Early Foundations of a Transactional Boosting Library for Scala and Java

A Masters Project Report
Authored by Marquita Ellis
Supervised by Maurice Herlihy
Conducted at Brown University Department of Computer Science
October 2012 - May 2013

I. Introduction

This project provides a foundation for a Scala transactional data structure library and further research into Transactional Boosting [1]. It also explores general issues in transactional data structure design. Transactional Boosting is a methodology proposed by Herlihy et al. in 2008, for transforming highly concurrent linearizable objects into highly concurrent transactional objects. The original work demonstrated the potential of such an approach on both a theoretical and a practical level. The motivation for creating a transactionally boosted data structure library in Scala is therefore manifold. One aspect of the motivation is inline with that of the original work. Software Transactional Memory provides composable synchronization, but efficient transactional data structures are not yet as commonly available as efficient concurrent data structures. As programs, transactional and otherwise, are composed of library calls, developing transactional libraries is essential for progress in the research and use of Software Transactional Memory. The original implementations and performance evaluation of Transactional Boosting were written in C++ and C, respectively. Implementing a transactionally boosted data structure library in Scala will make it available for widespread usage among Java and Scala programmers. Implementing it with transactional boosting will also take advantage of the significant effort that has gone into developing libraries such as the Java and Scala concurrency libraries. Furthermore, through developing the library, a more thorough evaluation of the transactional boosting technique can be conducted, and the performance of transactionally boosted data structures in these high level languages can be evaluated.

II. The Contributions of this Project

As part of this project, the following contributions were made. This list also serves as a rough outline for the rest of the report, following the necessary background sections.

- Three transactionally boosted maps were implemented in Scala with Scala Transactional Memory [3] and added to the transactional boosting library.

- A Scala transactionally boosted priority queue was also implemented and added to the library.

- A generic synthetic benchmark for maps was implemented and two versions of three STAMP benchmarks [4] were ported to Scala STM, for performance evaluation of the previously mentioned and future transactionally boosted data structures.

- A discussion of open issues, related work, and intriguing directions for future research is given at the end. Specific

recommendations and information for immediate extensions of the library are also given.

- Finally, comments for evaluating the strengths and weaknesses of the transactional boosting methodology are interleaved throughout.

III. Background

A. Software Transactional Memory

Software transactional memory is an alternative synchronization mechanism to traditional mutual exclusion primitives. Multistep concurrent operations in a transactional program are organized in *transactions*. Transactions are executed by the runtime system atomically. That is, they appear to take effect instantaneously and do not appear to be interleaved with the steps of other transactions. These properties are known as *atomicity* and *isolation*, respectively, in the database literature. A transaction may *commit* (take effect) or *abort* (appear to have never been executed). In the event of an abort, a transaction is *retried* unless an alternative execution path is specified.

A transactional memory system performs synchronization by tracking accesses to shared memory and detecting *conflicts*. A *conflict* exists between two transactions if one has written a location the other has read, or both have written to the same location. Conflict detection may be performed *eagerly*, detecting the conflict before the access, or *lazily*, detecting it afterward. If a conflict is detected, conflict resolution is performed. Conflicts can be resolved by aborting or (if conflict detection is eager) pausing one transaction until the other has been committed.

Shared memory modifications may also be *eager* or *lazy*, as long as atomicity and isolation guarantees are preserved. Modification is *eager* if shared memory is modified before the transaction commits. Modification is *lazy* if updates are buffered until it is known that the transaction can commit. If conflict detection is lazy and memory modification is eager, in the event of an abort, *recovery* actions are necessary to restore shared memory to its previous state.

B. Linearizability

Linearizability is a correctness condition for concurrent objects defined by Herlihy et al. in 1990 [9]. As the term is used frequently in this report, its definition is repeated here for the reader. An operation applied by a concurrent process is *linearizable* if it appears to take effect instantaneously at some point (a *linearization point*) between its invocation and response. An object is linearizable if each of its operations are linearizable.

