A Practical Wait-Free Simulation for Lock-Free Data Structures [*]

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

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

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. Wait-free implementations have been notoriously hard to design and often inefficient. In this work we present a transformation of lock-free algorithms 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, while still guaranteeing non-starvation.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords Wait-Freedom; Lock-Freedom

1. Introduction

Concurrent data structures are designed to utilize all available cores in order 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.

[*] This work was supported by the Israeli Science Foundation grant No. 283/10.

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.
Copyright © 2014 ACM 978-1-4503-2656-8/14/02…$15.00.
http://dx.doi.org/10.1145/2555243.2555261

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 finite number of steps, regardless of other threads' behavior. This worst-case guarantee has its theoretical appeal and elegance, but is also important in practice for making concurrent data structures useable with real-time systems. For real-time, it is not enough that progress is made in almost all executions. Progress (i.e., meeting the deadline) must be proven to happen in *all* executions. 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 finite 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 [17], the linked-list [14], the skiplist [17], 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 [15] using universal simulations. Universal simulation techniques have evolved dramatically since then (e.g., [1, 4–6, 8, 12, 16]), but even the state-of-the-art universal construction is too slow compared to the lock-free or lock-based implementations and cannot be used in practice.

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 [18, 19]. Next, an independent construction of a wait-free stack and queue appeared in [9]. And finally, a wait-free algorithm for the linked-list has been shown in [24].[1]

[1] In fact, [9] also presented an entirely new universal construction. However, in the general case (i.e., except for the stack and the queue) this construction required each thread to create a local copy of the entire data structure for every operation, which makes it impractical.

To obtain a fast wait-free queue and linked-list, the fast-path-slow-path methodology has been adopted for use with wait-freedom in [19, 24]. The fast-path-slow-path methodology is ubiquitous in systems in general and in parallel computing particularly (e.g., [2, 3, 20, 22]). It is typically used to partitions slow handling of difficult cases from fast handling of the more typical cases.

The way the fast-path-slow-path methodology was used in [19] was to let an operation start executing using a fast lock-free algorithm, and only move 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 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. The fast-path-slow-path methodology has been shown to make the wait-free queue [18] and linked-list of [24] almost as efficient as their lock-free counterpart.

The process of designing a fast wait-free algorithm for a new data structure is complex, difficult, and error-prone. One approach to designing new wait-free algorithms, which is also the one used in [18, 19, 24], is to start with a lock-free algorithm for the data structure, work (possibly hard) to construct a correct wait-free algorithm by adding a helping mechanism to the original algorithm, 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 fast-path-slow-path algorithm is nontrivial. One must design the lock- and wait-free algorithms to work in sync to obtain the overall combined algorithm with the required properties.

In this work we ask whether this entire design can be done automatically, and so also by non-experts. Given a lock-free data structure, can we apply a generic method to add an adequate helping mechanism and obtain a wait-free version for it, and then automatically combine the original lock-free version with the obtained wait-free version to create a fast wait-free algorithm for the same 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 transforms it to a practical wait-free data structure, that is almost as efficient as the original lock-free algorithm.

We next claim that the normalized representation we propose is meaningful in the sense that important known lock-free algorithms for data structures can be easily specified in this form. In fact, all linearizable lock-free data structures that we are aware of can be specified in a normalized form. We demonstrate the generality of the proposed normalized form by specifying several important lock-free data structures in their normalized form. We then obtain wait-free versions of them using the proposed transformation. In particular, we transform two linked-lists [11, 14], a skiplist [17], and a binary search tree [7] to create practical wait-free designs for them all. To the best of our knowledge, practical wait-free versions of the skiplist and the tree are not known in the literature and are presented here for the first time.

Next, in order to verify that the resulting algorithms are indeed efficient, 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 algorithm from which they were derived. Given these results, it seems like wait-freedom can be adopted for general use, and not only for real-time systems. Similar to car airbags, they are seldom deployed but good to have in case of need.

The contributions of this work are the following:

1. A 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 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. The first design of a practical wait-free skiplist.
4. The first design of a practical wait-free tree.
5. An implementation and reported measurements validating the efficiency of the proposed scheme.

The idea of transforming an algorithm to provide a practical algorithm with a different progress guarantee is not new. Taubenfeld [23], Ellen et al. [10] and Guerraoui et al. [13] are examples of such transformations. But none of them allows obtaining wait-free data structures in the standard asynchronous model.

