May 2015

Concurrent Lazy Splay Tree with Relativistic Programming Approach

Jaya Ram Sedai
University of Nevada, Las Vegas, jaya.sedai@gmail.com

CONCURRENT LAZY SPLAY TREE WITH RELATIVISTIC PROGRAMMING APPROACH

by

Jaya Ram Sedai

Bachelor of Computer Engineering
Tribhuvan University
Institute of Engineering, Pulchowk Campus
2009

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science Degree in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
May 2015

We recommend the thesis prepared under our supervision by

Jaya Ram Sedai

entitled

Concurrent Lazy Splay Tree with Relativistic Programming Approach

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

May 2015

Abstract

A splay tree is a self-adjusting binary search tree in which recently accessed elements are quick to access again. Splay operation causes the sequential bottleneck at the root of the tree in concurrent environment. The Lazy splaying is to rotate the tree at most one per access so that very frequently accessed item does full splaying. We present the RCU (Read-copy-update) based synchronization mechanism for splay tree operations which allows reads to occur concurrently with updates such as deletion and restructuring by splay rotation. This approach is generalized as relativistic programming. The relativistic programming is the programming technique for concurrent shared-memory architectures which tolerates different threads seeing events occurring in different orders, so that events are not necessarily globally ordered, but rather subject to constraints of per-thread ordering.

The main idea of the algorithm is that the update operations are carried out concurrently with traversals/reads. Each update is carried out for new reads to see the new state, while allowing pre-existing reads to proceed on the old state. Then the update is completed after all pre-existing reads have completed.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, *Dr. Ajoy K. Datta* for guiding, motivating and mentoring me to complete the thesis work.

I would also like to thank my committee members *Dr. Venkatesan Muthukumar*, *Dr. John Minor* and *Dr. Lawrence Larmore* for the support, I am grateful to have them in my thesis committee.

My sincere gratitude goes to my parents, sisters and brother in laws as they have always inspired me to work hard. I would like to thank my wife *Ajita* for supporting me to complete the thesis in time.

At last, I would like to thank all my friends, seniors and juniors for their love and support.

JAYA RAM SEDAI

University of Nevada, Las Vegas
May 2015

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

The hardware manufacturers are shifting towards multi-core computer design and the number of cores per computer is rising rapidly. More importantly, the scientific research is growing fast with the large data volume and requiring more computations. This demands the development of concurrent computation in the software system to utilize all the cores we have in hand so that the scientific computations can be performed fast enough on the ever-increasing big data volume. The main challenge of the multi-core computation is to perform task safely concurrent with optimum scalability. Many known techniques fails on scalability while focusing on safe concurrency [25].

Read-Copy-Update [23] is a novel synchronization mechanism that achieves scalability improvements by allowing reads to occur concurrently with updates. RCU supports concurrency between a updater and readers which is suitable for read-mostly data structures. Relativistic programming [14] is a generalization of the techniques developed for RCU. The terminology is introduced by Jonathan Walpole and his team at Portland State University developing the primitives and implementing the techniques in different data structures [24, 15]. Based on their use and applicability, RCU is described as "a way of waiting for things to finish" [19] and relativistic programming is described as "a way of ordering things" [14]. The relativistic programming primitives allow readers and writers to constrain the order in which the individual memory operations that compose the read and write are visible to each other. Unlike other synchronization mechanisms that try to impose a total order on operations to the same data, relativistic programming allows each reader to view writes in a different order. Each reader is allowed their own relative reference frame in which to view updates, and it is this property that gives relativistic programming its name.

The building block of the concurrent programs are the concurrent data structures that supports synchronization among the threads. Designing such data structure is far more difficult than the one for sequential software. Binary Search Tree is widely used in sequential context to implement lookup tables and dynamic sets. To access the items more quickly, self-adjusting version of BST called Splay tree is used which moves frequently used nodes near the root. In the concurrent context, moving the items to the root causes immediate sequential bottleneck. Different solutions have been proposed to resolve this issue. Yehuda Afek [1] suggests the technique to perform splay in a lazy manner to make it much more efficient using counters. We follow the counter based technique described by Afek and used by M. Regmee in his masters thesis dissertation [31] to perform the splay operation. Unlike the approach in [31] where the restructuring and deletion is postponed during the highly concurrent access of the tree and performed during the less contention, our approach is to carry out these operations concurrently with the reads using relativistic programming primitives as described in [14] to update operations involve multiple writes to multiple nodes.