C. The Transactional Boosting Methodology

Transactional Boosting is a methodology developed by Herlihy and Koskinen in 2008 [1]. The methodology describes correct implementation of a wrapper that provides transaction-level synchronization for a linearizable base object. The result is a transactional object that can synchronize at the same granularity as the base object in the absence of conflicts.

Transactional conflict detection and resolution in the wrapper are based on the object semantics. The methodology requires

specification of the object's abstract state, its methods and their affects on the state, and an inverse for every method. Conflict detection is based on commutativity of method calls. Two operations, *A* and *B*, *commute* if their invocations leave the object in the same state and they return the same values whether *A* completes first or *B* completes first. Commuting methods can therefore be executed concurrently without compromising semantic correctness. *Abstract locks* provide mutual exclusion for non-commutative method calls. The linearizable base object provides thread-level synchronization for commuting method calls.

Conflict resolution (or *recovery*) is based on method inverses. The *inverse* of a method "undoes" the affects of the method on the object's state. For each method invocation of the boosted object, an inverse for the method is recorded. If the transaction is rolled back, the log of inverses is executed.

D. The Scala Software Transactional Memory (STM) System

The transactional data structures and benchmarks described in the following sections were implemented with Scala Software Transactional Memory. Scala STM was developed by the Scala STM Expert Group at Stanford University [3]. It employs the Scala type system for conflict detection. Instead of tracking every memory access, it tracks accesses only to objects of type Ref, a type defined by Scala STM. Every shared memory object that the programmer wishes to protect for program correctness must be declared with this type. Scala STM conflict detection is eager.

The flexibility of the Scala STM model facilitated the implementation of the transactional boosting methodology, which obviates individual memory access synchronization for the boosted object. Furthermore, Scala closures and Scala STM's rollback and completion handlers provided a convenient means of registering method inverses for boosted data structure operations.

IV. Transactionally Boosted Data Structure Implementations

A. Overview

This section describes the implementation of three transactionally boosted maps and one transactionally boosted priority queue. Each description includes the abstract specification of the data structure, interesting implementation details, and comments introducing problems described more thoroughly in the future-research sections, sections VI and VII.

The Two Phase Locking (TPL) protocol was followed in the implementation of the following transactionally boosted data structures. Abstract locks are acquired at the beginning of a boosted operation and not released until the transaction either commits or aborts. They are released in the reverse of the order in which they were acquired. TPL is a technique adapted from the database literature for avoiding deadlock between transactions.

B. The Transactionally Boosted Maps

Three transactionally boosted maps were implemented in this project. The common abstract and commutativity specification of all three is given in Figure 1.

Abstract Map Specification

Method	Inverse
get(k)/v	noop()
contains(k)/-	noop()
remove(k)/v	put(k, v)
remove(k)/null	noop()
size()	noop()

Commutativity Specification

put(k,v) ⇔ put(l,v) , $k \neq l$

put(k,v) ⇔ put(k,w), $v \neq w$

get(k)/- ⇔ put(l,v) , $k \neq l$

get(k)/- ⇔ get(l)/- ⇔ contains(m)/- ⇔ size()

Figure 1. The Abstract Map Specification

Notice that map operations always commute if the keys on which they operate are different. (Also note, size operates on all the keys.) To therefore maximize potential concurrency, synchronization of operations in the implementation is based on keys.

More concretely, the abstract lock is implemented as an auxiliary map of keys to locks. An operation on key *k* first acquires the abstract lock for *k*, completes its operation on the underlying map, and registers its inverse before returning from the method call. Two examples in Scala with Scala STM are shown in Figures 1 and 2.

