Implementation of Scalable Lock-Free Vector with Combining

John Albury, Dax Borde, Matthew Garrison, Richard Snyder
University of Central Florida
{ jalbury, daxborde, matthewgarrison, richard.snyder }@knights.ucf.edu

***Abstract*— In *Scalable Lock-Free Vector with Combining* [1], Walulya and Tsigas introduce the first implementation of a concurrent, lock-free vector that utilizes combining to increase performance. In our project, we implement this concurrent vector as described in [1] and describe our approach here. We also implement a concurrent vector utilizing a Software Transactional Memory library. We compare the performance of our two implementations to each other and to the authors' performance evaluations.**

I. INTRODUCTION

The primary goal of this project is to deliver a working lock-free vector with combining as specified by Walulya *et al.* in [1]. The main difference between this algorithm and the one presented by Dechev *et al.* in [2] (LFVector) is the inclusion of combining. [1] indicates that this inclusion results in performance increases, especially on NUMA architectures. However, the clear cost is the complexity of the resulting code. Compared to the versions presented in LFVector, `pushback` and `popback` are significantly longer and more complex. Additionally, two new methods, `add_to_batch` and `combine`, were added.

Additionally, we implement a transactional version of this vector using the Rochester Software Transactional Memory (RSTM) library. We test several transaction sizes and compare the results to the results for our implementation of the Lock-Free Vector with Combining.

II. RELATED WORK

Dechev *et al.* [2] presented the first lock-free vector, utilizing descriptor objects. While concurrent threads can assist in completing pending tail operations, this implementation suffers from sequential bottleneck in the tail operations, since every tail operation attempts to modify a single descriptor object.

Feldman *et al.* [3] is a wait-free vector that also uses descriptor objects. However, this implementation uses a descriptor for the last element and the tail, instead of a global descriptor. This increases the number of compare-and-swaps needed to complete a `pushback` or `popback`. Additionally, resizing the vector requires copying over the entire vector to a new array. Finally, like LFVector, this implementation suffers from a sequential bottleneck in the tail operations.

Oyama *et al.* [4] presents a lock-based combining technique using an announcement LIFO stack. Unfortunately, this implementation can lead to starvation and has significant overhead.

Hendler *et al.* [5] uses flat-combining, a course-grained lock-based combining technique. Every thread accessing the vector will publish its request to an announcement list before attempting to acquire the global lock. The thread that acquires the lock (the combiner thread) will scan the list for pending requests, apply them to the vector, and then write the response back to the associated request. This implementation requires threads to spin while the combiner thread does its work, and the combiner thread must scan the entire announcement list for any pending requests. Because of these limitations, this implementation suffers from sequential bottlenecking.

Fatourou and Kallimanis [6] present an implementation of a combining technique with the hope of maintaining wait-free progress guarantees. Their technique has each thread announce its operation, copy the state, apply all announced operations to its local copy, and then change the global state to match its local state. This implementation suffers if the state (ie. the vector) is large. A solution was provided (Fatourou *et al.* [7]) to solve this issue, but that solution nullifies the wait-free progress guarantee.

III. CONCURRENT IMPLEMENTATION

Our paper's main improvement compared to previous approaches was using a bounded queue for the combining, rather than utilizing a combining tree. This modification allows the implementation to satisfy lock-free progress guarantees, whereas previous approaches that utilized combining did not. Within the queue, a single thread can have multiple requests. The combiner works in a way such that it traverses only a list of active requests posed in the queue.

The minimum methods available to the vector are:

- `pushback(elem)` - insert *elem* to the tail of the list, increment size
- `popback()` - return the tail node, decrement size
- `reserve(n)` - increase vector capacity to *n*
- `read(add)` - read the value at the address *addr*
- `write(addr, val)` - write value *val* at address *addr*
- `size()` - return the size of the vector

The structure of the data storage component is a two-level array. This structure is utilized to store data in non-contiguous memory, allowing for re-sizing of the vector without having to relocate previously stored data. The size of each block in this two-level array is a power of two, where each new block that is added is allocated twice the amount of memory as the previous block. The initial capacity can be any power of two.

