Wait-Free Linked-Lists [*]

Shahar Timnat Anastasia Braginsky Alex Kogan Erez Petrank
Dept. of Computer Science, Technion
{stimnat, anastas, sakogan, erez}@cs.technion.ac.il

Abstract

The linked-list data structure is fundamental and ubiquitous. Lock-free versions of the linked-list are well known. However, the existence of a practical wait-free linked-list has been open. In this work we designed such a linked-list. To achieve better performance, we have also extended this design using the fast-path-slow-path methodology. The resulting implementation achieves performance which is competitive with that of Harris's lock-free list, while still guaranteeing non-starvation via wait-freedom. We have also developed a proof for the correctness and the wait-freedom of our design.

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

General Terms Algorithms, Design, Performance

1. Introduction

A linked-list is one of the most commonly used data structures. The linked-list seems a good candidate for parallelization, as list modifications to different parts of the list may be executed independently and concurrently. Indeed parallel linked-lists with various progress properties abundant in literature. Most notably, lock-free linked-lists are well known. A lock-free data structure ensures that when several threads are concurrently accessing the data structure, at least one of them makes progress within a bounded number of steps. While this property ensures general system progress, it does not prevent starvation of a particular thread, or even of all the threads but one. Wait-free data structures ensure that each thread makes progress within a bounded number of steps, regardless of other threads' concurrent execution. However, practical concurrent data structures that ensure wait-freedom are notoriously hard to design. It was only recently that wait-free designs for the simple stack and queue data structures appeared in the literature [1, 6]. The stack and the queue have limited parallelism, as they have a limited number of contention points (e.g., the head of the stack, and the head and the tail of the queue). An interesting question that arises is whether it is possible to design an efficient wait-free algorithm for any highly parallel data structure. To the best of our knowledge, there is no wait-free linked-list algorithm available in the literature except for use of universal constructions, which are inefficient and impractical. We are not aware of any practical wait-free design for any more complex data structure as well.

The main contribution of this work is a practical, linearizable, fast and wait-free linked-list. Our construction builds on the lock-free linked-list of Harris [3], and extends it using a helping mechanism to become wait-free. The main technical difficulty is making sure that helping threads perform each operation correctly, apply each operation exactly once, and return a consistent result (of success or failure) according to whether each of the threads completed the operation successfully. Next, we extended our design using the fast-path-slow-path methodology of Kogan and Petrank [7], in order to achieve performance which is almost equivalent to that of the lock-free linked-list of Harris.

Our wait-free linked-list design follows the traditional practice, in which concurrent linked-list data structures realize a sorted list, where each key may only appear once [2, 3, 5, 8, 9]. An attempt to INSERT a new node with a key that already exists results in a failure.

We have implemented the wait-free linked-list and compared its efficiency with the implementation of Harris' lock-free linked-list. The naive algorithm has a substantial overhead, performing worse by a factor of three when compared to Harris' lock-free algorithm. However, the application of the fast-path-slow-path extension reduces the overhead significantly, bringing it to about 2-5%. This seems to be a reasonable overhead to be paid in practice for obtaining a non-starvation guarantee. In this brief announcement we present only a general description of our algorithm, and of the main challenges we encountered.

1.1 Background and Related Work

The first lock-free linked-list was presented by Valois [9]. A simpler and more efficient lock-free algorithm was designed by Harris [3], Michael [8] added a hazard-pointers mechanism to allow lock-free memory management for this algorithm. Fomitchev and Rupert achieved better theoretical complexity in [2]. Herlihy and Shavit implemented a variation of Harris's algorithm [5], and we used this implementation both for comparison and as the basis for the Java code that we developed. Recently, wait-free queues were presented in [1, 6]. A different approach for building concurrent lock-free or wait-free data structures, is the use of universal constructions [4, 5]. However, universal constructions (at least for the linked-list) are inefficient and non-scalable.

In these proceedings, Kogan and Petrank [7] present a technique called fast-path-slow-path, which makes wait-free algorithms almost as efficient as their lock-free counterparts. This technique combines a slower wait-free implementation of a data structure with a faster lock-free implementation, in order to achieve both the faster lock-free performance (or nearly so), and the stronger non-starvation guarantee of the wait-free implementation. We use the fast-path-slow-path methodology in this work to achieve an efficient and wait-free linked-list. The higher complexity of the

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

linked-list (compared to the queue) and its higher potential parallelism, makes the application of the fast-path-slow-path methodology more involved than the application of this method for the queue as presented in [7].