Each map was implemented with a different underlying base type. Each base type has its own consistency guarantees and performance characteristics. The base type of the first boosted map is a skip list map from the Java concurrency library. The expected average time cost for the contains, get, put, and remove operations is log(*n*). The size operation is O(*n*), since determining the number of elements requires traversal of the map. Furthermore, the Java documentation notes that iteration is *weakly consistent*, meaning the returned elements reflect the state of the map at some point since the creation of the iterator.

```
1.  def get(key: Key): Value = {
2.     AbstractLock lock key
3.     map get key
4.  }
```

Figure 2. Code for The Get Operation. In line 2, the lock is acquired for the input key. In line 3, the value for the input key in the underlying map is retrieved. There is no need to register an inverse, since the inverse of the get operation is a noop.

```
1. def remove(key : Key) : Value = {
2.      AbstractLock lock key
3.      if (map containsKey key) {
4.        var oldValue = map get key
5.        // on abort, restore old binding
6.        atomic { implicit txn =>
7.             Txn.afterRollback {
8.             status => map.put(key, oldValue)
9.           }
10.        }
11.       }
12.       map remove key
13.    }
```

Figure 2. Code for The Remove Operation. In line 2, the lock is acquired for the input key. In case the key is present in the map, lines 4 - 10 register the inverse of remove. If the key is not present, the inverse is a noop. Line 12 removes the key from the underlying map. The method returns the associated value.

The base type of the second boosted map is a hash map, also from the Java concurrency library. Its operations are non-blocking and resizing of the hash table is expected to be

"relatively slow", according to the documentation. The hash map iterator is also weakly consistent.

The third base type is a lock-free trie map from the Scala concurrent collection. The trie is an implementation of Prokopec et al's concurrent trie algorithm [5]. The implementation supports constant-time atomic lock-free snapshots, used to implement iteration and size among other operations.

For all three boosted maps, implementing the size operation efficiently in the wrapper presented an interesting challenge. Since size operates on every key in the data structure and does not commute with any operation that adds or removes elements from the data structure, it cannot be synchronized as simply as other operations which operate on only one key.

Several implementation strategies are possible. One is acquiring the lock for every key by traversing the auxiliary abstract lock map. However, this strategy is not only inefficient but also deadlock-prone. Other concurrent transactions may acquire a set of keys from the auxiliary map out-of-order for a corresponding set of single-key operations.

Another strategy is introducing a global lock, the acquisition of which would "freeze" the data structure. For the sake of illustration, suppose this were implemented as a single globally shared read-write lock. Single-key operations could acquire the lock in read mode before acquiring the lock of their key. The size operation could acquire the lock in write mode, preventing any changes to the data structure until size completes. This could be made more efficient by allowing operations that commute with size, such as get, to omit acquisition of the global lock. Both size and get operations could acquire individual abstract locks in read mode. There are several problems with this approach. Upgrading between read and write locks would need to be handled carefully. The approach complicates extending the data structure with other operations of equal complexity. It also adds overhead to single-key operations and the size operation itself would be delayed while concurrent readers drained-out.

The strategy the author settled on is as follows. A counter for the number of elements in the map since the last commit is maintained in the wrapper. Any operation that adds or removes elements from the map respectively increments or decrements this counter. Synchronization of transactions modifying this variable is provided via the Scala STM Ref type. Hence, the STM system guarantees that a modification (a put or remove operation) committed after the invocation of a size operation, but before the completion of its transaction, results in a subsequently resolved conflict. This approach is simple to implement correctly and does not compromise the extensibility or composability of the data structure or its operations. It is also generalizable to other similar operations. Its scalability can be easily improved by striping the counter as in [6].

B. A Transactionally Boosted Priority Queue

The base type of the transactionally boosted priority queue is from the Java concurrency library. While the base type supports both blocking and nonblocking operations, only nonblocking operations were employed in the

Abstract Priority Queue Specification

Method	Inverse
removeMin()/x	add(x)/-
min()/x	noop()
add(x)/-	addInverse(x)/-

Commutativity Specification

add(x)/- ⇔ add(y)/-

removeMin()/x ⇔ add(y)/- , $x \leq y$

min()/x ⇔ min()/x

Figure 4. The Priority Queue Specification given in [1].

boosting wrapper. The author's implementation otherwise follows the specification and design given in [1]. The specification is shown in Figure 4.

