Re-implementation of Lock-free Transactional Vector

Daniel Martel, Nyasha Frank, Philip Bettler

March 2020

1 Abstract

The vector is a fundamental data structure whose constant time traversal and re-sizable index range are desirable properties for concurrent structures. A transaction is a group of commands which execute as a package. Combining operations into one unit allows for execution to appear as a single atomic block.

The Lock-free Transactional Vector(LFTV), first of it's kind, is a concurrent version of the vector data structure, which supports a subset of vector operations. LFTV utilizes transactions to isolate groups of operations acting on the vector (size updates, reads and writes) and create an appearance of atomic execution. Each thread can only see the end result of full completed transactions.

Compared against the state-of-the-art transactional designs - GCC STM (Software Transactional Memory), Transactional Boosting, and STO (Software Transactional Objects) - the LFTV performs 18 percent faster on average.

2 Introduction

Vectors are a fundamental data structure in procedural and object-oriented languages; similar to dynamically re-sizable arrays. The elements stored in vectors benefit from contiguous data storage. Contiguous data storage allows for iterators to traverse vector elements. Inserting at the end of a vector takes constant time while inserting and deleting within the vector is linear.

A transaction is a group of commands which execute as a package. Combining operations into one unit allows for execution as a single block. Transactions follow ACID properties, atomicity, consistency, isolation and durability - these characteristics ensure transactions have data consistency and can be used to develop more complex and useful concurrent data structures.

3 LFTV

The Lock Free Transactional Vector establishes a concurrent transactional data structure. While some Lock-free vector designs already exist, LFTV is

the only transactional vector. LFTV boasts varied performance improvements over the state-of-the-art structures in different environments and with different operation ratios. In the case of only read operation testing, the compact LFTV outperforms all other data structures. In read-only cases the LFTV outperforms the state-of-the-art GCC Software Transactional Memory by 80 percent. Additionally, the compact LFTV outperforms transactional boosting by 17 percent on average and is 47 percent faster in the case of 66 percent reads and 33 percent writes.[7]

3.1 Methods

LFTV Implements a subsection of the standard vector operations, the details of implementation for these will be discussed in section 4.Implementation:

- Read: reads a value from the desired index and an out of bound index fails
- Write: writes to a specified index and will also fail with the out of bounds index
- pushBack: writes a pushed value to the end of the vector
- popBack: removes and returns the value at the end of the vector
- Size: returns the number of values stored in the vector
- Reserve: increases the vector's physical capacity

3.2 LFTV Properties: Guarantees, Progress and Correctness Conditions

The LFTV guarantees both progress and correctness as a concurrent data structure. Progress allows the user to assume that any thread will complete within a finite number of steps. Correctness guarantees that real time ordering will be respected, in regards to operation execution. The same transaction input should not produce different results per run. Strict Serializability guarantees that transactions will execute in an order consistent with their real-time execution order, serving as a correctness condition for LFTV. Opacity, which guarantees that non-committed transactions cannot access inconsistent states, is also supported by LFTV. The RWSet used by LFTV is dynamic, meaning that the memory locations are determined within the transaction and not known from the start. This property is essential because of operations such as pushBack and popBack, which operate on the last index of the vector. This last index can be variable as the vector is resized within the transaction. The LFTV also includes a helping scheme to guarantee progress within the lock-free environment. Whenever a thread intersects with an ongoing transaction, the new thread helps complete the work of the previous before beginning it's own transactions, thus eliminating any blocking.

