Re-Implementation of:
Practical Non-blocking Unordered Lists
Final Report

1st Michael Harris
University of Central Florida
mmph87@knights.ucf.edu

2nd Marcus Sooter
University of Central Florida
mdsooter@knights.ucf.edu

***Abstract*—One of the most important data structures in computer science is the list, but parallelizing it efficiently requires lock-free and wait-free algorithms that do not suffer from a large management overhead, preserving the performance of concurrent stack accesses. We use Java to implement existing work in the field, and this paper will outline the lock-free implementation of this algorithm.**

I. Introduction

The linked list is a data structure that is an integral tool to programmers, and is usually straightforward to implement. Adding concurrency adds a necessarily complex solution; The nature of a linked list is that to travel to one node, you must have already visited it's successor, In a sequential program this is normal, but in a concurrent configuration, this will cause contention between threads attempting to access the same node, so threads must be aware of other ongoing operations.

Careful design decisions must be made when implementing a lock-free stack to ensure that concurrency benefits are achieved without excessive management overhead. This implementation manages lock-freedom by utilizing helper nodes with a state that indicates in progress operations. Helper functions are critical to ensuring other threads are helped along during list traversals, physically deleting nodes in a linearizable way so that the stack always "looks" correct to each thread at any given time.

The paper goes further with the wait-free implementation to provide even stronger progress guarantees, including a fully wait-free enlist that helps other threads that were spawned at an earlier point in time, accomplished with a strictly increasing phase counter. This implementation is an in-progress work located on our team's GitHub project page. (1)

II. Related Work

This implementation is based on the work by Zhang, Zhao, Yang, Liu, and Spear. (2) A lock-free `enlist` function, called internally by `insert`, is the bottleneck in the otherwise fully wait-free implementation. Making `enlist` wait-free yields a fully wait-free algorithm. This is described within the paper as the 'slow-path/fast-path methodology', where the lock-free version is utilized when contention is low, which yields excellent performance, but when contention is high, the wait-free version is used to provide a higher progress guarantee at the cost of slightly lower throughput. Linearizability is accomplished in this data structure with a single `CAS` operation on a memory-word sized object.

The paper's inspiration to build upon existing work in the field cites the following recent advancements: Valois developed the first lock-free list to rely upon `CAS`, and utilized the technique of indicating ongoing operations in nodes. Harris implemented a technique using bit stealing to indicate deletion, where Michael improved on this by utilizing hazard pointers for deletion of nodes, yielding a viable fast-path algorithm for our purposes. The performance can be improved further by utilizing wait-free lookups, instituted by Heller, et al. Combining the Harris, Michael, and Heller algorithms results in unmatched performance as well as strong progress guarantees. (2)

III. Technical Approach Overview

When inserting or removing, these methods first create an intermediate node, with the state

INS or REM, which is linked to the head of the list by Enlist. HelpInsert then decides the outcome of this operation – if the key was already in the set, the intermediate node's status is set to INV to indicate the key was already in the set, and set the intermediate as logically deleted. If the key was not in the set, the intermediary is set to DAT and the insertion linearizes here. Updating the intermediate node requires a CAS to allow updating the state in a thread-safe way. A CAS failure kicks off HelpRemove so as to physically prune the newly deleted node. When removing a node, a REM intermediary node is stored at the head and HelpRemove is called to traverse the list and determine if the value is present. The intermediary node is set to INV to logically remove it from the list, lazily delaying the physical removal. Contains is a read-only function and will skip over any INV nodes while traversing, immediately return false if a REM node with a matching key is found, and return true only if the value found is an INS or DAT node.

HelpRemove and HelpInsert do the heavy lifting to coordinate concurrent access to this data structure. Both take logically deleted nodes and perform the actual physical removal. Key to the algorithm's success is that nodes must be added at the head, allowing each helper operation to complete a list traversal without running into any high contention areas which may delay progress.