Synchronization of non-commutative method calls is accomplished via a single reentrant read-write lock. Operations on the minimum priority element (min and removeMin) acquire the lock in write mode. Other operations acquire the lock in read mode.

As in [1], addInverse is implemented using the Holder Pattern and logical deletion. Instead of inserting elements directly into the underlying priority queue, a holder instance, with a reference to the element and a boolean representing whether the element has been deleted, is inserted. When a transaction that has added an element is rolled back, the element is marked as deleted. The removeMin operation conducts physical deletion of elements by removing all minimum priority elements in a loop until one that is not marked as deleted is encountered.

V. Benchmark Implementations

A. Overview

This section describes four contributed benchmarks, and their relevant implementation details, for evaluating the performance of the previously described and future boosted data structures. One is a synthetic benchmark for measuring the performance of map operations in fine detail. The other three are implementations of three Stanford Transactional Applications for Multi-Processing benchmarks [4].

B. The Synthetic Map Benchmark

The synthetic map benchmark can be used to measure the performance of any map implementing MapTrait, a Scala trait also contributed to the library by the author. The trait consists of the methods in the abstract map specification (see Figure 4). The benchmark is therefore generic enough to measure the performance of concurrent non-transactional maps as well as transactionally boosted maps, which is convenient for performance comparisons between the two.

The synthetic map benchmark follows a similar strategy as found in the literature for concurrent data structure performance measurement. The benchmark runs a user-specified number of threads, which concurrently perform operations on the shared map. The type of shared map is given by the user. The operations are chosen at random from the input distribution of operations. These operations can be executed either as standalone operations or inside a larger transaction - another user option. The benchmark reports the throughput as number of operations per second. It also reports the total time and - redundantly, for convenience

	Tx Length	R/W Set	Tx Time	Contention
Labyrinth	Long	Large	High	High
Vacation	Medium	Medium	High	Low/Medium
KMeans	Short	Small	Low	Low

Figure 5. Workload Characterization of *Labyrinth, Vacation,* and *KMeans* as reported in [4].

- the specified map class name and number of threads.

C. The Stanford Transactional Applications for Multi-Processing (STAMP) Benchmarks

The three Stanford Transactional Applications for Multi-Processing (STAMP) benchmarks [4] ported to the transactional boosting library were Labyrinth, Vacation, and KMeans. STAMP is currently a standard benchmarking suite in the transactional memory research community. Each benchmark in the suite was designed to emulate a real world application implemented with transactions. Each of the STAMP benchmarks chosen for this project has a significantly different workload characterization from the other two, shown in Figure 5.

The original benchmarks from [4] were implemented in C. Since the boosted data structures in this library project were implemented with Scala STM and are intended for use in Scala and Java, the three chosen benchmarks were ported to Java and Scala STM. Thanks to the Irvine Research Compiler (IRC) group, the author was able to start with versions of the benchmarks in the IRC dialect of Java. The benchmarks were ported from this dialect to Java SE, using the original benchmarks as a reference to preserve program semantics.

Next, Scala STM was installed. This involved identifying all shared variables and refactoring their declarations and accesses as the Scala STM Ref type, among other more minor ports. As an optimization, the set of shared variables protected by Scala STM was minimized by identifying and refactoring any unnecessarily shared or unmodified variables as local or final variables, respectively. Implementing atomic sections in Java with Scala STM involved implementing each atomic section as a Java Runnable or Callable and passing it to the Scala STM library for transactional invocation.

Two versions were created for each benchmark, the first to provide a baseline performance measurement of Scala STM without transactional boosting, preserved for reuse across different machines and across the lifetime of the library. The other version was created for measuring the performance of the benchmark with transactionally boosted data structures. Implementing the latter version involved identifying the major data structures in each application and replacing them with a transactionally boosted version. Further details, specific to each benchmark, are described in the following subsections.

