations during (t''_j, t_j). Hence, during p_j's LL/SC pair consisting of operations LL_j and SC_j, process q performs at least $2^b - 1$ successful SC operations, which is a contradiction to Assumption A. □

Lemma 107 *Let $SC_0, SC_1, \ldots, SC_K$ be the sequence of all successful SC operations in $\mathcal{H}$, ordered by the time they write into* X*. Let t_i, for all $i \in \{0, 1, \ldots, K\}$, be the time when SC_i writes into* X*, and p_i be the process executing SC_i. Let LL_i, for all $i \in \{1, 2, \ldots, K\}$, be the latest LL operation by p_i prior to SC_i. Then, for all $i \in \{1, 2, \ldots, K\}$, LL_i is executed entirely during (t_{i-1}, t_i).*

Proof. Let j be some index in $\{1, 2, \ldots, K\}$. Let t'_j be the time when LL_j starts

(i.e., when p_j executes Line 1 of LL_j). Then, we show that $t'_j > t_{j-1}$. Suppose not. Then, let v_j be the value that LL_j reads from X at time t'_j. By Corollary 3 and the fact that $t_{j-1} \in (t'_j, t_j)$, we have $v_j \neq (1, *)$. Consequently, $v_j = (0, k, s)$, for some k and s.

We know by Lemma 106 that if a value $(0, k, *)$ is written into X two or more times during $\mathcal{H}$, then all those writes are performed by the same process. Let q be the process that writes v_j into X. Let τ be the latest time prior to t'_j that q writes v_j into X, and OP′ be the SC operation by q during which it performs that write. Then, since SC_j is successful, it means that X holds value v_j at time t_j. Furthermore, since $t_{j-1} \in (t'_j, t_j)$, value v_j is written into X again at some time $\tau' \in (t_{j-1}, t_j)$.

Notice that, at times τ and τ', we have $seq_q = s$. Since, during OP′, q increments seq_q by one (at Line 24), it follows that during (τ, τ'), seq_q is incremented at least 2^b times. Therefore, q performs at least $2^b - 1$ successful SC operations during (τ, t_j). Since OP′ is the latest successful SC operation to write into X prior to time t'_j, it follows that q performs at least $2^b - 1$ successful SC operations during (t'_j, t_j). Hence, during p_j's LL/SC pair consisting of operations LL_j and SC_j, process q performs at least $2^b - 1$ successful SC operations, which is a contradiction to Assumption A. □

Lemma 108 *Let $SC_0, SC_1, \ldots, SC_M$ be the sequence of all first-successful SC operations in $\mathcal{H}$, ordered by the time they write into* X*. Let t_i, for all $i \in \{0, 1, \ldots, M\}$, be the time when SC_i writes into* X*. Let p_i, for all $i \in \{0, 1, \ldots, M\}$, be the process executing SC_i, and c_i be the value that p_i reads from* N *at Line 19 of SC_i. Then, we have $c_1 < c_2 < \ldots < c_M$.*

Proof. Let j be some index in $\{0, 1, \ldots, M-1\}$. Let $t < t_j$ be the time when p_j

reads N at Line 19 of SC_j. (If $j = 0$, then $t = 0$.) Then, by Lemma 104, the value of N is greater than or equal to c_j at all times after t. Since, by Lemma 107, SC_{j+1} starts after time t_j, and since $c_{j+1} \neq c_j$ (by Lemma 106), we have $c_{j+1} > c_j$, which proves the lemma. □

Lemma 109 *Let $SC_0, SC_1, \ldots, SC_M$ be the sequence of all first-successful SC operations in $\mathcal{H}$, ordered by the time they write into* X. *Let t_i, for all $i \in \{0, 1, \ldots, M\}$, be the time when SC_i writes into* X. *Let p_i, for all $i \in \{0, 1, \ldots, M\}$, be the process executing SC_i, and c_i be the value that p_i reads from* N *at Line 19 of SC_i. Then, we have $c_i = i$, for all $i \in \{0, 1, \ldots, M\}$.*

Proof. Suppose not. Let j be the smallest index such that $c_j \neq j$. (By initialization, $j > 0$). Then, by Lemma 108, $c_j > j$. Let t'_{j-1} and t'_j be the times when operations SC_{j-1} and SC_j, respectively, read the N at Line 19. (If $j = 1$, let $t'_{j-1} = 0$). Then, variable N is written at least two times during (t'_{j-1}, t'_j). Furthermore, by Lemma 107, we have $t'_j \in (t_{j-1}, t_j)$.