The upper level of the array consists of atomic pointers, as allocating new buckets needs to occur atomically. The lower level consists of atomic integers, as the values of the elements need to be changed atomically.

Note that the arrays on the second level of the data two-level are implemented as std::vectors. These vectors exist only to specify initial size at runtime as opposed to compile time. The size never changes after initialization, and each vector is treated as a normal array on any subsequent accesses.

A. *Progress and Correctness Guarantees*

Like [1], our data structure provides a lock free progress guarantee and linearizable correctness property. The linearization points of our implementation are in similar places to those of [1] and [2]. All of the following specified lines are in `parallel_vector.cpp`. For `pushback`, line 178 under normal circumstances and line 162 if the vector does not have any elements left to pop. For `popback`, line 541 under normal circumstances and line 483 if the operation is completed in the combining phase. The vector is also lock free because at each of these linearization points, a CAS is attempted. If the current thread fails the CAS, by the definition of CAS [8], some other thread must have linearized a different operation at this time. At least one thread is making progress at all times, so our data structure must be lock free.

B. *Synchronization Techniques*

The key synchronization techniques used to provide these guarantees are atomic compare and swap (CAS) operations, descriptors, and the combining queue. CAS allows us to guarantee lock free progress using low level atomic instructions. Even if a particular CAS fails and we must return to the top of a loop, lock free progress is still guaranteed, because a CAS failure implies that another thread's CAS succeeded [8]. Descriptors allow us to provide the lock free guarantee for multiple variables at a time by using CAS on pointers instead of word-sized primitives [2]. Finally, the fixed sized combining queue maintains this guarantee by allowing every active operation to both add items to an open queue and assist with the combining phase, guaranteeing that all operations in the queue will complete, even if a particular thread is starved of computing time [1].

C. *Modifications*

We made a few changes to the implementation described by [1] for our own convenience.

- The value `INT_MIN` (from the `climits` package) is reserved to mean "marked" (or "logically removed") in the underlying data storage. A node is marked by setting its value to `INT_MIN`, instead of bit stealing.
- The size of the first bucket is 2, changed from 8 in [2].

D. *Improvements*

To attempt to improve the vector, we made several additional modifications. First, in addition to keeping a thread-local copy of the size of the vector as described in the vector, we also keep a thread-local copy of the capacity of the vector. Our reasoning for why this might improve the performance of the vector is as follows: both `pushback` and `combine` check whether a new bucket needs to be allocated each time they attempt to insert a new item into the vector. To do this, they load the bucket where the new item would go from the shared vector and check whether the bucket is currently equal to null. This creates a point of contention and often times is unnecessary, especially as the size of the vector grows (since the bucket size doubles for each new bucket allocated, it is less frequent that a new bucket actually does need to be allocated as the vector becomes larger). To alleviate some of this contention in cases where it is obviously unnecessary to allocate a new bucket, we have each thread store a thread-local copy of the capacity, then, to check whether a new bucket needs to be allocated, the thread first checks the value of the thread-local copy of capacity and compares it to the index it would be inserting at next. If the index is smaller than capacity, then we know that a new bucket definitely does not need to be allocated (since the vector can never shrink in capacity, the thread-local value of capacity is a lower bound on the actual capacity of the vector). Otherwise, the thread loads the value of the bucket from the shared vector and checks if it is null in the same manner as before. We implement this modification by updating the thread-local copy of capacity every time a thread attempts to allocate a bucket. As stated above, the thread-local value of capacity is a lower bound on the actual capacity of the vector, and the thread still loads the bucket from memory when it thinks it may need to allocate a bucket, so this modification has no effect on the correctness of the vector.