Lock freedom means that at least one operation completes in a finite number of steps, with no guarantees made toward other threads progressing. The helper functions are both lock-free due to the fact that they will always terminate in O(len) time. This is because the list cannot form a cycle: each time a new node is added in, it is either a brand new node or the pred, curr ordering is maintained preventing a cycle from forming. CAS operations that fail turn the node's state into DAT or INV, preventing any further processing of this node. Enlist is lock-free since it is only called once by both Insert and Remove, and can only fail a maximum of n-1 times for n threaded instances. In other words, at least one CAS operation will always succeed, guaranteeing progress. Further, a CAS failure by Enlist means it has already succeeded elsewhere.

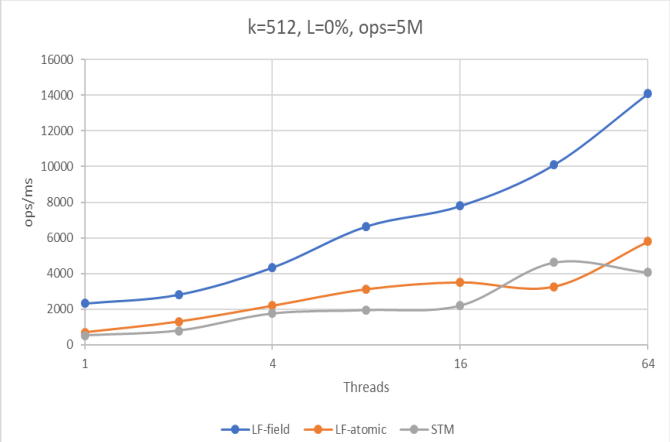
Loosely, each call to Insert or Remove will reach linearizability when the CAS operation is successful. Contains' linearizability point depends upon whether the element being searched for is located in the set with a valid mark. In both these cases, linearizability happens on line 16. If, however, Contains returns false, the linearizability point will happen after a different Remove call hits it's linearizability point on line 16, but before the Contains returns.

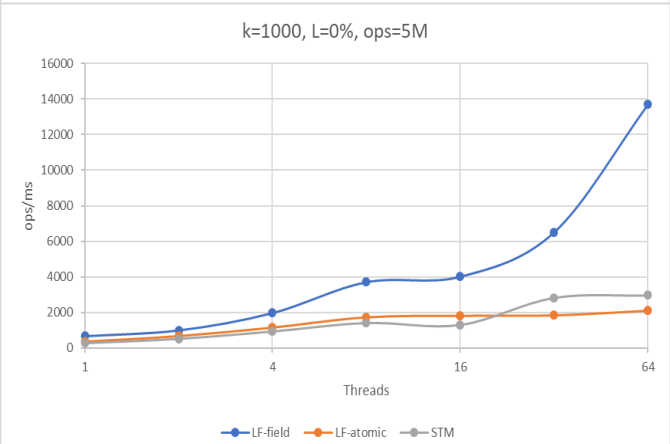
IV. Implementation Details

Original prototyping was performed in C++17, but Java SE 12 was chosen for the final implementation. This done was for a multitude of reasons, including using Java's garbage collection to avoid the ABA problem, the paper's reference to AtomicFieldUpdaters, and ease of use of software transactional memory tools.

Pseudocode provided by the Zhang, et al. paper is succinct and correct, with only minor adjustments needed, depending on the implementation. The Java implementation utilizes AtomicFieldUpdaters to avoid the overhead caused by performing compare and set operations on atomic references.

Originally, a state class was implemented, but later implementations saw a switch to AtomicInteger, which was included in the final benchmarking tests. A shift away from AtomicIntegers to AtomicFieldUpdaters yields the best performing version of the data structure, with the underlying data representation now held by the Integer class for the purposes of the field updaters. A further optimization could be to use the Character wrapper class to represent this information, since only 4 unique states exists, which can easily fit inside of an 8-bit byte. Wrapper classes are required to hold this information due to the constraints of the field updaters.





