A Practical Wait-Free Simulation for Lock-Free Data Structures

Erez Petrank
Dept. of Computer Science, Technion, Israel.
erez@cs.technion.ac.il

Shahar Timnat
Dept. of Computer Science, Technion, Israel.
stimnat@cs.technion.ac.il

March 7, 2017

Abstract

Lock-free data structures guarantee overall system progress, whereas wait-free data structures guarantee the progress of each and every thread, providing the desirable non-starvation guarantee for concurrent data structures. While practical lock-free implementations are known for various data structures, wait-free data structure designs are rare. In spite of the need for wait-free algorithms in practice, wait-free implementations have been notoriously hard to design and often inefficient. Recently, a couple of efficient wait-free data structures appeared in the literature, but designing each new such algorithm is difficult and error-prone.

In this work we present a mechanical transformation of lock-free data structures to wait-free ones allowing even a non-expert to transform a lock-free data-structure into a practical wait-free one. The transformation requires that the lock-free data structure is given in a normalized form defined in this work. Using the new method, we have designed and implemented wait-free linked-list, skiplist, and tree and we measured their performance. It turns out that for all these data structures the wait-free implementations are only a few percent slower than their lock-free counterparts.

Keywords: concurrent data structures, wait-freedom, lock-freedom.

1 Introduction

In today's world, where nearly every desktop and laptop contains several cores, parallel computing has become the standard. Concurrent data structures are designed to utilize all available cores to achieve faster performance. One of the important properties of concurrent data structures is the progress guarantee they provide. Typically, the stronger the progress guarantee is, the harder it is to design the algorithm, and often, stronger progress guarantees come with a higher performance cost.

Standard progress guarantees include *obstruction-freedom*, *lock-freedom* (a.k.a. *non-blocking*), and *wait-freedom*. The strongest among these is wait-freedom. A wait-free algorithm guarantees that *every* thread makes progress (typically completing a method) in a bounded number of steps, regardless of other threads' behavior. This worst-case guarantee has its theoretical appeal and elegance, but is also critical in practice for making concurrent data structures useable with real-time systems. Even when run on a real-time platform and operating system, a concurrent application must ensure that each thread makes its deadlines, i.e., has a bounded worst-case response time in worst-case scenarios. However, very few wait-free algorithms are known, as they are considered notoriously hard to design, and largely inefficient. The weaker lock-freedom guarantee is more common. A lock-free algorithm guarantees that at least *one* thread makes progress in a bounded number of steps. The downside of the lock-free guarantee is that all threads but one can starve in an execution, meaning that lock-freedom cannot suffice for a real-time scenario. As lock-free data structures are easier to design, constructions for many lock-free data structures are available in the literature, including the stack [16], the linked-list [13], the skiplist [16], and the binary search tree [7]. Furthermore, practical implementations for many lock-free algorithms are readily available in standard Java libraries and on the Web.

The existence of wait-free data structures has been shown by Herlihy [14] using universal simulations. Universal simulation techniques have evolved dramatically since then (e.g., [15, 4, 1, 12, 8, 5, 6]), but even the state of the art universal construction [5] is too slow compared to the lock-free or lock-based implementations and cannot be used in practice[1]. Universal constructions achieve a difficult task, as they go all the way from a sequential data structure to a concurrent wait-free implementation of it. It may therefore be hard to expect that the resulting wait-free algorithm will be efficient enough to become practicable.

The idea of mechanically transforming an algorithm to provide a practical algorithm with a different progress guarantee is not new, and not limited to universal constructions. Taubenfeld introduced contention-sensitive data structures (CSDS) and proposed various mechanical transformation that enhance their performance of progress guarantees [23]. Ellen et al suggested a transformation of obstruction-free algorithms into wait-freedom algorithms under a different computation model known as semisynchronous [10]. This construction does not extend to the standard asynchronous model.

Recently, we have seen some progress with respect to practical wait-free data structures. A practical design of a wait-free queue relying on *compare and swap* (CAS) operations was presented in [17]. Next, an independent construction of a wait-free stack and queue appeared in [9]. A wait-free algorithm for the linked-list has been shown in [24]. Finally, a wait-free implementation of a red-black tree appeared in [22].

One of the techniques employed in this work is the fast-path-slow-path method, which attempts to separate slow handling of difficult cases from the fast handling of the more typical cases. This method is ubiquitous in systems in general and in parallel computing particularly [19, 21, 2, 3], and has been adopted recently [18] for creating fast wait-free data structures.

According to the fast-path-slow-path methodology of [18], an operation starts executing using a fast lock-free algorithm, and only moves to the slower wait-free path upon failing to make progress in the lock-free execution. It is often the case that an operation execution completes in the fast lock-free path, achieving

[1]The claim for inefficiency of universal constructions has been known as a folklore only. In Section 11.3 we provide the first measurements substantiating this claim.

good performance. But some operations fail to make progress in the fast path due to contention, and in this case, the execution moves to the slower wait-free path in which it is guaranteed to make progress. As many operations execute on the fast (lock-free) path, the performance of the combined execution is almost as fast as that of the lock-free data structure. It is crucial to note that even the unlucky threads, that do not manage to make progress in the fast path, are guaranteed to make progress in the slow path, and thus the strong wait-free guarantee can be obtained. Thus, we obtain the best of both worlds: the performance and scalability of the lock-free algorithm combined with the wait-free guarantee; this enables concurrent algorithms to serve demanding environments without sacrificing performance. The fast-path-slow-path methodology has been shown to make the wait-free queue of [17] and the wait-free linked-list of [24] almost as efficient as their lock-free counterpart.

The process of designing a fast wait-free data structure for an abstract data type is complex, difficult, and error-prone. One approach to designing new wait-free data structures, which is also the one used in [17, 18, 24], is to start with a lock-free data structure, work (possibly hard) to construct a correct wait-free data structure by adding a helping mechanism to the original data structure, and then work (possibly hard) again to design a correct and efficient fast-path-slow-path combination of the lock-free and wait-free versions of the original algorithm. Designing a slow-path-fast-path data structure is non-trivial. One must design the lock- and wait-free algorithms to work in sync to obtain the overall combined data structure with the required properties.

In this work we ask whether this entire design can be done mechanically, and so also by non-experts. More accurately, given a lock-free data structure of our choice, can we apply a generic method to create an adequate helping mechanism to obtain a wait-free version for it, and then automatically combine the original lock-free version with the obtained wait-free version to obtain a fast, practical wait-free data structure?

We answer this question in the affirmative and present an automatic transformation that takes a linearizable lock-free data structure in a normalized representation (that we define) and produces a practical wait-free data structure from it. The resulting data structure is almost as efficient as the original lock-free one.

We next claim that the normalized representation we propose is meaningful in the sense that important known lock-free data structures can be easily specified in this form. In fact, all linearizable lock-free data structures that we are aware of in the literature can be stated in a normalized form. We demonstrate the generality of the proposed normalized form by stating several important lock-free data structures in their normalized form and then obtaining wait-free versions of them using the mechanical transformation. In particular, we transform the linked-list [13, 11], the skiplist [16], and the binary search tree [7], and obtain practical wait-free designs for them all.

Next, in order to verify that the resulting data structures are indeed practical, we implemented all of the above wait-free algorithms and measured the performance of each. It turns out that the performance of all these implementations is only a few percent slower than the original lock-free data structure from which they were derived.

The contributions of this work are the following:

1. A mechanical transformation from any normalized lock-free data structure to a wait-free data structure that (almost) preserves the original algorithm efficiency. This allows a simple creation of various new practical wait-free data structures.
2. A demonstration of the generality of the normalized representation, by showing the normalized representation for lock-free linked-list, skiplist and tree.
3. A formal argument for the correctness of the transformation, and thus also the obtained wait-free data structures.
4. An implementation and reported measurements validating the efficiency of the proposed scheme.

We limit our discussion to the field of lock-free linearizable data structures. We believe our ideas can be applied to other algorithms as well, such as lock-free implementations of STM, but this is beyond the scope of this work.

The paper is organized as follows. In Section 2 we provide an overview of the proposed transformation. In Section 3 we briefly discuss the specifics of the shared memory model assumed in this work. In Section 4 we examine typical lock-free data structures, and characterize their properties in preparation to defining a normalized representation. The normalized representation is defined in Section 5, and the wait-free simulating for a normalized lock-free data structure appears in Section 6. We prove the correctness of the transformation in 7. We discuss the generality of the normalized form in Section 8. Next, in Section 9, we show how to easily convert four known lock-free data structures into the normalized form, and thus obtain a wait-free version for them all. Some important optimizations are explained in Section 10 , and our measurements are reported in Section 11.

2 Transformation overview

The move from the lock-free implementation to the wait-free one is executed by simulating the lock-free algorithm in a wait-free manner. The simulation starts by simply running the original lock-free operation (with minor modifications that will be soon discussed). A normalized lock-free implementation has some mechanism for detecting failure to make progress (due to contention). When an operation fails to make progress it asks for help from the rest of the threads. A thread asks for help by enqueuing a succinct description of its current computation state on a wait-free queue (we use the queue of [17]). One modification to the fast lock-free execution is that each thread checks once in a while whether a help request is enqueued on the help queue. Threads that notice an enqueued request for help move to helping a single operation on the top of the queue. Help includes reading the computation state of the operation to be helped and then continuing the computation from that point, until the operation completes and its result is reported.

The major challenges are in obtaining a succinct description of the computation state, in the proper synchronization between the (potentially multiple) concurrent helping threads, and in the synchronization between helping threads and threads executing other operations on the fast lock-free path. The normalized representation is enforced in order to allow a succinct computation representation, to ensure that the algorithm can detect that it is not making progress, and to minimize the synchronization between the helping threads to a level that enables fast simulation.

The helping threads synchronize during the execution of an operation at critical points, which occur just before and just after a modification of the data structure. Assume that modifications of the shared data structure occur using a CAS primitive. A helping thread runs the operation it attempts to help independently until reaching a CAS instruction that modifies the shared structure. At that point, it coordinates with all helping threads which CAS should be executed. Before executing the CAS, the helping threads jointly agree on what the CAS parameters should be (address, expected value, and new value). After deciding on the parameters, the helping threads attempt to execute the CAS and then they synchronize to ensure they all learn whether the CAS was successful. The simulation ensures that the CAS is executed exactly once. Then each thread continues independently until reaching the next CAS operation and so forth, until the operation completes. Upon completing the operation, the operation's result is written into the computation state, the computation state is removed from the queue, and the *owner* thread (the thread that initiated the operation in the first place) can return.

There are naturally many missing details in the above simplistic description, but for now we will mention two major problems. First, synchronizing the helping threads before each CAS, and even more so synchronizing them again at the end of a CAS execution to enable all of them to learn whether the CAS was successful, is not simple. It requires adding version numbering to some of the fields in the data structure,

and also an extra `modified bit`. We address this difficulty in Section 6.

The second problem is how to succinctly represent the computation state of an operation. An intuitive observation (which is formalized later) is that for a lock-free algorithm, there is a relatively light-weight representation of its computation state. This is because by definition, if at any point during the run a thread stops responding, the remaining threads must be able to continue to run as usual. This implies that if a thread modifies the data structure, leaving it in an "intermediate state" during the computation, then other threads must be able to restore it to a "normal state". Since this often happens in an execution of a lock-free algorithm, the information required to do so must be found on the shared data structure, and not (solely) in the thread's inner state. Using this observation, and distilling a typical behavior of lock-free algorithms, we introduce a normalized representation for a lock-free data structure, as defined in Section 5. The normalized representation is built in a way that enables us to represent the computation state in a compact manner, without introducing substantial restrictions on the algorithm itself.

There is one additional key observation required. In the above description, we mentioned that the helping threads must synchronize in critical points, immediately before and immediately after each CAS that modifies the data structure. However, it turns out that with many of the CASes, which we informally refer to as *auxiliary* CAS*es*, we do not need to use synchronization at all. As explained in Section 4, the nature of lock-free algorithms makes the use of auxiliary CASes common. Most of Section 4.2 is dedicated to formally define *parallelizable methods*; these are methods that only execute auxiliary CASes, and can therefore be run by helping threads without any synchronization. These methods will play a key role in defining normalized lock-free representation in Section 5.

3 Model and General Definitions

We consider a standard shared memory setting. In each computation step, a single thread executes on a target address in the shared memory one of three atomic primitives: READ, WRITE, or CAS. A computation step may also include a local computation, which may use local memory.

A CAS primitive is defined according to a triplet: `target address`, `expected-value` and `new-value`. A CAS primitive atomically compares the value of the target address to the expected-value, and WRITES the new value to the target address if the `expected-value` and old value in the target address are found identical. A CAS in which the `expected-value` and old value are indeed identical returns true, and is said to be *successful.* Otherwise the CAS returns false, and is *unsuccessful*. A CAS in which the `expected-value` and `new-value` are identical is a *futile* CAS.

An abstract data type, ADT, is defined by a state machine, and is accessed via *operations*. An operation receives zero or more input parameters, and returns one result, which may be null. The state machine of a type is a function that maps a state and an operation (including input parameters) to a new state and a result of the operation.

A *method* is a sequence of code-instructions that specify computation steps, including local computation. The next computation step to be executed may depend on the results of previous computation steps. Similarly to an operation, a method receives zero or more input parameters, and returns one result, which may be null. A code instruction inside a method may invoke an additional method.

A special method, ALLOCATE, which receives as an input the amount of memory needed and returns a pointer to the newly allocated memory is assumed to be available. We assume automatic garbage collection is available. This means that threads need not actively invoke a DEALLOCATE method, and an automatic garbage collector reclaims memory once it is no longer reachable by the threads. For further discussion about memory management, see Section 11.1.

A data structure implementation is an implementation of an ADT. (e.g., Harris's linked-list is a data structure implementation). Such an implementation is a set of methods that includes a method for each

operation, and may include other supporting methods.

A *program* is a set of one or more methods, and an indication which method is the entry point of the program. In an *execution*, each thread is assigned a single program. The thread executes the program by following the program's code-instructions, and execute computation steps accordingly.

An execution is a (finite or infinite) sequence of computation steps, cleaned out of the local computation. A scheduling is a (finite or infinite) sequence of threads. Each execution defines a unique scheduling, which is the order of the threads that execute the computation steps. Given a set of threads, each of which coupled with a program, and a scheduling, a unique corresponding execution exists.

An execution must satisfy MEMORY CONSISTENCY. That is, each READ primitive in the execution must return the value last WRITTEN, or successfully CASed, to the same target address. Also. Each CAS must return true and be successful if and only if the `expected-value` is equal to the last value written (or successfully CASes) into the same target address. Most works do not particularly define MEMORY CONSISTENCY and take it for granted, but the way we manipulate executions in our correctness proof (Section 7) makes this definition essential.

4 Typical Lock-Free Algorithms

In this section we provide the intuition on how known lock-free algorithms behave and set up some notation and definitions that are then used in Section 5 to formally specify the normalized form of lock-free data structures.

4.1 Motivating Discussion

Let us examine the techniques frequently used within lock-free algorithms. We target linearizable lock-free data structures that employ CASes as the synchronization mechanism. A major difficulty that lock-free algorithms often need to deal with is that a CAS instruction executes on a single word (or double word) only, whereas the straightforward implementation approach requires simultaneous atomic modification of multiple (non-consecutive) words[2]. Applying a modification to a single-field sometimes leaves the data structure inconsistent, and thus susceptible to races. A commonly employed solution is to use one CAS that (implicitly) blocks any further changes to certain fields, and let any thread remove the blocking after restoring the data structure to a desirable consistent form and completing the operation at hand.

An elegant example is the delete operation in Harris's linked-list [13]. In order to delete a node, a thread first sets a special *mark* bit at the node's next pointer, effectively blocking this pointer from ever changing again. Any thread that identifies this "block" may complete the deletion by physically removing the node (i.e., execute a CAS that makes its predecessor point to its successor). The first CAS, which is executed only by the thread that initiates the operation, can be intuitively thought of as an *owner* CAS.

In lock-free algorithms' implementations, the execution of the owner CAS is often separated from the rest of the operation (restoring the data structure to a "normal" form, and "releasing" any blocking set by the owner CAS) into different methods. Furthermore, the methods that do not execute the owner CAS but only restore the data structure can usually be safely run by many threads concurrently. This allows other threads to unblock the data structure and continue executing themselves. We call such methods *parallelizable methods*.

4.2 Notations and Definitions Specific to the Normalized Form.

In this section we formally define concepts that can be helpful to describe lock-free data structures, and are used in this work to define the normalized form.

[2]This is one of the reasons why transactional memories are so attractive.

Definition 4.1 (Equivalent Executions.) *Two executions E and E' of operations on a data structure D are considered* equivalent *if the following holds.*

- *(Results:) In both executions all threads execute the same data structure operations and receive identical results.*
- *(Relative Operation Order:) The order of invocation points and return points of all data structure operations is the same in both executions.*
- *(Comparable length:) either both executions are finite, or both executions are infinite.*

Note that the second requirement does not imply the same timing for the two executions. It only implies the same relative order of operation invocations and exits. For example, if the ith operation of thread T_1 was invoked before the jth operation of T_2 returned in E, then the same must also hold in E'. Clearly, if E and E' are equivalent executions, then E is linearizable if and only if E' is linearizable.

In what follows we consider the invocation of methods. A method is invoked with zero of more input parameters. We would like to discuss situations in which two or more invocations of a method receive the exact same input parameters. If the method parameters do not include pointers to the shared memory, then comparing the input is straight-forward. However, if a method is invoked with the same input I at two different points in the execution t_1 and t_2, but I includes a pointer to a memory location that was allocated or deallocated between t_1 and t_2, then even though I holds the same bits, in effect, it is different. The reason for this this that in t_1 and t_2 I holds a pointer to a different "logical memory", which happens to be physically allocated in the same place. To circumvent this difficulty, we use the following definition.

Definition 4.2 (Memory Identity.) *For a method input I and an execution E, we say that I* satisfies memory identity *for two points in the execution t_1 and t_2, if no memory in I, or reachable from I, is allocated or deallocated between t_1 and t_2 in E.*

Next, we identify methods that can be easily run with help, i.e., can be executed in parallel by several threads without harming correctness and while yielding adequate output. For those familiar with Harris's linked-list, a good example for such a method is the search method that runs at the beginning of the DELETE or the INSERT operations. The search method finds the location in the list for the insert or the delete and during its list traversal it snips out of the list nodes that were previously marked for deletion (i.e., logically deleted entries). The search method can be run concurrently by several threads without harming the data structure coherence and the outcome of any of these runs (i.e., the location returned by the search method for use of insert or delete) can be used for deleting or inserting the node. Therefore, the search method can be easily helped by parallel threads. In contrast, the actual insertion, or the act of marking a node as deleted, which should happen exactly once, is a crucial and sensitive (owner) CAS, and running it several times in parallel might harm correctness by making an insert (or a delete) occur more than once.

To formalize parallelizable methods we first define a harmless, or *avoidable* parallel run of a method. Intuitively, an avoidable method execution is an execution in which each CAS executed during the method can potentially be avoided in an alternative scheduling. That is, in an avoidable method execution, there is an equivalent execution in which the method does not modify the shared memory at all.

Definition 4.3 (Avoidable method execution) *A run of a method M by a thread T on input I in an execution E of a program P is avoidable if there exists an equivalent execution E' for E such that in both E and E' each thread follows the same program, both E and E' are identical until right before the invocation of M by T on input I, and in E' each* CAS *that T executes during M either fails or is futile.*

Definition 4.4 (Parallelizable method.) *A method M is a parallelizable method of a given lock-free algorithm, if for any execution in which M is called by a thread T with an input I the following two conditions holds. First, the execution of a parallelizable method depends only on its input, the shared memory, and the*

results of the methods CAS *operations. In particular, the execution does not depend on the executing threads local state prior to the invocation of the parallelizable method.*

Second, At any point in E that satisfies memory identity to I with the point in E in which M is invoked, If we create and run a finite number of parallel threads, and the program of each of these threads would be to run method M on input I, then in any possible resulting execution E', all executions of M by the additional threads are avoidable.