The paper is organized as follows. In Section 2 we provide an overview of the proposed transfromation. In Section 3 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 4, and the details of simulating normalized lock-free algorithms in a wait-free manner appear in Section 5. In Section 6 we give an example of how to apply our technique using Harris's linked-list. We give highlights of the correctness proof in Section 7, and our measurements are reported in Section 8.

2. Tranformation 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. Asking for help is done by enqueuing a succinct description of its current computation state on a wait-free queue (we use the queue of [18]). 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 operation. A helping thread runs the operation it attempts to help locally and 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 5.

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 algorithm, as defined in Section 4. 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 denote *auxiliary CASes*, we do not need to use synchronization at all. As explained in Section 3, the nature of lock-free algorithms makes the use of auxiliary CASes common. Most of Section 3.2 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 4.

3. 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 4 to formally specify the normalized form of lock-free algorithms.

3.1 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 re-

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

move 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 [14]. 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 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*.

3.2 Notations and Definitions

In this section we formally define the essential concepts used in this work.

DEFINITION 3.1. (Futile CAS.) *A futile CAS is a CAS in which the expected value and the new value are identical.*

DEFINITION 3.2. (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.*

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 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. To formalize parallelizable methods we first define a harmless, or *avoidable* parallel run of a method. Loosely speaking, a run of a method is avoidable, if each CAS executed in it is avoidable. By avoidable, we mean that either: 1) the CAS fails, or 2) the CAS is futile, or 3) there exists an equivalent execution in which the CAS fails. Normally, in the equivalent execution, the CAS fails because a different thread executes the same CAS (i.e., same address, same expected-value, and same new-value), but this is not obligatory by the definition.

DEFINITION 3.3. (Avoidable method execution) *A run of a method M by a thread T on input I in an execution E is avoidable if each CAS that T attempts during the execution of M is avoidable in the following sense. Let S_1 denote the state of the computation right before the CAS is attempted by T. Then there exists an* equivalent execution *E' for E such that both executions are identical until reaching S_1, and in E' the CAS that T executes in its next step (after S_1) is either futile or unsuccessful. Also, in E' the first execution step from S_1 is executed by a thread who is the owner of an ongoing operation*[3].

We now move to defining parallelizable methods that can be executed on the data structure without "harming" its consistency. To this end, we conduct a mental experiment in which we consider additional parallel threads that execute methods concurrently with the run of the algorithm. This mental experiment creates executions that are not "legal". We will not have real executions in which a thread will just run a single method of an operation out of the blue with parameters supplied by some oracle. However, the resulting execution is well defined and we will use such "illegal" executions as hybrids for an argument that one real execution (the wait-free run) is equivalent to another real execution (an equivalent lock-free run).

DEFINITION 3.4. (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 hold. First, the execution of a parallelizable method depends only on its input, the shared data structure, and the results of the method's CAS operations. In particular, the execution does not depend on the executing thread's local state prior to the invocation of the parallelizable method. Second, at the point where M is invoked, if we create and run a finite number of parallel threads, each one executing M on the same input I concurrently with the execution of T, then in any possible resulting execution, 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. This is because every CAS it attempts 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 consider one such

[3] This implies that this owner thread is in the middle of executing some operation when arriving at S_1.

logically deleted node. Consider a CAS in which T attempts to physically remove this node from the list. We denote the state right before this attempt S_1. Now consider the thread T_1, which 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 (at S_1) to physically remove the node, and only then does T attempt the considered CAS and fails, is equivalent.

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. By the definition, parallelizable methods may be run out of the blue by threads that do not execute actual operations on the data structure.

In the proof, we will claim the equivalence of the wait-free execution and the lock-free one via several equivalent executions, some of them being "illegal" executions that run parallelizable methods "out of the blue" by additional threads that only execute these methods and cease to exist. Such "illegal" executions will only be used to incrementally argue that two legal executions of the real protocols are equivalent.

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 3.5. (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 $C(t) \leq d(t)$.* [4]
- *(Incremented periodically:) The method M does not run infinitely many steps without incrementing the contention failure counter.*

[4] 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 $C(t) = 0$.

A lock-free method must complete within a finite 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 to determine that a method in the fast-path is not making progress and so its executer 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.

4. Normalized Lock-Free Algorithms

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

4.1 The Normalized Representation

A normalized lock-free algorithm 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 algorithm, 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 operations 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 5.

