An Implementation of The Scalable and Correct Time-Stamped Stack

Henry Jetmundsen, Patrick Thompson, Matthew Villegas, and Justin Zabel

College of Engineering and Computer Science

University of Central Florida

Orlando, United States

hjetmund@knights.ucf.edu, patrick.cs@knights.ucf.edu, matthew.villegas@knights.ucf.edu, jzabel04@knights.ucf.edu

Abstract—Many concurrent data structures impose real time ordering over their elements. This is needed only if the insertions modifying the data structure ran sequentially. A new approach using time stamps was proposed to avoid unneeded ordering [1]. Our implementation is based on that time-stamped (TS) stack. Concurrent insertions can be left unordered and then ordered as necessary at removal. Because of this weak ordering, using linearizability to establish correctness is not possible. The original paper presents a new approach to proving correctness for the TS stack. This proof technique is a new, generic proof for correctness on concurrent data structures [1]. In this paper, we highlight our general approach to re-implementing the Time-Stamped stack, discuss our modifications made to the stack, give an overview of our implementation of a stack using software transactional memory, and analyze comparative performance graphs based on our experimental data.

I. INTRODUCTION

The Elimination Backoff Stack held the title of one of the most state-of-the-art concurrent stacks, in terms of scalability and performance. Elimination refers to the idea that pairs of pushes and pops made at the same time will eliminate each other before actually entering the stack. A metaphorical description of the elimination backoff stack might be to think of the data structure as a normal stack but with a funnel on top where a pool of recently pushed data waits to enter the structure. If high contention occurs without a possible elimination, the thread "backs off" and waits, to mitigate contention [9]. In this paper, we attempt to implement and analyze a stack that claims to be a successor to the Elimination Backoff stack, the Time-Stamped Stack. To expand further on the earlier metaphor, this new timestamped stack would be like removing the funnel and eliminating a high-contention choke point. The Time-Stamped stack has an array of individual linked-list based stacks named spPools. Each spPool is associated with a particular thread. These pools are single-producer, multi-consumer pools, a thread can only push to its own spPool, but any thread can pop from any spPool. This entirely eliminates contention on insertion. The nodes are associated with a timestamp to keep track of when each node was inserted. This is used for maintaining order in the stack and allowing for elimination. The advantages of using this timestamping approach will be analyzed in further detail in other sections of the paper.

II. APPROACH

Our implementation of the TS stack from [1] follows the original implementation closely in terms of the SP pool and overall stack class. In general, the key idea of the TS stack is to associate timestamps with the insertion of nodes, and have these timestamps order their removal. Our implementation also confirms that that insertion of a node and the creation of its timestamp do not have to be done atomically. This is possible because according to linearizability any overlapping method calls can be reordered. Order must be maintained only when method calls occur sequentially. This is exploited in our implementation by allowing out of order timestamps for overlapping push operations. Sequential push methods are still timestamped ordered.

This research paper has very easy to follow pseudocode that lightly matches the syntax of C++. It doesn't assume some of the language's features such as the pointer, generics, tuple, atomics, and null value implementations. Because of this ambiguity it makes it easier to use a language of our choice, we chose C++ for our implementation. Our general strategy this stack was to first code a fully sequential version of it, test it, and then convert the appropriate variables to atomics and use CAS operations where necessary for a proper parallel implementation.

III. MODIFICATIONS FROM ORIGINAL IMPLEMENTATION

The original implementation provided by the researchers is very optimized and uses techniques in C++ none of us have any experience with. We offer an in-between solution, it's modeled closely off of the pseudocode provided in the paper. Problems like solving the ABA problem require intricate implementations with tricky bit masking, ensuring memory alignment and more. As part of the memory alignment requirements the paper's implementation used a customized thread local calloc. We decided to use a thread cached malloc created by Google for Chromium. This doesn't help to solve the ABA problem but it does increase our stack's performance over using libg++. During the implementation of our program we ran into an issue where our thread local lists were being corrupted during multiple concurrent removes. In this case the pseudocode was just light enough that it made it very hard to fully understand where we had made mistakes and

so we were introduced to the wonderful world of debugging multithreaded software. The single producer - multi-consumer list implemented in the data structure isn't exactly like anything in our book. In retrospect we should have re-implemented the lists using a method discussed in our book and worked from there to make the optimizations discussed in the paper. Our implementation uses an atomic counter to generate timestamps. We chose this implementation because we felt it was more portable than RDTSCP, specifically on older x86 systems.