Loosely speaking, for every invocation of a parallelizable method M by one of the newly created threads, there is an *equivalent execution* in which this method's invocation does not change the data structure at all. In concrete known lock-free algorithms, this is usually because every CAS attempted by the newly created thread might be executed by one of the other (original) threads, thus making it fail (unless it is futile). For example, Harris's linked-list search method is parallelizable. The only CASes that the search method executes are those that physically remove nodes that are already logically deleted. Assume T runs the search method, and that we create an additional thread T_a and run it with the same input.

Consider a CAS in which T_a attempts to physically remove a logically deleted node from the list. Assume T_a successfully executes this CAS and removes the node from the list. Because the node was already logically deleted, this CAS does not affect the results of other operations. Thus, there exists an equivalent execution, in which this CAS is not successful (or not attempted at all.) To see that such an equivalent execution exists, consider the thread T_1 that marked this node as logically deleted in the first place. This thread must currently be attempting to physically remove the node so that it can exit the delete operation. An alternative execution in which T_1 is given the time, right before T_a executes the CAS, to physically remove the node, and only then does T_a attempt the considered CAS and fails, is equivalent.

It is important to realize that many methods, for example, the method that logically deletes a node from the list, are not parallelzable. If an additional thread executes CAS that logically deletes a node from the list, then this can affect the results of other operations. Thus, there exist some executions, that have no equivalent executions in which the additional thread does not successfully execute this CAS.

Parallelizable methods play an important role in our construction, since helping threads can run them unchecked. If a thread cannot complete a parallelizable method, helping threads may simply execute the same method as well.

We now focus on a different issue. In order to run the fast-path-slow-path methodology, there must be some means to identify the case that the fast path is not making progress on time, and then move to the slow path. To this end, we define the *Contention failure counter*. Intuitively, a contention failure counter is a counter associated with an invocation of a method (i.e. many invocations of the method imply separate counters), measuring how often the method is delayed due to contention.

Definition 4.5 (Contention failure counter.) *A contention failure counter for a method M is an integer field C associated with an invocation of M (i.e. many invocations of M imply many separate contention failure counters). Denote by C(t) the value of the counter at time t. The counter is initialized to zero upon method invocation, and is updated by the method during its run such that the following holds.*

- *(Monotonically increasing:) Each update to the contention failure counter increments its value by one.*
- *(Bounded by contention:) Assume M is invoked by Thread T and let $d(t)$ denote the number of data structure modifications by threads other than T between the invocation time and time t. Then it always hold that $C(t) \leq d(t)$.* [3]

[3] In particular, this implies that if no modifications were made to the data structure outside the method M since its invocation until time t, then $C(t) = 0$.

- *(Incremented periodically:) The method M does not run infinitely many steps without incrementing the contention failure counter.*

Remark 4.6 *The contention failure counter can be kept in the local memory of the thread that is running the method.*

A lock-free method must complete within a bounded number of steps if no modifications are made to the data structure outside this method. Otherwise, allowing this method to run solo results in an infinite execution, contradicting its lock-freedom. Thus, the requirements that the counter remains zero if no concurrent modifications occur, and the requirement that it does not remain zero indefinitely, do not contradict each other. The contention failure counter will be used by the thread running the method to determine that a method in the fast-path is not making progress and so the thread should switch to the slow path.

For most methods, counting the number of failed CASes can serve as a good *contention failure counter*. However, more complex cases exist. We further discuss such cases in Appendix B.

In order to help other threads, and in particular, execute CAS operations for them, we will need to have CASes published. For this publication act, we formalize the notion of a CAS description.

Definition 4.7 (CAS description.) *A* CAS *description is a structure that holds the triplet* $(addr, expected, new)$ *which contains an address (on which a* CAS *should be executed), the value we expect to find in this address, and the new value that we would like to atomically write to this address if the expected value is currently there. Given a pointer to a* CAS *description, it is possible to execute it and the execution can be either successful (if the* CAS *succeeds) or unsuccessful (if the* CAS *fails).*

5 Normalized Lock-Free Data Structures

In this section, we specify what a normalized lock-free data structure is. We later show how to simulate a normalized lock-free algorithm in a wait-free manner automatically.

5.1 The Normalized Representation

A normalized lock-free data structure is one for which each operation can be presented in three stages, such that the middle stage executes the owner CASes, the first is a preparatory stage and the last is a post-execution step.

Using Harris's linked-list example, the DELETE operation runs a first stage that finds the location to mark a node as deleted, while sniping out of the list all nodes that were previously marked as deleted. By the end of the search (the first stage) we can determine the main CAS operation: the one that marks the node as deleted. Now comes the middle stage where this CAS is executed, which logically deletes the node from the list. Finally, in a post-processing stage, we attempt to snip out the marked node from the list and make it unreachable from the list head.

In a normalized lock-free data structure, we require that: any access to the data structure is executed using a read or a CAS; the first and last stages be parallelizable, i.e., can be executed with *parallelizable methods*; and each of the CAS primitives of the second stage be protected by versioning. This means that there is a counter associated with the field that is incremented with each modification of the field. This avoids potential ABA problems, and is further discussed in Section 6.

Definition 5.1 *A lock-free data structure is provided in a* normalized representation *if:*

- *Any modification of the shared memory is executed using a* CAS *operation.*

- *Every* operation *of the data structure consists of executing three methods one after the other and which have the following formats.*
  1) CAS**-generator**, *whose input is the operation's input, and its output is a list of* CAS-descriptors. *The* CAS*-generator method may optionally output additional data to be used in the* WRAP-UP *method.*
  2) CAS**-executor**, *which is a fixed method common to all data structures implementations. Its input is the list of* CAS-descriptors *output by the* CAS*-generator method. The* CAS*-executor method attempts to execute the* CAS*es in its input one by one until the first one fails, or until all* CAS*es complete. Its output is the index of the* CAS *that failed (which is -1 if none failed).*
  3) Wrap-Up, *whose input is the output of the* CAS-executor method *plus the list of* CAS*-descriptors output by the* CAS*-generator, plus (optionally) any additional data output by the* CAS*-generator method to be used by the* WRAP-UP *method. Its output is either the operation's result, which is returned to the owner thread, or an indication that the operation should be restarted from scratch (from the* GENERATOR *method).*
- *The* GENERATOR *and the* WRAP-UP *methods are parallelizable and they have an associated* contention failure counter.
- *Finally, we require that the* CAS*es that the* GENERATOR *method outputs be for fields that employ versioning (i.e., a counter is associated with the field to avoid an ABA problem). The version number in the* `expected-value` *field of a* CAS *that the* GENERATOR *method outputs cannot be greater than the version number currently stored in the* `target address`*. This requirement guarantees that if the* `target address` *is modified after the* GENERATOR *method is complete, then the* CAS *will fail.*

All lock-free data structures that we are aware of today can be easily converted into this form. Several such normalized representations are presented in Section 9. This is probably the best indication that this normalized representation covers natural lock-free data structures. In Section 8 we show that all abstract data types can be implemented in a normalized lock-free data structure, but this universal construction is likely to be inefficient.

Intuitively, one can think of this normalized representation as separating owner CASes (those are the CASes that must be executed by the owner thread) from the other (denoted auxiliary) CASes. The auxiliary CASes can be executed by many helping threads and therefore create *parallelizable methods*. Intuitively, the first (generator) method can be thought of as running the algorithm without performing the owner CASes. It just makes a list of those to be performed by the executor method, and it may execute some auxiliary CASes to help previous operations complete.

As an example, consider the DELETE operation of Harris's linked-list. When transforming it to the normalized form, the GENERATOR method should call the search method of the linked-list. The search method might snip out marked (logically deleted) nodes; those are auxiliary CASes, helping previous deletions to complete. Finally, the search method returns the node to be deleted (if a node with the needed key exists in the list). The CAS that marks this node as logically deleted is the owner CAS, and it must be executed exactly once. Thus, the GENERATOR method does not execute this owner CAS but outputs it to be executed by the CAS-EXECUTER method. If no node with the needed key is found in the list, then there are no owner CASes to be executed, and the GENERATOR method simply returns an empty list of CASes.

Next, the CAS-EXECUTOR method attempts to execute all these owner CASes. In Harris's linked list, like in most known algorithms, there is only one owner CAS. The CAS-EXECUTER method attempts the owner CAS (or the multiple owner CASes one by one), until completing them all, or until one of them fails. After the CAS-EXECUTER method is done, the operation might already be over, or it might need to start from scratch (typically if a CAS failed), or some other auxiliary CASes should be executed before exiting. The decision on whether to complete or start again (and possibly further execution of auxiliary CASes) is done in the WRAP-UP method. In Harris' linked-list example, if the GENERATOR method outputted no CASes, then

it means that no node with the required key exists in the list, and the wrap-up method should return with failure. If a single CAS was outputted by the GENERATOR but its execution failed in the EXECUTER, then the operation should be restarted from scratch. Finally, if a single CAS was outputted by the GENERATOR and it was successfully executed by the EXECUTER, then the wrap-up method still needs to physically remove the node from the list (an auxiliary CAS), and then return with success. Removing the node from the list can be done similarly to the original algorithm, by calling the SEARCH method again.

We note that the normalized representation requires all data structure modifications to be executed with a CAS, and allows no simple WRITE primitives. This is in fact the way most lock-free data structures work. But this requirement is not restrictive, since any WRITE primitive can be replaced by a loop of repeatedly reading the old value and then trying to CAS it to the new value until the CAS is successful.

To see that this does not foil the lock-free property, replace the WRITES with such loop CASes one by one. Now, for a single such replacement note that either the CASes always succeed eventually and then the algorithm is still lock-free, or there exists an execution of this loop that never terminates. In the later case, other threads must be executing infinitely many steps that foil the CASes, while the current thread never modifies the data structure. This is similar to a case where this thread is not executing at all, and then the other threads must make progress, since the algorithm (without the looping thread) is lock-free.

6 Transformation Details

In this section, we provide the efficient wait-free simulation of any normalized lock-free data structure. To execute an operation, a thread starts by executing the normalized lock-free algorithm with a *contention failure counter* checked occasionally to see if contention has exceeded a predetermined limit. To obtain non-starvation, we make the thread check its *contention failure counter* periodically, e.g., on each function call and each backward jump. If the operation completes, then we are done. Otherwise, the contention failure counter eventually exceeds its threshold and the slow path must be taken.

There is also a possibility that the *contention failure counter* never reaches the predetermined limit for any execution of a single method, but that the WRAP-UP method constantly indicates that the operation should be restarted from fresh. (This must also be the result of contention, because if an operation is executed alone in the lock-free algorithm it must complete.) Thus, the thread also keeps track of the number of times the operation is restarted, and if this number reaches the predetermined limit, the slow path is taken as well. The key point is that an operation cannot execute infinitely many steps in the fast-path. Eventually, it will move to the slow-path.

The slow path begins by the thread creating an `operation record` object that describes the operation it is executing. A pointer to this operation record is then enqueued in a wait-free queue called the `help queue`. Next, the thread helps operations on the `help queue` one by one according to their order in the queue, until its own operation is completed. Threads in the fast path that notice a non-empty `help queue` provide help as well, before starting their own fast-path execution.

6.1 The Help Queue and the Operation Record

The description of operations that require help is kept in a wait-free queue, similar to the one proposed by Kogan and Petrank in [17]. The queue in [17] supports the standard ENQUEUE and DEQUEUE operations. We slightly modify it to support three operations: ENQUEUE, PEEK, and CONDITIONALLY-REMOVE-HEAD. ENQUEUE operations enqueue a value to the tail of the queue as usual. The new PEEK operation returns the current head of the queue, without removing it. Finally, the `conditionally-remove-head` operation receives a value it expects to find at the head of the queue, and removes it (dequeues it) only if this value is found at the head. In this case it returns *true*. Otherwise, it does nothing and returns *false*. This

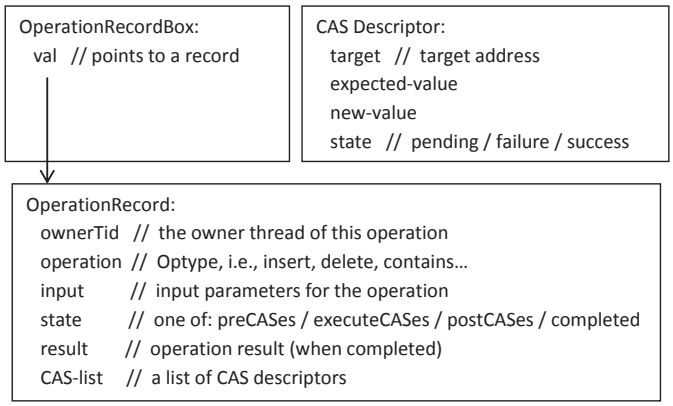

Figure 1: Operation Record

queue is in fact simpler to design than the original queue, because DEQUEUE is not needed, because PEEK requires a single read, and the `conditionally-remove-head` can be executed using a single CAS. (Therfore, `conditionally-remove-head` can be easily written in a wait-free manner.) Some care is needed because of the interaction between ENQUEUE and CONDITIONALLY-REMOVE-HEAD, but a similar mechanism already appears in [17], and we simply used it in our case as well. The Java implementation for our variation of the queue is given in Appendix A.

We use this queue as the `help queue`. If a thread fails to complete an *operation* due to contention, it asks for help by enqueuing a request on the `help queue`. This request is in fact a pointer to a small object (the operation record box) that is unique to the operation and identifies it. It is only reclaimed when the operation is complete. In this operation record box object there is a pointer to the `operation record` itself, and this pointer is modified by a CAS when the operation's status needs to be updated. We specify the content of this object and record in Figure 1.

6.2 Giving Help

When a thread T starts executing a new operation, it first PEEKs at the head of the `help queue`. If it sees a non-null value, then T helps the enqueued operation before executing its own operation. After helping to complete one operation, T proceeds to execute its own operation (even if there are more help requests pending on the queue).

To participate in helping an operation, a thread calls the HELP method, telling it whether it is on the fast path, and so willing to help a single operation, or on the slow path, in which case it also provides a pointer to its own operation record box. In the latter case, the thread is willing to help all operations up to its own operation. The HELP method will PEEK at the head of the `help queue`, and if it sees a non-null operation record box, it will invoke the HELPOP method. A null value means the help queue is empty, and so no further help is needed.

The HELPOP, invoked by the HELP method, helps a specific *operation* O, until it is completed. Its input is O's operation record box. This box may either be the current head in the `help queue` or it is an operation that has been completed and is no longer in the `help queue`. As long as the operation is not complete, HELPOP calls one of the three methods, PRECASES, EXECUTECASES, or POSTCASES, as determined by the operation record. If the operation is completed, HELPOP attempts to remove it from the queue using CONDITIONALLY-REMOVE-HEAD. When the HELPOP method returns, it is guaranteed that the operation record box in its input represents a completed operation and is no longer in the `help queue`.

The PRECASES method invokes the CAS-GENERATOR method of the normalized lock-free algorithm,

```
1:  void help (boolean beingHelped, OperationRecordBox myHelpBox) {
2:      while (true) {
3:          OperationRecordBox head = helpQueue.peekHead();
4:          if (head != null)
5:              helpOp(head);
6:          if (!beingHelped || myHelpBox.get().state == OpState.completed)
7:              return;
8:      }
9:  }
```

Figure 2: The help method

```
1:  void helpOp(OperationRecordBox box) {
2:      OperationRecord record = null;
3:      do {
4:          record = box.val;
5:          OpState state = record.state;
6:          if (state == OpState.preCASes) {
7:              preCASes(box, record);  ▷ Executes the CAS generator supplied by the normalized algorithm plus attempt to make
    the result visible.
8:          }
9:          if (state == OpState.executeCASes) {
10:             int failedIndex = executeCASes(record.list);        ▷ carefully execute the CAS list outputted by the CAS generator.
11:             record.failedCasIndex = failedIndex;
12:             record.state = OpState.postCASes;
13:         }
14:         if (state == OpState.postCASes) {
15:             postCASes(box, record);                              ▷ execute the wrap-up method, plus some administrative work
16:         }
17:     } while (state != OpState.completed);
18:     helpQueue.conditionallyRemoveHead(box);
19: }
```

Figure 3: The helpOp method

```
1:  void preCASes(OperationRecordBox box, OperationRecord record) {
2:        cas-list = MonitoredRun(Of GeneratorMethod on record);
3:        if (cas-list != null) {
4:           newRecord =
                new OperationRecord(record.ownerTid, record.operation, record.input, OpState.executeCASes, null, cas-list);
5:           CAS(box.val, record, newRecord);
6:        }
7:     }
```

Figure 4: The preCASes method

which generates the list of CAS-descriptions for the CAS-EXECUTOR. As the CAS-GENERATOR method is parallelizable, it can be run by several threads concurrently at no risk[4]. It runs a monitored version of the generator, which occasionally checks the contention failure counter in order to guarantee this method will not run forever. If the contention failure counter reaches the predetermined threshold, the thread simply quits this method with null and reads the operation record box to see if another thread has made progress with this operation (if not, the HELPOP method will call the PRECASES method again).

The PRECASES method allocates a new operation record that holds the result of the run of the CAS-GENERATOR method. The outcome of the PRECASES can either be a null pointer if the method was stopped by the contention failure counter, or a list of CAS-descriptors if the method completed successfully. If the result of the CAS-GENERATOR execution is not a null, the PRECASES method creates a new operation record and attempts to make it the official global operation record for this operation by attempting to atomically change the operation record box to reference it. There is no need to check whether this attempt succeeded as the CAS-GENERATOR method is a parallelizable method and any result by any of its concurrent executions is a proper result that can be used to continue the operation.

If the `OperationRecord` is not replaced by a new one, then soon enough all threads will only run this method, all helping the same operation. In that case, it is guaranteed to be completed because the simulation is equivalent to running this operation solo[5]. After the *OperationRecord* is successfully replaced by a CAS, some threads might still be executing the GENERATOR method. Since we monitor the execution with a contention failure counter, and since the counter is required to be incremented repeatedly (cannot maintain any value forever), then we know that these threads do will not execute infinitely many steps in these methods.

The CAS-EXECUTOR method is not parallelizable and therefore helping threads cannot simply run it concurrently. Only one execution of each CAS is allowed, and it should be clear to everyone whether each CAS execution succeeded or failed. So we replace it with a carefully designed concurrent method, named EXECUTECASES (Figure 5) .

The EXECUTECASES method receives as its input a list of CAS-descriptors to be executed. Each CAS description is also associated with a `state` field, which describes the execution state of this CAS: succeeded, failed, or still pending. The controlled execution of these critical CASes requires care to ensure that: each CAS is executed exactly once, the success of the CAS gets published even if one of the threads stops responding, and an ABA problem is not created by letting several threads execute this sensitive CAS instead of the single thread that was supposed to execute it in the original lock-free algorithm. The ABA problem is introduced because a thread may be inactive for a while and then successfully execute a CAS that had been executed before, if after its execution the `target address` was restored back to its old value.

Ideally, we would have liked to execute three instructions atomically: (1) read the `state`, (2) attempt the CAS (if the `state` is pending), and (3) update the CAS `state`. Unfortunately, since these three instructions work on two different locations (the CAS's `target address` and the descriptor's `state` field) we cannot

[4]This is formally proved at Section 7.

[5]A formal argument for the wait-freedom is given in Section 7.2.

run this atomically without using a heavy mutual exclusion machinery that foils wait-freedom (and is also costly).

To solve this atomicity problem, we introduce both a versioning mechanism to the fields being CASed, and an additional bit, named `modification-bit`, to each CASed field. (In a practical implementation, the `modified-bit` is on the same memory word as the version number.)