DEFINITION 4.1. *A lock-free algorithm is provided in a* normalized representation *if:*

- *Any modification of the shared data structure is executed using a CAS operation.*
- *Every* operation *of the algorithm 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[5]

[5] A CAS descriptor is a triplet: $(address, expectedvalue, newvalue)$

2) CAS Executor, *which is a fixed method common to all data structures and all algorithms. Its input is the list of* CAS descriptors *output by the CAS generator method. The CAS executor method attempts to execute the CASes in its input one by one until the first one fails, or until all CASes complete. Its output contains the list of* CAS Descriptors *from its input and the index of the CAS that failed (which is zero if none failed).*
3) Wrap-Up, *whose input is the output of the* CAS Execution method *plus the operation's input. Its output is either the operation 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 CASes 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).*

All lock-free algorithms for data structures that we are aware of today can be easily converted into this form. As an example, a normalized representation of Harris's linked-list is given in Section 6. We have also devised normalized representations for the binary-search-tree of Ellen et al. [7], the skiplist of Herlihy and Shavit [17], and the linked-list of Fomitchev and Ruppert [11]. This is probably the best indication that this normalized representation covers natural lock-free algorithms. We remark that all abstract data types can be implemented in a normalized lock-free algorithm, using a simplified version of the universal construction of Herlihy [15], but this 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* in the lock-free algorithm) 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 executor 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 EXECUTOR 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 EXECUTOR 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.

5. Transformation Details

In this section, we provide the efficient wait-free simulation of any normalized lock-free algorithm. We start with an overview.

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. If the operation completes, then we are done. Otherwise, the contention failure counter exceeded its threshold and the slow path must be taken. 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.

To provide help, a thread examines the first operation record enqueued on the *help queue*. The operation record describes which operation should now be executed, and also which of its three methods needs to be run. Running the CAS GENERATOR method or the WRAP-UP method is easier, as they are parallelizable methods and they can be run by several threads concurrently at no risk. To execute one of these two, the helping thread executes their code using the input in the operation record. Upon completion, the helping thread creates a new updated operation record that includes the output of the method. It then tries (using an atomic CAS) to let the new operation record replace the original one, and become visible to all threads. If this CAS fails, then another thread has already completed the execution of this method and reported a (perhaps different) output, which has been publicly set as this method's output. The helping thread proceeds to help using the new publicly visible operation record, whether this operation record has been created by itself or by a different helping thread. It remains to describe helping the CAS EXECUTOR method.

The CAS EXECUTOR method is not parallelizable and therefore helping threads cannot simply run it concurrently. To support a concurrent execution of it, fields that can potentially be modified by the CAS EXECUTOR are paired with a versioning field and a `modified bit`. These additional fields allow executing each CAS exactly once, and publicly report success or failure. The success or failure of each CAS is reported in a special field on the CAS list. This controlled execution of the 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. To achieve this careful execution, we first assume that each CAS is versioned, so that a belated CAS that occurs long after the original CAS completed, cannot foil the execution and it must fail. Furthermore, a common problem that appears when designing wait-free algorithms with helping threads naturally arises here. We need to execute the CAS in one memory location and report that the execution succeeded (or not) in a different memory location (in the CAS list pointed to by the operation record). To achieve this, we use, in addition to the version number (or in fact as part of the version number), an extra `modified bit` to signal whether the CAS has already been executed (successfully) or not. The details follow.

5.1 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 [18]. The queue in [18] supports the standard ENQUEUE and DEQUEUE operations. We slightly modify it to support three operations: ENQUEUE, PEEK, and CONDITIONALLY-REMOVE-HEAD. The ENQUEUE operation just enqueues 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 queue is in fact simpler to design than the original queue, because DEQUEUE is not needed, because PEEK requires a single

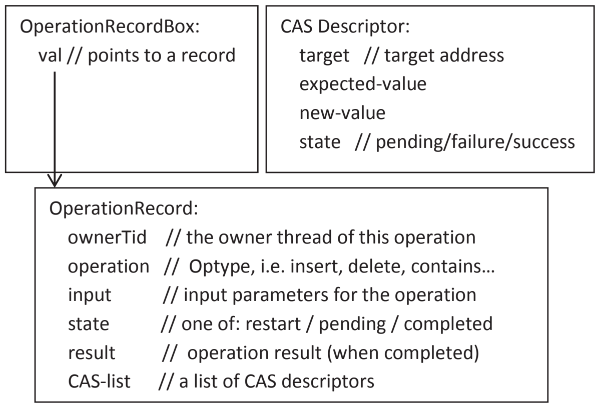
OperationRecordBox:
val // points to a record
CAS Descriptor:
target // target address
expected-value
new-value
state // pending/failure/success
OperationRecord:
ownerTid // the owner thread of this operation
operation // Optype, i.e. insert, delete, contains…
input // input parameters for the operation
state // one of: restart / pending / completed
result // operation result (when completed)
CAS-list // a list of CAS descriptors