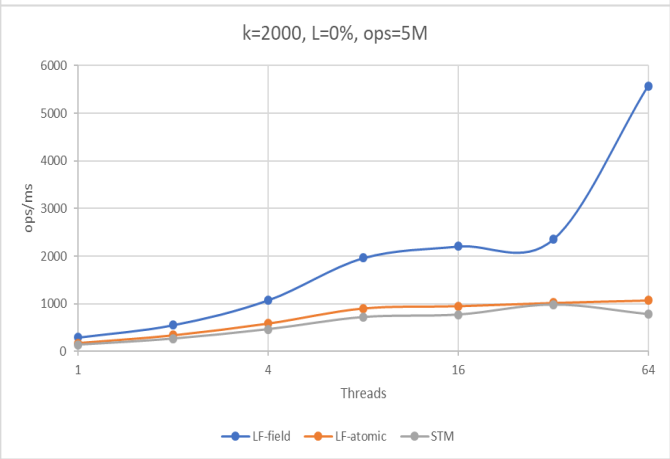
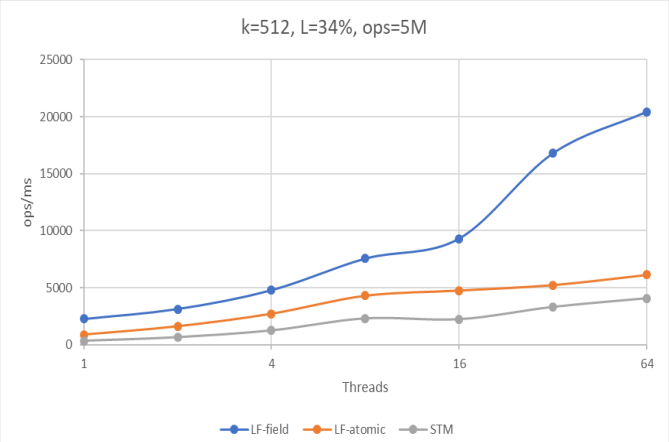
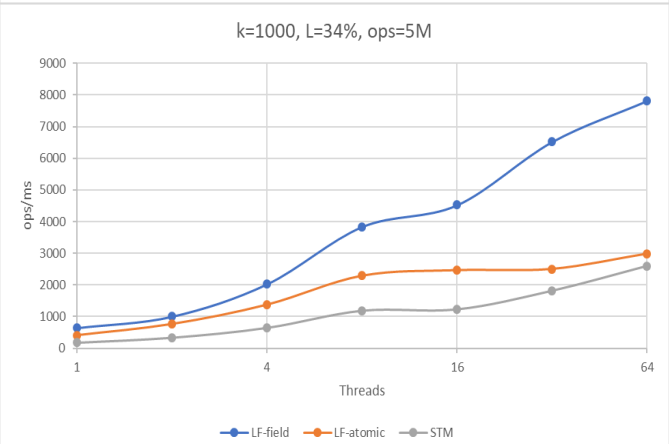


Fig. 1. k: varies, L=0%, ops=5M





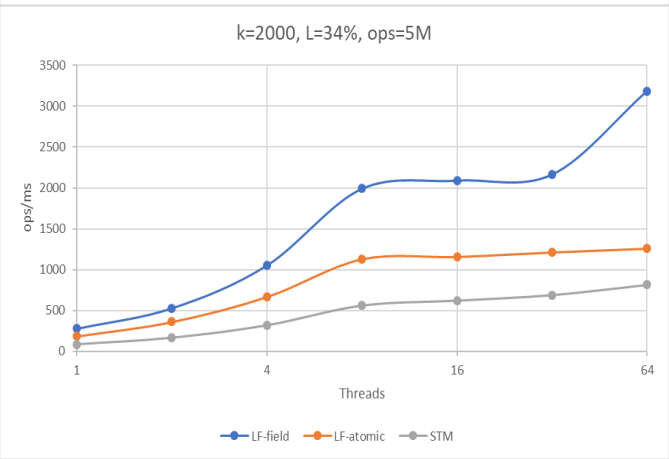


Fig. 2. k: varies, L=34%, ops=5M

V. Benchmarks

Benchmarks were performed on an Intel Core i7-6700 (3.40 GHz base clock) Quad-core CPU with Hyper-threading (8 logical cores), with 32 GB of RAM @ 2133 MHz running 64-bit Windows 10 Pro Version 1803.