1.2 Objective

Most of the concurrent data structure implementations use either blocking (coarse grained or fine grained locks) or non-blocking synchronization mechanisms and some uses transactional memory mechanism. Lock based technique is safe but does not scale as it does not allow multiple threads to access the common data at the same time and hard to manage effectively. Non-blocking mechanisms are complex to design. Transactional Memory solves the complexity problem inherent in most NBS techniques, but it still suffers from poor performance [14]. It provides no more concurrency than is theoretically available with fine grained locking.

Our objective in this thesis is to study the new synchronization mechanism namely Read-Copy-Update and the generalized version of it called relativistic programming, then apply their primitives in designing the concurrent self stabilizing Splay Tree in Lazy Splaying manner. Relativistic programming technique and its primitives are chosen to be applied for the low overhead readers and joint access parallelism between readers and writers. This helps splaying and restructuring of the tree in presence of the concurrent read operation in progress.

1.3 Related Work

The RCU-like access to a binary search tree [30] was described by H. T. Kung and Q. Lehman in September 1980 in their paper "Concurrent Maintenance of Binary Search Trees". After that several RCU-like mechanisms were studied and proposed by different researchers. It became popular when it was added in Linux kernel in October of 2002. Paul E. Mckenney is one of the inventors of RCU. Mckenney presented RCU in his Ph.D. dissertation [19] and maintains useful contents about RCU and RP in the web [25, 32, 21] and published research papers on RCU [20, 18]. Relativistic Programming research group in Portland State University is generalizing and standardizing the RCU programming model. They gave the term "Relativistic Programming" borrowing from Einsteins theory of relativity in which each observer is allowed to have their own frame of reference. Jonathan Walpole is the founder of the Relativistic Programming research group. In relativistic programming, each reader is allowed to have their own frame of reference with respect to the order of updates. Different papers [24, 15, 16] and Ph.D. dissertation [14, 35] are published related to relativistic programming model. Hagit Attiya and Maya Arbel from Israel Institute of Technology are examining the Read Copy Update (RCU) synchronization mechanism, investigating its implementations and use in concurrent data structures [2, 3]. There are some academic research on prototype implementation of RCU other than linux kernel namely OpenSolaris and HelenOS operating systems [28, 17]. The RCU and RP research focuses on read-mostly data structures and reclamation environment.

1.4 Outline

In chapter 1, we briefly discussed about the background of choosing the RCU based synchronization technique on a concurrent data structure as a thesis research topic. We then briefly discussed about the objective of the study and the related research work currently being done in the the field of RCU and RP synchronization techniques. Many papers related to RCU and RP implementations have been researched.

In chapter 2, we give brief overview of shared memory system. Then we discuss various synchronization mechanisms used in shared memory system to work properly in the shared data. We focus more on Read Copy Update and Relativistic programming techniques and how they are free from the issues that are in lock based and other synchronization mechanisms to work in the concurrent search tree implementation. We also describe about the concurrent data structure and the correctness criteria for the concurrent data structure.

In chapter 3, we will go over Binary Search Trees, Splay tree fundamentals and splaying operations in details.

In chapter 4, We will discuss about the RCU fundamentals, its properties and the way it works. We will go over the ordering and correctness criteria in Relativistic Programming and also present RP primitives that are used in our implementation. Then we will briefly describe about our implementation approach.

In chapter 5, we will present the proposed algorithm on binary search tree operations including lazy splaying using relativistic primitives discussed in previous chapter. We will present the pseudo code for the proposed algorithm. Before that we will briefly introduce the various supporting fields/variables and operation on the binary search tree used.

In chapter 6, we conclude the work and outline the remaining and future work.

Chapter 2

Background

2.1 Multi-Core Processors

As the transistor size is being decreased day by day, more transistors fit into the single chip but the clock speed cannot be increased because of the overheating problem. This is the reason that no matter how many transistors can fit in the single chip, the speed can not be increased enough to solve the current need. So the manufacturers are developing multicore architectures. Multiple cores communicate directly through shared hardware caches. Multiprocessor chips make computing more effective by exploiting parallelism: harnessing multiple processors to work on a single task [12]. Multi-core is a design in which a single physical processor contains the core logic of more than one processor [6]. Fig 2.1 [29] is a basic block diagram of a generic multi-core processor.