Another way we attempted to improve performance was by updating the thread-local value of size in several places in addition to the locations described in [1]. Having a thread-local copy of a value be more consistent with the actual value makes it less likely for the actual value to be loaded unnecessarily. In [1], the thread-local copy of size is updated on read operations, write operations, and other places that modify the vector. We extend that to potentially updating the thread-local value of size *any* time a value from the vector is read at any point during execution. This includes several points in `pushback` and `popback` that were not included in [1] (to our knowledge, based on the pseudocode and explanations given). For example, the thread-local value of size is potentially updated any time a write descriptor is created. Our logic for this is as follows: if the value read from the vector is marked, that means that the index that was read from is greater than (or equal to) the actual size of the vector. Thus, if that index is smaller than the current value of the thread-local copy of size, then we know the index is a closer estimate to the actual size of the vector than the current value of thread-local copy is, and we should set the thread-local value of size to the index. Since the thread-local copy is used only as an optimization in [1] (its value is only used for bounds-checking, and if a given index seems out of bounds when compared to the thread-local copy of size, the actual size of the vector is loaded to confirm), this modification has no effect on the correctness of the vector.

E. Advantages and Disadvantages

The major advantage of this implementation over other lock free implementations is its greatly improved performance compared to other resizable arrays that support `pushback` and `popback`. (More details are available in the performance section.) The major disadvantage of this implementation is shared by many other lock free data structures: it is quite complex. Even when compared to its close relative [2], the code doubles in length. Additionally, implementing this data structure in a practical environment requires the use of a non-standard lock free memory management solution.

F. Difficulties

While we eventually created a working implementation of the paper's vector, we did encounter some roadblocks, due to the following issues.

- The code is very complex. The four most important methods (`pushback`, `popback`, `AddToBatch`, and `Combine`) are all quite long, with a lot of if-statements. This makes following the flow of the program difficult.
- The pseudocode has very few comments. Additionally, while the authors do explain how each method works, these explanations are not always detailed enough.
- Some things, like when to update the thread-local copy of size, are not included in the pseudocode and are only briefly mentioned in the explanations of the methods.
- The variable names are short, which makes understanding what their purpose is difficult.
- Some parts of the methods are omitted from the pseudocode, presumably for brevity. For example, the explanation of `Combine` mentions that the global combining queue should be nullified. However, this step is not present in the pseudocode.

IV. TRANSACTIONAL IMPLEMENTATION

For the Software Transactional Memory (STM) version of the vector, we used the Rochester Software Transactional Memory (RSTM) C++ library. To accomplish this, we first developed a sequential version of the vector described in [1] (i.e., the vector is implemented as a two-level array to avoid having to move elements upon re-sizing). Then, we replaced all operations on the shared vector with their transactional equivalents, allowing the vector to be shared across multiple threads in a thread-safe manner.

When testing our STM implementation, we varied the transaction size, which is defined as the number of operations completed in a single transaction. Initially, we set the transaction size to 1 and then increased the transaction size to 2 and then 4. There are various reason why the transaction size can affect the performance of a shared object that utilizes STM. With smaller transaction sizes, a transaction is less likely to fail since it is accessing less memory locations (and, thus, it is less likely to have to restart the transaction), so this can result in better performance. However, larger transaction sizes can have performance benefits as well. There is less overhead in creating and executing one large transaction as opposed to creating and executing many small transactions. The ideal transaction size can also be affected by the system the program is run on and the behavior of the program itself. We compare and explain the results across the different transaction sizes for our vector in the *Performance* section.

A. Improvements

Originally, when implementing the STM version of our vector, we simply placed the sequential versions of the methods in atomic blocks and replaced all operations on shared memory with their transactional equivalents. In our second iteration, the first modification we made was moving code that does not need to be executed atomically to outside of the atomic blocks. For example, calculating the indices in the two-level array that corresponds to a given index does not need to be calculated in the atomic block. Code in atomic blocks may need to be executed many times (if the transaction fails), so leaving this code in the atomic block can lead to repeated, unnecessary computation. The second modification we made was that, if a variable is being modified multiple times within an atomic block, we modified the code to only call "TM_WRITE" on that variable once at the end of the block. Having these extra writes in the atomic block leads to more shared memory accesses than are necessary. For example, when reserving space in the vector, the value of capacity is incremented each time a new bucket is allocated. Instead of writing this value each time it is incremented, we store the new value in a local variable, then write the value of capacity at the very end of the atomic block.