IV. CORRECTNESS CONDITION

Due to the weakly ordered internal structure of the stack, it is difficult to use linearization points to establish the correctness of the stack. Since concurrent pop operations are not ordered later pop operations occur, finding a sequential ordering internally is not possible. The original paper [1] provides a new stack theorem that proves linearizability with respect stack semantics without needing a total linearization order. The proof for this theorem is beyond the scope of our paper, but it is available in the paper for the original implementation.

V. PROGRESS GUARANTEE

Our implementation of the TS Stack provides two functions, push and pop, with push being wait free and pop being lock free. Due to each thread having its own local stack to manage alone, the threads are able to push onto their stack without worrying about conflicting with any other threads pushing since they will also be pushing to their own local stack. When a push and pop happen on the same stack we are still able to guarantee wait freedom for the push call since there is no need for total linearization order as shown in the new proof of correctness proposed in the paper.

The pop operation maintains lock freedom but not wait freedom. The success of a pop call relies on the success of a call to the remove function. If the remove function succeeds then this means that there is progress and if remove fails then this means that there was progress in another thread. There are two cases in which the pop call can fail. The first happens when the pop call sees that all spPools are empty but then later during a final check finds that one of the spPools is no longer empty. The other case is when pop finds an element to remove, however when it goes to remove the element it can no longer find the element meaning that another pop call made it there first. A failed pop call will result in it starting over and trying again. This means that pop is not wait free, however it is lock free because we can guarantee that progress is being made in some thread even if the pop fails. The key synchronization technique used to guarantee this progress is the use of atomic compare and exchange operations for the top of the spPool when removing nodes from the spPool in the remove function. This guarantees that all other threads see that action as the top will have change, and the action occurs in a single, instantaneous and atomic step.

VI. ADVANTAGES AND DISADVANTAGES

The TS stack provides quite a few significant advantages over the Elimination Backoff stack while only having a couple disadvantages. Unlike the Elimination Backoff stack, the TS stack provides no contention on push operations due to every thread having its own list of elements. This allows for every thread to be able to insert elements without worrying about another thread changing the list or trying to insert at the same time. The TS stack also provides much less contention with the pop operation when compared to the Elimination Backoff stack. This is because while both stacks allow for elimination of operations on pop calls, the TS stack provides a wider range of calls to be cancelled resulting in a much higher success rate overall. Due to the new proof of correctness proposed with the TS stack, the need for total linearization order has also been removed which allows us to implement a wider variety of weakly ordered algorithms to improve performance on the stack. The only disadvantages we saw when implementing the TS stack is that the code and proof are more complex than the Elimination Backoff stack making it a little harder to read and implement. The TS stack might also not be as portable due to the need for platform specific time-stamping algorithms.

VII. CHALLENGES FACED

One of our biggest challenges faced was dealing with allocating memory for each spPool to use which we solved with the use of TCMalloc from the Google gperftools library. Another obstacle was implementing the addressing mechanism for our spPools array. In the original paper, the pseudocode implied using the ID of the thread as an index to that thread's spPool in the spPool array. In C++, `std::this_thread::get_id()` returns the operating system's assigned id for the thread which does not easily map well to a sequential index. To solve this, we paired a sequential index with the thread when spawning the thread, and modified the push() and pop() operations to also receive the id of the thread as a parameter in order to easily index the spPools array. Another issue faced was with properly implementing popping from the stack. Our pop function suffered from an improper reassignment of a next pointer in the spPools, causing there to be a cycle either arbitrarily within the spPool, or a cycle at the top of the spPool. We realized that there was a lot of implementation details left out of the pseudocode in the paper that were in the implementation provided by the authors in [1], but we avoided looking at their implementation for a long time because we wanted to come up with an original implementation based solely on their pseudocode.