Figure 1. Operation Record

read, and the CONDITIONALLY-REMOVE-HEAD can be executed using a single CAS. (Therefore, 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 [18], and we simply used it in our case as well.

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.

5.2 Asking for Help and 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).

A thread starts executing its own operation by running the (normalized) lock-free version of the operation. As lock-freedom does not guarantee non-starvation, the thread may run for a long time without making progress. To obtain non-starvation, we make the thread check periodically that its *contention failure counter* does not exceed a predetermined limit. This check should be performed periodically, e.g., on each function call and each backward jump. If the operation completes while executing the lock-free (fast) path, then we are done. This must be the case when no contention occurs, because the given lock-free algorithm must make progress when no contention is encountered. Otherwise, the contention failure counter will notify that the operation is delayed due to contention. In this case, the thread creates

```
void help (boolean beingHelped, OperationRecordBox myHelpBox)
{
    while (true) {
        OperationRecordBox head = helpQueue.peekHead();
        if (head != null)
            helpOp(head);
        if (!beingHelped || myHelpBox.val.state == OpState.completed)
            return;
    }
}
```

Figure 2. The help method

an operation record, encapsulates it with an operation record box and requests help by enqueuing the operation record box on the wait-free `help queue`. Next, the enqueuing thread starts helping the operations on the help queue one by one, until its own operation is completed (in practice, its own operation is likely to be the only one in 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, which generates the list of CAS descriptions for the CAS EXECUTOR. It runs a monitored version of the generator, which occasionally checks the contention counter in order to guarantee this method will not run forever. If the CAS GENERATOR completes its execution without being halted prematurely due to the contention failure counter, The PRECASES method allocates a new operation record that holds the result returned by the CAS GENERATOR. Then, the PRECASES method attempts to make its operation record 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, any operation

```
void helpOp(OperationRecordBox box) {
    OperationRecord record = null;
    do {
        record = box.val;
        if (record.state == OpState.restart) {
            preCASes(box, record);        ▷ CAS generator plus extras.
        }
        if (record.state == OpState.pending) {
            executeCASes(record);   ▷ carefully executes the CAS list.
            postCASes(box, record);     ▷ wrap-up method, plus extras.
        }
    } while (record.state == OpState.restart || record.state == OpState.pending);
    helpQueue.conditionallyRemoveHead(box);
}
```

Figure 3. The helpOp method

```
void preCASes(OperationRecordBox box, OperationRecord record) {
    ArrayList<ICasDesc> cas-list = MonitoredRun(Of GeneratorMethod on record);
    if (cas-list != null) {                                              ▷
        newRecord = new OperationRecord(record.ownerTid, record.operation, record.input, OpState.pending, null, cas-list);
        box.val.compareAndSet(record, newRecord);
    }
}
```

Figure 4. The preCASes method

record installed by any of the concurrently executing threads is a proper record that can be used to continue the operation.

If during the execution of the CAS GENERATOR method the contention 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.) If the `OperationRecord` is not replaced by a new one, then soon enough all threads will only run this CAS GENERATOR method, all helping the same operation.

Using the fact that the original algorithm is lock-free, it is possible to show that eventually some thread will successfully complete the operation without being interrupted by the contention failure counter. Intuitively, parallelizable methods can only execute a finite number of CASes while no owner CASes are exeucted[6]. For example, in the normalized form of Harris's linked list, given in Section 6, parallelizable methods can only execute physical deletions of logically deleted nodes. Thus, the number of CASes executed in the parallelizable methods is bounded by the number of logical deletions, and logical deletions are never executed inside parallelizable methods. Once the helping threads complete all the remaining physical deletions, they will execute no more CASes, and will not stop each other from completing.

[6] Formally, this is not accurate and some pathological exceptions exist. However, even in those cases some thread must successfully complete the operation eventually.