4 Implementation

4.1 Class Structures - Compact Vector Operations

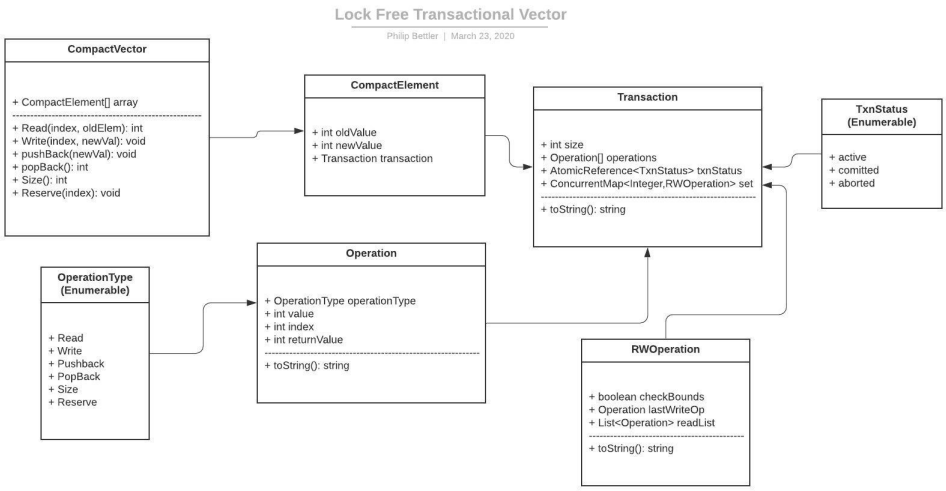

Figure 1: Lock Free Transactional Vector Class Diagrams

The Compact Vector is an array of compact elements wrapped in a class with the 6 crucial transactional vector operations; Read, Write, PushBack, PopBack, Size and Reserve. In Figure 1. The parameters and returns of each of these functions are listed.

Each Transaction has within it an array of operations, and a status indicating where in the process of being committed the specific transaction is. The transaction also holds its own RWSet, labeled "set" in Figure 1. Each of the RWOperations has a checkbounds boolean and lastWriteOp by which to determine the last value to commit to the specific index. The readList is all the reads executed in order for the transaction.

Read uses the index to determine the value of the compact element at the index. Each element has an old value and new value, to determine which one will be read we look to the status of the transaction linked to the element. If committed, we read the new value; aborted transactions would return the old value, and active transactions would need to be helped before continuing. Write checks if the index is in bound and replaces the current value with newVal. PushBack adds newVal to a new element at the end of the vector. Size returns the current size of the compact vector. Reserve expands the vector to the

specified index.

4.2 Preprocessing

Majority of our implementations inherit directly from the method designs of LFTV. Though LFTV is meant to have the ability to create dynamic or static RWSets during preprocessing, we limited our sequential design to only build static RWSets for the time being. Following the approach of the LFTV's preprocessing design, we only insert operations that read or write (read, write, popBack, and pushBack) into the RWSet.

4.3 Sequential Design

For the design of our sequential approach, we relax all synchronization techniques on the vector. We first pre-populate our vector by simulating a fixed number of pushBack transactions on the vector. After the vector is pre-filled, we follow a three step process - build a transaction, preprocess the transaction, then update the necessary elements. When building a transaction, we assign random integers to each variable in the operation class. If the random operation type is popBack, then we store the max integer into the operation's value field. We build each transaction to have 5 operations inside, passing it to the preprocessing stage.

After preprocessing, we pass the transaction descriptor to the CompleteTransaction function, making sure to sort the keys from high to low. Then each index in the set is passed to the UpdateElement function where we follow LFTV's approach for updating elements in a vector.

```
1: function CompleteTransaction(Transaction desc)
2:     indexes = desc.set.keySet();
3:
4:     while indexes.hasNext() {
5:         index = indexes.next()
6:         RWOperation rwop = desc.set.get(index)
7:
8:         res = UpdateElement(desc, index, rwop)
9:         if !res then return false;
10:    }
11:    desc.status = Committed
12:    return true;
```

Figure 2: Algorithm for CompleteTransaction

4.4 MRLOCK

This lock algorithm allows threads to efficiently compete for k shared resources in a deadlock-free and starvation-free manner. A single thread may request anywhere from 1 to k shared resources simultaneously. The request for these resources is represented with a BitSet of length k. Each bit in the BitSet

represents one shared resource. A 1 indicates a request to that resource and a 0 means that resource is not requested. The algorithm uses a size n array buffer to create a lock-free FIFO queue. This queue has a pointer to the head and the tail. A new request of resources is inserted at the tail. It then compares its BitSet value to the other requests starting from the head until it reaches itself. If it finds a request that needs at least one of the same resources it spins on that request. It is released when that conflicted request has completed and unlocked. Once a request lands upon itself it can acquire the lock and use the resources it needs.