Let τ_1 and τ_2 be the first two times that N is written during (t'_{j-1}, t'_j). Let LL_1 and LL_2 be the LL operations that write into N at times τ_1 and τ_2, respectively. Then, the CAS at Line 6 of LL_1 and LL_2 is of the form CAS(N, $j-1$, j) and CAS(N, j, $j+1$), respectively. Let $t < t'_j$ be the time when LL_2 reads X at Line 1, and v be the value that it reads from X. Then, since LL_2 executes Line 6, it follows that $v = (1, l)$, for some value l. Let $t' < t$ be the time when value $(1, l)$ is written into X, and OP be the operation that performs that write. Then, OP $= SC_k$, for some $k < j$. Therefore, by definition of j, OP reads k from N at some time $t'' < t'$ (at Line 19), and writes k into $l \rightarrow$ name at some time $t''' \in (t'', t')$. (If $t' = 0$, then $t''' = t'' = t' = 0$, and $k = 0$.) Furthermore, by Corollary 4, $l \rightarrow$ name $= k$ at all

times after t'''. Therefore, LL_2 reads k from $l\rightarrow$`name`, which is a contradiction to the fact that the CAS at Line 6 of LL_2 is of the form `CAS(N,` j, $j+1$). □

Lemma 110 *Let $SC_0, SC_1, \ldots, SC_M$ be the sequence of all first-successful SC operations in $\mathcal{H}$, ordered by the time they write into* X. *Let t_i, for all $i \in \{0, 1, \ldots, M\}$, be the time when SC_i writes into* X. *Let p_i, for all $i \in \{0, 1, \ldots, M\}$, be the process executing SC_i. Let* OP *be any LL operation in $\mathcal{H}$. If* OP *reads $(1, l)$ from* X *at Line 1, then we have (1) $l = \&\text{loc}_{p_k}$, for some $k \in \{0, 1, \ldots, M\}$, and (2)* OP *executes* `da_write`$(k, \&\text{loc}_{p_k})$ *at Line 5.*

Proof. The first part of the lemma follows immediately from the algorithm. We now prove the second part of the lemma. Let $t'_k < t_k$ be the time when SC_k reads N at Line 19 of OP′. (If $k = 0$, then $t' = 0$.) Then, during (t'_k, t_k), p_k writes k into loc_{p_k}.name (by Lemma 109). Consequently, by Lemma 4, at all times after t_k, loc_{p_k}.name holds value k.

Let t be the time when OP executes Line 1. Then, by Corollary 3, we have $t > t_k$. Therefore, OP reads k from loc_{p_k}.name at Line 5, and hence executes `da_write`$(k, \&loc_{p_k})$ at Line 5, which proves the lemma □

Lemma 111 *Let $SC_0, SC_1, \ldots, SC_M$ be the sequence of all first-successful SC operations in $\mathcal{H}$, ordered by the time they write into* X. *Let p_i, for all $i \in \{0, 1, \ldots, M\}$, be the process executing SC_i, and t_i be the time when p_i writes into* X *during SC_i. Let SC'_i, for all $i \in \{0, 1, \ldots, M\}$, be the first successful SC operation to write into* X *after SC_i, and t'_i be the beginning of the execution of SC'_i. Then, at least one* `da_write`$(i, \&\text{loc}_{p_i})$ *operation is executed entirely during (t_i, t'_i).*

Proof. Let j be any index in $\{0, 1, \ldots, M\}$. Let, p'_j be the process executing SC'_j. Let LL'_j be the latest LL operation by p'_j prior to SC'_j. Then, by Lemma 107, LL'_j is executed entirely during (t_j, t'_j). Furthermore, since X doesn't change during (t_j, t'_j), LL'_j reads $(1, \&loc_{p_j})$ from X at Line 1. Consequently, by Lemma 110, LL'_j executes da_write$(j, \&loc_{p_j})$ at Line 5, which proves the lemma. □

Lemma 112 *Let $SC_0, SC_1, \ldots, SC_M$ be the sequence of all first-successful SC operations in $\mathcal{H}$, ordered by the time they write into* X. *Let t_i, for all $i \in \{0, 1, \ldots, M\}$, be the time when SC_i writes into* X. *Let p_i, for all $i \in \{0, 1, \ldots, M\}$, be the process executing SC_i. Let* OP *be any LL operation in $\mathcal{H}$. If* OP *reads $(0, k, *)$ from* X *at Line 1, then we have (1) $k \in \{0, 1, \ldots, M\}$, and (2) at least one* da_write$(k, \&\text{loc}_{p_k})$ *operation is executed entirely before the start of* OP.