Next, we carefully execute the CASes obtained by the PRECASES method. This is done in the EXECUTECASES method (Figure 5). It receives as its input the operation record, which holds the list of CAS descriptions 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. 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 CASed memory address and the CAS state) we cannot run this atomically without using a heavy mutual exclusion machinery that foils wait-freedom (and is also costly).

We solve this atomicity problem and an additional ABA problem by adding the versioning mechanism to the fields that are being CASed. The ABA problem is introduced because a thread may be inactive for a while and then attempt to execute a CAS that was already executed. Returning to the atomicity issue, we must report the outcome of a CAS execution correctly, even if the thread that performed the successful CAS is delayed before reporting the success and all other threads fail to execute the CAS (as the field has already been modified). The ABA and the atomicity problems nicely represent two major standard problems that arise with the incorporation of a help mechanism to make a lock-free algorithm wait-free.

To solve the first ABA problem we employ versioning with all fields that require CASes. To solve the second problem we use an additional bit with each CASed field. (Practically, this would be one of the version bits.) This bit, denoted the `modified bit`, will signify that a successful CAS has been executed but (possibly) not yet reported. So when a CAS is executed, a successful execution will put the new value together with the `modified bit` set. All further attempts to modify this field must fail as the old value of any CAS never has this bit set. Thus, the operation that follows a successful CAS operation must be a CAS that clears this bit. However, before any thread attempts to clear this bit, the thread must first update the `state` of the CAS to reflect this success.

To summarize, the EXECUTECASES method 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 and the `modified bit` is set, and if the version has not yet been incremented, then it uses a CAS to clear the modified bit and increment the version and moves on to help the next CAS. Otherwise, if the CAS `state` is currently set to failure, then the EXECUTECASES method immediately returns. 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` to success and only then attempts to clear the modified bit. Now if

```
private void executeCASes(OperationRecord record) {
    for (int i = 0; i < record.numOfCases(); i++) {
        ICasDesc cas = record.list.get(i);
        if (cas.GetState() == CasState.success) {
            cas.ClearBit(); ▷ clears modified bit (if set). (See Remark 1)
            continue;
        }
        if (cas.GetState() == CasState.failure)
            return;
        cas.ExecuteCas();
        if (cas.ModifiedBitSet()) {          ▷ Checking not only that the
    modified bit is checked, but also that the version has not changed.
            cas.SetState(CasState.success);    ▷ Cas Succeeded. State is
    modified atomically with a CAS.
            cas.ClearBit();                              ▷ (See Remark 1)
        }
        if (cas.GetState() != CasState.success) {
            cas.SetState(CasState.failure);                      ▷ Either
    this state change will fail because it has been previously updated, or
    the CAS really failed.
            return;
        }
    }
}
Remarks:
1. Clearing the modified bit is done with a CAS. The expected value
   for this CAS instruction is the new value of the original CAS.
```

Figure 5. The executeCASes Method

the CAS `state` is not set to success, then EXECUTECASES sets this state to failure and returns. Otherwise, success is achieved and EXECUTECASES proceeds to the next CAS.

An invariant of this mechanism is 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.

The existence of the `modified bit` requires a minor modification to the fast-path. 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. If so, the thread should pause its current execution and call the help method to participate in helping the current operation to complete (clearing this bit in the process). Failing to do so will foil the lock-freedom property (and the wait-freedom property as well), because the thread may forever fail in attempts to modify that field, even when running solo.

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

```
void postCASes(OperationRecordBox box, OperationRecord record)
{
  shouldRestart, operationResult = MonitoredRun(of Wrapup Method on record);
  if (operationResult == Null) Return
  if (shouldRestart)
    newRecord = new OperationRecord(record.ownerTid, record.operation, record.input, OpState.restart, null, null);
  else
    newRecord = new OperationRecord(record.ownerTid, record.operation, record.input, OpState.completed, operationResult, null);
  box.val.compareAndSet(record, newRecord);
}
```

Figure 6. The postCASes Method

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.

6. Example: Harris's Linked-List

Harris designed a practical lock-free linked-list [14]. A Java implementation of it is available in [17]. Harris's 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.

We start by noting that Harris's SEARCH method, which is used by both the INSERT and DELETE operations, is a *parallelizable method* as is (there is no need to change it to make it parallelizable). 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 the times this CAS failure happens.

We now specify a normalized form 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 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). For a delete operation, call the original search(key) prior to returning true, to ensure the node is physically removed.
  - 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 [17] (which is already a parallelizable method) and exit with the same result.