We utilize the MRLOCK to Execute the operations of a LFTV with this code:

```
ExecuteOps(Transaction desc) {
        BitSet request = new BitSet();
        Set<Integer> indexes = desc.set.keySet();
        Iterator<Integer> it = indexes.iterator();
        // Setup the resource request
        while(it.hasNext()) {
            // set each bit for every index
            // that we want to access
            request.set(it.next());
        }
        Integer position = lockManager.lock(request);
        // Complete operations here
        lockManager.unlock(position);
}
```

4.5 MR Simple Lock

Though we created an MRLock class, we ended up using the MR Simple Lock for our performance tests. We followed the algorithm written by Deli Zhang - the pseudocode is below. Once a thread has acquired the lock on its requested indexes, it calls UpdateElemNoHelping which replaces the elements in the vector without worrying about helping other threads complete first.

```
1: function mrSimpleLock(Bitset resources)
2:     for(;;) {
3:       BitSet b = bits.get()
4:       if !bits & resources
5:         if bits.compareAndSet(bits, bits | resources)
6:           break
```

Figure 3: Algorithm for MR Simple Lock

```
1: function mrSimpleUnlock(Bitset resources)
2:     for(;;) {
3:       BitSet b = bits.get()
4:       if !bits & resources
5:         if bits.compareAndSet(bits, bits &  ~resources)
6:             break
```

Figure 4: Algorithm for MR Simple Unlock

To run performance tests on the transactional vector using MR simple lock, we prepopulated the vector with 1024 elements and varied the number of threads from 1-32. Each thread was given a total of 5000 transactions to execute. We tested on an Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 2701 Mhz, 2 Core(s), 4 Logical Processor(s) machine. Below are our results.