1. Labyrinth

The primary source of contention in Labyrinth is a three dimensional uniform grid. Threads iteratively retrieve two points, a start and end pair, from a work queue until the queue is empty. The threads attempt to compute a *route* in the grid between the start and end points. The *route* is a set of adjacent points that can be followed from the start point to the end point without intersecting any other previously found routes. It represents an unobstructed path in the grid. Success for all pairs of points in the work queue is not always possible. Once a valid route is found, it is stored in the shared grid.

The STAMP implementation of Labyrinth employs privatization to reduce conflicts during route calculation. Each thread copies the shared grid before calculating its route. After finding a route, it validates it against the shared grid. If validation fails, the thread starts over with a new copy of the grid.

In the pure Scala STM implementation, the shared grid is a three dimensional array of Refs. Threads create copies of the grid before beginning the main work of the transaction but to less of an advantage. It reduces the number of Ref access invocations, thereby minimizing the associated overhead. However, since Scala STM tracks any accesses to objects of the Ref type, not just those within an explicit outer transaction, a Scala STM implementation cannot take full advantage of the privitization technique.[1]

The benchmark implementation with transactional boosting replaces the shared grid with a transactionally boosted map of cell locations to values. Similarly to the pure Scala STM version, there is no way to non-transactionally access values in the map, either inside or outside a transaction,[2] hence attempting privatization helps little. It is noteworthy that, because locks need be acquired on every cell access and threads access cells out of order, using a transactionally boosted map in place of the grid is highly deadlock-prone. Deadlock resolution mechanisms are necessary in either the STM or the lock but add even more execution overhead. The current implementation relies on Scala STM's deadlock resolution.

2. *Vacation*

Vacation emulates a database travel reservation system. The primary sources of contention are four red black tree instances. Travel reservations are stored in and retrieved from the red black tree nodes. Tree modifications, which may or may not involve rebalancing the tree, are executed transactionally. Since threads executing transactions randomly select operations and randomly select the trees on which to execute the operation, contention is fairly spread out over the four trees.

---

[1] This was true at the time the last version of the benchmark was released. Since then, Scala STM has been extended to provide other access methods that may be used to, essentially, allow non-transactional access to Ref protected values. However, these new accessor specifications are such that they may be better purposed to implementing early-release for this benchmark. See getWith and relaxedGet, http://nbronson.github.io/scala-stm/api/0.7/index.html#scala.concurrent.stm.Ref

[2] Boosted data structures could be extended to support non-transactional retrieval. However, implementing such an extension while ensuring the same correctness guarantees is nontrivial.

In the implementation of Vacation with transactional boosting, the red black trees were replaced with transactionally boosted maps. In the pure Scala STM version, nodes of the red black tree are protected with Refs, and multi-operation modifications of the tree are protected by atomic notations. The version with transactional boosting outperforming the Scala STM version on this benchmark is expected. This is because the synchronization of the former is coarser-grained. It may be argued that a more fair comparison could be conducted between red black trees in each version. However, the transactional synchronization in the transactionally boosted version would still be at the granularity of operations. The outcome of the competition would therefore depend on the performance of the underlying concurrent red black tree.

3. KMeans

KMeans is an implementation of K-means clustering, a data mining application. Less than half the execution time is transactional. Since the work is partitioned across threads and data points are processed privately, the biggest source of contention in KMeans is a two dimensional array storing cluster centers. It is updated transactionally during each iteration of the clustering algorithm. The amount of contention is dependent on the value of *K*, with lower values resulting in higher contention.

In the implementation of KMeans with transactional boosting, the two dimensional array of cluster centers is represented with a transactionally boosted map. In the pure Scala STM version, it is a two dimensional array of Refs. Any two concurrent modifications of a cell result in a conflict. Since the granularity of data structure synchronization is relatively the same, the pure Scala STM version outperforming the version with transactional boosting on this benchmark is expected, due to its lower per-access overhead.