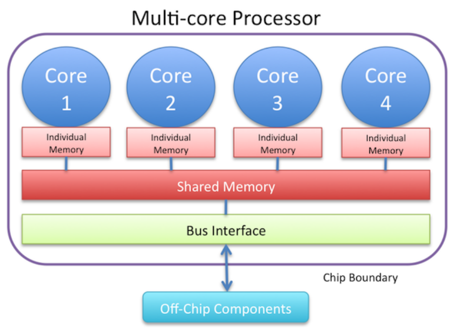

Figure 2.1: A basic block diagram of a generic multi-core processor

The availability of multicore system, in fact , provides the environment for parallel and/or concurrent computation, it does not speed up by itself. So, to utilize the parallel environment and perform in optimum speed, we have to rethink about the data structure, algorithm and the software that works exploiting the parallelism. But the parallel programs are very difficult to design, write, debug, and tune than sequential software. Concurrent computing is one of the outstanding challenges of modern Computer Science.

2.2 Concurrent Computing

Concurrent computing is a form of computing in which several computations are executing during overlapping time periods i.e, concurrently instead of sequentially (one completing before the next starts)[13]. In a concurrent system, a computation can make progress without waiting for all other computations to complete where more than one computation can make progress at the same time. A process is an instance of a program running in a computer. It is the basic entity that can be executed in a computer. A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. A thread is a component of a process. A computer program contains several processes and each process may have multiple threads. concurrency arises when multiple software threads running at different cores tries to access some shared resources. In single CPU system the concurrency is observed only logically. They use the time-sharing technique to share the same CPU within multiple threads.

2.3 Shared Memory Computation

A shared-memory computation [12] consists of multiple threads, each of which is a sequential program and they communicate by calling methods of objects that reside in a shared memory. Threads are asynchronous as they run at different speeds are and can halt for an unpredictable duration at any time. Thread delays are unpredictable, ranging from microseconds to even seconds. The shared memory can be centrally located or distributed into different computation nodes and connected via some form of network. UMA(uniform memory access) is a kind of shared memory architecture where the shared memory appears to be at equal distance from each processor and hence have the same response time. The other kind is NUMA(non-uniform memory access) architectures in which a shared memory might appear to be closer to one processor while another shared memory might appear closer to another processor and hence they may not have the same response time.

The concurrent execution in sharing resources may cause race condition and behave in unexpected manner. So the asynchronous concurrent processes have to be synchronized when they were accessing the shared data. A critical section is a piece of code that accesses a shared data that must not be concurrently accessed by more than one thread of execution. A critical section will usually terminate in fixed time, and a thread, task, or process will have to wait for a fixed time to enter it [8]. Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use.

2.4 Shared Memory Synchronization

In this section, we discuss about different synchronization mechanisms being used to work with shared data in the concurrent environment.

2.4.1 Lock Based Synchronization

Lock based synchronization protects critical sections allowing only one thread to enter a critical section at a time which in fact restricts concurrency and preserves data safety. Lock is used to achieve mutual exclusion. This is also known as blocking technique. As the locking restricts concurrency, this mechanism does not scale. More-ever , there are other disadvantages of lock based synchronization such as deadlock which may occur in a scenario where two processes running concurrently and locking their current resource waits for another to release the lock to get each-others current resources, that makes them wait forever. Priority Inversion is another common disadvantage of a lock based system, where a low-priority thread/process holding a common lock can prevent high-priority threads/processes from proceeding.

One of the locking techniques that our study in this thesis can be related is the reader-writer

locking. Reader writer lock allows concurrent access but for readers only. Multiple threads can read the data concurrently but an exclusive lock is needed for modifying data. When a writer is writing the data, readers will be blocked until the writer is finished writing.
There exists two kinds of locking depending on granularity of the lock. They are Coarse-grained and Fine-grained locking. Coarse-grained locking is used to protect the entire data structure by a single lock which may cause sequential bottleneck and progress delay [26]. Fine-grained locking can be used to obtain some degree of concurrency and hence scalability as multiple locks of small granularity is used to protect different partition of the data to allow concurrent operations to proceed in parallel as long as they do not access the same partition of the data. In some cases the fine grained locking may cause negative impact on performance. For example, if a linked list is partitioned such that each node has a separate lock, then the cost of acquiring a lock for each node can outweigh any gains through additional concurrency [14].