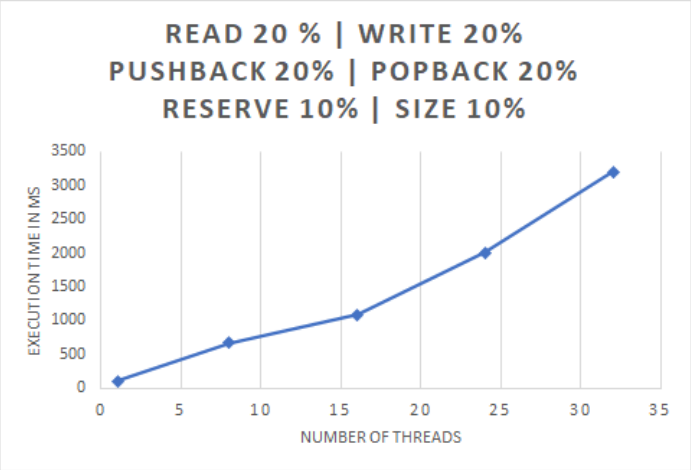

Figure 5: 20% Read and Write, 20% Pushback and Popback, 10% Reserve and Size

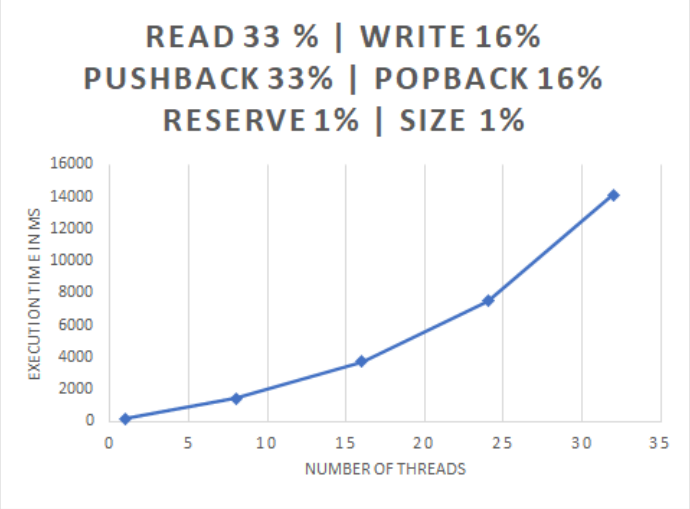

Figure 6: 33% Read, 16% Write, 33% Pushback,16% Popback, 1% Reserve, 1% Size

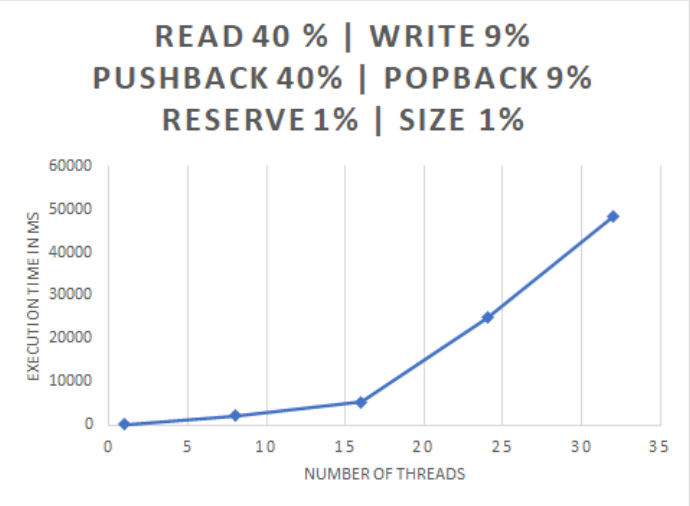

Figure 7: 40% Read, 9% Write, 40% Pushback, 9% Popback, 1% Reserve, 1% Size

4.6 Lock-free

This lock-free design allows t threads to concurrently complete transactions by helping each other when conflict arises. We allow threads that attempt to

```
1: function acquireSize(Transaction t, Integer oldSize)
2:     if(!sizeAcquired) {
3:         if(oldSize != null && t == oldSize.desc)
4:             possibleSize = oldSize.oldValue
5:
6:         do {
7:             oldSize = v.size.get()
8:             newSize = new CompactElement(oldSize.oldValue, oldSize.newValue, t)
9:             while (oldSize.desc.status == TxnStatus.active)
10:                CompleteTransaction(oldSize.desc, 0)
11:
12:            if(oldSize.desc.status == TxnStatus.committed)
13:                newSize.oldValue = oldSize.newValue
14:            else {
15:                newSize.oldValue = oldSize.oldValue
16:                newSize.desc = t
17:            }
18:        } while (!v.size.CAS(oldSize, newSize))
19:        possibleSize = newSize.oldValue
20:        sizeAcquired = true
21:    }
```

Figure 8: Algorithm for Acquire Size

update indexes apart of an already active transaction begin helping the current active transaction at that index. However threads that conflict with size always begin helping from the very beginning and start with preprocessing. The first thread to complete preprocessing for a transaction with set a global RWSet, and from that point on it will not change, even if other helping threads had repeated the preprocessing step.

To run performance tests on the lockfree transactional vector, we prepopulated the vector with 1024 elements and varied the number of threads from 1-32. Each thread was given a total of 5000 transactions to execute. We tested on an Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 2701 Mhz, 2 Core(s), 4 Logical Processor(s) machine. We compared the results with those of the MR Simple Lock implementation.

