Concurrent Wait-Free Red Black Trees

Group 16

David Ferguson, Jacob Spigle

March 20, 2018

Abstract. 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.

1 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. Each of these windows is a balanced 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* variable) 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). An modify operation may also help during a search operation to ensure that the search operation eventually terminates [1]. This is necessary because this implementation avoids copying windows unnecessarily, and instead traversing to the next root when such a transaction would occur. 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: 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].

2 Current State, Problems, and Planning

We do not foresee many changes between the given implementation in [1] and our own, because the psuedocode written in the document seems to be written with C++ in mind. However, our own implementation's class structure will diverge from their own, because we will be reverse engineering it from references in the algorithms provided.

Because we are developing in a multithreaded fashion in C++, we are using the POSIX Threads library (pthread.h) to create, join, and monitor our threads. For atomic operations such as Compare and Swap (CAS), and for utilization of atomic variables we will be using the C++ Standard Library (std::atomic). The usage of these libraries will be the tools we use to create the concurrent data structure.

As of right now, our task is to properly be able to handle the external record for a node's value, as it is not being stored inside the node itself for this implementation. We have encountered some issues simply with translation of these algorithms, figuring out the best way to bring them into the code without sacrificing correctness. We struggled a bit with deciding whether keeping the decision made by the authors of this implementation to "assume the tree is never empty and always contains at least one node"[1]. For the purpose of getting our solution up and running, we are current following the practices of the authors.

Our plan of action for completing this project is to have a working solution for the concurrent data structure by the end of March. Moving forward onto part two from April till the final due date. In the event that we are able to complete both parts earlier than expected, we hope to conduct further testing and experimentation, possibly further improving upon our solution or rectifying any overlooked sections where our implementation is unoptimized.

References

[1] Aravind Natarajan, Lee H. Savoie, and Neeraj Mittal. *Concurrent Wait-Free Red Black Trees.* The University of Texas at Dallas, Richardson, TX 75080, USA, 2013.

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