V. PERFORMANCE

The results for our concurrent and transactional implementations are shown in Fig. 1. We test each version of the vector with 1, 2, 4, and 8 threads and record the average time to execute 500,000 operations. We use three different operation ratios: (25% `pushback`, 25% `popback`, 25% `read`, 25% `write`); (60% `pushback`, 20% `popback`, 10% `read`, 10% `write`); and (40% `pushback`, 20% `popback`, 40% `read`, 0% `write`). For the STM version of the vector, we use transaction sizes of 1, 2, and 4 operations. All tests were run on a virtual machine running Ubuntu 18.04, using g++ version 7.3.0 as the C++ compiler.

As can be seen in Fig. 1, our concurrent implementation generally scaled very well as the thread count increased, especially compared to the STM implementation. One reason for this is that STM implementations generally do not scale well compared to other concurrent implementation. As the number of threads (and, thus, contention) increases, threads have to abort and restart transactions more often. The concurrent implementation, however, utilizes the combining queue

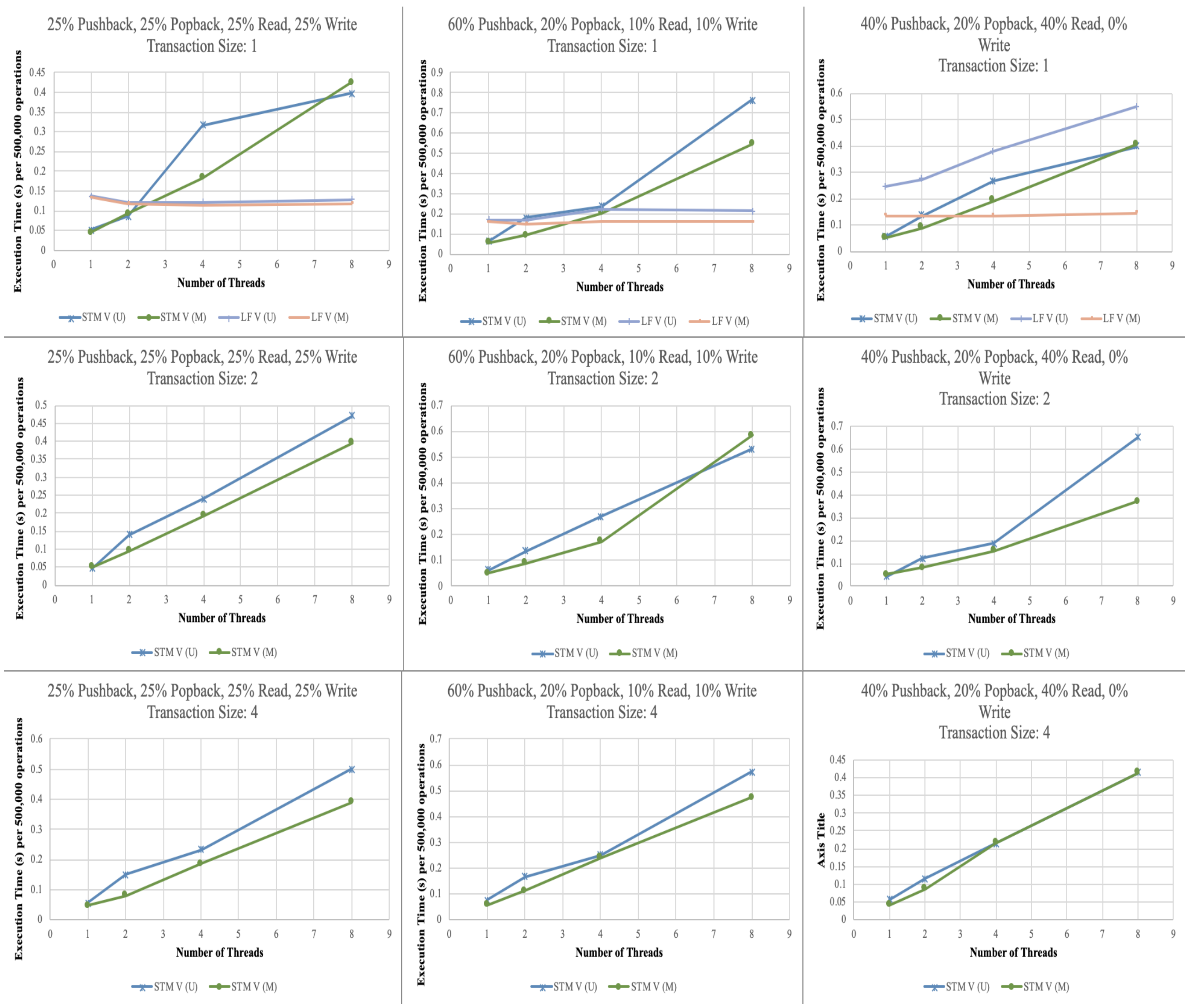

Fig. 1. Performance results for different implementations with *pushback*, *popback*, random access write, and random access read.

described in [1], which is specifically designed to improve performance under high contention.

When comparing our unmodified ("LF V (U)") and modified ("LF V (M)") versions of our concurrent implementation, the modifications we introduced seemed to significantly improve performance, as our modified vector (whose modifications are described in Section III-D) scales much better than our unmodified vector. This could be due to the reduced contention on the vector as a result of introducing the thread-local copy of the vector's capacity.

For the unmodified ("STM V (U)") and modified ("STM V (M)") versions of our STM implementation, our modifications generally seemed to improve performance as well. This is likely due to removing unnecessary accesses to shared memory and reducing repeated computation, as described in Section IV-A.