2.4.2 Lock Free Synchronization

As single thread that holds a lock in a blocking technique may prevent progress in all other threads, a non-blocking synchronization technique were developed [11] which guarantees that some process will complete an operation in a finite number of step. Lock Free implementation is a non-blocking which has guaranteed system-wide progress. Lock-freedom allows individual threads to starve but guarantees system-wide throughput. An algorithm is lock-free if it satisfies that when the program threads are run sufficiently long at least one of the threads makes progress.

2.4.3 Obstruction Free Synchronization

Obstruction free is the weakest non-blocking progress guarantee. An algorithm is obstruction free if at any point, a single thread executed in isolation (i.e., with all obstructing threads suspended) for a bounded number of steps will complete its operation. All lock free algorithms are obstruction free.

2.4.4 Wait Free Synchronization

In wait-free synchronization implementation, every operation has a bound on the number of steps the algorithm will take before the operation completes and any process that has invoked an operation eventually completes unless it fails itself irrespective of number of active processes or their state. All wait-free algorithms are lock-free.

Above non-blocking mechanisms are better for concurrency than Locking but such algorithms are complex to implement as they must accommodate any arbitrary interleaving from different threads; and preserve the liveness property in the presence of arbitrary delays in any other thread [14].

2.4.5 Transactional Memory

Transactional memory is an emerging programming model that was developed to provide a solution to the problem associated with blocking and non-blocking synchronization we discussed above. It is an approach extended from the approach used in database transactions. Transaction works on the principle that either all of the operations within a transaction complete or non of them completes. Two fundamental properties of TM implementations are disjoint access parallelism and the invisibility of read operations. Disjoint access parallelism ensures that operations on disconnected data do not interfere. The invisibility of read operations means that their implementation does not write to the memory reducing the memory contention. But [4] proves an inherent trade-off for implementations of transactional memories: they cannot be both disjoint-access parallel and have read-only transactions that are invisible and always terminate successfully. TM can be implemented in hardware (HTM), in software (STM), or both. Hardware transactional memory systems may comprise modifications in processors, cache and bus protocol to support transactions and Software transactional memory provides transactional memory semantics in a software runtime library or the programming language,[6] and requires minimal hardware support [34]. TM still suffers from poor performance and provides no more concurrency than is theoretically available with fine grained locking [14].

2.4.6 Read-Copy-Update

Read-Copy-Update (RCU) is synchronization mechanism which works as a framework for implementing concurrent algorithms. It allows extremely low overhead, wait-free reads that occur concurrently with updates. This makes RCU implementation more scalable for read-mostly algorithm implementations in the concurrent environment. RCU can be used to replace reader-writer locking. It has also been used in a number of other ways. As readers do not directly synchronize with RCU updaters RCU read paths extremely fast, and also permits RCU readers to accomplish useful work even when running concurrently with RCU updaters. RCU ensures that reads are coherent by maintaining multiple versions of objects and ensuring that they are not freed up until all pre-existing read-side critical sections complete [25]. RCU defines and uses efficient and scalable mechanisms for publishing and reading new versions of an object, and also for deferring the collection of old versions. These mechanisms distribute the work among read and update paths in such a way as to make read paths extremely fast. In some cases (non-preemptable kernels), RCU's read-side primitives have zero overhead. RCU is made up of three fundamental mechanisms namely Publish-Subscribe Mechanism (for insertion),Wait For Pre-Existing RCU Readers to Complete (for deletion) and maintain Multiple Versions of Recently Updated Objects (for readers). RCU API does not provide any means of synchronization among writers. We will revisit RCU in details in chapter 4.

2.4.7 Relativistic Programming

Relativistic programming is a programming technique for concurrent shared-memory architectures which is the generalization of the RCU model. It basically has two following properties [32].

1. It tolerates different threads seeing events occurring in different orders, so that events are not necessarily globally ordered, but rather subject to constraints of per-thread ordering, and in a few cases, partial-order constraints on global ordering.