VIII. SOFTWARE TRANSACTIONAL MEMORY

We chose to use the Rochester Software Transactional Memory package to implement a stack supporting transactions as a basis for comparing against our implementation of the TS Stack. This stack implementation uses a linked-list where nodes are pushed and popped onto the head or top of the linked-list. To modify this sequential stack into one that use the RSTM library, we replaced operations on

shared memory with various macros provided by the library. The shared memory accessed by all threads was the stack's top pointer, so any operations modifying that pointer were wrapped in macros. For example, reading from the top was done by calling *TM_READ(top)* and writing to it was done with *TM_WRITE(top, newVal)*. Our initial implementation of the STM version of the stack involved replacing all memory allocation and deallocation calls with *TM* calls. This was a more fine-grained approach where each call on shared memory was wrapped in its own *TM_BEGIN* and *TM_END* transaction blocks.

A. Modifications to the STM Stack

Our second version of the stack involved applying a more coarse-grained approach to wrapping operations in transaction blocks. This was done in an attempt to see if we could get any performance improvements from the stack. Anytime we want to run any of the *TM* calls, the operations must be wrapped within a *TM_BEGIN* and *TM_END* block. For our initial version, we had a fine-grained approach to these blocks, wrapping individual calls to push and pop in *TM_BEGIN* and *TM_END* transaction blocks. To improve performance, we tried wrapping our whole looping structure that called push and pop in a *TM_BEGIN* and *TM_END* block. Our rationale for this was that it would lower the amount of times that transaction blocks are called, meaning that there would be less overhead in constantly expanding *TM_BEGIN* and *TM_END* macros. We analyze the differences in performance between these two versions in the Performance Analysis section of this paper.

B. Proof of Correctness for STM Stack Version 2

The basis for correctness for software transactional memory is linearizability. With Software Transactional Memory, a group of operations are made to appear to all happen in one atomic step, satisfying the need for an instantaneous operation demanded by linearizability. The two operations to analyze the correctness of are push and pop. With push, the operation linearizes with respect to a concurrent push or pop when the transaction it is a part of is committed. Pop operates in a similar manner, when RSTM commits a transaction that the operation is in, pop is linearized with respect to concurrent pushes and pops. Since the STM Stack is made up of linearizable components, the stack overall is linearizable.

IX. PERFORMANCE ANALYSIS

Performance for the TS Stack and the STM Stack was evaluated using a Google Compute Engine instance running Ubuntu 18.04 with 8 vCPUs. According to the Google Cloud Compute Engine FAQ, a vCPU is, "implemented as a single hardware hyper-thread on one of the available CPU platforms." For our testing we purchased 8 vCPUs with 2.3 GHZ Intel Xeon E5 v3 (Haswell) as the CPU platform.

A. Graph Details

Each line present in the performance graphs corresponds to a particular ratio of pushes to pops. The y axis represents the number of operations per second. The x axis represents the number of threads. Each ratio of pushes and pops in the legend is also paired with what type of stack it ran with. In the case of comparing STM stacks, the lines in the legend either say v1 or v2 for pairing it with a particular version of the STM stack.

Fig. 1. Performance comparison between the TS Stack and STM Stack v2

B. TS Stack Performance

Looking at the performance of the TS Stack, we can see significant improvements up to 8 threads, after which the performance stagnates and performs similarly up until our 32 thread maximum. This makes sense because the cloud instance we tested on had 8 cores so it's safe to say that performance likely would have continued to increase with the number of cores in the system. The stack has a low of 2 million operations per second with 1 thread and increased linearly up to about 6 million operations per second with 8 or more threads.