In order to test the different versions of the List, a test driver was implemented. The driver is given the number of threads to run, the range of inputs allowed, the percentage of calls which are contains (the remaining percentage are evenly split between insert and remove), and lastly the number of instructions. Each thread is passed pre generated random data for both the keys and operations it will use. The total number of instructions to

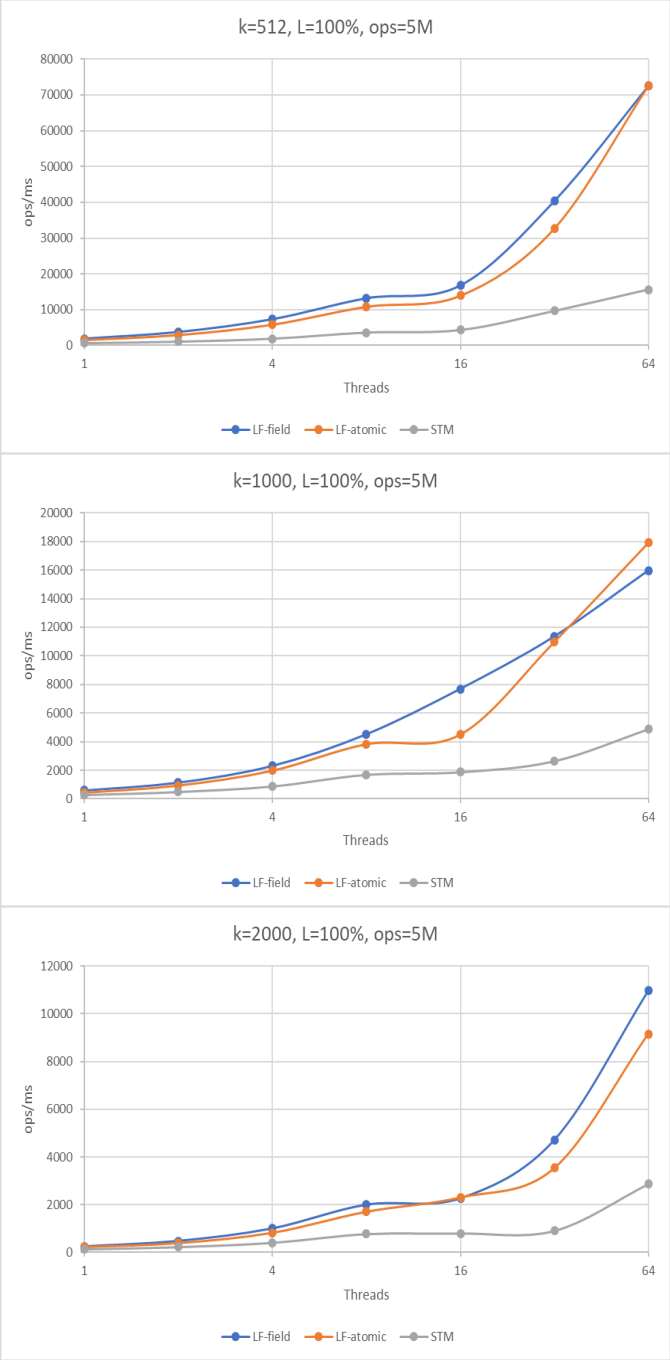


Fig. 3. k: varies, L=100%, ops=5M

complete are split between each thread. At this point the threads are started, when they complete their instructions, each thread returns the number of milliseconds it took to complete it's instructions. These times are summed and averaged, and then used with the total instruction count to find an average operations per millisecond.

Due to all of the threads being given both random data, and the same number of instructions, some threads will complete their set of instructions earlier than others; However, due to all of the threads averaging out their times after they all complete, any inaccuracy that would result from threads either taking too long, or completing abnormally quickly will be minimized through both the number of threads and a high instruction count.

Each of the 9 graphs display a thread count of 1, 2, 4, 8, 16, 32, and 64 threads, with varying levels of lookup percentages, with the remaining operations split evenly between insert and remove method calls. Key size was also restrained in tests, and a uniform 5 million operations were performed for each test.

The benchmark testing performed 63 individual tests, which each did averages to prevent statistical aberrations from skewing the test results. The below times show how long it took for each version of the program to complete the 63 benchmarks, which correspond to 320 million total operations. Specific testing methodology is discussed in the sections ahead.