2. It tolerates conflicts, for example, one thread can safely modify a memory location despite the fact that other threads might be concurrently reading that same memory location. Howard [14] states that the relativistic programming primitives allow readers and writers

to constrain the order in which the individual memory operations that compose the read and write are visible to each other. Unlike other synchronization mechanisms that try to impose a total order on operations to the same data, relativistic programming allows each reader to view writes in a different order. Each reader is allowed their own relative reference frame to view updates, and it is this property that gives relativistic programming its name. Relativistic programming constrains ordering in a pairwise manner between a writer and each reader. Each reader forms a different pair with the writer so the ordering constraints can be applied differently to each reader-writer pair. By not requiring a total order agreed on by all threads minimizing the ordering constraints and thus the overhead necessary to impose those ordering constraints. By minimizing the overhead, relativistic programming holds the promise for better performance and scalability. Read Copy Update and hence Relativistic programming focus on the read performance that is beneficial for read-mostly data structures. Research on relativistic programming aims to standardize the programming model of Read-Copy Update. Josh Triplett proposed a new memory ordering model in his PHD thesis in 2011 [35] for relativistic programming model called relativistic causal ordering, which combines the scalability of relativistic programming and Read-Copy Update with the simplicity of reader atomicity and automatic enforcement of causality. We will relativistic Relativistic Programming primitives and details along with Read Copy Update in chapter 4.

2.5 Correctness in Concurrent Implementations

As long as there is no guarantee that algorithm or implementation is correct to give desired output, it can not be accepted only based on the performance, code beauty or any other factors. But checking correctness of concurrent implementations is not that straight forward as compared to sequential implementations. In addition to defining the correct sequential behavior, correct interaction between threads also has to be defined. Threads being asynchronous in nature, adds the difficulties in defining the interaction between them.Concurrency is meant for improved performance but adds complexity in designing as well as verifying the correctness of the implementations. The correctness is the behavior of concurrent objects which gives the safety property. The other property of concurrent object is the liveness and is referred as progress behavior. It is easier to reason about concurrent objects if we can somehow map their concurrent executions to sequential ones, and limit our reasoning to these sequential executions [12].

In lock based implementations, correctness is defined in terms of data structure invariants and threads are allowed to violate the invariants while they hold the lock as they affect critical section sequentially so no other threads will access the data and see an invalid state. But to be correct implementation, all invariants have to be restored prior to releasing the lock.

In Non-blocking implementations, as data can be changed by one thread during another threads operation, invariants cannot be used in the same way as in lock based implementations. Following are the correctness criteria and principles for the concurrent implementations [12].

2.5.1 Quiescent Consistency

An object is quiescent if it has no pending method calls. Quiescent consistency can be defined by following two principles [12]

1. Method calls should appear to happen in a one-at-a-time,sequential order (Principle 3.3.1) [12].

2. Method calls separated by a period of quiescence should appear to take effect in their real-time order (Principle 3.3.2) [12].

It means any time an object becomes quiescent,then the execution so far is equivalent to some sequential execution of the completed calls. Formally, the object is called quiescently consistent, given a concurrent execution history of an object, if all operations appear to occur in some sequential order and non overlapping operations appear to occur in real-time order assuming each operation accesses a single object.

2.5.2 Sequential Consistency

The order in which a single thread issues method calls is called its program order. Sequential Consistency is defined by the Principle 3.3.1 stated above collectively with following principle: Method calls should appear to take effect in program order (Principle 3.4.1) [12]. Sequential consistency requires that method calls act as if they occurred in a sequential order consistent with program order. That is, in any concurrent execution, there is a way to order the method calls sequentially so that they are consistent with program order, and meet the objects sequential specification. sequential consistency is not compositional that means the result of composing sequentially consistent components is not itself necessarily sequentially consistent.

2.5.3 Linearizability

Linearizability is a correctness condition for concurrent objects. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain [13]. Linearizability can be defined by the following principle
Each method call should appear to take effect instantaneously at some moment between its invocation and response (Principle 3.5.1) [12].

This principle states that the real-time behavior of method calls must be preserved. Formally, the object is called linearizability given a concurrent execution history of an object,if the system is sequentially consistent and the sequential order is consistent with real time; i.e., all operations appear to happen between their invocation and response assuming each operation accesses a single object. Every linearizable execution is sequentially consistent, but not vice versa.
We will discuss about the correctness criteria in for relativistic programming in section 4.2.

Chapter 3

Literature Review

There are lots of research is being done to develop the scalable version of the concurrent data structures from the sequential version for the common data structures such as stack, queue, linked list, hash tables and skip lists and different varieties of binary search trees like splay tree.