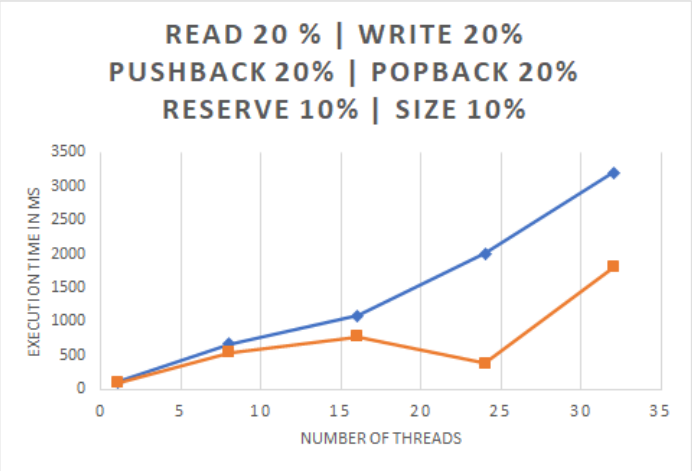

Figure 9: Performance tests, blue = locked, orange = lockfree

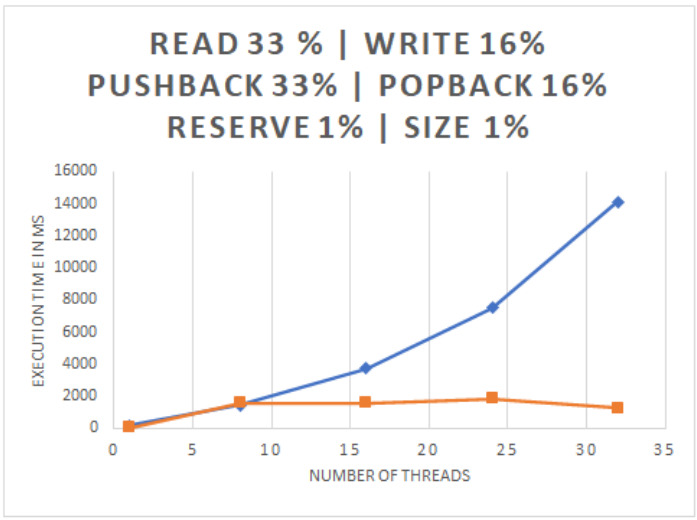

Figure 10: Performance tests, blue = locked, orange = lockfree

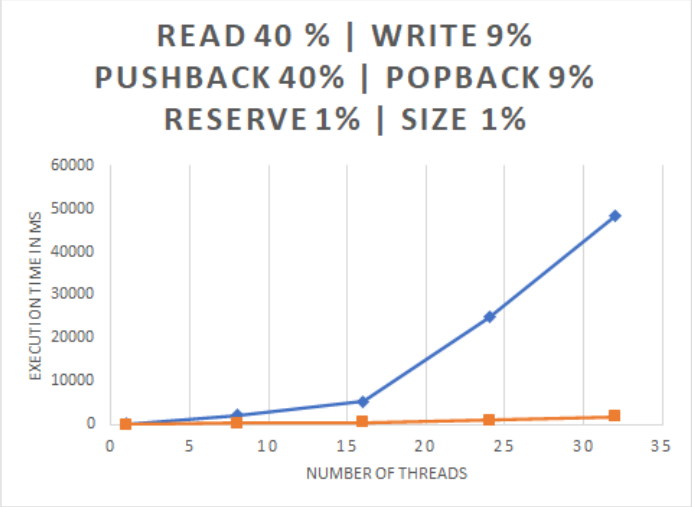

Figure 11: Performance tests, blue = locked, orange = lockfree

5 Conclusion

The LFTV is the first transactional lock free vector. With marked increases in performance from the state-of-the-art data structures currently in use, LFTV is a promising innovation in concurrent data structures. Our implementation shows the applicability of the transactional lock free vector in a java run-time environment while maintaining the progress and correctness conditions.

6 References

[1] MRLOCK: Deli Zhang, Brendan Lynch, Damian Dechev, Fast and Scalable Queue-Based Resource Allocation Lock on Shared-Memory Multiprocessors, In Proceedings of 17th International Conference on Principles of Distributed Systems (OPODIS 2013), Nice, France, December 2013.

[2] Use of descriptor objects: Deli Zhang, Pierre LaBorde, Lance Lebanoff, Damian Dechev, Lock-free Transactional Transformation, ACM Transactions on Parallel Computing (ACM TOPC), Vol. 5, No. 1, Article 6, June 2018.

[3] Wait-free data structures: Steven Feldman, Pierre LaBorde, Damian Dechev, Tervel: A Unification of Descriptor-based Techniques for Non-blocking Programming, In Proceedings of the 15th IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XV), Samos, Greece, July 2015.

[4] Performance testing of lock-free data structures: Ramin Izadpanah, Steven Feldman, Damian Dechev, A Methodology For Performance Analysis of Non-Blocking Algorithms Using Hardware and Software Metrics, In Proceedings of the 19th IEEE International Symposium on Object/component/service-oriented Real-time Distributed Computing (IEEE ISORC 2016), York, UK, May 2016.

[5] Correctness testing of concurrent data structures: Christina Peterson, Pierre LaBorde, Damian Dechev, CCSpec: A Correctness Condition Specification Tool, In the 27th IEEE/ACM International Conference on Program Comprehension (ICPC '19), Montreal, QC, Canada, May 2019.

[6] Progress verification of concurrent data structures: Christina Peterson, Victor Cook, Damian Dechev, Practical Progress Verification of Descriptor-Based Non-Blocking Data Structures, In Proceedings of the 27th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2019), Rennes, France,October 22-25, 2019.

[7] Kenneth Lamar, Christina Peterson, and Damian Dechev. 2020. Lock-free transactional vector. In Proceedings of the Eleventh International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '20). Association for Computing Machinery, New York, NY, USA, Article 6, 1–10.