The `modified-bit` will signify that a successful CAS has been executed by a helping thread, but (possibly) not yet reported. So when a CAS is executed in the slow path, a successful execution will put the new value together with the `modified-bit` set. As a result, further attempts to modify this field must fail, since the `expected-value` of any CAS never has this bit set. When a field has the `modified-bit` set, it can only be modified by a special CAS primitive designated to clear the `modified-bit`. This CAS, which we refer to as a CLEARBIT CAS, is the only CAS that is executed *without* incrementing the version number. It only clears the `modified-bit`, and nothing more. However, before any thread attempts the CLEARBIT CAS, it must first update the `state` of the CAS to reflect success.

Our transformation keeps the invariant that in the entire data structure, only a single `modified-bit` might be set at any given moment. This is exactly the bit of the CAS that is currently being helped by all helping threads. Before clearing this `modified-bit`, no other CAS execution can be helped.

Let us examine an execution of the EXECUTECASES method. The executing thread goes over the CASes in the list one by one, and helps execute them as follows. First, it reads the CAS `state`. If it is successful, it attempts the CLEARBIT CAS to clear the `modified-bit`, in case it hasn't been done before. The `expected-value` of the CLEARBIT CAS exactly matches the `new-value` of the CAS-descriptor except that the `modified-bit` is set. Thus, due to the version number, the CLEARBIT CAS can only clear a `modified-bit` that was switched on by the same CAS-descriptor. (This is formally proved in Section 7.1.)

Otherwise, if the CAS `state` is currently set to failure, then the EXECUTECASES method immediately returns with the index of the failing CAS. Otherwise, the `state` is pending, and EXECUTECASES attempts to execute the listed CAS and set the `modified-bit` atomically with it. Next, it checks whether the `modified bit` is set, and if it is, it sets the (separate) CAS `state` field to success and only then attempts to clear the `modified-bit`.

Setting the `state` field to success is done with an atomic CAS, which only succeeds if the previous `state` is pending. This is required to solve a race condition in which the execution of the CAS-descriptor has failed, yet the `modified-bit` is set to true is the result of a successful execution of a later CAS. Afterwards, and only if the `state` is now indeed success, the CLEARBIT is attempted. Next, if at that point the CAS `state` field is still not set to success, then it means the CAS has failed, and thus EXECUTECASES sets this `state` to failure and returns. Otherwise, success is achieved and EXECUTECASES proceeds to the next CAS in the list.

The existence of the `modified-bit` requires minor modifications to the fast-path. First, READ primitives should ignore the `modified-bit` (always treat it as if the bit were off.) This should be easy: the `modified-bit` is adjacent to the version number, which does not normally influence the execution (only when calculating the next version number for the `new-value` of a CAS.)

Second, when a thread attempts a CAS and the CAS fails in the fast-path, it should check to see whether the CAS failed because the `modified-bit` in the required field is set, and if so, whether the CAS would have succeeded were the bit turned off.

Thus, after a CAS in the fast-path fails, instead of continuing as usually, the thread that attempted the CAS READS the value from the CAS's `target address`. If this value differs from the CAS's `expected-value` in other bits than the `modified-bit`, then the thread continues the execution as usual, since the CAS has "legitimate" reasons for failure. However, if the value in the CAS's `target address` is identical to the CAS's `expected-value` in all the bits but the `modified-bit`, then the thread pauses its current execution and calls the help method to participate in helping the current operation to complete (clearing this bit

```
 1: private void executeCASes(CAS-list cl) {
 2:       for (int i = 0; i < cl.size(); i++) {
 3:          ICasDesc cas = cl.get(i);
 4:          if (cas.GetState() == CasState.success) {
 5:             cas.ClearBit();
 6:             continue;
 7:          }
 8:          if (cas.GetState() == CasState.failure)
 9:             return i;
10:          cas.ExecuteCas();
11:          if (cas.ModifiedBitSet()) {                    ▷ Checks whether the modified bit in the target address is set.
12:             cas.CASStateField(CasState.pending, CasState.success);    ▷ Attempt with a CAS to change the descriptor's state
    from pending to success.
13:             if (cas.GetState == CasState.success) { cas.ClearBit(); }
14:          }
15:          if (cas.GetState() != CasState.success) {
16:             cas.WriteStateField(CasState.failure);  ▷ CAS MUST HAVE FAILED, SET THE DESCRIPTOR'S STATE TO FAILURE.
17:             RETURN I;
18:          }
19:       }
20:       RETURN -1;                                          ▷ THE ENTIRE CAS-LIST WAS EXECUTED SUCCESSFULLY
21:    }
```

Figure 5: The executeCASes Method

```
1: void postCASes(OperationRecordBox box, OperationRecord record) {
2:    shouldRestart, operationResult = MonitoredRun(of Wrapup Method on record);
3:    if (operationResult == Null) Return
4:    if (shouldRestart)
5:       newRecord = new OperationRecord(record.ownerTid, record.operation, record.input, OpState.preCASes, null, null);
6:    else
7:       newRecord = new OperationRecord(record.ownerTid, record.operation, record.input, OpState.completed, operationRe-
   sult, null);
8:    box.val.compareAndSet(record, newRecord);
9: }
```

Figure 6: The postCASes Method

in the process.)

After the help method returns the `modified-bit` is guaranteed to have been cleared. Thus, the CAS is attempted again, and the execution continues as usual from that point. Even if the re-execution fails, there is no need to READ the `target address` again. It is guaranteed that the value in the `target address` is now different from the CAS's `expected-value`: if the `modified-bit` is turned back on after being cleared, it can only be done together with incrementing the version number.

After the CASes are executed, the HELPOP method calls the POSTCASES method (Figure 6), which invokes the WRAP-UP method of the original lock-free algorithm. If the WRAP-UP method fails to complete due to contention, the monitored run will return null and we will read again the `operation record box`. If the WRAP-UP method was completed without interruption, the POSTCASES method attempts to make its private operation record visible to all by atomically attempting to link it to the `operation record box`. Note that its private `operation record` may indicate a need to start the operation from scratch, or may indicate that the operation is completed. When the control is returned to the HELPOP method, the record is read and the execution continues according to it.

7 Correctness

Our goal is to prove that given a normalized linearizable lock-free data structure implementation for a particular abstract data type, our transformation generates a wait-free linearizable implementation for the same abstract data type. As a preliminary step, we first prove that the implementation of the EXECUTECASES method, as given in Figure 5 is correct. The exact definition of a correct behavior of the EXECUTECASES method is given in the following subsection (Definition 7.1). Subsection 7.1 proves that our implementation is indeed correct. Given this result, Subsection 7.2 proves that the generated algorithm of our transformation is linearizable and wait-free.

7.1 Correctness of the EXECUTECASES Implementation

In the EXECUTECASES method, potentially many threads are working together on the same input (same CAS list). A CAS-list is a structure that holds zero or more CAS-descriptors, and a field indicating the length of the list. Each CAS-descriptor consists of four fields: `target address`, `expected-value`, `new-value`, and `status`. The three first fields are final (never altered) after a CAS-descriptor has been initialized.

Loosely speaking, to an "outside observer" that inspects the shared memory, many threads executing the EXECUTECASES method on a certain CAS-list should appear similar to a single thread executing the CAS-EXECUTER method (the second method in the normalized form) on a private (but identical) CAS-list input. Recall that in the CAS-EXECUTER method, the CASes are executed according to their order until they are completed or the first one among them fails. The output of the method is the index of the first CAS that failed, or minus one if no CAS failed.

The main difference between an execution of the CAS-EXECUTER method by a single thread, and concurrent executions of the EXECUTECASES method by many threads, is that in the latter each CAS is executed in two steps. We refer to the first (main) step simply as executing the CAS-descriptor, and to the second step as executing the CLEARBIT of the CAS-descriptor. An execution of a CAS-descriptor (which occurs in line 10 of the EXECUTECASES method) is an execution of a CAS for which the `target address`, `expected-value` and `new-value` are the same as the CAS-descriptor's, except that the `new-value` is altered such that the `modified-bit` is set. An execution of a CLEARBIT of a CAS-descriptor (which occurs in lines 5 and 13) is an execution of a CAS for which the `target address` and the `new-value` are the same as the CAS-descriptor's, and the `expected-value` is identical to the `new-value` except that the `modified-bit` is set. (Thus, the `expected-value` of the second step is the `new-value` of the first step.)

In what follows, we formally define what is a correct concurrent behavior for the EXECUTECASES method and prove that the implementation of it given in Figure 5 is indeed correct. The correctness of the transformation, as detailed in subsection 7.2, relies on the correctness of the EXECUTECASES method stated here. However, the two proofs are independent of each other, and the reader may skip the proof in this subsection if he chooses to, without loss of clarity.

Definition 7.1 (Correct Behavior of the EXECUTECASES method.) *When one or more threads execute concurrently the* EXECUTECASES *method using the same* CAS*-list input, the following should hold.*

- *All computation steps inside the* EXECUTECASES *method are either: a) an execution of a* CAS*-descriptor, b) a* CLEARBIT *of a* CAS*-descriptor, or c) applied on the memory of the* CAS*-list (e.g., altering the* `state` *field of a* CAS*-descriptor).*
- *For every* CAS*-descriptor c: a) any attempt to execute c except the first attempt (by some thread) must fail, and b) any attempt to execute the* CLEARBIT *of c except the first attempt (by some thread) must fail.*

- *Before a* CAS*-descriptor* c *in a* CAS*-list* cl *is executed for the first time: a) all the previous* CAS*-descriptors in* cl *have been* successfully *executed, and b)* CLEARBIT *has already been executed for all the previous* CAS*-descriptors in* cl*.*

- *Once some thread has completed executing the* EXECUTECASES *method on an input* CAS*-list* cl *the following holds.*
  1) Either all the CAS*-descriptors have been successfully executed, or all the* CAS*-descriptors have been executed until the first one that fails. Further* CAS*-descriptors (after the first one that fails) have not been executed, and will not be executed in the rest of the computation.*
  2) A CLEARBIT *was successfully executed for each* CAS*-descriptor that was successfully executed.*

- *The return value of the* EXECUTECASES *for* every *thread that completes it is:*
  1) The index of the first (and only) CAS*-descriptor whose execution failed the first time it was attempt, if such exists.*
  2) -1 otherwise.

Our goal is to prove that the EXECUTECASES method as implemented in Figure 5 is correct by Definition 7.1, assuming that its input is legal. More precisely, we consider an execution E in which the EXECUTECASES method is invoked (possibly many times). We use several assumptions on E, (all fulfilled by an execution of an algorithm that results from applying our transformation on a normalized form algorithm) about how the EXECUTECASES method is used, and prove that E fulfills definition 7.1. We assume the following.

Assumption 7.2 *Only a single* CAS*-list for which the execution (by some thread) is not yet completed is active at any given moment. More precisely: whenever the* EXECUTECASES *method is invoked in* E *with an input* CAS*-list* cl*, then for all prior invocations of the* EXECUTECASES *method with an input* CAS*-list* cl'*, by any thread, one of the following holds.*
1) cl *and* cl' *are equal.*
2) An execution of the EXECUTECASES *method for which the input was* cl' *is already completed.*

Remark 7.3 *Note that we do not assume that* all *executions of the* EXECUTECASES *method with input* cl' *are already completed.*

Assumption 7.4 *Any address that is used as a* `target address` *of any* CAS*-descriptor is only ever modified in* E *with a* CAS *(no writes). Outside the* EXECUTECASES *method, all the* CAS*es that modify this address has the* `modified-bit` *off both in the* `expected-value` *and in the* `new-value`*.*

Assumption 7.5 *A version number is associated with every address that is used as a* `target address` *of a* CAS*-descriptor. For every* CAS *in* E *that attempts to modify such an address outside the* EXECUTECASES *method, the version number of the* `new-value` *is greater by one than the version number of the* EXPECTED VALUE*. (That is, each successful* CAS *increments the version number by one.)*

Assumption 7.6 CAS*-descriptors are initialized with a pending* `state`*, and the* `state` *field is never modified outside the* EXECUTECASES *method.*

Assumption 7.7 *When a* CAS*-descriptor is initialized, the version number in the* `expected-value` *field is no greater than the current version number stored in the* `target address` *of the* CAS*. (That is,* CAS*-descriptors are not created speculatively with "future" version numbers.)*

Remark 7.8 *Usually in a* CAS*, the* `expected-value` *is a value that was previously read from the* `target address`*. If that is the case, this assumption will always hold.*

To simplify the proof, we first define a few terms used in the proof. First, we define a total order between all the CAS-lists that are used as an input to the EXECUTECASES method in E, and to all the CAS-descriptors used in these CAS-lists.

Definition 7.9 (Total order of CAS-lists.) *Given two different* CAS*-lists cl_1 and cl_2 used as an input to the* EXECUTECASES *method in E, we say that cl_1 is before cl_2 (or prior to cl_2) if the first time that an* EXECUTECASES *method with input cl_1 is invoked in E is prior to the first time that an* EXECUTECASES *method with input cl_2 is invoked in E.*

Remark 7.10 *Note that by Assumption 7.2, if cl_1 is prior to cl_2, then some thread completes executing the* EXECUTECASES *method on input cl_1 before the first time that the* EXECUTECASES *method is invoked with cl_2.*

Definition 7.11 (Total order of CAS-descriptors.) *Given a* CAS*-descriptor c_1 that belongs to a* CAS*-list cl_1, and a* CAS*-descriptor c_2 that belongs to a* CAS*-list cl_2, we say that c_1 is before c_2 (or prior to c_2) if either: 1) cl_1 is before cl_2, or 2) cl_1 and cl_2 are equal, and c_1 appears before c_2 in the* CAS*-list.*

Next, we define the most recent CAS-list, most recent EXECUTECASES iteration, and most recently active CAS-descriptor for a given point in time t. For an execution E, time t is the point in the execution after exactly t computation steps.

Definition 7.12 (Most recent CAS-list, most recent EXECUTECASES iteration, most recently active CAS-descriptor) *At time t, the* most recent CAS-list *cl is the latest* CAS*-list (Definition 7.9) such that an* EXECUTECASES *method is invoked with cl as an input before time t. The* most recent EXECUTECASES iteration *at time t is the latest iteration (with the largest i variable) of the loop in lines 2–17 of the* EXECUTECASES *method that any thread was executing at or before t on the most recent cl of time t. The* most recently active CAS-descriptor *is the* CAS*-descriptor that is read at the beginning of the most recent* EXECUTECASES *iteration.*

Remark 7.13 *Note that if the first time the* EXECUTECASES *method is invoked in E is after time t, then the most recent* CAS*-list, most recent* EXECUTECASES *iteration, and most recently active* CAS*-descriptor are undefined for time t.*

Definition 7.14 (`modified-bit` belongs to a CAS-descriptor.) *We say that a* `modified-bit` *that is true at time t* belongs *to the* CAS*-descriptor whose execution switched this bit to true most recently prior to t. (Note that a* `modified-bit` *can only be set to true in line 10 of the* EXECUTECASES *method. (Assumption 7.4.))*

Claim 7.15 *At any point in the computation, if a* `modified-bit` *is on, it belongs to some* CAS*-descriptor.*

Proof: By Assumption 7.4, a `modified-bit` cannot be switched on outside of the EXECUTECASES method. Inside the EXECUTECASES method, it can only be switched on by executing a CAS-descriptor in line 10. It follows from Definition 7.14 that when a `modified-bit` is on, it belongs to some CAS-descriptor.

In what follows we state several invariants that are true throughout execution E. After stating them all, we will prove them using induction on the computation steps of the execution. The induction hypothesis is that all the following invariants are correct after i computation steps, and we shall prove they all hold after $i + 1$ computation steps. When proving that an invariant holds for $i + 1$ steps, we will freely use

the induction hypothesis for any one of the invariants, and may also rely on the fact that previously proved invariants hold for $i+1$ steps. All the invariants trivially hold for $i = 0$ steps: the first invariant holds since by Assumption 7.6 all CAS-descriptors are initialized as pending, and the rest of the invariants hold for $i = 0$ steps vacuously, since they refer to a condition that is always false before a single execution step is taken.

Invariant 7.16 *The* `state` *of a* CAS*-descriptor that has not yet been executed is pending.*

Invariant 7.17 *If the* `state` *of a* CAS*-descriptor is failure, then the first attempt to execute the* CAS*-descriptor has already occurred, and it has failed.*

Invariant 7.18 *If the* `state` *of a* CAS*-descriptor is success, then the first attempt to execute the* CAS*-descriptor has already occurred, and it has succeeded.*

Invariant 7.19 *If a* CAS*-descriptor's* `state` *is not pending (i.e., either success or failure), then it is final (never changes again).*

Invariant 7.20 *An attempt to execute a particular* CAS*-descriptor in a given* CAS*-list, except the first attempt by the first thread that attempts it, must fail.*

Invariant 7.21 *If some thread t is currently executing the nth iteration of the loop in some instance of the* EXECUTECASES *method (formally: if the last computation step taken by t is inside the nth iteration of the loop), then the* `states` *of the* CAS*-descriptors read in iterations 0 to $n-1$ of the same* EXECUTECASES *instance are success.*

Invariant 7.22 *If a* CAS*-descriptor c in a* CAS*-list cl has been executed, then the* `states` *of all the previous* CAS*-descriptors in cl are success.*

Invariant 7.23 *If the* `state` *of a* CAS*-descriptor c in a* CAS*-list cl is not pending, then the* `states` *of all the previous* CAS*-descriptors in cl are success.*

Invariant 7.24 *If some thread t has already completed the execution of an* EXECUTECASES *method with input* CAS*-list cl, then either 1) the* `states` *of all the* CAS*-descriptors in cl are success, or 2) the* `state` *field of exactly one* CAS*-descriptor c in cl is failure, the* `states` *of all the* CAS*-descriptors before c in cl (if any) are success, and the* `states` *of all the* CAS*-descriptors after c in cl (if any) are pending.*

Invariant 7.25 *If a* CAS*-descriptor has already been successfully executed, then one of the following holds.*
1) The CAS*-descriptor's* `state` *field indicates success, or*
2) The CAS*-descriptor's* `state` *field indicates a pending state, and the* `target address` *of the* CAS *still holds the* CAS*'s* `new-value`*, and in particular, the* `modified-bit` *is set to true.*

Invariant 7.26 *If some thread t is currently executing the loop in lines 2–19 (formally: if the last execution step taken by t is inside the loop), in which the* CAS*-descriptor c is read, but the iteration t is executing is not the most recent* EXECUTECASES *iteration, (which means that c is not the most recently active* CAS*-descriptor), then c's state is not pending.*

Invariant 7.27 *If some thread t has already completed executing the loop in lines 2–19 (either by breaking out of the loop in line 9 or 17, or by continuing to the next loop from line 6, or simply by reaching the end of the iteration), in which the* CAS*-descriptor c is read, then there is no* `modified-bit` *that is set to true and that belongs to c.*

Invariant 7.28 *If a certain* `modified-bit` *is true, then this* `modified-bit` *belongs to the most recently active* CAS*-descriptor.*

Proof: (Invariant 7.16.) Each CAS-descriptor is initialized as pending, and its `state` can potentially be changed only in lines 12 and 16 of the EXECUTECASES method. Before a thread t executes one of these lines for a certain CAS-descriptor, it first attempts to execute the same CAS-descriptor in line 10. Thus, if a CAS-descriptor has never been executed, its `state` must be pending.

Proof: (Invariant 7.17.) Assume by way of contradiction that in step $i+1$ a thread t sets a CAS-descriptor c's `state` to failure, and that the first attempt to execute c has not yet occurred or has been successful. Step $i+1$ must be an execution of line 16, since this is the only line that sets a `state` field to failure (Assumption 7.6). Consider the execution right after t executed line 10 of the same iteration of the loop in lines 2–19. t has just executed c, so it is impossible that c has not yet been executed. Thus, the first attempt to execute c must have been successful.

By the induction hypothesis (Invariant 7.25), in each computation step after c was first executed (and in particular, after thread t executed it in line 10), and until step i, c's `state` is either success, or it is pending and the `modified-bit` is set to true. Thus, when t executes line 11, there are two cases.