The performance of the STM vector was generally best with a transaction size of 1, although the difference is not significant. This is likely because aborts at higher transaction sizes are more costly than at lower transaction sizes, since more memory locations are being accessed in each transaction. This difference is more pronounced at higher thread counts, where higher contention causes aborts to be more likely.

VI. Conclusion

In [1], Walulya and Tsigas present a highly scalable vector. The cost of this performance boost is the resulting complexity of the code. We implement our own version of the lock-free vector with combining and add modifications to attempt to improve its performance. We then compare the performance of our implementations to a Software Transactional Memory (STM) implementation of the vector. The performance of these two versions of the vector is comparable at very low thread counts, but the lock-free vector with combining significantly outperforms the STM vector at higher thread counts. This is likely due to the lock-free vector's use of a combining queue in order to alleviate contention.

ACKNOWLEDGMENT

Very special thanks to Dr. Walulya for answering our questions about implementing his data structure.

REFERENCES

[1] I. Walulya and P. Tsigas. Scalable lock-free vector with combining. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 917–926, May 2017.

[2] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Lock-free dynamically resizable arrays. In Mariam Momenzadeh Alexander A. Shvartsman, editor, *Principles of Distributed Systems*, pages 142–156, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[3] Steven Feldman, Carlos Valera-Leon, and Damian Dechev. An Efficient Wait-Free Vector. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):654–667, March 2016.

[4] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, 11 1999.

[5] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures - SPAA '10*, page 355, Thira, Santorini, Greece, 2010. ACM Press.

[6] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. *ACM SIGPLAN Notices*, 47(8):257, September 2012.

[7] Panagiota Fatourou and Nikolaos D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures - SPAA '11*, page 325, San Jose, California, USA, 2011. ACM Press.

[8] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, revised first edition edition, 2012.