VI. Discussion: Directions for Future Research and Related Work

A. Pools

Sharing a queue among threads for retrieving work is a common pattern in multithreaded programs, as exemplified by the STAMP benchmarks. Often, however, the program semantics do not require elements of the work queue to be processed in any particular order. Ensuring the ordering guarantees requires costly synchronization, and in this case, is unnecessary. It therefore appears that concurrent queues are used for their availability rather than their suitability in this context. A good direction for future research would be developing efficient concurrent and transactional pool algorithms and implementations. Moreover, the observation that synchronization overhead may be reduced by allowing weaker consistency guarantees is relevant not only in the context of queues versus pools.

B. Iteration

In the course of this study, the weak consistency of the Java concurrent map and priority queue iterators highlighted an interesting challenge for transactional boosting. Implementing consistent iteration in the wrapper is non-trivial given weaker consistency guarantees of the base object. However, it is necessary for ensuring transactional guarantees. Consistency

requires that every element contained in the data structure is observed by the time of linearization of the transaction. This requires ensuring that items added or removed by transactions committed between the start and end of the concurrent iteration are "seen" by the iterating transaction. This may be done by preventing or detecting such modifications, but each of these approaches poses challenges and efficiency tradeoffs. Implementing iteration is similar to implementing size in that it touches (or may touch) every element. As discussed in the context of the map size operation implementation, acquiring abstract locks for every element while avoiding deadlock is inefficient at best. Efficient mechanisms for ensuring consistent iteration is an interesting problem for future research.

There has been related work on non-transactional concurrent and transactional iteration since Transactional Boosting was proposed. In 2012, A. Prokopec et al. published an algorithm for concurrent tries with non-blocking snapshots [5]. This algorithm is implemented in the Scala concurrent trie map, used as a base object for one of the transactionally boosted maps in this project. The algorithm relies on the specifics of the concurrent trie algorithm. It is therefore difficult (though perhaps not impossible) to generalize the approach to other data structures.

In 2013, E. Petrank et al. published a technique for adding a linearizable wait-free iterator to a wait-free or lock-free data structure [10]. The algorithm allows multiple iterators to collect a snapshot of the data structure. Implementing this technique (or other techniques for consistent iteration) in the transactionally boosted base object would simplify the problem of ensuring consistent iteration at the transaction level, while exploiting the performance of wait-free synchronization at the thread-level. It is also possible to adapt this algorithm to a transactional data structure, but the overhead introduced by the snapshot mechanisms in this context is assumed to be beyond an acceptable level.

In 2010, N. Bronson et al. published Transactional Predication [6]. Transactional Predication decouples testing element presence in the data structure from retrieving or modifying its value via per-element transactionally managed predicate variables. The problem of "missing" an element inserted concurrently with an iteration is addressed by introducing a shared counter associated with predicates. The counter is striped across multiple transactionally managed memory locations to reduce the occurrence of false conflicts. An iterator reads all the transactionally managed counters at the beginning of iteration. Later modification of a counter via concurrent insertion by a thread results in a conflict. See [6] for further details. Because this data structure implementation approach relies on the existing conflict detection mechanisms of the software transactional memory system, it composes cleanly therewith. Transactional Predication is an alternative transactional data structure implementation approach.

C. Optimism versus Pessimism

Optimistic Transactional Boosting [8] was proposed by Hassan et al. in 2013. It modifies the Transactional Boosting methodology by performing synchronization *optimistically*. Operations are performed on the data structure initially without modifying

shared memory or acquiring locks, both are deferred to the commit phase of the transaction.

One general advantage this approach may have over the original Transactional Boosting proposal is that pessimism can be "heavy handed", in that it requires transactions to obtain locks for which no conflict may occur. However, the original proposal was for data structures in a highly concurrent setting. In such a setting, a high degree of contention is expected and pessimism leads to higher productivity in the common case.

Since the relative performance benefit of each approach is workload dependent, further research should explore balancing the tradeoffs between optimism and pessimism more fully. Implementing a dynamic strategy for switching between the two when appropriate would be a particularly interesting research direction.