The first case is that c's `state` is success. Since there is no code line that changes a `state` back to pending, and since until step $i+1$ the `state` cannot be failure by the induction hypothesis (Invariant 7.25), then the `state` must also be success when t executes line 15. Thus, the condition in this line is false, line 16 is not reached, and t cannot set c's `state` to failure at step $i+1$, yielding contradiction for the first case.

The second case is that c's `state` field is pending and that the `modified-bit` is set. In that case, t will attempt by a CAS to switch the `state` from pending to success in line 12. After executing this line, c's `state` must be success (since it cannot be failure by the induction hypothesis (Invariant 7.25), and if it were pending the CAS would have changed it to success). Similarly to the previous case, the `state` must also be success when t executes line 15, and thus line 16 is not reached, yielding contradiction for the second case.

Proof: (Invariant 7.18.) Assume by way of contradiction that in step $i+1$ a thread t sets a CAS-descriptor c's `state` to success, and that the first attempt to execute c has not yet occurred or has been unsuccessful. Step $i+1$ must be an execution of line 12, since this is the only line that sets a `state` field to success (Assumption 7.6). t has already executed line 10 of the same iteration of the loop, thus the first attempt to execute the CAS-descriptor has already occurred, and thus it must have failed.

Consider the execution when t executes line 11 of the same iteration of the loop. The `modified-bit` must have been on, otherwise line 12 would not have been reached. By Claim 7.15, this `modified-bit` must belong to a CAS-descriptor. We consider three cases. The first case is that the `modified-bit` belongs to c. In this case c's first execution attempt must have been successful, yielding contradiction.

The second case is that the `modified-bit` belongs to a CAS-descriptor prior to c. However, when t executes line 11, then by the induction hypothesis (Invariant 7.28), the `modified-bit` must belong to the most recently active CAS-descriptor. Since c is active at that point, then any CAS-descriptor prior to c cannot be the most recently active one by definition, and thus the `modified-bit` cannot belong to it, yielding contradiction for the second case.

The third case is that the `modified-bit` belongs to a CAS-descriptor that comes after c. Thus, by the induction hypothesis (Invariant 7.26), after i computation steps c's `state` cannot be pending. (t is executing the loop in lines 2–19 after i steps, but c cannot be the most recently active CAS-descriptor since a later CAS-descriptor has already been active to set the `modified-bit` to true.) If c's `state` is not pending after i steps, then t cannot set it to success in step $i+1$ via an execution of line 12, yielding contradiction for the third case.

Proof: (Invariant 7.19.) This follows directly from Invariants 7.17 and 7.18, which are already proven for $i+1$ steps. That is, if c's `state` is failure after i steps, then by Invariant 7.17, the first attempt to execute c must have failed. Thus, by Invariant 7.18, the `state` cannot be success after $i+1$ steps. Similarly, if c's `state` is success after i steps, then by Invariant 7.18, the first attempt to execute c must have succeeded. Thus, by Invariant 7.17, the `state` cannot be failure after $i+1$ steps. Finally, a `state` cannot be changed from success or failure to pending, because no line in the EXECUTECASES method changes a `state` to pending, and by Assumption 7.6, no line in the code outside the EXECUTECASES does that either.

Proof: (Invariant 7.20.) Assume that in step $i+1$ a CAS-descriptor c is attempted, and this is not the first attempt to execute this CAS. We shall prove this attempt must fail. By Assumption 7.5, each CAS-descriptor is to a `target address` that is associated with a version number. Furthermore, by combining Assumption 7.5 with Assumption 7.7, the version number of the `expected-value` is never greater than the current value stored in the `target address`. Thus, we consider two cases. The first case is that the first attempt to execute a c had succeeded. In this case, after this execution, the version number is greater than the expected-value's version number, and thus the attempt to execute it again in step $i+1$ must fail.

The second case is that the first attempt to execute a c had failed. If it failed because at the time of the attempt the version number stored in the `target address` had already been greater than the version number of the `expected-value`, then this must still be true, and the attempt to execute c in step $i+1$ must also fail. If the first attempt to execute c failed because even though the version numbers matched, the value stored in the `target address` differed from that of the `expected-value`, and the difference was not limited to the `modified-bit`, then in order for the execution attempt in step $i+1$ to succeed the value stored in the `target address` must then be changed, but in such a case the version number must be incremented, and thus again c's execution in step $i+1$ is doomed to failure.

The last possibility is that the first attempt to execute c had failed only because the `modified-bit` was set to true at the time. Since the `modified-bit` can be switched off by executing a CLEARBIT *without* incrementing the version number, this could theoretically allow c to be successfully executed later. However, this is impossible. Consider c's first execution. Since this happens before step $i+1$, then by the induction hypothesis (Invariant 7.28), if the `modified-bit` was set, the `modified-bit` must belonged to the most recently active CAS-descriptor. This cannot be c, since c was not successfully executed at the time. Thus, by the induction hypothesis (Invariant 7.26) c's state at the time was not pending. And thus, by Invariants 7.17 and 7.18, c must have been executed before, and this cannot be c's first execution.

Proof: (Invariant 7.21.) To reach the nth iteration, t must have first completed iterations 0 to $n-1$. Consider t's execution of line 15 for each of these iterations. In this line, the `state` of the CAS-descriptor that is read in the same iteration is checked. If the `state` is set to success, then by Invariant 7.19 (which is already proved for $i+1$ steps), the `state` is also success after $i+1$ steps, and we are done. If the `state` is not success, then t will break out of the loop in line 15, and the nth iteration would not be reached.

Proof: (Invariant 7.22.) By the induction hypothesis for the same invariant (Invariant 7.22), the invariant holds after i steps. Assume by way of contradiction that the invariant does not hold after $i+1$ steps. Thus, the $i+1$-st step must be one of the following.
1) A thread t executes a CAS-descriptor c in a CAS-list cl while the `state` of a previous CAS-descriptor in cl is not success.
2) The `state` of a CAS-descriptor c_2 in a CAS-list cl changes from success to a different value, while a later CAS-descriptor in cl has already been executed.

The first case yields contradiction because if t is executing a CAS-descriptor c, then the `states` of all the previous CAS-descriptors in the same list must be success by Invariant 7.21, which is already proved for $i+1$ steps. The second case yields a contradiction because a non-pending state is final by Invariant 7.19, which is also already proved for $i+1$ steps.

Proof: (Invariant 7.23.) If the `state` of a CAS-descriptor c is not pending, then c has already been executed by Invariant 7.16 (which is already proved for $i+1$ steps). If c has already been executed, then the `states` of all the previous CAS-descriptors in the same cl are success by Invariant 7.23 (which is also already proved for $i+1$ steps).

Proof: (Invariant 7.24.) By the induction hypothesis for the same invariant (Invariant 7.24), the invariant holds after i steps. Assume by way of contradiction that the invariant does not hold after $i+1$ steps. Thus, the $i+1$-st step must be one of the following.
1) A thread t completes the execution of the EXECUTECASES method on input CAS-list cl, yet cl does not meet the requirements.
2) A thread t changes the `state` field of a CAS-descriptor in a CAS-list cl that met the requirements after i steps. (And this cl was used as an input to an EXECUTECASES invocation that is already completed.)

Consider the first possibility, and in particular, consider which computation step could be the last computation step that t executes when completing the execution of the EXECUTECASES method on input cl. For each of them, we will demonstrate that after it, cl must meet the requirements of Invariant7.24, thus reaching contradiction for the first possibility. The last computation step in an execution of the EXECUTECASES method can be one of the following. a) Reading a failure value out of a CAS-descriptor's `state` field and breaking out of the loop (lines 8-9)[6]. Thus, by Invariant 7.23, which is already proved for $i+1$ steps, the fact that the CAS-descriptor's `state` field is failure (not pending), proves that the `states` of all the previous CAS-descriptors in the list are success, and the fact that the CAS-descriptor's `state` field is failure (not success), proves that the `states` of all the later CAS-descriptor in the list are pending.

b) Writing a failure value to a CAS-descriptor's `state` field and breaking out of the loop (lines 16-17). Again, by Invariant 7.23, the fact that the CAS-descriptor's `state` is failing implies that earlier CAS-descriptors's `states` are success and later CAS-descriptor's `states` are pending.

c) attempting to clear the `modified-bit` and "continuing" after the last iteration of the loop in lines 5-6. In this case, the fact that the condition in line 4 was true implies that the `state` of the last CAS-descriptor in the list was success, and by Invariant 7.19, which is already proved to $i+1$ steps, the `state` of the last CAS-descriptor must still be success after $i+1$ steps. Thus, using Invariant 7.23, which is also proved for $i+1$ steps, the `states` of all the previous CAS-descriptors must be success as well.

d) Reading a success value out of the last CAS-descriptor in a CAS-list and finishing the last loop iteration (line 15). In this case, again, the `state` of the last CAS-descriptor is success, and thus, using Invariant 7.23, the `states` of all the previous CAS-descriptors are also success. In all of the cases (a)-(d), the CAS-list meets the requirements of Invariant 7.24, and thus the invariant is not violated, yielding contradiction for the first possibility.

Now consider the second possibility. By Invariant 7.19, which is already proved for $i+1$ steps, if the `state` of a CAS-descriptor is not pending then it never changes again. Thus, in step $i+1$ thread t must be changing the `state` of c from pending to a different value. However, since cl met the requirements of Invariant 7.24 for a CAS-list used as input for a completed EXECUTECASES method after i steps, and yet c, which belongs to cl, has it `state` set to pending, it means that after i steps there must be a CAS-descriptor in cl before c, whose `state` is failure. By Invariant 7.23, which is already proved for $i+1$ steps, after $i+1$ steps, if a CAS-descriptor's `state` is not pending, then the `states` of all previous CAS-descriptors in the same CAS-list are success. Thus, changing c's state to anything other than pending in step $i+1$ yields contradiction.

Proof: (Invariant 7.25.) Assume by way of contradiction that in step $i+1$ thread t executes a step that violates Invariant 7.25 for a CAS-descriptor c. By using the induction hypothesis for the same Invariant

[6]note that breaking out of the loop is not a computation step by itself, since it is neither a READ, WRITE or CAS to the shared memory, but just an internal computation.

7.25, such a step must be one of the following.
1) A successful execution of c (after which neither of the post conditions holds).
2) Changing c's `state` field either from pending to failure, of from success to a different value.
3) Changing the value stored in c's `target address` from the `new-value` with a set `modified-bit` to a different value (while the `state` is pending).

We will go over each of these possibilities. In the first case, step $i+1$ must be the first execution of c (by Invariant 7.20, which is already proved for $i+1$ steps). Thus, by the induction hypothesis (Invariant 7.16) c's `state` must be pending after i steps. Thus, after $i+1$ steps, c's state is still pending (since executing c does not change its `state` field), and since the execution in step $i+1$ is successful, then after $i+1$ steps the value stored in c's `target address` is c's `new-value`, with the `modified-bit` set. It follows that after step $i+1$ Invariant 7.25 still holds, yielding contradiction for the first case.

Consider the second case. Recall we assumed that step $i+1$ violates Invariant 7.25. For the second case (i.e., a change of c's state field) to violate the invariant, c must have been successfully executed at some step before step $i+1$. By Invariant 7.20, any attempt but the first attempt to execute c cannot be successful. Thus, the first attempt to execute c must have been successful. It follows that step $i+1$ cannot change the `state` of c to failure, by using Invariant 7.17, which is already proved for $i+1$ steps. Furthermore, step $i+1$ also cannot change c's state to pending, simply because no line in the code does that, yielding contradiction for the second case.

Finally, consider the third case. By Assumption 7.4, changing the value stored in a `target address` of any CAS-descriptor, while the `modified-bit` is set, cannot be done outside the EXECUTECASES method. The only places in the code where the contents of an address with a set `modified-bit` can be switched are the CLEARBIT instructions in lines 5 and 13. However, note that in order to reach a contradiction, we need to refer both to the possibility that step $i+1$ changes the value stored in c's `target address` because it is an execution of the CLEARBIT of c, and that step $i+1$ changes the value stored in c's `target address` because it is an execution of a CLEARBIT of a different CAS-descriptor c', that shares the same `target address`.

If step $i+1$ is a CLEARBIT of c, then in order to execute it either in line 5 or 13, c's `state` must be previously checked and found to be success. By using the induction hypothesis (Invariant 7.19) the state of c must still be success after i steps, and since changing the value stored in the target address does not change the `state`, then also after $i+1$ steps. Thus, the invariant holds after step $i+1$, yielding contradiction for this particular sub-case of the third case.

Now consider the possibility that step $i+1$ is a CLEARBIT of a CAS-descriptor c' different than c that shares the same `target address`. By the assumption of the third case, the value stored in the `target address` after i computation steps is the `new-value` of c with the `modified-bit` set. Thus, in order for the CLEARBIT of c' to successfully change this value, c and c' must both have the exact same `new-value`, including the version number. Thus, it is impossible for both c and c' to be executed successfully, since the first one of them that is executed successfully increments the version number. We assumed (contradictively) that c was executed successfully, and thus c' cannot be successful. Thus, by the induction hypothesis (Invariant 7.18) the `state` of c' cannot be success in the first i computation steps, and thus a CLEARBIT instruction of c' cannot be reached for the $i+1$-st step, completing the contradiction.

Proof: (Invariant 7.26.) Assume by way of contradiction that after $i+1$ steps 1) thread t_1 is executing the loop in lines 2–17 in which the CAS-descriptor c is read, 2) c's `state` is pending, and 3) c is not the most recently active CAS-descriptor. By the induction hypothesis for the same invariant (Invariant 7.26), one of these three is not true after i steps. Thus, one of the following holds.
1) In step $i+1$ t_1 starts executing a new iteration of the loop in lines 2–17. (This could also be the first iteration in a new EXECUTECASES invocation.) c is the CAS-descriptor for this new iteration, c's `state` is pending, and c is not the most recently active CAS-descriptor.

2) In step $i+1$ c's state is changed back to pending.
3) In step $i+1$ a thread t_2 starts executing a new iteration of the loop in lines 2–17 (possibly the first iteration in a new EXECUTECASES invocation), thus making c no longer the most recently active CAS-descriptor.

We consider each of these cases. In the first case, let t_2 be the thread that executed (or is executing) an iteration that is after the iteration t_1 is currently executing. (If no such thread exists, then c is the most recently active CAS-descriptor and we are done. Also, note that we do not assume $t_1 \neq t_2$.) If t_2 is executing (or was executing) a later iteration than t_1 is currently executing, then we consider two possibilities. The first possibility is that t_2 is executing (or was executing) a later iteration on the same CAS-list that t_1 is iterating on. This case leads to a contradiction because c's `state` cannot be pending by Invariant 7.21, which is already proved for $i+1$ iterations. The second possibility is that t_2 is iterating (or was iterating) on a different cl than t_1 is currently iterating on. Thus, by Assumption 7.2, some thread already completed the execution of an EXECUTECASES method with cl as the input. This leads to a contradiction because by Invariant 7.24, which is already proved for $i+1$ steps, either the `states` of all the CAS-descriptor are success (and then c's `state` cannot be pending), or that there is a CAS-descriptor with a `state` failure before c (and then, by using Invariant 7.21, t_1 cannot be executing the iteration in which c is read).

We now turn to consider the second case. This case yields a contradiction immediately, because no line of code inside the EXECUTECASES changes a state back to pending, and by Assumption 7.6, no line of code outside the EXECUTECASES method does that either.

Finally, we consider the third case. The proof here is very similar to the first case. We consider two possibilities. The first possibility is that t_2 is executing a later iteration on the same CAS-list that t_1 is iterating on. This case leads to a contradiction because c's `state` cannot be pending by Invariant 7.21, which is already proved for $i+1$ iterations. The second possibility is that t_2 is iterating on a different cl than t_1 is iterating on. Thus, by Assumption 7.2, some thread already completed the execution of an EXECUTECASES method with cl as the input. This leads to a contradiction because by Invariant 7.24, which is already proved for $i+1$ steps, either the `states` of all the CAS-descriptor are success (and then c's `state` cannot be pending), or that there is a CAS-descriptor with a `state` failure before c (and then, by using Invariant 7.21, t_1 cannot be executing the iteration in which c is read).

Proof: (Invariant 7.27.) By the induction hypothesis for the same invariant (Invariant 7.27), the invariant holds after i steps. Assume by way of contradiction that the invariant does not hold after $i+1$ steps. Thus, the $i+1$-st step must be one of the following.
1) A thread t_2 successfully executes a CAS-descriptor c (line 10), while a different thread t has already completed a loop iteration in which c was read.
2) A thread t completes the execution of an iteration in which c is read, while there is still a `modified-bit` that is set to true and that belongs to c.

If the first case is true, then by Invariant 7.20, which is already proved for $i+1$ steps, step $i+1$ must be the first step in which c is executed. Consider t's execution of the iteration in which c is read. If t reached line 6, then c's `state` much have been success, which by the induction hypothesis (Invariant 7.18) means c had been executed before. If t reached line 9, then c's `state` must have been failure, which by the induction hypothesis (Invariant 7.17) also means c had been executed before. If t did not complete the loop in either line 6 or 9, then t must have reached and executed line 10, which again means that c was executed before step $i+1$. Whichever way t completed the iteration, CAS-descriptor c must have been executed before step $i+1$, thus it cannot be executed successfully in step $i+1$, yielding contradiction for the first case.

If the second case is true, then consider the different possibilities for t to complete the loop. If t breaks out of the loop in line 9 or in line 17, then c's `state` is failure. By Invariant 7.17, which is already proved for $i+1$ steps, this means the first attempt to execute c was not successful. By Invariant 7.20, it follows that no execution of c is successful until step $i+1$. It follows that there is no `modified-bit` that belongs to c, yielding contradiction for this sub-case of the second case.

If t completes the loop via the continue in line 6 then in t's last execution step inside the loop (which is assumed to be step $i+1$ of the execution) t attempts by a CAS to clear the `modified-bit`. If the `modified-bit` is previously set to true and belongs to c, then the value stored in c's `target address` is the same as the `expected-value` for the CLEARBIT CAS, and the `modified-bit` will be successfully cleared, yielding contradiction for this sub-case of the second case.

If t completes the loop by reading a success value out of c's `state` field and then reaching the end in line 15, then consider the execution when t executes line 11 of the same iteration. If the `modified-bit` is off at that time, then a `modified-bit` cannot belong to c at step $i+1$, since c has already been executed at least once, and thus further attempts of it until step $i+1$ must fail (Invariant 7.20). If the `modified-bit` is on, then t will reach line 12. When t executes the CAS in this line, then either the `state` is changed from pending to success, either the `state` is already success (the `state` cannot be failure, otherwise t would not have read a success value from it in line 15, because a non-pending state is final (by the induction hypothesis (Invariant 7.19). It follows that when t reached line 13, it attempted a CLEARBIT CAS to clear the `modified-bit`. If the `modified-bit` is previously set to true and belongs to c, then the value stored in c's `target address` is the same as the `expected-value` for the CLEARBIT CAS, and the `modified-bit` will be successfully cleared, yielding contradiction.

Proof: (Invariant 7.28.) By the induction hypothesis for the same invariant (Invariant 7.28), the invariant holds after i steps. Assume by way of contradiction that the invariant does not hold after $i+1$ steps. Thus, the $i+1$-st step must be one of the following.
1) A thread t successfully executes a CAS-descriptor c (line 10), while c is not the most recently active CAS-descriptor.
2) A thread t starts a new iteration of the loop in lines 2–17, thus making c no longer the most recently active CAS-descriptor, while a `modified-bit` that belongs to c is on.

Consider the first case. Since c is successfully executed at step $i+1$, then by Invariant 7.20, which is already proved for $i+1$ steps, this must be the first attempt to execute c. Thus, by using the induction hypothesis (Invariant 7.16), c's `state` must be pending. Thus, by the fact that t is currently executing the loop iteration in which c is read, and by using the induction hypothesis (Invariant 7.26), c is the most recently active CAS-descriptor, yielding contradiction for the first case.