Our Implementation of the TS Stack managed to outperform the TS Atomic Stack from the original paper in all scenarios that the TS Atomic Stack was tested in. The TS Atomic Stack presented in the paper showed little improvement with an increase of threads and sometimes even a decrease in performance, usually hanging around 1 to 1.5 million operations per second. We performed better than Stutter Stack and in some cases the Interval Stack as well, however this is likely because we don't free any memory in the program. This lack of time spent freeing memory likely makes our program run fairly faster than the results seen in the original paper.

Fig. 2. Performance of gperftool's tcmalloc vs libg++ malloc

C. tcmalloc vs libg++ malloc

If one does not specify an implementation of malloc terrible things can happen. Will the current default implementation use coarse grain locks? Or even worse, an unsafe technique that might even crash? In class we've discussed how we should avoid using new for both performance and correctness reasons. Addressing these concerns are custom implementations of malloc such as tcmalloc, jemalloc, and more. The primary strategy behind these mallocs is thread local caching. Each thread has access to a fairly large pool of memory and a way of quickly securing more memory from a central heap. For small objects each thread rarely contends with another in the heap. Between our group members various computer / OS configurations, a fairly large variance could be seen in performance. Using MinGW on Windows, for example, a test might take about 3 seconds while the same test using a recent version of GCC would take under a second. By specifying an implementation of malloc we avoid issues that would otherwise cause massive headaches. After we had a mostly working version of the stack we decided to look into the low level implementation details and how the authors tackled memory management.

Fig. 3. Performance for both versions of our STM Stack with Transaction size of 1

Fig. 4. Performance for both versions of our STM Stack with Transaction size of 2

Fig. 5. Performance for both versions of our STM Stack with Transaction size of 10

D. TS Stack vs STM Stack Version 2

As we expected, our implementation of the TS Stack outperformed the STM Stack. As the graph shows, there is a steady increase in performance up to 8 threads, which reflects the fact that we were using 8 vCPUs on the Google Cloud Platform Instance. Unlike the STM Stack, the TS Stack did not suffer from a severe performance decrease when running on 4 threads. This might indicate that there is some odd behavior within the RSTM library. While the STM Stack decreased in performance as threads were added, the TS Stack improved when increasing the number of threads from 1 to 8. The TS Stack had stable performance when running with 16 and 32 threads while the STM Stack performance slowly decreased. One cause for this could be the high overhead of creating and committing transactions, especially as the thread count grows.

E. STM Stack Version 1 vs. STM Stack Version 2

We observed that the second version of the STM Stack performed better overall than version one. Version two performed better when there were uneven amounts of push operations. Our tests consisted of 70% percent push operations and 30% pop operations. Version one performed best when there were

an even amount of push and pop operations. Overall, the 30% push and 70% pop test performed the best. This is likely because there were a large amount of failed pop operations, or trying to pop from an empty stack. When a pop operation reads that the top of the stack is null, it immediately returns without having to execute any further transactional memory operations. This is the same for both versions of the stack which is why the 30% push versus 70% pop test performed best. For the 50% push versus 50% pop and the 70% push versus 30% pop, the results were mixed. Neither version of the stack performed better than the other. This shows that a more coarse grained implementation of transactional blocks does not seem to affect performance. The RSTM library may be able to recognize when there are overlapping transactions and be able to remove one of them to achieve optimal performance.

A significant observation from our testing is the poor performance across the board when running the stack on four threads. There is no obvious answer that we could find and it may stem from the hardware used in testing. Because we used a virtual machine we could not control what other programs the CPU was running. Also, we don't know exactly how the execution was distributed across physical CPUs. For example, our eight vCPUs might have been clustered on the same physical CPU, spread across multiple physical CPUs, or dynamically distributed for load balancing.

F. Transaction Size Differences

Changing the size of the transactions involves increasing the number of operations that can occur with each transaction. In RSTM, this is trivially achieved. This can be done with through a Config object provided by the library. The config object has a field that allows us to set the number of operations per transaction. During testing, we pass in our desired transaction size as a command line argument. Moving from a transaction size of 1 to a transaction size of 2 had a few different effects. When the STM Stack performed well, specifically with 1, 2 and 8 threads, the performance saw a slight increase, especially when the majority of the operations were pops. The worst performance was with 4 threads, and moving to a transaction size of 2 worsened that performance for the initial version of our stack. The second version actually improved with a transaction size of 2 with 4 threads. The biggest differences in performance were when running the stack with more pop operations than push operations.