Data structures that allows the efficient retrieval of an element from the set of elements are called search structures. Binary Search Tree is the mostly used search structure. We will describe about the binary search tree and the splay tree in the following sections.

3.1 Binary Search Tree

Binary search trees(BST) are a class of data structures used to implement lookup tables and dynamic sets. They store data items, known as keys and support three operations Find(key), Insert(key) and Delete(key) operations.

A binary search tree is a node-based binary tree data structure where each node has a comparable key and an associated value and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left sub-tree and smaller than the keys in all nodes in that node's right sub-tree [5]. Fig 3.1 is a BST of size 10 and depth 3 with root 17 and leaves 1, 5, 20 and 25. Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree, they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right subtrees. Each lookup/insertion/deletion takes O(n) worst case time complexity where n is the number of items stored in the tree.

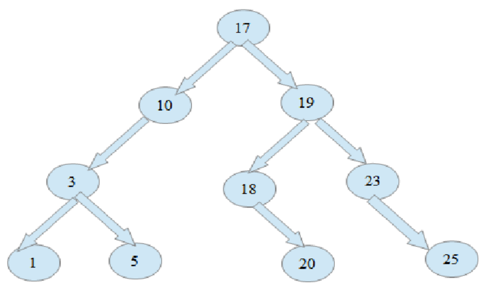

Figure 3.1: Binary Search Tree

Concurrent implementation of any search tree can be done using single global lock. Some level of concurrency can be obtained using reader-writer lock which allows readers execute concurrently with each other. Using fine grained locking with one lock per nodes can improve the performance. RCU based synchronization further allows reader and writer execute together which scales even more. But this technique does not specify the synchronization among the updaters.

There are various self adjusting binary search trees which restructures itself based on certain conditions to make the overall operations efficient. The restructuring is done using tree rotations that does not effect the BSTs property. AVL tree, Red Black Tree and Splay tree are the popular example of such trees.

3.2 Splay Tree

A splay tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log n) amortized time. For many sequences of non-random operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown [33]. The Splay tree is suitable for the applications where fraction of the entries are the targets of most of the find operations. The splay tree was invented by Daniel Dominic Sleator and Robert Endre Tarjan in 1985.

The find() operation in a splay tree begins like the find() operation in an ordinary binary search tree. then tree is traversed until the node is found or reach a node from which the next step leads to a null pointer. Then it performs the splaying on the node where the search ended node for find() operation irrespective of the result whether it found the searched key or not. The node is splayed up the tree through a sequence of rotations, so that the node will be on the root of the tree. This will bring the recently accessed entries near the root and improve the balance along the branch if the node being splayed lies deeply down an unbalanced branch of the tree prior to the splay operation. So if the node being splayed is deep, many nodes on the path to that node are also deep and by restructuring the tree, we make access to all of those nodes cheaper in the future.

3.3 Splaying

splaying is the operation on the node of interest through which a recently accessed nodes are kept near the root and the tree remains roughly balanced to achieve the desired amortized time bounds .

Each particular step depends on three factors:

Case I: Whether node of interest is the left or right child of its parent node.

Case II: Whether parent node is the root or not, and if not.

Case III: Whether parent is the left or right child of its parent.

Based on above cases, there are three types of splay steps. Additionally, there may be right or the left handed cases based on the position of the nodes being splayed and its parent(right or left). As they are symmetric, only one of the two cases is explained here.

Zig step: This is done when parent is the root. The tree is rotated on the edge between the node of interest and its parent. Fig 3.2 shows the tree before and after the zig step, not it does not break the BST property.