Proof. Let OP′ be the latest successful SC operation prior to OP. Then, OP′ writes $(0, k, *)$ into X at Line 22. Let q be the process executing OP′, and t be the time when OP′ performs that write. Then, at time t, we have loc_q.name $= k$.

Let OP″ be the first successful SC operation by q in $\mathcal{H}$. Let t' be the time when q reads N at Line 19 of OP″. Then, since we have loc_q.name $= k$ at time t, it follows by Lemma 4 that q reads k from N at time t'. Therefore, by Lemma 109, we have (1) $q = p_k$, (2) OP″ $= SC_k$, and (3) $k \in \{0, 1, \ldots, M\}$. Then, by Lemma 111, it follows that at least one da_write$(k, \&loc_{p_k})$ operation is executed between SC_k and OP′. Since OP′ starts before OP starts, we have the lemma. □

Lemma 113 *Let $SC_0, SC_1, \ldots, SC_M$ be the sequence of all first-successful SC operations in $\mathcal{H}$, ordered by the time they write into* X. *Then, the algorithm performs*

read and write operation on $\mathcal{D}$ in accordance with the specification of DynamicArray object. Furthermore, for any `da_write`$(k, *)$ *or* `read`(k) *operation, we have* $k \in \{0, 1, \ldots, M\}$.

Proof.

Let $SC_0, SC_1, \ldots, SC_M$ be the sequence of all first-successful SC operations in $\mathcal{H}$, ordered by the time they write into X. Let t_i, for all $i \in \{0, 1, \ldots, M\}$, be the time when SC_i writes into X. Let p_i, for all $i \in \{0, 1, \ldots, M\}$, be the process executing SC_i. Then, by Lemma 110, any LL operation that reads $(1, \&loc_{p_i})$ from X at Line 1 executes `da_write`$(i, \&loc_{p_i})$ at Line 5, for all $i \in \{0, 1, \ldots, M\}$. Since no other LL operations invoke `da_write` at Line 5, it means that all the values written into the same location are the same. Furthermore, for any `da_write`$(k, *)$ operation, we have $k \in \{0, 1, \ldots, M\}$.

Notice that, by Corollary 3, t_i is the first time that value $(1, \&loc_{p_i})$ is written into X, for all $i \in \{0, 1, \ldots, M\}$. Then, by the above argument, `da_write`$(i, *)$ is not invoked until time t_i, for all $i \in \{0, 1, \ldots, M\}$. Let t'_i, for all $i \in \{0, 1, \ldots, M\}$, be the time when SC_i begins its execution. Then, by Lemma 111, at least one `da_write`$(i, *)$ operation is executed entirely during (t_i, t'_{i+1}), for all $i \in \{0, 1, \ldots, M-1\}$. Consequently, before `da_write`$(i+1, *)$ operation is invoked, at least one `da_write`$(i, *)$ operation completes, for all $i \in \{0, 1, \ldots, M-1\}$.

Notice that, by the algorithm, `da_read`(k) is invoked by an LL operation OP only after OP reads $(0, k, *)$ from X at Line 1. Then, by Lemma 112, at least one `da_write`$(k, *)$ operation completes before the start of `da_read`(k). Furthermore, we have $k \in \{0, 1, \ldots, M\}$. Therefore, the lemma holds. □

In the following, let $\mathrm{SC}_{p,i}$ denote the ith *successful* SC by process p, and $v_{p,i}$ denote the value written in $\mathcal{O}$ by $\mathrm{SC}_{p,i}$. The operations are linearized according to the following rules. We linearize each SC operation at the instant it attempts to write into X (either at Line 20 or Line 22). We linearize each VL at Line 29. Let OP be any execution of the LL operation by p. The linearization point of OP is determined by two cases. If OP returns at Line 11, then we linearize OP at Line 1. Otherwise, let $\mathrm{SC}_{q,k}$ be the latest successful SC operation to write into X prior to Line 1 of OP, and let v' be the value that p reads at Line 12 of OP. We show that there exists some $i \geq k$ such that (1) at some time t during the execution of OP, $\mathrm{SC}_{q,i}$ was the latest successful SC operation to write into X, and (2) $v' = v_{q,i}$. Then, we linearize OP at time t.