Now consider the second case. We claim that since t starts an iteration that is after the iteration in which c is read, then *some* thread t' (which may be t) has previously completed an iteration of the EXECUTECASES method in which c is read. To see this, consider the iteration that t starts. If it is a later iteration on the same CAS-list to which c belong, then t itself must have completed the iteration in which c is read (thus, $t' = t$). If it is a later iteration on a different CAS-list, then by Assumption 7.2, some thread (which is t') has already completed an execution of the EXECUTECASES method on the CAS-list to which c belong. To complete the EXECUTECASES method, t' must either complete the iteration in which c is read, or break out of the loop earlier. However, t' cannot break out of the loop earlier, because that requires a CAS-descriptor with a failure `state` to be in the CAS-list before c, and if that were the case, then by the induction hypothesis (Invariant 7.22) c could not have been executed, and thus there could not have been a `modified-bit` belonging to c. To conclude, some thread t' has completed an iteration of the EXECUTECASES method in which c is read. It follows by Invariant 7.27, which is already proved for $i+1$ steps, that there is no `modified-bit` belonging to c, yielding contradiction.

At this point, Invariants 7.16–7.28 are all proved to be correct throughout the execution. Relying on these invariants, we now complete the proof for the correctness of the EXECUTECASES method.

Observation 7.29 *All execution steps inside the* EXECUTECASES *method are either: a) an execution of a* CAS*-descriptor, b) a* CLEARBIT *of a* CAS*-descriptor, or c) applied on the memory of the* CAS*-list.*

Proof: True by observing the code. Line 10 (execution of a CAS-descriptor) and lines 5,13 (CLEARBIT of a CAS-descriptors) are the only lines that execute on shared memory that is not inside the CAS-list. The other computation steps either read a `state` field of a CAS-descriptor, write to a `state` field, execute a CAS on a `state` field, or read the number of CASes in the CAS-list.

Claim 7.30 *Before a* CLEARBIT *of a* CAS*-descriptor c is executed for the first time, c has been successfully executed.*

Proof: A CLEARBIT for a CAS-descriptor c can only be attempted (either in line 5,13) if the `state` of the c was previously read and turned out to be success. By Invariant 7.18, this means that c had been successfully executed before.

Claim 7.31 *For every* CAS*-descriptor c:*
1) Any attempt to execute c except the first attempt (by some thread) must fail.
2) Any attempt to execute the CLEARBIT *of c except the first attempt (by some thread) must fail.*

Proof: (1) is simply restating the already proved Invariant 7.20. It remains to prove (2). Recall that an execution of a CLEARBIT is an execution of a CAS in which the `target address` is c's `target address`, the `expected-value` is c's `new-value` (including the version number) except that the `modified-bit` is on, and the `new-value` is the exact `new-value` of c. By Claim 7.30, when c's CLEARBIT is executed, c has already been successfully executed, and it follows that the version number stored in the `target address` is already at least equals to the version number of the `expected-value` of the CLEARBIT CAS. By Assumption 7.5, the version number is incremented in every successful CAS outside the EXECUTECASES method. It follows that the version is incremented in any successful CAS excluding the CLEARBIT CAS, in which it remains the same. Thus, If the first execution of the CLEARBIT CAS fails, every further execution of it must fail as well, since the value stored in the `target address` can never hold the `expected-value` of the CAS. Similarly, if the first execution c's CLEARBIT is successful, then after it the `modified-bit` is off, and cannot be set on again without the version number being incremented. And thus, additional executions of c's CLEARBIT CAS must fail.

Claim 7.32 *A* `modified-bit` *that belongs to a* CAS*-descriptor c can only be turned off by executing the* CLEARBIT *of c.*

By Assumption 7.4, a `modified-bit` cannot be turned off outside the EXECUTECASES method since CASes outside the EXECUTECASES method always expect the `modified-bit` to be off. Inside the EXECUTECASES method, a `modified-bit` can only potentially be turned off when executing a CLEARBIT CAS. It remains to show that a `modified-bit` that belongs to a CAS-descriptor c cannot be turned off by executing a CLEARBIT of a different CAS-descriptor c'.

If any `modified-bit` belongs to c, it follows that c has been successfully executed. By Claim 7.30, to execute the CLEARBIT of c', c' must first also be successfully executed. In order for the CLEARBIT of c' to turn off a `modified-bit` that belongs to c, both c and c' must have the same `target address`, and, moreover, the same `new-value`, otherwise executing the CLEARBIT of c' would fail. However, if both c and c' have the same `new-value`, both must share the same version number in the `expected-value`, which implies that only one of them can possibly succeed. Thus, c' couldn't have been successfully executed, and thus it cannot clear the `modified-bit` of c.

Claim 7.33 *Before a* CAS*-descriptor c in a* CAS*-list cl is executed for the first time:*
1) All the previous CAS*-descriptors in cl have been* successfully *executed.*
2) CLEARBIT *has already been executed for all the previous* CAS*-descriptors in cl.*
(Note: the claim vacuously holds for CAS*-descriptors that are never executed.)*

Proof: By Invariant 7.21, when c is executed, all the previous CAS-descriptors in cl has their `state` set to success, which by Invariant 7.18 means they have all been successfully executed, proving (1). By Invariant 7.27, all `modified-bits` of all the previous CAS-descriptors have already been switched off, which by Claim 7.32 implies that the CLEARBIT of all the previous CAS-descriptors in cl has already been executed, proving (2).

Claim 7.34 *For any* CAS*-descriptor c, the first attempt to execute the* CLEARBIT *of* c *(by some thread) is successful. (Note: the claim vacuously holds for* CAS*-descriptors for which a* CLEARBIT *is never executed.)*

Proof: Immediately after executing c, the value stored in the `target address` is exactly the `expected-value` of the CLEARBIT CAS. This value cannot be changed before a CLEARBIT CAS is executed, since no CAS except the CLEARBIT expects to find the `modified-bit` on, and there are no writes (without a CAS) to the `target address` (Assumption 7.4). Thus, until a CLEARBIT is executed on this address, the value remains unchanged. By Claim 7.32, a CLEARBIT of a CAS-descriptor other than c cannot be successful. Thus, the value in the `target address` remains the expected value of the CLEARBIT CAS until the CLEARBIT is executed, and thus, the first attempt to execute the CLEARBIT of c is successful.

Claim 7.35 *Once some thread has completed executing the* EXECUTECASES *method on an input* CAS*-list* cl *the following holds.*
1) Either all the CAS*-descriptors have been successfully executed, or all the* CAS*-descriptors have been executed until one that fails. Further* CAS*-descriptors (after the first one that fails) have not been executed, and will also not be executed in the rest of the computation.*
2) A CLEARBIT *was successfully executed for each* CAS*-descriptor that was successfully executed.*

Proof: By Claim 7.24, once some thread has completed the EXECUTECASES method on the input cl, either the `state` field of all the CAS-descriptors cl is set to success, or that one of them is set to failure, the ones previous to it to success, and the ones after it to pending. By Invariants 7.17 and 7.18, the CAS-descriptors whose `state` is success were executed successfully, and the CAS descriptor whose `state` is failure failed. By Invariant 7.22, CAS-descriptors after the CAS-descriptor that failed are not executed. Thus, (1) holds.

The thread that completed executing the EXECUTECASES method on cl, has completed executing an iteration for each successful CAS-descriptor in cl, and thus by Invariant 7.27, all the `modified-bits` have already been switched off. By Claim 7.32, a `modified-bit` can only be turned off by a CLEARBIT of the CAS-descriptor that previously set the bit on, and thus, it follows that a CLEARBIT was successfully executed for each successful CAS-descriptor, and (2) holds.

Claim 7.36 *The return value of the* EXECUTECASES *for* every *thread that completes it is:*
1) The index of the first (and only) CAS*-descriptor whose execution failed the first time it was attempted, if such exists.*
2) -1 otherwise.

Proof: Each thread that executes the EXECUTECASES method may exit it via one of three possible code-lines: 9, 17 or 20. If the thread exited via line 9, or via line 17, and returned i (the loop variable), then the `state` of the ith CAS-descriptor is failure, and thus its execution has failed by Invariant 7.17. By Claim 7.35 (1), this must be the only CAS that failed. Thus, in the case that a thread exits via line 9 or via line 17, the returned value is that of the first and only CAS-descriptor whose execution failed the first time it was attempted.

If a thread reaches line 20 and returns -1, then immediately before that it must be executing the last iteration of the loop in lines 2–19. Thus, by Invariant 7.21, the `states` of all the previous CAS-descriptors are success, and thus, by Invariant 7.18, all the CAS-descriptors before the last one were executed successfully.

As to the last one, its `state` must be success as well (and thus, it must also have succeeded), otherwise when the thread reads the CAS-descriptors `state` and compares it to success in line 15, it would enter the if clause and leave through line 17. Thus, in the case that a thread exits reaches 20, all the CAS-descriptors were executed successfully, and -1 is returned.

Lemma 7.37 *The implementation of the* EXECUTECASES *method as given in Figure 5, is correct, meaning that it satisfies Definition 7.1.*

Proof: Follows from Observation 7.29, and Claims 7.31, 7.33, 7.35, and 7.36.

7.2 Linearizability and WaitFreedom

Assume that LF is a linearizable lock-free algorithm given in the normalized form for a certain abstract data type, ADT. Let WF be the output algorithm of our transformation as described in Section 6 with LF being the simulated lock-free algorithm. Our goal is to prove that WF is a linearizable wait-free algorithm for the same abstract data type, ADT.

We claim that for every execution of WF, there is an *equivalent execution* (Definition 4.1) of LF. Since we know that LF is correct and linearizable, it immediately follows that WF is correct and linearizable as well. We start from a given execution of WF, denoted E_0, and we reach an equivalent execution of LF in several steps.

For each intermediate step, we are required to prove two key points. First, that the newly created execution preserves memory consistency. That is, each READ returns the last value written (or put via CAS) to the memory, and each CAS succeeds if and only if the value previously stored in the `target address` equals the `expected-value`. Proving memory consistency is required in order to prove that the newly created execution is indeed an execution.

Second, for each intermediate step, we are required to prove equivalency. That is, that each thread executes the same data structure operations in both executions, that the results are the same, and that the relative order of invocation and return points is unchanged. For the last execution in the series of equivalent executions, we will also prove that it is an execution of LF.

7.2.1 Step I: Removing Steps that Belong to the Additional Memory used by WF

WF uses additional memory than what is required by LF. Specifically, WF uses a `help queue`, in which it stores operation record boxes, which point to operation records. Operation records hold CAS-lists, which are in fact also used by LF, only that the CAS lists used by WF holds an extra `state` field for each CAS, not used in the original LF algorithm. In this step we erase all the computation steps (READS, WRITES, and CASES) on the additional memory used by WF.

Let E_1 be the execution resulting from removing from E_0 all the execution steps on the additional memory (the memory of the `help queue`, the operation record boxes, and the operation records excluding the CAS-lists - yet including the `state` field of each CAS in the CAS-lists).

Claim 7.38 *E_0 and E_1 are equivalent, and E_1 preserves memory consistency.*

Proof: E_1 has the same invocations and results of operations as E_0, and their relative order remain unchanged, thus E_0 and E_1 are equivalent by definition. E_1 preserves memory consistency since E_0 is memory consistent, and each memory register used in E_1 is used in E_1 in exactly the same way (same primitives with same operands, results, and order) as in E_0.

7.2.2 Step II: Tweaking CASes of the EXECUTECASES Method

Most of the steps of E_0 that belong to neither the GENERATOR, WRAPUP or CAS-EXECUTER method were dropped in E_1. However, in E_1 there are still two sources for steps that should be dropped. The main source is the EXECUTECASES method (the other source will be reminded shortly). Recall that E_0 is an execution of WF, which employs both the CAS EXECUTER method (in the fast path) and the concurrent EXECUTECASES method (in the slow path), while the original algorithm LF only employs the CAS EXECUTER method. By Lemma 7.37, all the computation steps of the EXECUTECASES method are either executing a CAS-descriptor, executing a CLEARBIT of a CAS-descriptor, or steps on the `state` field of a CAS-descriptor in the CAS list.

Steps on the `state` field were already dropped in the move from E_0 to E_1. Next, according to Lemma 7.37, each execution of a CAS-descriptor that is not the first attempt to execute a given CAS-descriptor, and each execution of a CLEARBIT that is not the first attempt to execute the CLEARBIT for the same CAS-descriptor, must fail. It follows that these CASes do not modify the memory and can be dropped without violating memory consistency. Afterwards, according to Lemma 7.37, what remains of the EXECUTECASES are pairs of successful CASes: each successful execution of a CAS-descriptor is followed by a successful execution of a CLEARBIT CAS of the same descriptor. Possibly, at the end of these successful pairs remains a single unsuccessful execution of a CAS-descriptor.

We now tweak these pairs CASes to be identical to an execution of the (fast path) CAS-EXECUTER method. To do that, each pair is merged into a single CAS. More precisely, the `new-value` of each execution of a CAS-descriptor is changed such that the `modified-bit` is off (this alternative `new-value` is the same as the original `new-value` of the following CLEARBIT CAS), and each CLEARBIT CAS is dropped. After this change what remains of the EXECUTECASES method is identical to the CAS-EXECUTER method (except that the CASes are executed by several thread instead of by a single thread, but this will be handled when moving from E_2 to E_3). However, the last change can potentially violate memory consistency.

Memory consistency is potentially violated for READ primitives that were originally (that is, in E_0 and E_1) executed between an execution of a CAS-descriptor to the following CLEARBIT CAS. Memory consistency is violated because the value stored in the `target address` now has the `modified-bit` switched off immediately after the first execution of the CAS, instead of being switched off only after the CLEARBIT CAS. More importantly than READ primitives, the memory consistency of CAS primitives executed (in E_0 and E_1) between a CAS-descriptor and the following CLEARBIT CAS is also potentially violated.

To regain memory consistency, READ primitives in between a pair are changed such that their returned value indicates that the `modified-bit` is unset. Recall that when we described the changes induced to the fast-path in our transformation, we mentioned that all READ operations always disregard the `modified-bit` (the fast-path acts as if the bit were off). Thus, changing the execution such that now the bit is really off only takes us "closer" into an execution of LF.

CAS primitives that occurred in between a pair of CASes are handled as follows. Recall that in order to be compatible with the `modified-bit`, the fast path in WF is slightly altered. This is the second source of computation steps (the first being the CLEARBIT CASes) that belong to WF and that do not originate from the three methods of the normalized structure. Whenever a CAS is attempted and failed in the fast-path of WF, the same memory address is subsequently read. If the value is such that implies that the CAS could have succeeded were the `modified-bit` switched off, then HELP is called, and then the CAS is retried. In what follows we simultaneously remove the extra READs and CASes originating from this modification of the fast-path and restore memory consistency.

For each CAS that failed in the fast-path, examine the corresponding READ following it. If the result of this READ indicates that the CAS should fail regardless of the `modified-bit`, then move the CAS forward in the execution to be at the place where the READ is, and drop the READ. If the results of the READ

indicates that the CAS should succeed (or can succeed if the `modified-bit` would be switched off), then drop both the CAS and the READ. (The re-attempt of the CAS is guaranteed to be after the `modified-bit` is switched off.) We are now ready to formally define E_2.

Let E_2 be the execution resulted from applying the following changes to E_1.

- Each execution of a CAS-descriptor in the EXECUTECASES method, excluding the first attempt for each CAS-descriptor, is dropped.
- Each execution of a CLEARBIT CAS is dropped.
- The remaining execution of CAS-descriptors in the EXECUTECASES method are changed such that their `new-value` has the `modified-bit` off.
- For each unsuccessful CAS executed in the fast path:
  - If the CAS was re-attempted as a result of the subsequent corresponding READ, drop both the CAS and the READ, and keep only the re-attempt of the CAS (regardless whether this re-attempt succeeds or fails.)
  - Otherwise, move the CAS later in the execution to the place where the subsequent READ is, and drop the READ.
- (Remaining) READ primitives that were originally between a pair of a CAS-descriptor execution and the corresponding CLEARBIT execution, and that target the same memory address such as these CASes, are modified such that their returned value has the `modified-bit` switched off.

Claim 7.39 *E_2 and E_1 are equivalent, and E_2 preserves memory consistency.*

Proof: E_2 has the same invocations and results of operations as E_1, and their relative order remain unchanged, thus E_1 and E_2 are equivalent by definition. Dropping executions of CAS-descriptors that are not the first attempt of a given CAS-descriptor cannot violate memory consistency, because these CASes are unsuccessful by Lemma 7.37, and thus do not change the memory. Dropping the CLEARBIT CASes together with modifying the execution of the CAS-descriptors such that they set the `modified-bit` to off changes the state of the memory only for the time between each such pair of CASes, and thus can only violate memory consistency at these times. Consider the primitives that occur at these time frames.

By the definition of the normalized form, WRITE primitives are not used on these addresses. Furthermore, there could be no successful CASes between such a pair of CASes, because the `modified-bit` is on at these times, and the CLEARBIT CAS is the only CAS that ever has the `modified-bit` set in its `expected-value`. An Unsuccessful CAS receives special treatment. It is followed by a designated READ. If this READ determines the CAS can fail regardless of the `modified-bit`, then at the time of the READ, the CAS can fail without violating memory consistency in E_2 as well. Since in E_2 this CAS is moved in place of the READ (and the READ is dropped), then memory consistency is preserved for these CASes as well.

If the designated READ determines that the CAS may succeed, then the CAS is re-attempted. In such a case the CAS (together with the READ is dropped, and thus it does not violate memory consistency anymore. As for the re-attempt CAS, because it is only attempted after HELP is called, it is guaranteed to be executed after the CLEARBIT CAS. There are thus two options. Either the re-attempt CAS succeeds (both in E_1 and in E_2), and thus it is certainly not between a a pair of CASes, or the re-attempt CAS can fail. If it fails, then this cannot violate memory consistency. This is true even if the re-attempt CAS occurs between a (different) pair of CASes, because the fact that the CAS is re-attempted implies that its version number suits the previous pair of CASes, and cannot suit the new pair that is surrounding the re-attempt CAS.

As for other READ primitives between a pair of CASes(other than the designated READ that are specially inserted after a failure in a CAS), they are modified to return the value with the `modified-bit` off. Thus, memory consistency is restored for these READ primitives as well.

7.2.3 Step III: Changing the Threads that Executed the Steps

In E_2 all the execution steps belong, or could legitimately belong, to one of the GENERATOR, WRAPUP, and cas executer methods. However, the threads that executes the steps are still mixed up differently than in LF. In this step the execution steps or their order are not altered, but the threads that execute them are switched. In E_3, the original threads of E_2 (which are the same as the threads of E_1 and of E_0) act accordingly to LF, and other additional threads (not present in E_2) are created to execute redundant runs of the GENERATOR and WRAPUP methods.

While a thread executes an operation in the fast path, without helping other operations, he follows the original LF algorithm. However, this changes when a thread moves to the slow path. First, a thread can move to the slow path because the contention failure counter of either the GENERATOR or WRAPUP methods causes it to stop. In such a case, the method has not been completed and will be executed again in the slow path. The execution steps originating from this uncompleted method are thus moved to an additional thread created for this purpose.

In the slow path, we examine all the executions of the GENERATOR and WRAPUP methods. For each execution of such a method, we go back and examine what happens afterwards in E_0. If the thread that executes this method in E_0 later successfully CAS the operation record with the method's result to the operation record box (either in line 5 in the PRECASES method (Figure 4) or in lines 6 or 8 in the POSTCASES method (Figure 6)), then the computation steps of this method are moved to the owner of the operation being helped (the thread that asked for help). Note that it is also possible that these steps belong to this owner thread in the first place, and are not moved at all.