Note that there is a difference between calling the search method from inside the help mechanism via the GENERATOR method, and calling it from outside the help mechanism while executing the fast lock-free path. When calling from outside the GENERATOR method, if a thread tries to remove a logically deleted node and fails due to a checked `modified bit`, the thread switches to the help mechanism[7]. While when called from inside the Generator method, it is already inside the help mechanism, and thus recalling it should be avoided. For this reason, in the actual Java code, the search method receives a second parameter, a boolean that signifies whether the calling thread is already inside the slow path or not.

7. Correctness Highlights

We now discuss some key issues in the correctness proof for the simulation described in this paper. Assume that the given algorithm, denoted LF, is a linearizable lock-free algorithm presented in a normalized form for a certain abstract data type, ADT. Let WF be the output algorithm as described in

[7] As explained in Section 5.2, failing to do so will foil the lock-freedom property.

Section 5 with LF being the simulated lock-free algorithm. Then we claim that WF is a linearizable wait-free algorithm for the same abstract data type, ADT.

We show that for every execution of WF, there is an *equivalent execution* (Definition 3.2) of LF. Since we know that LF is correct and linearizable, it follows that WF is correct and linearizable as well. Consider any given execution of WF. We build an equivalent execution of LF in three steps.

Step 1: (Segregate generator and wrap-up.) Given an execution E_0 of WF we build an equivalent execution E_1 in which any operation executed in the slow path in E_0 by several helping threads is replaced by an operation that is executed by one single thread in the slow path. This single thread in E_1 executes the GENERATOR and WRAP-UP methods as they were performed by threads that successfully reported their respective results into the `operation record box` of this operation in E_0. Other threads may execute the GENERATOR or WRAP-UP *parallelizable methods* concurrently in E_1, but the output they produce is discarded and not used.

Step 2: (Move redundant help operations to auxiliary threads.) Given an execution E_1 output by Step 1, we build an equivalent execution E_2 in which all the extra executions of the GENERATOR or WRAP-UP *parallelizable methods*, whose output is discarded, are executed by new additional unrelated threads, each performing only one such extra method execution.

Step 3: (Drop auxiliary threads.) Given an execution E_2 output by Step 2, we construct an equivalent execution E_3 in which the additional unrelated threads are simply dropped. Using the fact that these additional threads execute *avoidable* methods, it can be shown that the resulting execution is equivalent. The execution E_3 is a legal linearizable execution of LF, which is equivalent to E and we are done.

Wait Freedom An easier yet critical part of the proof is showing that each operation in WF is completed after a finite number of steps. This is done in two steps. First, we show that if there is an operation in WF that takes an unbounded number of steps to complete, then from some point in the execution, no operation at all can be completed (since all the threads will be helping this single operation). Second, we use again the transition from an execution of WF (E_0) to an execution of LF (E_3). If for infinitely many steps no operation is linearized in E_0, then no operation is linearized in E_3 as well, in contradiction to the lock-free property of the original algorithm. Thus, WF is shown wait free by way of contradiction.

8. Performance

We chose four well-known lock-free algorithms of three widely-used data structures, and used the transformation described in this paper to derive a wait-free algorithm for each. In our implementation we applied a (generally applicable) optimization: we used the original (non-normalized) lock-free algorithm for the fast path with an additional contention failure counter for each of the methods of the original algorithm. This optimization is possible whenever the normalized algorithm and the original algorithm can safely run concurrently on the same data structure.

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 [14], the binary-search-tree of Ellen et al. [7], the skiplist of Herlihy and Shavit [17], and the linked-list of Fomitchev and Ruppert [11]. All implementations were coded in Java. The Java implementations for Harris's linked-list and the skiplist were taken from [17]. We implemented the binary search tree and the list of Fomitchev and Ruppert in the most straightforward manner, following the papers.

In this work we do not specifically address the standard problem of memory management for lock-free (and wait-free) algorithms. If the original lock-free algorithm reclaims unused objects, then the obtained simulation works well with it, except that we need to reclaim objects used by the generated algorithm: the operation records, the operation record boxes, and the nodes of the help queue. This can be done using a few (constant) number of hazard pointers [21] per thread. The implementation is tedious, but does not introduce a significant difficulty.