Claim 24 *At the beginning of* $SC_{p,i}$, seq_p *holds the value* $i \bmod 2^b$.

Proof. Prior to $\mathrm{SC}_{p,i}$, exactly $i-1$ successful SC operations are performed by p. Therefore, variable seq_p was incremented exactly $i-1$ times prior to $\mathrm{SC}_{p,i}$. Since seq_p is initialized to 1, it follows that at the beginning of $\mathrm{SC}_{p,i}$, seq_p holds the value $i \bmod 2^b$. □

Claim 25 *During* $SC_{p,i}$, *p writes* $(i-1) \bmod 2^b$ *into* loc_p.`oldseq` *at Line 25.*

Proof. According to Claim 24, at the beginning of $\mathrm{SC}_{p,i}$, seq_p holds the value $i \bmod 2^b$. Therefore, p writes $(i-1) \bmod 2^b$ into loc_p.`oldseq` at Line 25. □

Claim 26 *From the moment p performs Line 17 of* $SC_{p,i}$, *until p completes* $SC_{p,i+1}$, loc_p.`val`$i \bmod 2$ *holds value* $v_{p,i}$.

Proof. According to Claim 24, at the beginning of $\mathrm{SC}_{p,i}$, seq_p holds value $i \bmod 2^b$. Since $(i \bmod 2^b) \bmod 2 = i \bmod 2$, it follows that p writes $v_{p,i}$ into $loc_p.\texttt{val}i \bmod 2$ at Line 17 of $\mathrm{SC}_{p,i}$. Furthermore, since p increments seq_p at Line 26 of $\mathrm{SC}_{p,i}$, value $v_{p,i}$ (in $loc_p.\texttt{val}i \bmod 2$) will not be overwritten until seq_p reaches value $(i+2) \bmod 2^b$, which in turn will not happen until p executes Line 26 of $\mathrm{SC}_{p,i+1}$. Therefore, variable $loc_p.\texttt{val}i \bmod 2$ holds value $v_{p,i}$ from the moment p performs Line 17 of $\mathrm{SC}_{p,i}$, until p completes $\mathrm{SC}_{p,i+1}$. □

Claim 27 *During $SC_{p,i}$, p writes $v_{p,i-1}$ into* $\mathrm{loc}_p.\texttt{oldval}$ *at Line 24.*

Proof. By Claim 26, $loc_p.\texttt{val}(i-1) \bmod 2$ holds the value $v_{p,i-1}$ at all times during $\mathrm{SC}_{p,i}$. As a result, p writes $v_{p,i-1}$ into $loc_p.\texttt{oldval}$ at Line 24 of $\mathrm{SC}_{p,i}$. □

Claim 28 *Let* OP *be an LL operation by process p, and $SC_{q,k}$ the latest successful SC operation to write into* X *prior to Line 1 of* OP. *If* OP *terminates at Line 11, then it returns the value $v_{q,k}$.*

Proof. Let $\mathcal{I}$ be the time interval starting from the moment q performs Line 17 of $\mathrm{SC}_{q,k}$ until q completes $\mathrm{SC}_{q,k+1}$. According to Claim 26, variable $loc_q.\texttt{val}k \bmod 2$ holds value $v_{q,k}$ at all times during $\mathcal{I}$. Furthermore, if $k = 1$, q writes $(1, \&loc_q)$ into X at Line 20 of $\mathrm{SC}_{q,k}$; otherwise, q writes $(0, loc_q.\texttt{name}, k \bmod 2^b)$ into X at Line 22 of $\mathrm{SC}_{q,k}$. In either case, p reads $k \bmod 2^b$ at either Line 4 of OP or at Line 8 of OP. Furthermore, by Lemmas 112 and 113, p reads $\&loc_q$ at either Line 3 of OP or at Line 7 of OP. Consequently, since $(k \bmod 2^b) \bmod 2 = k \bmod 2$, p reads $loc_q.\texttt{val}k \bmod 2$ at Line 9 of OP. Our goal is to show that p executes Line 9 of OP during $\mathcal{I}$, and therefore reads $v_{q,k}$ from $loc_q.\texttt{val}k \bmod 2$.