VII. Starting Points for Future Research: Specific Recommended Library Extensions

The library should be expanded with more data structure types. Two types that may be particularly interesting are trees (with varying properties) and pools. Trees that perform complex operations such as rebalancing may be good candidates for boosting, since performing transactional synchronization at the granularity of operations rather than locations may be beneficial for these complex operations. This seemed to be the case for the red black trees in the Vacation benchmark, as discussed earlier.

All of the STAMP benchmarks in the project used queues for distributing tasks among threads. As discussed earlier, this is believed to be a common pattern in multithreaded programs. Providing a transactional pool to replace such queues would be convenient for developers and increase potential concurrency in programs.

The author placed developer instructions and resource links for porting other benchmarks to Scala STM in the repository. Since this project, three other benchmarks have been added to the library, thanks to other students. Lixing Lian added a port of StmBench7 [7], another research standard transactional memory system benchmark. Fan Gao added ports of Intruder and Yada from the STAMP benchmark suite. The STAMP benchmarks that therefore remain to be ported to our library are Bayes, Genome, and SSCA2.

The main data structure in Bayes is a directed acyclic graph. Threads transactionally calculate dependencies between variables, represented by the nodes, and add new edges when dependencies are found. Due to its high degree of contention, Bayes may benefit from the pessimism of a transactionally boosted tree.

Genome emulates a two phase gene assembly application where each phase is executed transactionally. Although most of the program execution time is transactional, there is little contention since threads perform few modifications to the shared sets in each phase.

Scalable Synthetic Compact Application 2 (SSCA2) is a scientific application for efficient graph representation. Adjacency arrays and auxiliary arrays are used to construct a directed weighted multi-graph. Accesses to the adjacency arrays are protected by transactions. These transactions are short and have small read and write sets.

Furthermore, due to the large number of nodes in the graph, contention on adjacency lists is infrequent. It is not expected that transactional boosting will perform well on this benchmark, due to the low contention and simple operations.

In summary, while the already ported benchmarks span the range of workloads provided by STAMP, porting the remaining benchmarks may be useful for a thorough comparison against other current and future transactional data structures in the literature. Of these remaining benchmarks, Bayes may provide the most new information, as it tests the performance of a boosted tree in a high contention setting.

Acknowledgements

The author is deeply grateful to Maurice Herlihy for providing a partial implementation of the first transactionally boosted map and its initial abstract lock to start the project, not to mention his valuable feedback throughout.

References

[1] Maurice Herlihy , Eric Koskinen, Transactional boosting: a methodology for highly-concurrent transactional objects, Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, February 20-23, 2008, Salt Lake City, UT, USA.

[2] Maurice Herlihy , Nir Shavit, The Art of Multiprocessor Programming, Revised Reprint, Morgan Kaufmann Publishers Inc., San Francisco, CA, 2012.

[3] Scala Software Transactional Memory, http://nbronson.github.io/scala-stm/index.html

[4] C.C. Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: Stanford transactional applications for multi-processing, in: IISWC'08: Proc. IEEE International Symposium on Workload Characterization, 2008.

[5] Aleksandar Prokopec , Nathan Grasso Bronson , Phil Bagwell , Martin Odersky, Concurrent tries with efficient non-blocking snapshots, Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, February 25-29, 2012, New Orleans, Louisiana, USA.

[6] Nathan G. Bronson , Jared Casper , Hassan Chafi , Kunle Olukotun, Transactional predication: high-performance concurrent sets and maps for STM, Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing, July 25-28, 2010, Zurich, Switzerland.

[7] Rachid Guerraoui , Michal Kapalka , Jan Vitek, STMBench7: a benchmark for software transactional memory, ACM SIGOPS Operating Systems Review, v.41 n.3, June 2007

[8] Ahmed Hassan, Roberto Palmieri, Binoy Ravindran, Optimistic transactional boosting, Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming, 2014, New York, NY, USA.

[9] Maurice P. Herlihy , Jeannette M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Transactions on Programming Languages and Systems (TOPLAS), v.12 n.3, p.463-492, July 1990

[10] Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In DISC, pages 224--238, 2013.