Relativisitc Red-Black Trees

Haerunnisa Dewindita, Josue Martinez
University of Central Florida

Spring 2018

1 Introduction

The Red-Black Tree (RBTree) was invented by Rudolf Bayer, a German Computer Scientist, and is a self-balancing binary search tree that maintains Big O time complexity *O(log n)* for search, insertion and deletion of a node. The nodes within this tree contain (key, value) pairs and are sorted based on their keys. At any given node, the left subtree will contain nodes with keys smaller than the current node, and similar with the right subtree. Each node also contains an extra color bit, signifying a red or black color, which aids in the balancing of the tree to maintain logarithmic runtimes.
The RBTree is used extensively in operating systems - the Completely Fair Scheduler within the Linux Kernel is implemented with Red-Black Rrees to by using the left most node of the tree.
In this paper, a concurrent and wait-free RBTree will be re-implemented based on the following paper by Philip W. Howard and Jonathon Walpole: Relativistic red-black trees. The authors have created a linearly scalable implementation of the RBTree by using a new methodology called relativistic programming.

2 Relativistic Programming

Relativistic programming is a concurrent methodology which focuses on the casual relationships of an operation rather than imposing a strict total order on execution, which linearizability defines. Linearizability is a correctness condition to reason about a concurrent history by matching it with an equivalent, legal sequential history. By confining operations from different threads to a definitive sequence, there leaves little room for reordering. It also imposes strict rules on the order of programs when it is not necessary. Even though linearizability is a correctness condition, it is not a required one - this is where relativistic programming comes into play. The central idea behind this new approach is that any operation will have its own frame of referene of a structure, and will proceed with its execution based on that snapshot. The timing of events can vary between threads, or observers, due to the cost of communication along with

the order of the observations. This is seen as problematic under the hood of linearizability, since ambiguity can lead to correctness issues. However, relativistic programming roots itself into preserving casual relationships and supporting a more relaxed order through this approach, and therefore deems the potential wrong timing of events as natural. It is also considered correct, so long as the casuality is preserved.
So what is a casual relationship? Casual relationships are the ordering of events in a specific operation in which its sequence is vital to the correct outcome. For example, when creating a new node with key K and value V, it is necessary that the memory allocation of the new node occur before the initialization of it with K and V. Furthermore, when updating a tree with a new node, there is a casual relationship between the existence of the new, correct node, with the linking of it to the tree. This new node will have to exist before it can be linked into the structure.
When there is no casuality between operations, they are free to be reordered. Since the maintenance of casuality is weighted more on the write operations, reads can proceed at any time and bear no effect on the structure. Reads will also never have to wait for other reads, or any other writes. Since statistically the read operations of any given data structure occurs more often than writes, this balance will work in favor of the performance of the RBTree.

2.1 Primitives

To implement this RBTree relativistically, there are a few primitives used to ensure the stability and correctness of the data:

wait-for-readers - This primitive waits until all current reads complete to ensure that those reads gather the correct data before a write could potential change it.
rp-free - This primitive schedules the freeing of memory whenever an update is to delete a component of the tree. This operation is done asynchronously so the calling thread is not required to wait.
rp-publish - This primitive is used by writers when a change is made and they would like for this update to be visible to future readers.
rp-read - This primitive dereferences pointers to nodes.

3 RBTree Implementation

This RBTree contain the following functions:

insert(key, value) - To insert a node into the RBTree, the new node is initially given a the color red and traverses the tree based on the key that is given to the node. Once a new node is placed we then check to insure that all the properties of the Red-black tree are maintained. If the properties are no longer maintained then we will see if we can restore all properties of the Red-black tree by swapping the colors of the parent nodes. If the does not restore the properties of the RBTree we then will perform a rotation depending on the structure of the RBTree at that position.

delete(key) - The process of deletion is dependent on the nodes state of being a left or a right child. If the node is a left child, the parent pointing to the node will then point to the right child of the deleted node then perform the operations to maintain the Red-black tree. The same process is followed for a right child but we will reference that nodes left child.

lookup(key) - To find a node we then search the tree based on the value you want to find and the current node. If the key is less than the current node we move on to the left child, if the key is greater we will move on to the right child.

first()/last() - Return the leftmost (lowest-keyed) and rightmost (highest-keyed) node, respectively.

next()/prev() - Returns the next/previous node in key-sorted order from the tree, respectively.

4 Techniques and Guarantees

4.1 Progress

This relativistic RBTree depends heavily on the RP primitives defined above to maintain progress. Since relativistic programming is tailored to data structures who have more read operations done on them than writes, it allows more flexibility in the ordering of operations and focuses on the casual relationships of operations as mentioned earlier. The normal linearizability questions as to a structure's progress could be, "What if during an update, a reader thread repetitively chimes in while the writing thread is performing its wait-for-readers()? This would mean the writing thread is endlessly waiting for readers to execute, hence making no progress." However, any operation done on this RBTree relies on the tree's state at the time point A in which the function was invoked, so the only readers some writing thread T would have to wait on were the readers that occurred before or on time point A. Some function's operations are done on the data structure's state at the time it was invoked, and therefore brings in the possibility that two threads see different versions of the data - hence the *relativistic* keyword in the methodology's name. Since a function call only has to wait for the threads that occurred before itself, the number of threads it has to wait for will be finite. The wait will conclude at some point, hence the structure is able to make progress.

4.2 Correctness

Linearizability is widely used to reason about a structure's correctness by trying to make an equivalent sequential history out of the concurrent history. However, this imposes a total order on the string of operations and function calls - this strict ordering doesn't allow a structure to scale well. Even though the relativistic approach is more lax than linearizability, the way we reason about the correctness of this relativistic RBTree depends on some properties:

Immutable nodes - once a node has been created for an *insert()* function, its key and value will not change.