Time to complete 320M ops:
STM (both): 584 seconds
LF (field updaters): 219 seconds
LF (atomic state): 334 seconds

Here we can see that the atomic field updaters provided a 34% performance over using `Atomic` integers to directly `CAS` the state. The software transactional versions of the data structure, each with varying transaction sizes, both performed all tests in the same amount of time, performing 63% slower than the optimal version with field updaters. This is a slight performance hit, but not the extensive hit that a more intricate or less optimally designed data structure would suffer under software transactional memory.

VI. Concurrent Data Structure

For this project, two different versions of the lock free list were implemented. The first utilizes

AtomicReferences, and AtomicIntegers; While the second implementation utilizes AtomicFieldUpdaters. The first implementation of the lock free version utilizes an AtomicReference to the head, and an AtomicInteger as the state. The main advantage of this version is that it allows for an extremely simple and intuitive implementation in Java while still providing comparable results to the field updater version. Because the Atomic version is so simple to implement, it allows users to more easily expand the same ideas to other data structures like stacks, or hash maps.

The field updater version is slightly more complicated, as at any point wherever the user would normally perform a CAS on the state, the field updater is utilized. In this version, there is no longer any need for the state to be atomic, and now only needs to be marked as volatile. This raises performance as now there is no need to acquire the state as an atomic before changing it. Both of these versions have the same linearization points and progress guarantees. However, the field updater version is significantly more efficient and performed far better in the benchmarks than the atomic version.

VII. Software Transactional Data Structure

DeuceSTM 1.2 was used to analyze the software transactional performance of this data structure. Deuce was primarily developed by Guy Korland (3) and runs as a Java agent, performing real time instrumentation of the application bytecode. Version 1.2 of the package was chosen over 1.3 as it appeared more stable with fewer faults.

Methods are indicated as transactional by including an @Atomic flag at the top of the method as a javadoc, with the transaction being able to be re-initiated or aborted through the mechanism of throwing an exception. This is a very useful approach, since catching the exception allows us to re-initiate fields and prepare for a retry of the transaction. This allows only the most critical code to be wrapped in atomic helper methods, indicated in our STM codebase by methodnameDoAtomic().

Performance for the STM data structure is lower than the concurrent version, but surprisingly optimizing the transaction size to be as small as possible does not yield an increase in the efficiency of the data structure. Optimizing the STM version involved only putting key portions of code within synchronization blocks, and moving re-initialization code into the catch block so that the transaction is instantly ready to go when it retries.

However, due to the straightforward nature of this data structure, these minor tweaks appeared to not make any difference during benchmarking, as each version finished in the same amount of time. This is likely due to built in efficiencies within Java and the data structure itself, where the code did not really lend itself to large, inefficient transactional blocks. Most code, derived from the paper's pseudocode, is located within tight loops that do not perform extra actions or unnecessary, unoptimized CAS operations. Each function within the program executes its Atomic counterpart for only the absolute smallest window necessary, due to the well planned out design by the authors. (2)

VIII. Code

Both the code for this reimplementation and the graphs for the compared benchmarking, each version with varied inputs can be found at: https://github.com/mmph87/Practical-Non-blocking-Unordered-Lists (1)

References

[1] M. Harris and M. Sooter, "Re-implementation of practical non-blocking unordered lists," 2019.

[2] K. Zhang, Y. Zhao, Y. Yang, Y. Liu, and M. Spear, "Practical non-blocking unordered lists," 2013.

[3] K. Guy, S. Nir, P. Felber, and B. Igor, "Deuce stm java software transactional memory," 2008.

[4] D. Dechev, "The aba problem in multicore data structures with collaborating operations," *International Conference on Collaborative Computing: Networking, Applications and Worksharing*, vol. 7, pp. 201–213, 10 2011.

[5] D. Dechev, P. Pirkelbaur, and B. Stroustrup, "Understanding and effectively preventing the aba problem in descriptor-based lock-free designs," *IEEE International Symposium on Object/component/service-oriented*

Real-time Distributed Computing, vol. 13, pp. 201–213, 5 2010.

[6] Z. Painter, C. Peterson, and D. Dechev, "Lock-free transactional adjacency list," *International Workshop on Languages and Compilers for Parallel Computing*, vol. 30, pp. 201–213, 10 2017.