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 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 Xeon, and 1 to 32 in the Opteron. The keys were randomly and uniformly chosen in the range $[1, 1024]$. 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 compares the four algorithms when running on the Opteron (the left graph of each couple) and on the Xeon (right). The figure shows the execution times (seconds) as a function of the number of threads. The performance of the wait-free algorithms is comparable to the lock-free al-

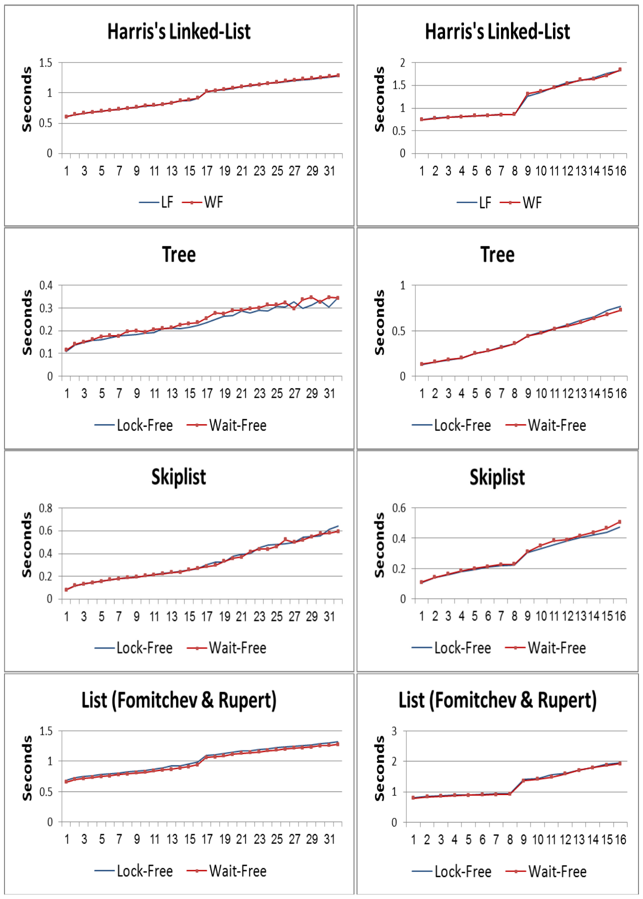

Figure 7. Lock-Free versus Wait-Free algorithms. Left: Opteron. Right: Xeon

gorithms, the difference being 2% on average. We also measured how frequently the slow path was invoked. The fraction of operations running the slow path is small (up to 1/3,000). This allows the wait-free algorithms to retain similar performance to the lock-free algorithms. Still, a minority of the operations require the help mechanism to guarantee completion in a bounded number of steps, necessary for achieving wait-freedom.

References

[1] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *STOC*, pages 538–547, 1995.

[2] James H. Anderson and Yong-Jik Kim. Fast and scalable mutual exclusion. In *DISC*, pages 180–194, 1999.

[3] James H. Anderson and Yong-Jik Kim. Adaptive mutual exclusion with local spinning. In *DISC*, 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*, pages 335–344, 2010.

[6] Tyler Crain, Damien Imbs, and Michel Raynal. Towards a universal construction for transaction-based multiprocess programs. In *ICDCN*, pages 61–75, 2012.

[7] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *PODC*, 2010.

[8] Panagiota Fatourou and Nikolaos D. Kallimanis. The redblue adaptive universal constructions. In *DISC*, pages 127–141, 2009.

[9] Panagiota Fatourou and Nikolaos D. Kallimanis. A highly-efficient wait-free universal construction. In *SPAA*, pages 325–334, 2011.

[10] Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-free algorithms can be practically wait-free. In *DISC*, pages 78–92, 2005.

[11] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *PODC'04*, pages 50–59, 2004.

[12] Michael Greenwald. Two-handed emulation: how to build non-blocking implementation of complex data-structures using dcas. In *PODC*, pages 260–269, 2002.

[13] Rachid Guerraoui, Michal Kapalka, and Petr Kouznetsov. The weakest failure detectors to boost obstruction-freedom. *Distributed Computing*, 20(6):415–433, 2008.

[14] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC'01*, 2001.

[15] Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *PPOPP*, pages 197–206, 1990.

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

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

[18] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *PPOPP*, pages 223–234, 2011.

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

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

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

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

[23] Gadi Taubenfeld. Contention-sensitive data structures and algorithms. In *DISC*, pages 157–171, 2009.

[24] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *OPODIS*, 2012.