At the moment when p reads X at Line 1 of OP, q must have already executed Line 17 of $\mathrm{SC}_{q,k}$ and not yet executed Line 22 of $\mathrm{SC}_{q,k+1}$. Hence, p executes Line 1 during $\mathcal{I}$. From the fact that OP terminates at Line 11, it follows that p satisfies the condition at Line 11. Therefore, the value that p reads from loc_q.oldseq at Line 10 of OP is either $(k-2) \bmod 2^b$ or $(k-1) \bmod 2^b$. Hence, by Claim 25 and Assumption A, it follows that, when p performs Line 10, q did not yet complete $\mathrm{SC}_{q,k+1}$. So, p executes Line 10 during $\mathcal{I}$. Since p performed both Lines 1 and 10 during $\mathcal{I}$, it follows that p performs Line 9 during $\mathcal{I}$ as well. Therefore, by Claim 26, p reads $v_{q,k}$ from loc_q.val$k$ mod 2, and the value that OP returns is $v_{q,k}$. □

Claim 29 *Let* OP *be an LL operation by process p, and $SC_{q,k}$ be the latest successful SC operation to write into* X *prior to Line 1 of* OP. *If* OP *terminates at Line 13, let v' be the value that p reads at Line 12 of* OP. *Then, there exists some $i \geq k$ such that (1) at some time t during the execution of* OP, *$SC_{q,i}$ is the latest successful SC operation to write into* X, *(2) $SC_{q,i+1}$ writes into* X *during* OP, *and (3) $v' = v_{q,i}$.*

Proof. If $k = 1$, then q writes $(1, \&loc_q)$ into X at Line 20 of $\mathrm{SC}_{q,k}$. Otherwise, q writes $(0, loc_q.\mathtt{name}, k \bmod 2^b)$ into X at Line 22 of $\mathrm{SC}_{q,k}$. In either case, p reads $k \bmod 2^b$ at either Line 4 of OP or at Line 8 of OP. Furthermore, by Lemmas 112 and 113, p reads $\&loc_q$ at either Line 3 of OP or at Line 7 of OP.

Since OP terminates at Line 13, the condition at Line 11 of OP doesn't hold. Therefore, p reads a value different than $(k-1) \bmod 2^b$ and $(k-2) \bmod 2^b$ from loc_q.oldseq at Line 10 of OP. Then, by Claim 25, q completes Line 25 of $\mathrm{SC}_{q,k+1}$ before p performs Line 10 of OP. Consequently, q completes Line 24 of $\mathrm{SC}_{q,k+1}$ before p performs Line 12 of OP. As a result, the value v' that p reads at Line 12

of OP was written by q (into loc_q.oldval at Line 24) in either $SC_{q,k+1}$ or a later SC operation by q. We examine two cases: either v' was written by $SC_{q,k+1}$, or it was written by $SC_{q,i}$, for some $i \geq k+2$.

In the first case, by Claim 27, we have $v' = v_{q,k}$. Furthermore, by definition of $SC_{q,k}$, at the time when p executes Line 1 of OP, $SC_{q,k}$ is the latest successful SC to write into X. Finally, by definition of $SC_{q,k}$ and an earlier argument, $SC_{q,k+1}$ writes into X between Lines 1 and 5 of OP. Hence, the lemma holds in this case.

In the second case, by Claim 27, we have $v' = v_{q,i-1}$. Since, at the time when p executes Line 1 of OP, $SC_{q,k}$ is the latest successful SC to write into X, it means that $SC_{q,i-1}$ and $SC_{q,i}$ did not write into X prior to Line 1 of OP. Furthermore, since $SC_{q,i}$ had executed Line 24 prior to Line 12 of OP, it follows that $SC_{q,i-1}$ and $SC_{q,i}$ had written into X prior to Line 12 of OP. Consequently, $SC_{q,i-1}$ and $SC_{q,i}$ had written into X during OP, which proves the lemma. □

Lemma 114 (Correctness of LL) *Let* OP *be any LL operation, and* OP′ *be the latest successful SC operation such that LP(*OP′*) < LP(*OP*). Then,* OP *returns the value written by* OP′.