X. RELATED WORKS

The original paper refers to Attiya et al.'s Laws of Order paper as inspiration for the Timestamped stack when they discovered this stack as a counter-example for the proof that linearizable stacks, queues, and dequeues use read-after-write and atomic-write-after-read operations for removal [2]. Hoffman, Shalev, and Shavit's Basquet Queue implementation is thought to be one of the first queue implementations to allow unordered enqueues relative to atomic operations [3]. The AFC queue by Gorelik and Hendler implements timestamps in single producer buffers similar to the timestamped stack

but, unlike it, a thread merges the producer buffers into a total order, which creates a blocking removal. Additionally the AFC queue generates timestamps before insertion [4]. The LCRQ and SP queues use atomic counters for indexing similar to how timestamps can be implemented as atomic counters, but those implementation match indexes exactly or falls back to a slow path whereas the timestamped stack is not concerned with maintaining order or unique counters, meaning the performance for the stack is only dependent on the amount of matching timestamps instead of a fast path or slow path [5] [6]. The proof that the stack is linearizable with respect to a sequential stack is inspired by Henzinger et al.'s queue theorem [7]. Using SPPools as partial data-structures are used in Haas et al's distributed queues [8]. This implementation uses elimination from the elimination-backoff stack but differs in that elimination is done by timestamp comparisons instead of a collision array [9].

XI. CONCLUSIONS

In conclusion, unfortunately we were not able to complete the implementation of the stack. We were also not able to meet our stretch goal of implementing the queue and deque structures talked about in [1]. The main issue we encountered was chasing a bug in the remove operation of the stack. We found that at some point, an SPPool linked list was getting corrupted. This corruption is caused by either never encountering the sentinel node or because the sentinel node is no longer pointing to itself. This causes an infinite loop while trying to find the youngest node to remove.

In hindsight, we should have written our implementation in a language other than C++, most likely Java. This would have allowed us to look at the code supplied by [1], and would also avoid some problems inherent to non garbage collected languages, like the ABA problem. A few days before the submission, we looked at the original implementation to try and get an idea of how to fix the bug in the remove method. This is when we noticed that there are many details in the implementation that are never talked about in [1] or the supplied pseudocode. If we had chose a different language for our implementation, we could have followed the code of [1]'s implementation.

REFERENCES

- [1] M. Dodds, A. Haas, C. M. Kirsch, "A Scalable, Correct Time-Stamped Stack" presented at the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Mumbai, India, Jan. 15-17, 2015
- [2] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. Michael, and M. Vechev. Laws of order: expensive syn-chronization in concurrent algorithms cannot be eliminated. In *POPL*, 2011.
- [3] M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. In *OPODIS*. Springer, 2007.
- [4] M. Gorelik and D. Hendler. Brief announcement: an asym-metric flat-combining based queue algorithm. In *PODC*, 2013.
- [5] Y. Afek and A. Morrison. Fast concurrent queues for x86processors. In *PPoPP*. ACM, 2013
- [6] T. Henzinger, H. Payer, and A. Sezgin. Replacing compe-tition with cooperation to achieve scalable lock-free FIFO queues. Technical Report IST-2013-124-v1+1, *IST Austria*, 2013.

- [7] T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR*, 2013.
- [8] A. Haas, T. Henzinger, C. Kirsch, M. Lippautz, H. Payer, A. Sezgin, and A. Sokolova. Distributed queues in shared memory multicore performance and scalability through quantitative relaxation. In *CF*. ACM, 2013.
- [9] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*. ACM, 2004.
- [10] D. Dechev. The ABA problem in multicore data structures with collaborating operations. In *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, Orlando, FL, 2011, pp. 158-167. doi: 10.4108/icst.collaboratecom.2011.247161