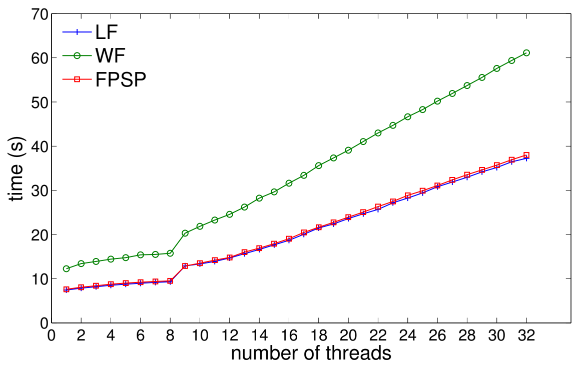
2. The Algorithm: an Overview

The wait-free linked-list supports three operations: INSERT, DELETE, and CONTAINS. All of them run in a wait-free manner. The algorithm builds on Harris's lock-free linked-list [3], and uses an enhanced helping mechanism to make it wait-free. Similarly to Harris's linked-list, our list contains sentinel head and tail nodes, and the *next* pointer in each node can be marked using a special *mark bit*, to signify that the entry in the node is logically deleted.

To achieve wait-freedom, a helping mechanism is used. The helping mechanism employs a special *state* array, with an entry for each thread. When a thread wishes to perform an operation on the list, it first chooses a phase number, higher than all phase numbers previously selected, and posts an operation-descriptor in its *state* array entry. The operation descriptor describes the operation it wishes to perform and also contains the phase number. Next, the thread goes through all the *state* array, and helps perform operations with smaller or equal phase numbers. This ensures wait-freedom: a delayed operation eventually receives help from all threads and soon completes.

Designing a concurrent algorithm in the presence of helping threads is a lot more difficult than designing the lock-free counterpart, because the wait-free version with all the helping threads must deal with many more potential races. One major difficulty with helping threads occurs when several threads are attempting a similar operation (such as DELETE 6) concurrently and several more threads are helping them to apply it. At the end, we must apply this operation only once (correctly) and properly report success or failure to each of the threads that initiated the operation (even though the operation might have been applied by a helping thread who later stopped responding). While in the lock-free list, each thread knows whether or not it succeeded because it is the one who performs the CAS that causes the (logical) deletion, the same is not true for the wait-free linked-list. To deal with this added complexity, we employed a designated additional *success* bit in each node's *next* pointer, on which the concurrent DELETE operations compete to determine which of them completed *successfully*. The idea is that when several DELETE operations of the same key are executed concurrently, the actual deletion of the node is done using the help mechanism, but once the node is already deleted, only the threads that initiated the operation (and not the helping threads) compete on setting the additional *success* bit, to determine which of them owns this deletion. This settles the owner of the success in a wait-free manner.

To use this mechanism, we partitioned the DELETE operation into two distinct steps, and provided help for each of them independently. The first step in the DELETE operation is a *search_delete* step, in which the physical node, nominated for deletion, is selected (with help). Next, an *execute_delete* step is executed (with help), in which this selected node is logically deleted by marking its *next* pointer. This delicate partition ensures that for a single DELETE operation, all helping threads might only try to delete the same specific node, selected in the *search_delete state*. The thread that initiate the DELETE operation will compete on the *success* bit of this particular node. We believe that this technique for determining success of a thread in executing an operation in the presence of helping threads can be useful in future constructions of wait-free algorithms.



3. Performance

We compared both our basic wait-free linked-list implementation and the fast-path-slow-path version of it against the lock-free linked-list of Harris. The implementations we used were written in Java. The lock-free Java implementation we compared against is by Herlihy and Shavit [5] and available on the Internet. All the tests were run on SUN's Java SE Runtime, version 1.6.0. We ran the measurements on 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 8MB per processor. In this brief announcement we only report a single micro-benchmark. In this benchmark each thread did one million operations, out of which 50% were insertions, and 50% were deletions. The keys were randomly and uniformly chosen in the range $[1, 1024]$. Each test was run with the number of threads ranging from 1 to 32. Each test was repeated 20 times, and the average of the runs is reported. The standard deviation remained below 2% in most measurements, once peaking above 5% and reaching 6.2%. It turns out that the fast-path-slow-path results nearly matches those of the lock-free algorithm.

Acknowledgments

We thank Maurice Herlihy for his enlightening comments.

References

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

[2] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *PODC'04*, pages 50–59, New York, NY, USA, 2004. ACM.

[3] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC '01*, London, UK, 2001. Springer-Verlag.

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

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

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

[7] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *PPOPP*, 2012.

[8] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA '02*, pages 73–82, New York, NY, USA, 2002. ACM.

[9] John D. Valois. Lock-free linked lists using compare-and-swap. In *PODC '95*, pages 214–222, New York, NY, USA, 1995. ACM.