If the thread that executes the method (either GENERATOR or WRAPUP) does not successfully CAS the result of the method into the operation record box, then the results of the method are simply discarded and never used. In this case, the computation steps of this method are moved to an additional thread created for this method only.

It remains to switch the owner of the CASes originating from the EXECUTECASES method of the slow path. Some of them were dropped in the move from E_1 to E_2, and the rest were modified. We set the owner of the operation being helped (the thread that asked for help) to be the thread that executes these remaining CASes.

Let E_3 be the execution resulted from applying the following changes to E_2.

- For each GENERATOR method or WRAPUP method that is not completed due to contention (either in the fast path or in the slow path), create an additional thread, and let it execute the computation steps originating from this method.
- For each GENERATOR method or WRAPUP method executed in the slow path, whose results are not later successfully CASed into the operation record box, create an additional thread, and let it execute the computation steps originating from this method.
- For each GENERATOR method or WRAPUP method executed in the slow path, whose results *are* later successfully CASed into the operation record box, let the owner thread of the operation being helped execute the computation steps originating from this method.
- For each execution of the EXECUTECASES method, let the owner of the operation being helped execute the CASes that originated from this method (if any remained in E_2).

Since E_3 includes additional threads that are not a part of E_2, we can only claim that E_3 and E_2 are equivalent when considering only the threads that participate in E_2. We formalize this limited equivalency as follows.

Definition 7.40 (Limited Equivalency of Executions.) *For two executions E and E' we say that E limited to the threads of E' and E' are equivalent if the following holds.*

- *(Results:) The threads of E' execute the same data structure operations and receive identical results in both E' and E.*
- *(Relative Operation Order:) The order of invocation points and return points of all data structure operations is the same in both executions. In particular, this means that threads of E that do not participate in E' execute no data structure operations.*
- *(Comparable length:) either both executions are finite, or both executions are infinite.*

Claim 7.41 *E_3 limited to the threads of E_2 and E_2 are equivalent, and E_3 preserves memory consistency.*

Proof: All the threads of E_2 have the same invocations and results of operations in E_3 that they have in E_2, and their relative order remains unchanged, thus E_3 and E_2 are equivalent by definition. By Claim 7.39, E_2 preserves memory consistency. E_3 only differs from E_2 in the threads that execute the primitive steps, but the steps themselves and their order remain unchanged, thus E_3 preserves memory consistency as well.

Claim 7.42 *E_3 is an execution of LF, possibly with additional threads executing the* GENERATOR *and* WRAPUP *methods.*

Proof: By Claim 7.41, E_3 preserves memory consistency. It remains to show that each thread in E_3 either 1) follows the LF program structure of GENERATOR, CAS EXECUTER and WRAPUP methods, or 2) executes a single parallelizable method (either the GENERATOR or WRAPUP). To do this, we need to simultaneously consider executions E_3 and E_0. Note that each computation step in E_3 originates from a single computation step in E_0. (Some computation steps from E_0 were dropped and have no corresponding computation steps in E_3. Some computation steps in E_0 were slightly altered by changing the value of the `modified-bit`, and some were transferred to a different thread. Still, each computation step in E_3 originates from a single specific computation step in E_0.)

Fix an operation executed in E_3 and follow the thread that executes it. Originally, in E_0, the thread starts by offering help. However, all the computation steps that involve reading the help queue and operation records were already dropped in the move from E_0 to E_1; the remaining computation steps that involve helping the operation of a different thread are transferred either to the thread being helped or to an additional thread in the move from E_2 to E_3. Thus, in E_3 the thread starts executing the GENERATOR directly.

Originally, in E_0, while the execution is in the fast-path it is similar to LF with three small modifications. The first modification is that after executing a CAS that fails, the thread executes a READ on the `target address`, and then possibly re-executes the CAS. These extra steps were dropped in the transition from E_1 to E_2. The second modification is that the execution of the GENERATOR and WRAPUP methods is monitored, in the sense that a contention failure counter is updated and read periodically. However, there is no need for the contention failure counter to be in the shared memory. It is in a thread's local memory, and thus such monitoring occurs in the local steps and is not reflected in the execution. It only affects the execution if the contention threshold is reached and help is asked. The third modification is that the number of times that the WRAPUP method indicates that the operation should be restarted from scratch is also monitored, in order to move to the slow-path if this number reaches a predetermined limit. Similarly to the contention failures counter, this monitoring is done within a threads's local computation.

Thus, as long as the execution of an operation in E_0 is in the fast-path (which could very well be until its completion), the corresponding execution in E_3 of the operation's owner thread is according to LF. Next,

we examine what happens in E_0 when the thread asks for help and move to the slow-path. The method that was interrupted by the contention failure counter (if any) is transferred to an additional thread.

Once an operation in E_0 is in the slow path, the owner thread, and possibly helping threads, start executing one of three methods: the GENERATOR, EXECUTECASES, or WRAPUP, depending on the `state` of the operation record pointed by the operation record box. We examine how this execution is reflected in E_3.

For the GENERATOR and WRAPUP methods, the owner thread (the thread that asked for the help) executes in E_3 the steps of the thread that in E_0 successfully replaced the operation record with a CAS. These steps were transferred to the owner thread in the transition from E_2 to E_3. Other executions of the GENERATOR and WRAPUP methods, by threads that did not successfully replaced the operation record, are transferred to additional threads. Since only one thread may successfully CAS the operation record box from pointing to a given operation record to point to a new one, then in E_3 the owner thread executes the required parallelizable method (either GENERATOR or WRAPUP) once, as is done in an execution of LF. Afterwards, in E_0, helping threads will start executing the next required method (if any) according to the new `state` of the operation record.

The case is different for the EXECUTECASES method. Executions of the EXECUTECASES method are not transferred to additional threads, and the steps that are transferred to the owner in the transition from E_2 to E_3 were possibly executed by several different threads in E_0. To see that the steps that are executed in E_3 by the owner are indeed an execution of the CAS-EXECUTER method, we rely on Lemma 7.37. By this lemma, the first attempts of all the CAS-descriptors in the CAS-list are done according to their order, and once the first CAS-descriptor fails, the following CAS-descriptors in the list will not be attempted. In the transition from E_1 to E_2, only these first attempts of each CAS-descriptor in the list are kept, and further attempts are dropped. Also, the attempted CASes are changed and have the `modified-bit` of the `new-value` switched off. These modified CASes are transferred to the owner thread in the transition from E_2 to E_3.

Thus, in E_3, the owner thread executes the CASes of the list one by one according to their order, until one of them fails. This is simply an execution of the CAS-EXECUTER method. By Lemma 7.37, before the first thread exits the EXECUTECASES method, all these CASes (all first attempts of CAS-descriptors) have already occurred. Thus, when in E_0 the operation's `state` is changed to post-CASes, and helping threads might start executing the WRAPUP method, all the computation steps of the EXECUTECASES (possibly apart from steps that are dropped in the transition from E_0 to E_1 or from E_1 to E_2) are already completed.

Regarding the output of the EXECUTECASES method, according to Lemma 7.37, the returned value of the EXECUTECASES method is the index of the first CAS that fails, or -1 if all CASes are executed successfully. In E_0, this value is stored inside the operation record and is used as the for the threads that read the operation record and execute the WRAPUP method. Thus, in E_0, and also in E_3, the WRAPUP method execution have the correct input.

We conclude that the execution of each operation in E_3 is according to LF. If in E_0 the operation is completed in the fast-path, then the operation owner executes the operation similarly in E_3, minus extra steps that were dropped, and steps that give help that are transferred either to additional threads or to the owner of the helped operation.

If an operation in E_0 starts in the fast-path and then moves to the slow-path, then the parallelizable methods (GENERATOR and WRAPUP) are transferred to the operation owner if their output was used, or to additional threads if the output was discarded. The execution of the EXECUTECASES is modified to an execution of CAS-EXECUTER and is transferred to the owner thread. Thus, E_3 is an execution of LF, possibly with extra threads, each of them executes once either the GENERATOR method, or the WRAPUP method.

7.2.4 Step IV: Dropping Additional Threads

The purpose of this step is to drop all of the additional threads along with the parallelizable methods they are executing. Each additional thread executes a single parallelizable method. Each additional thread executes only a finite number of steps (because the method it executes is monitored in E_0 by a contention failure counter), and thus only a finite number of successful CASes. Thus, to drop an additional thread along with the parallelizable method it executes, we use the characteristic property of parallelizble methods, as given in Definition 4.4.

For each additional t executing a parallelizable method, we replace the execution with an equivalent execution in which all the threads follow the same program, but t's execution is avoidable. That is, t executes only futile and non-successful CASes. Such an execution, which is also an execution of LF plus additional threads executing parallelizable methods, exists by Definition 4.4. Then, t is simply dropped from the execution entirely. This does not violate memory consistency, because t's execution steps do not alter the data structure at all. This process is repeated for every additional thread.

Let E_4 be the execution resulted from the process describe above. Specifically, for each additional thread t, we replace the execution with an equivalent execution in which t's executed method is avoidable, as is guaranteed by Definition 4.4, and then each additional thread is dropped.

Claim 7.43 *E_3 limited to the threads of E_4 and E_3 are equivalent, and E_4 preserves memory consistency.*

Proof: For each additional thread, the transition to an equivalent execution as guaranteed by Definition 4.4 preserves equivalence and memory consistency. An additional thread that only executes READs, failed CASes, and futile CASes can be dropped without harming memory consistency (as it does not alter the shared memory).

Claim 7.44 *E_2 and E_4 are equivalent.*

Proof: E_2 and E_4 has the same set of threads: threads that are added in the transition from E_2 to E_3 are dropped in the transition from E_3 to E_4. Both E_2 and E_4 are equivalent to E_3 limited to their threads (Claims 7.41 and 7.43). It follows that E_2 and E_4 are equivalent.

Claim 7.45 *E_4 is an execution of LF.*

Proof: By Claim 7.42, E_3 is an execution of LF with possibly additional threads executing parallelizable methods. The equivalent execution guaranteed in Definition 4.4 is such in which each thread follows the same program. Thus, each (non-additional) thread follows the same program in E_3 and in E_4, which means that each thread in E_4 follows an execution of LF. All the additional threads of E_3 are dropped, and thus E_4 is an execution of LF.

7.2.5 Linearizability of WF

Corollary 7.46 *For each execution of WF, there exists an equivalent execution of LF.*

Proof: Follows directly from Claims 7.38, 7.39, 7.44, and 7.45.

Theorem 7.47 *WF is a linearizable.*

Proof: It is given that LF is linearizable. For each execution of WF there exists an equivalent execution of LF (Corollary 7.46). Thus, each execution of WF is linearizable, and WF itself is linearizable.

7.2.6 Wait Freedom of WF

To show that WF is wait-free, we first claim that it is lock-free. Then, we show that due to the helping mechanism, WF cannot be lock-free without being wait-free as well.

Claim 7.48 *WF is lock-free.*

Proof: Assume by way of contradiction that WF is not lock-free. Thus, there exists an infinite execution of WF in which only a finite number of operations are completed. By Corollary 7.46, for each execution of WF exists an equivalent execution of LF. By definition of equivalent executions, the equivalent execution of LF must also be infinite, and only a finite number of operations may be completed in it. This contradicts the fact that LF is lock-free.

Theorem 7.49 *WF is wait-free.*

Proof: Assume by way of contradiction that WF is not wait-free. Thus, there exists an infinite execution of WF, in which some thread executes infinitely many steps yet completes only a finite number of operations. Let E be such an execution, and T the thread that completes only a finite number of operations. Consider the last operation that T starts (which it never completes).

T cannot execute infinitely many steps in the fast-path: executions of the GENERATOR and WRAPUP methods are monitored by a contention failures counter, and at some point in an infinite execution of them the threshold must be reach, and help will be asked. Thus, it is impossible to execute infinitely many steps in a single method of the fast-path. However, it is also impossible to execute infinitely many loops of the GENERATOR, CAS-EXECUTER and WRAPUP methods, since when a certain threshold is reached, help is asked. Thus, at some point, T must ask for help.

When asking for help, T enqueues a help request into the wait-free `help queue`. Since this queue is wait-free, then after a finite number of steps the help request must be successfully enqueued into the queue, with only a finite number of help requests enqueued before it.

While the `help queue` is not empty, each thread, when starting a new operation, will first help the operation at the head of the `help queue` until it is completed and removed from the help queue. Only then, the thread will go and execute its own operation. It follows that once a help request for an operation *op* is enqueued to the `help queue`, each thread can only complete a finite number of operations before *op* is completed. To be accurate, if at a given moment *op* is the n'th operation in the queue, then each thread can complete a maximum of n operations before *op* is completed.

Thus, once T successfully enqueues the help request into the `help queue`, only a finite number of operations can be completed before T completes its operation. Since T never completes its operation, then only a finite number of operations can be completed at all. Thus, in the infinite execution E, only a finite number of operations is completed. This contradicts the fact that WF is lock-free (Claim 7.48).

8 On the Generality of the Normalized Form

Our simulation can automatically transform any lock-free linearizable data structure given in a normalized form into a wait-free one. A natural question that arises is how general the normalized form is. Do all abstract data types (ADT) have a normalized lock-free implementation? We answer this question in the affirmative. However, the value of this general result is theoretical only as we do not obtain *efficient* normalized lock-free implementations. The main interest in the transformation described in this paper is that it attempts to preserve the efficiency of the given lock-free data structure. Thus, it is not very interesting to invoke it on an inefficient lock-free implementation.

We claim that any ADT can be implemented by a normalized lock-free algorithm (given that it can be implemented sequentially). This claim is shown by using (a simplified version of) the universal construction of Herlihy [14], which transforms any sequential data structure to a linearizable lock-free one. Recall that in this universal construction, there is a global pointer to the shared data structure. To execute an operation, a thread reads this pointer, creates a local copy of the data structure, executes the operation on the local copy, and attempts by a CAS to make the global pointer point to its local copy. If the CAS succeeds the operation is completed, and if it fails, the operation is restarted from scratch. We observe that this construction is in effect already in the normalized form, it just needs to be partitioned correctly into the three methods.

Specifically, the CAS-GENERATOR method creates the local copy of the data structure, executes the operation on it, and outputs a list with a single CAS descriptor. The CAS defined in the CAS-descriptor is the attempt to make the global pointer point to the local copy that was prepared in the CAS-generator. The CAS-executer method is the fixed method of the normalized representation, which simply attempts this CAS and (since it is the only one) reports the result. The WRAP-UP method then indicates a restart from scratch if the CAS failed, or returns with the appropriate results if it succeeded.

Of course, this construction is not practical. A lock-free data structure built in this manner is likely to be (very) inefficient. But this construction shows that each ADT can be implemented using the normalized form.

9 Examples: the Transformation of Four Known Algorithms

In this section we will present how we converted four known lock-free data structures into wait-free ones, using the described technique. The four data structures are: Harris's linked-list, Fomitchev & Ruppert's linked-list, a skiplist, and a binary-search-tree. During this section we will also explain how to wisely construct the *parallelizable* GENERATOR and WRAP-UP methods, in a manner which is easy to implement, efficient, and strait-forward.

9.1 Harris's linked-list

Harris designed a practical lock-free linked-list. His list is a sorted list of nodes in which each node holds an integer key, and only one node with a given key may be in the list at any given moment. He employed a special `mark bit` in the `next` pointer of every node, used to mark the node as logically deleted. Thus, a node is deleted by first marking its next pointer using a CAS (in effect, locking this pointer from ever changing again) and then physically removing it from the list by a CAS of its predecessor's `next` field. Inserting a new node can be done using a single CAS, making the new node's designated predecessor point to the new node. In this section we assume familiarity with Harris's linked-list. A reader not familiar with it may skip this section and read on.

We start by noting that Harris's SEARCH method, which is used by both the INSERT and DELETE operations, is a *parallelizable method*. The SEARCH method's input is an integer key, and its output is a pair of adjacent nodes in the list, the first with a key smaller than the input value, and the second with a key greater than or equal to the input value. The SEARCH method might make changes to the list: it might physically remove marked nodes, those nodes that are logically deleted. The search method is restarted in practice anytime an attempted CAS fails. (Such an attempted CAS is always an auxiliary CAS, attempting to physically remove a logically deleted node.) A simple enough *contention failure counter* for this method can be implemented by counting number of failed CASes.

We now specify a normalized version of Harris's linked-list:

- A *contention failure counter* for all of the methods in Harris's linked-list can be implemented by counting the number of failed CASes.

- The (parallelizable) GENERATOR method is implemented as follows: For an insert(key) operation:
  - Call the original SEARCH(KEY) method.
  - If a node is found with the wanted key, return an empty list of CAS-descriptors. (The insert fails.)
  - If a pair (pred, succ) is returned by the search method, create a new node n with the key, set n.next = succ, and return a list with a single CAS descriptor, describing a change of pred.next to point to n.

  The GENERATOR method for a delete(key) operation is:
  - Call the original SEARCH(KEY) method.
  - If no node is found with the given key, return an empty list of CAS-descriptors.
  - If a node n was found appropriate for deletion, return a list with a single CAS-descriptor, describing a change of n.next to set its `mark-bit`.

  The GENERATOR method for a contains(key) operation is:
  - return an empty list of of CAS-descriptors.
- The (parallelizable) WRAP-UP method is implemented as follows: For an insert(key) or a delete(key) operation:
  - If the list of CAS-descriptors is empty, exit with result false (operation failed).
  - If the CAS-descriptor was executed successfully, exit with result true (operation succeeded).
  - If the CAS-descriptor was not successful, indicate that a restart of the operation is required.

  For a contains(key) operation:
  - Call the original contains(key) method (which is already a parallelizable method) and exit with the same result.

We would like to make a remark concerning the contention failure counter. Implementing a counter that simply counts the number of CAS failures is good enough for a linked-list of integers (like the one Harris and others have implemented), but is insufficient for a linked-list of strings, and other data types as well. This is because infinitely many insertions before the key searched for by a CONTAINS method or a SEARCH method, can delay a thread forever without it ever failing a CAS operation. In such cases a more evolved contention failure counter is needed. Its implementation requires holding an approximation counter on the number of keys in the list. Holding the exact count is possible, but inefficient, whereas maintaining an approximation with a bounded error can be achieved with a negligible time overhead and is enough. The more evolved contention failure counter reads the approximation at the beginning of each method and its value is #failed CASes + Max(0, traversed keys - (approximation + max error)). The full details for implementing this contention failure counter along with the needed approximation appear in Appendix B.

9.2 Binary Search Tree

The first practical lock-free binary search tree was presented in [7]. The algorithm implements a leaf-oriented tree, meaning that all the keys are stored in the leaves of the tree, and each internal node points to exactly two children. When a thread attempts to insert or delete a node, it begins its operation by a CAS on an internal node's *state* field. It stores a pointer to an *Info* object, describing the desired change. This (owner) CAS effectively locks this node, but it can be unblocked by any other thread making the desired

(*auxiliary*) CASes. In [7], storing the initial pointer to the *Info* object is also referred to as *Flagging*, and we shall use this notation as well. In a DELETE operation, they also use *Marking*, that permanently locks the internal node that is about to be removed from the tree. Familiarity with [7] is required to fully understand this part. In a nutshell, an INSERT is separated into three CASes:

- I-1. Flagging the internal node that its child sub-tree is needed to be replaced.
- I-2. Replacing the child pointer to point to the new sub-tree
- I-3. Unflagging the parent.

A DELETE operation is separated into four CASes:

- D-1. Flagging the grandfather of the leaf node we wish to delete.
- D-2. Marking the parent of the node we wish to delete(this parent will be removed from the tree as well, but the child is the only leaf node that is to be removed).
- D-3. Changing the grandfather's child pointer to point to a new sub-tree.
- D-4. Unflagging the grandparent.

The neat design of this algorithm makes it very easy to convert it into the normalized structure and thus into a wait-free algorithm, since the methods in it are separated by their functionality. It contains a SEARCH method, designed to find a key or its designated location. This method does not change the data structure, and is thus trivially a *parallelizable method*.
It contains additional parallelizable methods designed to help intended operations already indicated by Info fields: The HELP, HELP-DELETE, HELP-MARKED and HELP-INSERT methods.