Ordering - a node is inserted into the tree BST style based on it's key, so at any given node N, N's left subtree will contain nodes with keys strictly less than N's key, and similarly for N's right subtree.

Structure - any update methods will not contort the RBtree in such a way that it's ordering is damaged.

With these properties, we can reason about the correctness of the RBTree in respect to it's abilities:

Lookup - When scanning the tree to find a node with the provided key, the traversal will always read the correct key of the current node to compare due to the *immutable* property. On top of this, the traversal will always go down the correct branch due to the *ordering* property. Finally, since the *structure* of the tree is always maintained, there will never be a read operation in which a node below the current node is moved in such a way that the reading thread will not come across it.

Insert - Inserting into the RBTree depends on the *rp-publish* primitive. *Rp-publish* serves as a linearization point since the node exists in the tree once *rp-publish* is called. Insertion can leave the tree imbalanced - however, the structure, ordering, and immutability properties depends on the restructure operations.

Delete - Deletes can occur in two places: leaf nodes or internal nodes. When a leaf node is to be deleted from the tree, it is a simple pointer assignment in which the parent node's reference to the deleted node is invalidated, which is atomic in all hardware. For internal nodes, a deletion brings a long multiple copies of nodes and different pointer assignments. These node copies still adhere to the immutable property, and the pointer assignments are made in a way that the structure and ordering of the tree is maintained. Once some node B is to be deleted, a copy of its child node *C'* is used to take its place, following an *rp-publish.* Finally, *C'* takes C's (original child node) children followed by another rp-publish.

4.3 Synchronization

This RBTree was implemented for heavy read operations with light write operations. The synchronization techniques to be used will have the write operations bear most of the weight of delay. Locks are used for write operations and provide the synchronization needed between writers. This synchronization has no effect on readers since reads are done oblivious to writes. Since writes depend on the conclusion of current reads, any areas where data is to be read, they are bounded by start-read and end-read RP primitives.

5 Reflections and Evaluation

Implementing this relativistic RBTree was a **huge** challenge. This methodology was not introduced during the duration of this class, and its a concept that grounds itself in being different from linearizability. The original paper itself is also quite centered on defending the benefits and advantages of RP against linearizability that it fails to go in depth in the actual RP implementation of the RBTree. There are so many comparisons to other implementations against RP, their downfalls, and a generic blanket statement on the RP primitives that the actual algorithm for the insert/delete/lookup for the relativistic RBTree is glossed over with no tangible link to the primitives for us to code. Researching *relativistic programming* only brought us to the same writers of our assigned paper, hence same research materials and wording, or an RCU implementation (used in Linux OS and is a form of RP, but **not** the implementation described in the paper). Phillip, an author of the original paper, provides a Github link to the RCU implementation using the URCU library. However, his source code for this spans over 20 files with no documentation or comments, and the code doesn't even compile. Since the Github holds code that was used to **compare** itself to the RP-RBTree, the contents of it were only minimally useful.

Although relativistic programming is meant to be more relaxed in its implementation of a data structure, its a concept thats difficult to grasp and reason about. It is also quite new and minimally documented by different people, so research on this topic kept us circling around the same paper. Since the actual understanding of RP-RBTree was not fully achieved, we were not able to think up of any improvements on this structure without completely toppling it to linearizability.

6 Performance

Sadly, we were not able to accomplish the stage in our RP-RBTree to have it functioning for testing and acquiring performance results. This is due to the structure's difficulty and hard to grasp concepts in implementation. However, the author provides graphs of his performance results and it shows that the

P. W. HOWARD AND J. WALPOLE

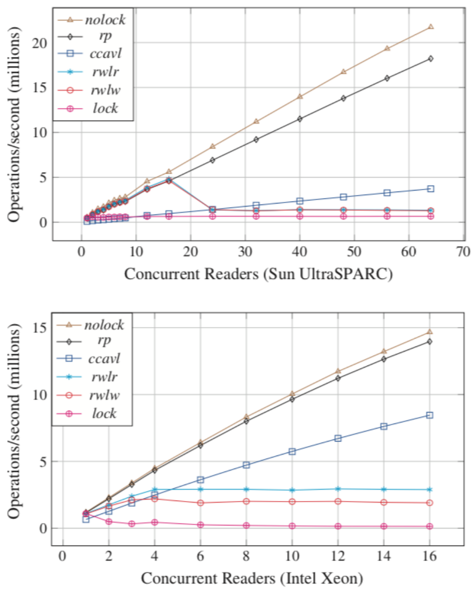

Figure 10. Read performance of 64K node RBtrees. In the UltraSparc data, the *rwlr* and *rwlw* lines are on top of each other because there is little difference between these for read-only workloads.

Figure 1: Original author's RP-RBTree performance.

structure had achieved what it was aimed to do: be highly scalable in concurrent reads. The current point of our project has landed itself in the hands of RCU rather than the RP implementation the author had proposed. The RCU functionality is supported by Userspace's RCU library, which is quite picky in the environment it runs in. It seems to only work in Linux environments with specific versions of *gcc* and *clang*. This library has also been lightly documented for its use, and research leads to the obscure errors other users encounter. We have matched up the RP-RBTree algorithm with its RCU equivalent in our code logically, however, since RCU is a *type* of relativistic programming, we were also not able to implement this fully due to issues with Userspace's RCU library compatibility and the default linux/rcu.h was also nowhere to be found in our development systems.