Proof. Let p be the process executing OP. Let $SC_{q,k}$ be the latest successful SC operation to write into X prior to Line 1 of OP. We examine the following two cases: (1) OP returned at Line 11, and (2) OP returned at Line 13. In the first case, since all SC operations are linearized at the instant they attempt to write into X, and since OP is linearized at Line 1, we have $SC_{q,k} =$ OP′. Furthermore, by Claim 28, OP returns the value written by $SC_{q,k}$. Therefore, the lemma holds. In the second case, by Claim 29, there exists some $i \geq k$ such that (1) at some time t during the execution of OP, $SC_{q,i}$ is the latest successful SC operation to write into X, and (2)

OP returns $v_{q,i}$. Since all SC operations are linearized at the instant they attempt to write into X, and since OP is linearized at time t, it follows that $\text{SC}_{q,i} = \text{OP}'$. Therefore, the lemma holds in this case as well. □

Lemma 115 (Correctness of SC) *Let* OP *be any SC operation by some process p, and* OP′ *be the latest LL operation by p that precedes* OP. *Then,* OP *succeeds if and only if there does not exist any successful SC operation* OP″ *such that* $LP(\text{OP}') < LP(\text{OP}'') < LP(\text{OP})$.

Proof. Suppose that OP returned *false*. Let $\text{SC}_{q,k}$ be the latest successful SC operation to write into X prior to Line 1 of OP′. Let t be the time when p executes Line 1 of OP′, and t' be the time when p performs the CAS at Line 20 or 22 of OP. We examine the following two cases: (1) OP′ returned at Line 11, and (2) OP′ returned at Line 13. In the first case, from the fact that OP returned *false*, it follows that the CAS at Line 20 or Line 22 of OP has failed. Then, during (t, t'), variable X has changed. Consequently, some successful SC operation OP″ writes into X during (t, t'). Since all SC operations are linearized at the instant they attempt to write into X, and since OP′ is linearized at Line 1, it follows that $LP(\text{OP}') < LP(\text{OP}'') < LP(\text{OP})$, and OP is therefore correct to return *false*.

In the second case, by Claim 29, there exists some $i \geq k$ such that (1) at some time t during the execution of OP′, $\text{SC}_{q,i}$ is the latest successful SC operation to write into X, (2) $\text{SC}_{q,i+1}$ writes into X during OP′, and (3) $v' = v_{q,i}$. Since OP′ is linearized at time t and since all SC operations are linearized at the instant they attempt to write into X, we have $LP(\text{OP}') < LP(SC_{q,i+1}) < LP(\text{OP})$. Hence, OP is correct to return *false*.

If OP returned *true*, let t'' be the time when OP writes into X. Let t''' be the latest time prior to t'' that some successful SC operation writes into X. Then, by Lemma 107, OP$'$ is executed entirely during (t''', t''). Since all SC operations are linearized at the instant they attempt to write into X, it follows that no successful SC is linearized between $LP(\text{OP}')$ and $LP(\text{OP})$ (regardless as to where OP$'$ is linearized). Hence, OP is correct to return *true*. □

Lemma 116 (Correctness of VL) *Let* OP *be any VL operation by some process p, and* OP$'$ *be the latest LL operation by p that precedes* OP. *Then,* OP *returns true if and only if there does not exist any successful SC operation* OP$''$ *such that* $LP(\text{OP}') < LP(\text{OP}'') < LP(\text{OP})$.

Proof. Similar to the proof of Lemma 115. □

Theorem 13 *Algorithm 7.4 is wait-free and, under Assumption A, implements a linearizable* 64*-bit LL/SC object from* 64*-bit CAS objects and registers. The time complexity of Join, LL, SC, and VL is* $O(1)$. *The space complexity of the algorithm is* $O(K^2 + KM)$, *where* K *is the total number of processes that have joined the algorithm and* M *is the number of implemented LL/SC objects.*

Proof. This theorem follows immediately from Lemmas 114, 115, and 116. □