In this algorithm, the linearization points of the operations happens **after** the blocking (owner) CASes, inside the *parallelizable methods*, thus the normalized version would have to do some work after the CAS-EXECUTOR method is completed. This is naturally done in the WRAP-UP method.

- A contention failure counter implementation consists of the following.
  - Count the number of times CASes failed.
  - Count the number of times parallelizable methods were called (except the first time for each method).
- The GENERATOR, For an insert(key) operation:
  - Call the original SEARCH(KEY) method.
  - If a node with the requested key is found, return an empty list of CASes.
  - If the parent is *Flagged*: call the (original) HELP method, and afterwards restart the Generator.
  - Return a list with a single CAS-descriptor containing a CAS to change the state of the designated parent to point to an *Info* object describing the insertion (CAS-I-1).
- The WRAP-UP method for an insert(key) operation:
  - If the list of CASes is empty, exit with result false (operation failed).
  - If CAS-I-1 failed, return *restart operation from scratch*.

  - Else, call (the original parallelizable method) HELPINSERT (which will perform CAS-I-2 and CAS-I-3) and exit with true (operation succeeded).

- The GENERATOR method, for a delete(key) operation:
  - Call the original SEARCH(KEY) method.
  - If a node with the requested key was not found, return an empty list of CASes.
  - If the grandparent is *Flagged*: call the (original) HELP method, and afterwards restart the GENERATOR method.
  - If the parent is *Flagged*: call the (original) HELP method, and afterwards restart the GENERATOR method.
  - Return a list with a single CAS-descriptor, containing a CAS to change the state of the grandparent to point to an *Info* object describing the deletion (CAS-D-1).

- The WRAP-UP method, for a delete(key) operation:
  - If the list of CASes is empty, exit with result false (operation failed).
  - If CAS-D-1 failed, return *restart operation from scratch*.
  - Else, call the (original) HELPDELETE method (which potentially executes CAS-D-2, CAS-D-3, and CAS-D-4, but may fail).
    * if HELPDELETE returned true, return *operation succeeded*.
    * else, return *restart operation from scratch*.

- The GENERATOR method, for a contains(key) operation:
  - Return an empty list of CASes.

- The WRAP-UP method, for a contains(key) operation:
  - call the original SEARCH(KEY) method.
  - If a node with the requested key was found, exit with result true.
  - Else, exit with result false.

Note that the binary-search-tree algorithm is designed in a way that during a single operation, each *parallelizable* method can only be called more than once as a result of contention (since other thread had to make a change to the tree that affects the same node). Additionally, the remark about Harris's linked-list (the additional effort needed in some cases in order to implement a contention failure counter) applies here as well.

9.3 Skiplist

Let us refer to the lock-free skiplist that appears on [16]. It is composed of several layers of the lock-free linked-list of Harris. Each node has an array of `next` fields, each point to the next node of a different level in the skiplist. Each `next` field can be *marked*, signifying the node is logically deleted from the corresponding level of the skiplist. The keys logically in the list are defined to be those found on *unmarked* nodes of the lowest list's level. To delete a key, first the FIND(KEY) method is called. If a corresponding node is found, its `next` fields are marked by a CAS from its top level down to level zero. To insert a key, again, the FIND(KEY) method is called first, returning the designated predecessor and successor for each level. The

node is inserted to the lowest (zero) level first, and then to the rest of the levels from bottom up. Familiarity with chapter 14.4 of [16] is required to fully understand the process.

When designing this algorithm, a subtle design decision was made that carries interesting implications for our purposes. As the algorithm appears in [16], the only auxiliary CASes are snipping out marked nodes in the FIND method, similar to Harris's linked-list. Fully linking a node up after it has been inserted to the lowest level is done only by the thread that inserted the node. Thus, in order to achieve lock-freedom, operations by other threads must be allowed to complete while some nodes are incomplete (not fully linked). These operations might include inserting a node immediately after an incomplete node, or even deleting an incomplete node. Allowing such operations to complete causes some difficulties. One result is that when two nodes are being inserted concurrently, and they are intended to be adjacent nodes at some level of the skiplist, it is possible that the node that should come first will bypass the link to its designated successor, skipping over it, and even past other nodes entered concurrently to the same level. This cannot happen at the bottom level, and so it does not hamper the algorithm's correctness, but it can cause higher levels to hold less nodes than they were supposed to, arguably foiling the log(n) complexity of the skiplist.

It is a small and relatively simple change to make the linking up of an inserted node to be done by auxiliary CASes, which are attempted by each thread that traverse that node in the FIND method, instead of doing it by owner CASes only attempted by the thread that inserts the node. If we would make this change, these CASes could be done by other threads in their GENERATOR method. As it is, however, they can only be done in the WRAP-UP method, and only by the owner thread. Since our purpose here is to show how our technique should be used to convert a **given** lock-free algorithm into a wait-free one, and not to suggest variants to the lock-free algorithm, we shall focus on showing how to normalize the algorithm of [16] this way.

- A contention failure counter for each method can be implemented by counting the number of failed CASes.
- The GENERATOR for an insert(key) operation:
  - Call the original FIND(KEY) method.
  - If a node is found with the desired key, return an empty list of CASes.
  - Else, create a new node n with the key, set its `next` field in each level to point to the designated successor, and return a list with a single CAS-desciptor, to change the prev.next at the bottom level to point to n.
- The WRAP-UP method for an insert(key) operation:
  - If the CAS-list is empty, return false (operation failed).
  - If the CAS in the CAS-list failed, return *restart operation from scratch*.
  - Else, follow the original algorithm's linking up scheme. That is, until the new node is fully linked:
    - Call FIND(KEY).
    - Try by a CAS to set the predecessor's next field to point to the newly inserted node for each unlinked level. Use the successor returned from the FIND method as the expected value for the CAS. Restart the loop if the CAS fails.
- The GENERATOR method for a delete(key) operation:
  - Call the original FIND(KEY) method.

  - If no node is found with the given key, return an empty CAS-list.
  - If a node n was found appropriate for deletion, return a list with a CAS-descriptor for each level in which the node is linked, from the highest down to level zero, to mark its `next` field.

- The WRAP-UP method for a delete(key) operation is as follows.
  - If the CAS-list is empty, return false (operation failed).
  - Else, if all CASes were successful, return true (operation succeeded).
  - Else, return *restart operation from scratch*.

- The GENERATOR method for a contains(key) operation:
  - Return an empty list of CASes.

- The WRAP-UP method for a contains(key) operation is as follows.
  - Call the original FIND(KEY) method.
  - If a node with the requested key was found, exit with result true.
  - Else, exit with result false.

The remark about Harris's linked-list (the additional effort needed in some cases in order to implement a contention failure counter) applies here as well.

9.4 The Linked-List of Fomitchev and Ruppert

In the list of Fomitchev and Ruppert, before deleting a node, a backlink is written into it, pointing to its (last) predecessor. This backlink is later used to avoid searching the entire list from the beginning the way Harris did when a node he used was deleted. Fomitchev and Ruppert employ two special bits in each node's `next` field. The `mark` bit, similarly to Harris's algorithm, to mark a node as logically deleted, and the `flag` bit, that is used to signal that a thread wants to delete the node pointed by the flagged pointer. Deletion is done in four phases:

- Flagging the predecessor
- Writing the backlink on the victim node to point to the predecessor
- Marking the victim node
- physically disconnecting the node and unflagging the predecessor (both done in a single CAS).

The main (owner) CAS in this case, which must be done in the CAS-EXECUTER method, is the first (flagging the predecessor). This flagging blocks any further changes to the predecessor until the flag is removed. Removing the flag can be done by any thread in the parallelizable HELPFLAGGED method. The second phase, of writing the backlink, is actually not done by a CAS, but by a direct WRITE. This is safe, since the algorithm is designed in a way that guarantees that for a specific node, there is only a single value that will be written to it (even if many threads will write it). Keeping this non-CAS modification of the data structure will not harm our transformation and it will still provide a correct wait-free algorithm, yet it does not technically match our definition of the normalized representation. To solve this, we can replace this WRITE action with a CAS that uses $NULL$ as the expected-value. This change have no algorithmic applications. The insert operation is done similarly to the insert operation in Harris's linked-list, except that it uses the backlinks to avoid searching the list from the beginning, and that it calls the HELPFLAGGED method to remove the "lock" on a flagged node, if needed.

- A contention failure counter implementation consists of the following.
  - Count the number of times CASes failed.
  - Count the number of times the HELPFLAGGED method is called (except the first time).
- The GENERATOR, for an insert(key) operation:
  - Call the original search(key) method.
  - If a node is found with the wanted key, return an empty list of CAS-descriptors.
  - Else, if a window(pred, succ) is returned, and pred is flagged, call the (original) HELPFLAGGED method.
  - If a window (pred, succ) that is fit for inserting the key is found, create a new node n with the key, set n.next = succ, and return a list with a single CAS-descriptor, describing a change of pred.next to point to n.
- The WRAP-UP method for an insert(key) operation:
  - If the list of CAS-descriptors is empty, exit with result false (operation failed).
  - If the CAS-descriptor was executed successfully, exit with result true (operation succeeded).
  - If the CAS-descriptor was not successful, indicate *restart operation from scratch*.
- The GENERATOR, for a delete(key) operation:
  - Call the original search(key) method.
  - If no node is found with the given key, return an empty list of CAS-descriptors.
  - If a victim node and its predecessor were found, return a list with a single CAS-descriptor, describing a change of the predecessor.next so that its `flag-bit` will be set.
- The WRAP-UP method for a delete(key) operation:
  - If the list of CAS-descriptors is empty, exit with result false (operation failed).
  - If the CAS-descriptor was executed successfully, call the (original) HELPFLAGGED method, and afterwards exit with result true (operation succeeded).
  - If the CAS-descriptor was not successful, indicate *restart operation from scratch*.
- The GENERATOR method for a contains(key) operation:
  - Return an empty list of CASes.
- The WRAP-UP method for a contains(key) operation:
  - Call the original SEARCH(KEY) method.
  - If a node with the requested key was found, exit with result true.
  - Else, exit with result false.

As with all the examples, the remark appearing after Harris's linked list applies here as well. In the following section, we describe an important optimization that is especially important in the case of the transformation of the list of Fomitchev & Ruppert; the normalized representation of the algorithm does not fully utilize the strength of the backlinks, which is a key feature of this algorithm when comparing it to Harris's. Using the optimization in 10.1 guarantees that most operations will still fully utilize the backlinks, while the few operations that will complete in the slow path may extract only part of its benefits.

10 Optimizations

10.1 Using the Original Algorithm for the Fast Path

In order to use our simulation technique and obtain a wait-free practical algorithm, the first thing we need to do is to express the lock-free data structure in the normalized form. As mentioned above, in our work we expressed four data structures this way. Our intuition is that the data structure in the normalized form is in some way "the same" as the original algorithm, only expressed differently. In what follows, we provide some formalization for this intuition and then use it for an optimization.

Definition 10.1 (Interoperable Data Structures.) *We say that two lock-free data structure algorithms are* interoperable *if they can be run on the same memory concurrently and maintain linearizability and correctness.*

The above definition means that for each data-structure operation that we would like to perform, we can arbitrarily choose which of the two algorithms to use for running it, and the entire execution remains linearizable for the same ADT. All of the four normalized algorithms we created are *interoperable* with their original versions[7]. We would like to exploit this fact in order to use the original lock-free algorithm, and not the normalized version of it, as the fast-path for the simulation. The slow path, in which help is given, still works in the normalized manner. This optimization is possible, but requires some care. To safely allow the original algorithm to work with the help mechanism of the normalized algorithm, we require that a slightly stronger parallelism property will be kept by the parallelizable methods. Recall that a *parallelizable method* is a one whose executions are avoidable. In what follows we strengthen the definition of avoidable method execution.

Definition 10.2 Strongly avoidable method execution: *A run of a method M by a thread T on input I in an execution E of a program P is strongly avoidable if there exists an equivalent execution E' for E such that in both E and E' each thread follows the same program, both E and E' are identical until right before the invocation of M by T on input I, in E' each* CAS *that T executes in M either fails or is futile, and (the new requirement): In E and E' the shared memory reaches the same states in the same order.*

A state of the shared memory is simply the contents of all memory. Failed CASes, futile CASes, and READ primitives, do not alter the state of the shared memory. The new requirement does not mean that after n computation steps the state of the shared memory is the same in E and in E', since each one of them can have a different set of computation steps that do not alter the memory. The meaning of the extra requirement is that the alternative execution E' is not only equivalent to E, but is also indistinguishable from it, in the sense that an observer who examines the shared memory cannot tell whether E or E' has taken place.

This stronger definition is not needed for our technique to work, only to ensure a safe use of this specific optimization. All of the four algorithms we expressed in the normalized form naturally fulfill this stronger requirement. Thus, since the original algorithm can work interoparably with the normalized one, it can also work interoparably with the normalized one in the presence of "extra" avoidable executions of parallelizable methods, and we can safely use it as the fast-path, given that we adjust it to have contention failure counters for its methods.

10.2 Avoiding versions

As explained in Section 6.2, while executing the CASes, a helping thread may create an ABA problem if it is delayed and then returns to execute when the CAS it is attempting to simulate has already been

[7]Excluding the fact that version numbers must be added to the original algorithms as well.

completed and the algorithm has moved on. To ensure that this helping thread does not foil the execution, we introduced versioning to make sure its CAS fails and it can continue executing properly. For some data structures, ABA problems of this type cannot occur because the original data structure is designed to avoid them. For example, the tree algorithm of Ellen et al. [7] allows helping threads to operate within the original lock-free algorithm and it supports such help with a special mechanism that eliminates such ABA problems. Therefore, for the tree there is no need to add the versioning mechanism to each CAS, and indeed we did not use versioning when making the tree wait-free. This does not eliminate the need to use the `modified-bit` for a structured execution of the public CASes.

11 Performance

11.1 Memory Management

In this work we do not specifically address the standard problem of memory management for lock-free (and wait-free) algorithms. In the Java implementation we just use Java's garbage collector, which is probably not wait-free. If the original lock-free algorithm has a solution for memory management, then the obtained simulation works well with it, except that we need to reclaim objects used by the generated algorithm: the operation records and the operation record boxes. This can be done using hazard pointers [20]. The implementation is tedious, but does not introduce a significant difficulty and we do not deal with it in the current submission.

11.2 Our Wait-Free Versions vs. the Original Lock-Free Structures

We chose four well-known lock-free algorithms, and used the transformation described in this paper to derive a wait-free algorithm for each. We implemented these algorithms and, when possible, used the optimizations described in Section 10. The performance of each wait-free algorithm was compared against the original lock-free algorithm. We stress that we compared against the original lock-free version of the algorithm without adding versioning to the CAS operations and without modifying it to fit a normalized representation.

The four lock-free algorithms we chose were Harris's linked-list [13], the binary-search-tree of Ellen et al. [7], the skiplist of Herlihy and Shavit [16], and the linked-list of Fomitchev and Ruppert [11]. All implementations were coded in Java. The Java implementations for the lock-free algorithms of Harris's linked-list and the skiplist were taken from [16]. We implemented the binary search tree and the list of Fomitchev and Ruppert ourselves, in the most straightforward manner, following the papers.

All the tests were run on SUN's Java SE Runtime, version 1.6.0. We ran the measurements on 2 systems. The first is an IBM x3400 system featuring 2 Intel(R) Xeon(R) E5310 1.60GHz quad core processors (overall 8 cores) with a memory of 16GB and an L2 cache of 4MB per processor. The second system features 4 AMD Opteron(TM) 6272 2.1GHz processors, each with 8 cores (overall 32 cores), each running 2 hyper-threads (overall 64 concurrent threads), with a memory of 128GB and an L2 cache of 2MB per processor.

We used a micro-benchmark in which 50% of the operations are *contains*, 25% are *insert*, and 25% are *delete*. Each test was run with the number of threads ranging from 1 to 16 in the IBM, and 1 to 32 in the AMD. In one set of tests the keys were randomly and uniformly chosen in the range $[1, 1024]$, and in a different set of tests the keys were chosen in the range $[1, 64]$. In each test, each thread executed 100,000 operations overall. We repeated each test 15 times, and performance averages are reported in the figures. The maximum standard deviation is less than 5%. The contention threshold was set to $k = 2$. In practice, this means that if one of the three simulation stages encounters k failed CASes, it gives up the fast path and moves to the slow path.

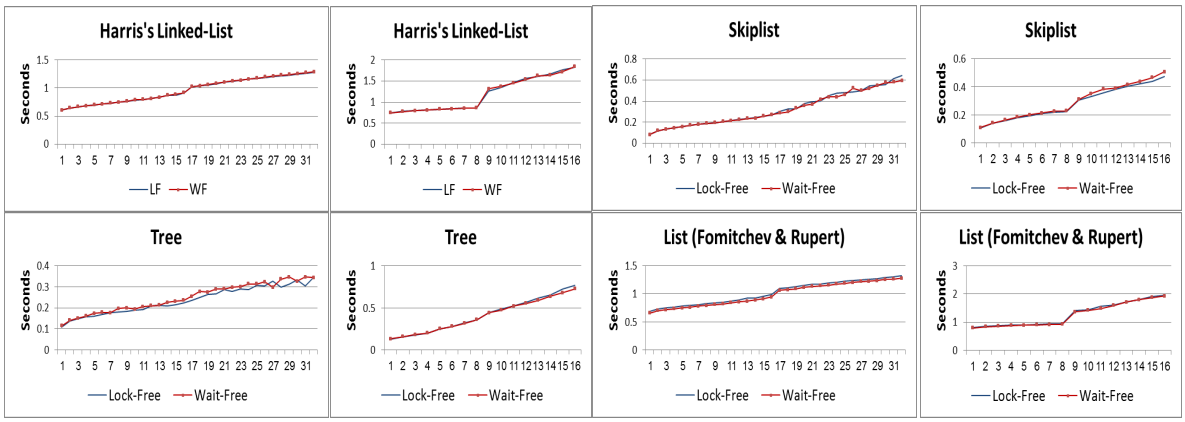

Figure 7: Lock-Free versus Wait-Free algorithms, 1024 keys. Left: AMD. Right: IBM

Figure 7 compares the four algorithms when running on the AMD (the left graph of each couple) and on the IBM (right) for 1024 possible keys. The figure show the execution times (seconds) as a function of the number of threads.

For 1024 keys, the performance of the wait-free algorithms is comparable to the lock-free algorithms, the difference being 2% on average. The close similarity of the performance between the original lock-free algorithms and the wait-free versions produced using our simulation suggests that the slow-path is rarely invoked.

Figure 8 indicates how many times the slow path was actually invoked in each of the wait-free data structures as a function of the number of threads. Keep in mind that the overall number of operations in each run is 100,000 multiplied by the number of threads. The results reported are again the averages of the 15 runs (rounded to whole numbers). As expected, the fraction of operations that require the slow path is very small (maximum fraction of about 1/3,000 of the operations). The vast majority of the operations complete in the fast-path, allowing the algorithm to retain performance similar to the lock-free algorithm. Yet, a minority of the operations require the help mechanism to guarantee completion in a bounded number of steps, thus achieving wait-freedom.

The results for 64 keys are depicted in figures 9 and 10. The behavior for 64 keys is different than for 1024 keys. The smaller range causes a lot more contention, which in turn causes a lot more operations to ask for help and move to the slow-path. Asking for help in the slow path too frequently can dramatically harm the performance. This is most vividly displayed in the tree data structure on the AMD. When running 32 parallel threads, about 1 in 64 operations asks for help and completes in the slow-path. This means that roughly during half of the execution time there is an operation running in the slow-path. As a result, all threads help this operation, sacrificing scalability for this time. Thus, it is not surprising that the performance are down by about 50%.

