Concurrent Wait-Free Red Black Trees

David Ferguson and Jacob Spigle

Abstract— **Our re-implementation of Natarajan, Savoie, and Mittal's wait-free algorithm [1] seeks to not only imitate the concurrent data structure presented, but also transform it into a transactional data structure using the RSTM library. We are implementing a concurrently managed red-black tree using wait-free algorithms designed and presented in [1]. We will be writing these algorithms using C++ as our programming language. Concurrency during manipulation of a tree data structure is not plausible without additional (and creative) data structures because of the multiple instructions that rotations perform during the re-balancing of the tree. Using the techniques outlined in [1], we will have a concurrent algorithm that makes progress, is linearizable, and correct. Experiments run by the authors of this implementation prove that their solution provides "significantly better performance"[1] than other attempts that preceded it, including both attempts at concurrency and lock-based implementations.**

I. Introduction

This wait-free implementation of the Red Black tree data structure boasts search(), insert(), update(), and delete() functions, all executed utilizing single-word compare-and-swap instructions. The data structure's concurrent implementation employs the use of "windows", which are overlapping snapshots of the current state of the Red Black tree within the scope of the windows' root node. Windows are copied from a global window, then edited locally before being pushed back to the global view of the Red Black tree itself, and pushing a modified window into the windows' origin will result in a correct, linearizable solution. This is because the window itself can be atomically swapped, where rotations are done inside a modified window, and using a single-word compare-and-swap, are placed back into the node where the window originated. This solution also strives for optimal concurrency by introducing an array that holds pending instructions (using the *announce* array) and decides whether or not a thread will assist by checking for conflicts with it's own update operation (using a *gate* variable, given to each record in the tree). A modify operation also is tasked to help another operation, choosing which to help in a round robin manner, so that it may also help during a search operation to ensure that it may eventually terminate [1]. This is necessary because this implementation avoids copying windows unnecessarily and instead traverses to the next window's root node when such a transaction would occur. Because this skipping occurs during a search operation, it would be possible for the operation to be overtaken,, These additions to the traditional sequential Red Black tree allow for an efficient algorithm that has outperformed other attempts at this implementation of the concurrent wait-free Red Black tree data structure.

Related Work: A previous attempt at creating a concurrent Red-Black tree data structure was implemented by Kim *et. al.* [3]. It is built upon Ma's [4] work with specifically insertions upon a lock-free Red-Black tree data structure, and [3] extends [4] to include both modification and deletion techniques. There have been a few more recent attempts at this wait-free implementation of manipulating red-black trees concurrently. Notably, in 2014, there was a thesis written proposing that instead of a Top-Down approach to obtaining ownership of nodes within the tree, that working from Bottom-Up approach would "[allow] operations interested in completely disparate portions of the tree to execute entirely uninhibited"[2].

Contributions: In this paper we seek to present our re-implementation of the Concurrent Wait-Free Red-Black Tree data structure presented in [1]. We also wish to present our solution to transforming [1] into a Transactional data structure by using the RSTM C++ Library presented by Marathe *et. al.* [6]. To do this, we will also need to employ POSIX Threads (pthreads) in our C++ code.

II. Preliminaries

The Concurrent Wait-Free Red Black Tree algorithm that was presented by [1] is derived from a few other presented algorithms. [1] presents a conglomeration of these algorithms in their modified state that creates the presented concurrent data structure.

Tsay & Li's Wait-Free Framework for Concurrent Tree-Based Data Structures

Tsay & Li's [5] TL-Framework is a construction that can be applied to any tree-based data structure attempting concurrency, provided the hardware supports load-linked, store-conditional and check-valid synchronization primitives. [5] presents the ability to perform wait-free top-down operations by utilizing 'windows', which are described as "rooted subtree[s] of [a] tree structure, that is, a small, contiguous piece of the tree" [1]. Essentially, as a process progresses through a tree while attempting to perform its' given operation, the process will continuously create copies of each node when

[1]D. Ferguson is a student at Department of Electrical Engineering and Computer Science, University of Central Florida, Orlando, Florida, 32816-2450

[2]J. Spigle is a student at Department of Electrical Engineering and Computer Science, University of Central Florida, Orlando, Florida, 32816-2450

utilizing this framework, creating a 'window' accessible only by the local process. We call the current windows of the tree $\mathrm{W_G}$ for window global. $\mathrm{W_G}$ is the only window accessible from the root node of the tree. With a firm grasp on the concept of windows, we dive into the steps that must be taken to perform any operation when utilizing the TL-Framework as described in [1]:

1) Explore-Help-And-Copy:

III. CONCLUSIONS

Our presentation of both the concurrent and transactional data structure implementations deriving from [**?**] demonstrates the linearizibility of the resulting C++ program. Searching in the Red-Black tree using these algorithms will be significantly less expensive due to the lack of helping. Modify operations that are performed in separate windows are able to execute concurrently without loss of correctness at any point because of the single word compare and swap instructions used when swapping out windows during a window transaction. Our implementation of [**?**]'s presented Wait-Free Memory Reclamation allows for nodes that have been essentially removed from the data structure (passive nodes) and are no longer in danger of being referenced by an operation inside that window may be effectively freed or reallocated accordingly.

APPENDIX

Appendixes should appear before the acknowledgment.

REFERENCES

[1] A. Natarajan, L. Savoie, & N. Mittal 2013. 'Concurrent Wait-Free Red Black Trees'. The University of Texas at Dallas, Richardson, TX 75080, USA.

[2] V. Kubushyn 2014. 'Concurrent Localized Wait-Free Operations on a Red Black Tree'. University of Nevada, Las Vegas.

[3] J. H. Kim, H. Cameron, & P. Graham 2006. 'Lock-Free Red-Black Trees Using CAS'. *Concurrency and Computation: Practice and Experience*, 1-40.

[4] J. Ma, 2003. 'Lock-Free Insertions on Red-Black Trees. MSc thesis. University of Manitoba.

[5] J.J. Tsay & H.C. Li 1994. 'Lock-Free Concurrent Tree Structures for Multiprocessor Systems'. *Parallel and Distributed Systems, 1994*, 554-549. IEEE.

[6] V.J. Marathe, M.F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W.N. Scherer III & M.L. Scott. 'Lowering the Overhead of Nonblocking Software Transactional Memory'. Computer Science Department, University of Rochester.