In such circumstances, it is advisable to set the contention threshold to a higher level. Setting it to 3 (instead of 2) causes a significant improvement in the performance. This comes with the cost of allowing some operations to take longer, as some operations will first fail 3 times, and only then ask for help.

11.3 Our Wait-Free Transformation vs. a Universal Construction

Universal constructions achieve a difficult task, as they go all the way from a sequential data structure to a concurrent wait-free implementation of it. It may therefore be difficult to also make the resulting wait-free algorithm efficient enough to become practicable. Our technique builds on a tailored made lock-free data structure and achieve the smaller step from lock-freedom to wait-freedom. This may be the reason why we

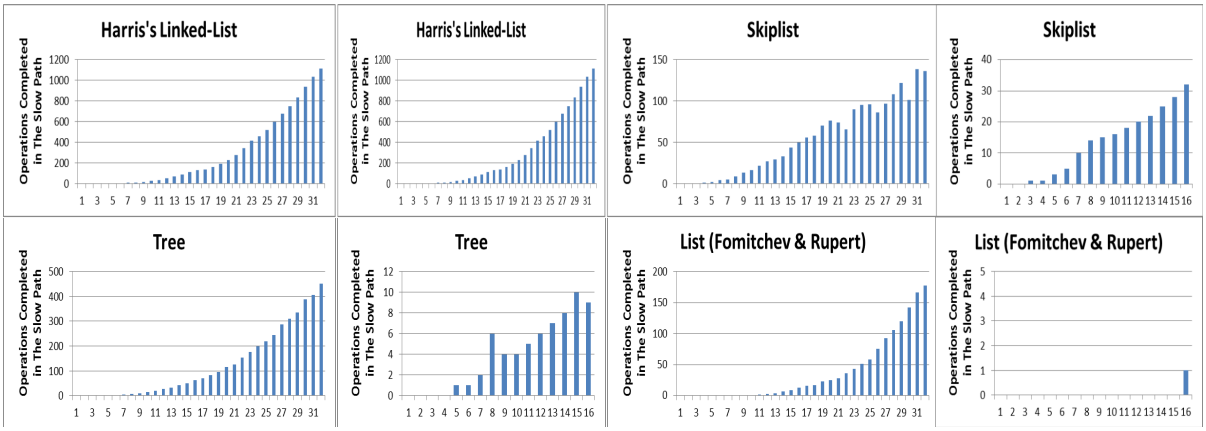

Figure 8: Number of Operation Completed in the Slow Path., 1024 keys. Left: AMD. Right: IBM

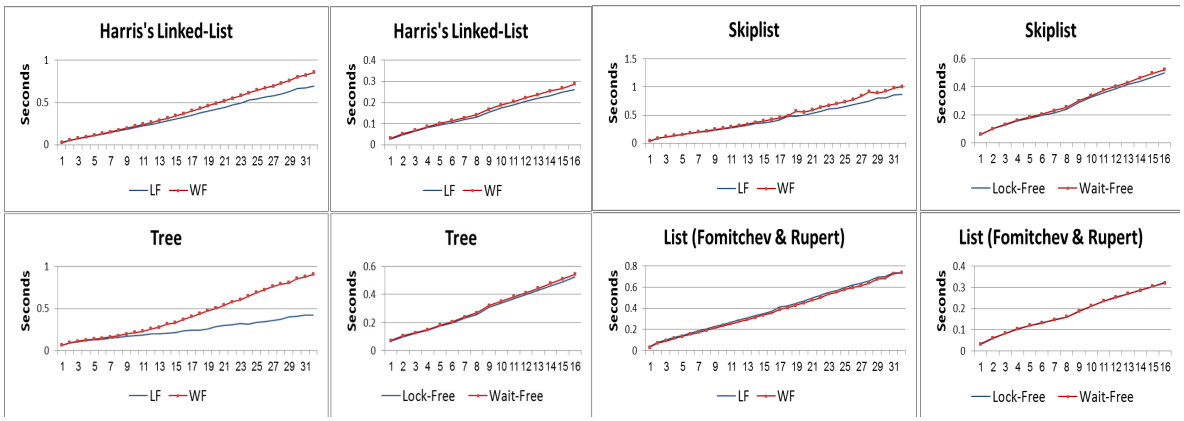

Figure 9: Lock-Free versus Wait-Free algorithms, 64 keys. Left: AMD. Right: IBM

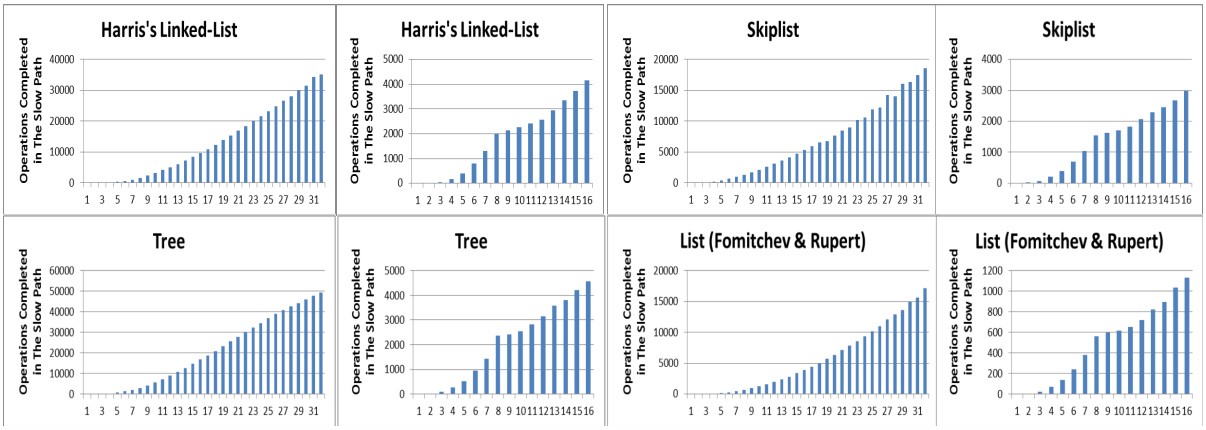

Figure 10: Number of Operation Completed in the Slow Path, 64 keys. Left: AMD. Right: IBM

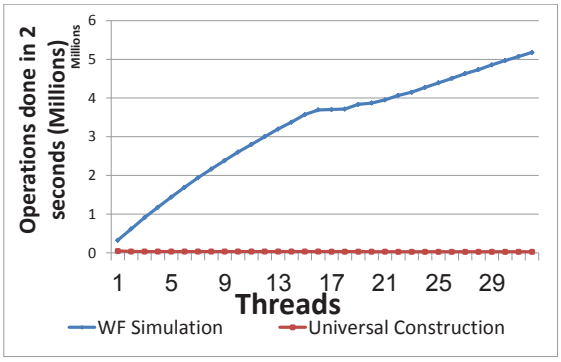

Figure 11: Our Wait-Free List against a Universal Construction List

are able to retain practicable performance.

To demonstrate the performance difference, we implemented the state of the art universal construction of Chuong, Ellen, and Ramachandran [5] for a standard sequential algorithm of a linked-list. The obtained wait-free linked-list was compared against the wait-free linked-list generated by applying our technique to Harris's lock-free linked-list. [8]

We ran the two implementations on our AMD Opetron system featureing 4 AMD Opteron(TM) 6272 2.1GHz processors, each with 8 cores (overall 32 cores), each running 2 hyper-threads (overall 64 concurrent threads), with a memory of 128GB and an L2 cache of 2MB per processor. In the micro-benchmark tested, each thread executed 50% contains, 25% insert, and 25% delete operations. The keys were randomly and uniformly chosen from the range $[1, 1024]$. The number of threads was ranging from 1 to 32. In each measurement, all the participating threads were run concurrently for 2 seconds, and we measured the overall number of operations executed. Each test was run 10 times, and the average scores are reported in the figures.

In Figure 11 the total number of operations (in millions) done by all the threads is reported as a function of the number of the threads. It can be seen that the wait-free list obtained in this paper (and so also the lock-free linked-list) drastically outperforms the universal construction for any number of threads. Also, while our list scales well all the way up to 32 threads, the list of the universal construction does not scale at all. Figure 12 is based on the same data, but demonstrates the ratio between our construction of the wait-free linked-list and the universal construction of wait-free linked list. For a single thread, our list is 6.8 times faster and this ratio grows with any additional thread, up to a factor of 198 times faster than the universal construction for 32 threads.

References

[1] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast (extended abstract). In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995, Las Vegas, Nevada, USA*, pages 538–547, 1995.

[8] Note that implementing the universal construction of [5] on Harris's lock-free linked-list, instead of using the universal construction on a standard sequential list, is possible, but ill-advised. Although both implementations would result in a wait-free list, the one based on a lock-free algorithm would undoubtedly be slower. The universal construction already handles the inter-thread race conditions, and implementing it on Harris's linked-list would force it to also use the (unneeded) synchronization mechanisms of Harris.

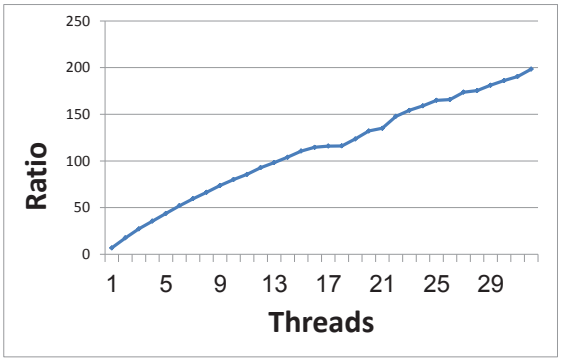

Figure 12: Ratio between Our List and a Universal Construction List

[2] James H. Anderson and Yong-Jik Kim. Fast and scalable mutual exclusion. In *Distributed Computing, 13th International Symposium, Bratislava, Slavak Republic, September 27-29, 1999, Proceedings*, pages 180–194, 1999.

[3] James H. Anderson and Yong-Jik Kim. Adaptive mutual exclusion with local spinning. In *Distributed Computing, 14th International Conference, DISC 2000, Toledo, Spain, October 4-6, 2000, Proceedings*, pages 29–43, 2000.

[4] James H. Anderson and Mark Moir. Universal constructions for large objects. *IEEE Trans. Parallel Distrib. Syst.*, 10(12):1317–1332, 1999.

[5] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A universal construction for wait-free transaction friendly data structures. In *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*, pages 335–344, 2010.

[6] Tyler Crain, Damien Imbs, and Michel Raynal. Towards a universal construction for transaction-based multiprocess programs. In *Distributed Computing and Networking - 13th International Conference, ICDCN 2012, Hong Kong, China, January 3-6, 2012. Proceedings*, pages 61–75, 2012.

[7] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 131–140, 2010.

[8] Panagiota Fatourou and Nikolaos D. Kallimanis. The redblue adaptive universal constructions. In *Distributed Computing, 23rd International Symposium, DISC 2009, Elche, Spain, September 23-25, 2009. Proceedings*, pages 127–141, 2009.

[9] Panagiota Fatourou and Nikolaos D. Kallimanis. A highly-efficient wait-free universal construction. In *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, pages 325–334, 2011.

[10] Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-free algorithms can be practically wait-free. In *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, pages 78–92, 2005.

[11] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, PODC '04, pages 50–59, New York, NY, USA, 2004. ACM.

[12] Michael Greenwald. Two-handed emulation: how to build non-blocking implementation of complex data-structures using DCAS. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC 2002, Monterey, California, USA, July 21-24, 2002*, pages 260–269, 2002.

[13] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, London, UK, UK, 2001. Springer-Verlag.

[14] Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Princiles & Practice of Parallel Programming (PPOPP), Seattle, Washington, USA, March 14-16, 1990*, pages 197–206, 1990.

[15] Maurice Herlihy. A methodology for implementing highly concurrent objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993.

[16] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[17] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 223–234, 2011.

[18] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *PPOPP*, pages 141–150, 2012.

[19] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.

[20] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[21] Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Program.*, 25(1):1–39, 1995.

[22] Aravind Natarajan, Lee Savoie, and Neeraj Mittal. Concurrent wait-free red black trees. In *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings*, pages 45–60, 2013.

[23] Gadi Taubenfeld. Contention-sensitive data structures and algorithms. In *Distributed Computing, 23rd International Symposium, DISC 2009, Elche, Spain, September 23-25, 2009. Proceedings*, pages 157–171, 2009.

[24] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *Principles of Distributed Systems, 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings*, pages 330–344, 2012.

A The Wait-Free Queue

In our algorithm, we rely on a wait-free queue supporting the operations *enqueue, peek* and *conditionally-remove-head*, rather then *enqueue* and *dequeue* as given in [17]. Adjusting the queue from [17] to our needs was a very easy task. The java implementation of the adjusted queue that we used is provided here.

```
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.atomic.AtomicReference;
import java.util.concurrent.atomic.AtomicReferenceArray;


public class WFQueueAd<V> {

        class Node {
                public V value;
                public AtomicReference<Node> next;
                public int enqTid;
                public AtomicInteger deqTid;
                public Node (V val, int etid) {
                        value = val;
                        next = new AtomicReference<Node>(null);
                        enqTid = etid;
                        deqTid = new AtomicInteger(-1);
                }
        }

        protected class OpDesc {
                public long phase;
                public boolean pending;
                public boolean enqueue;
                public Node node;
                public OpDesc (long ph, boolean pend, boolean enq, Node n) {
                        phase = ph;
                        pending = pend;
                        enqueue = enq;
                        node = n;
                }
        }

        protected AtomicReference<Node> head, tail;
        protected AtomicReferenceArray<OpDesc> state;

        public AtomicInteger enqed = new AtomicInteger(0);
        public AtomicInteger deqed = new AtomicInteger(0);

        public WFQueueAd () {
                Node sentinel = new Node(null, -1);
                head = new AtomicReference<Node>(sentinel);
```

```
        tail = new AtomicReference<Node>(sentinel);

        state = new AtomicReferenceArray<OpDesc>(Test.numThreads);

        for (int i = 0; i < state.length(); i++) {
                state.set(i, new OpDesc(-1, false, true, null));
        }
}

public void enq(int tid, V value) {
        long phase = maxPhase() + 1;
        state.set(tid,
     new OpDesc(phase, true, true, new Node(value, tid)));
        help(phase);
        help_finish_enq();
}

public V peekHead() {
        Node next = head.get().next.get();
        if (next == null)
                return null;
        return next.value;
}

public boolean conditionallyRemoveHead(V expectedValue) {
        Node currHead = head.get();
        Node next = currHead.next.get();
        if (next == null || !next.value.equals(expectedValue))
                return false;
        if (head.compareAndSet(currHead, next)) {
                help_finish_enq();
                currHead.next.set(null);
                return true;
        }
        else
                return false;
}

protected void help(long phase) {
        for (int i = 0; i < state.length(); i++) {
                OpDesc desc = state.get(i);
                if (desc.pending && desc.phase <= phase) {
                        if (desc.enqueue) {
                                help_enq(i, phase);
                        }
                }
        }
}
```

```
protected void help_enq(int tid, long phase) {
        while (isStillPending(tid, phase)) {
                Node last = tail.get();
                Node next = last.next.get();
                if (last == tail.get()) {
                        if (next == null) {
                                if (isStillPending(tid, phase)) {
                                        if (last.next.compareAndSet
                                    (next, state.get(tid).node)) {
                                                help_finish_enq();
                                                return;
                                        }
                                }
                        } else {
                                help_finish_enq();
                        }
                }
        }
}

protected void help_finish_enq() {
        Node last = tail.get();
        Node next = last.next.get();
        if (next != null) {
                int tid = next.enqTid;
                OpDesc curDesc = state.get(tid);
                if (last == tail.get() && state.get(tid).node == next) {
                        OpDesc newDesc = new OpDesc
                    (state.get(tid).phase, false, true, next);
                        state.compareAndSet(tid, curDesc, newDesc);
                        tail.compareAndSet(last, next);
                }
        }
}

protected long maxPhase() {
        long maxPhase = -1;
        for (int i = 0; i < state.length(); i++) {
                long phase = state.get(i).phase;
                if (phase > maxPhase) {
                        maxPhase = phase;
                }
        }
        return maxPhase;
}

protected boolean isStillPending(int tid, long ph) {
```

```
            return state.get(tid).pending &&
            state.get(tid).phase <= ph;
        }
}
```

B Implementing a Contention Failure Counter in the Presence of Infinite Insertions

A somewhat hidden assumption in the fast-path-slow-path technique (and consequently, in our technique as well), is the ability to be able to identify effectively when a thread fails to complete an operation due to contention. Failing to recognize contention will foil wait-freedom, as the relevant thread will not ask for help. Counting the number of failed CASes is generally a very effective way of identifying contention. However, it is not always enough. For example, in the binary search tree, a thread may never fail a CAS, and yet be held forever executing auxiliary CASes for other threads' operations. Identifying such a case is generally easy. For the binary tree algorithm, we did so by counting invocations of the parallelizable help methods.

However, there is one problem that often presents a greater difficulty. We refer to this problem as *the infinite insertions problem*. This is a special case in which a thread in a lock-free algorithm may never complete an operation and yet never face contention.

Consider what happens when a data structure keeps growing while a thread is trying to traverse it. For example, consider what happens in a linked-list, if while a thread tries to traverse it to reach a certain key, other threads keep inserting infinitely many new nodes before the wanted key. The thread might never reach the needed key. The complexity of searching the key in this case is linear at the size of the list, but this size keeps growing. If the list size is some how limited (for example, if all the keys in the list must be integers), then this cannot go on forever, and eventually the traversing thread must reach the key it seeks (or discover it is not there). Such a bound on the size of the data structure can be used to assert for the wait-freedom of some of the algorithms we have discussed in this paper, but it provides a rather poor bound for the wait-freedom property, and it cannot at all be used at some cases. (Such as in a list that employs strings, instead of integers, as keys.)

To implement a contention failure counter that is robust to this problem, we offer the following mechanism to enable a thread to identify if the data structure is getting larger while it is working on it. The idea is that each thread will read a field stating the size of the data structure prior to traversing. For example, in a list, a skiplist or a tree, it can read the number of nodes of the data structure. During the operation, it will count how many nodes it traverses, and if the number of traversed nodes is higher than the original total number of nodes (plus some constant), it will abort the fast-path and will ask for help.

However, a naive implementation of this basic idea performs poorly in practice, since maintaining the exact number of nodes in a wait-free manner can be very costly. Instead, we settle for maintaining a field that approximates the number of keys. The error of the approximation is bounded by a linear function of the number of threads operating on the data structure. Thus, before a thread starts traversing the data structure, it should read the approximation, denoted SIZE-APP, and if it traverses a number of nodes that is greater than SIZE-APP + MAX-ERROR + CONST, switch to the slow path and ask for help.

To maintain the approximation for the number of nodes, the data structure contains a global field with the approximation, and each thread holds a private counter. In its private counter, each thread holds the number of nodes it inserted to the data structure minus the number of nodes it deleted from it since the last time the thread updated the global approximation field. To avoid too much contention in updating the global field, each thread only attempts to update it (by a CAS) once it reaches a certain *soft_threshold* (in

absolute value). If the CAS failed, the thread continues the operation as usual, and will attempt to update the global approximation field at its next insert or delete operation. If the private counter of a thread reaches a certain *hard_threshold*, it asks for help in updating the global counter. This is done similarly to asking help for other operations: it should enqueue a request into the help-queue. The input for the operation of UPDATEGLOBALCOUNTER is an integer stating the required adjustment. The *Generator* method here is reading the global counter, and then output a single CAS description, describing a CAS that alters the old counter value with the wanted new one. The WRAP-UP METHOD exits the operation if the CAS succeeded, or indicates that the operation should be restarted if the CAS failed[9]. Such an approximation of the size of the data structure can be maintained very cheaply, and is enough to solve the problem of the infinite insertions.

[9] In essence, we have just described the normalized lock-free algorithm for a shared counter.