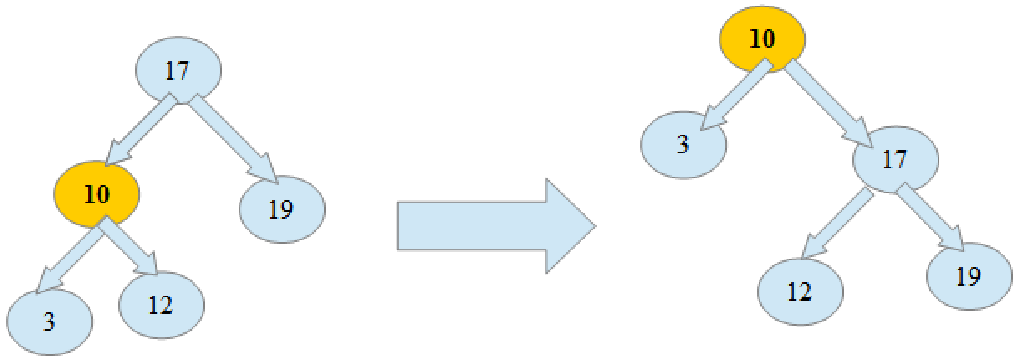

Figure 3.2: Zig step

Zig-zig step: This is done when parent of the node being splayed is not the root and the node and parent are either both right children or are both left children. The Fig 3.3 shows the case where both are the left children. The tree is rotated on the edge joining parent with its parent, then rotated on the edge joining the node with parent.

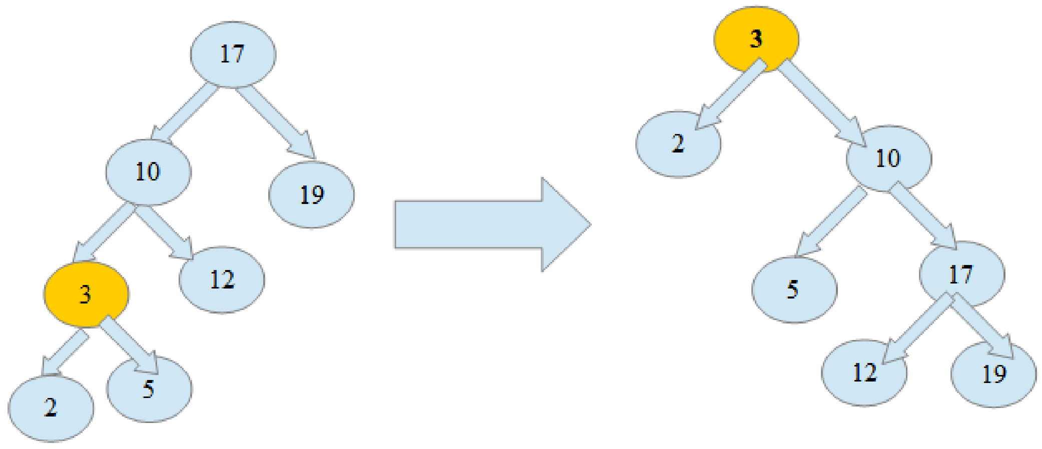

Figure 3.3: Zig-zig step

Zig-zag step: This step is done when parent of the node being splayed is not the root and the node is a right child and parent is a left child or vice versa. The tree is rotated on the edge between parent and the node to be splayed, and then rotated on the resulting edge between the node and garend parent. Fig 3.4 shows the Zig-zag operation.

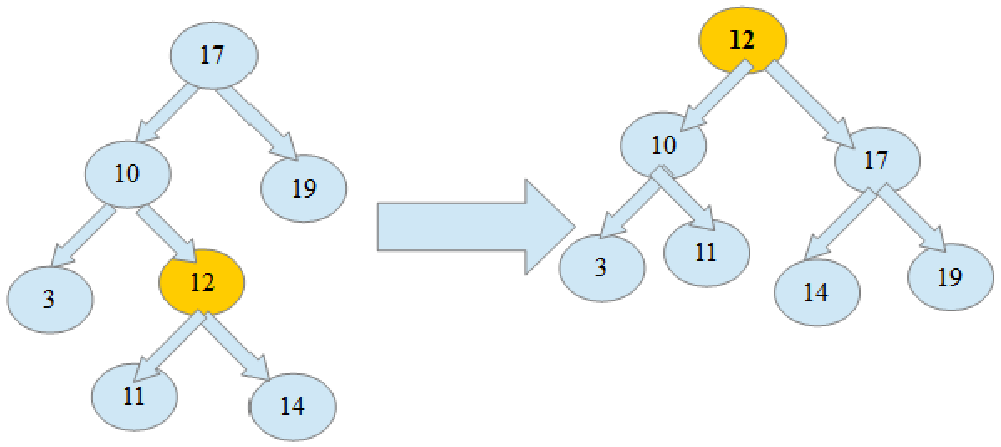

Figure 3.4: Zig-zag step

Bottom-up splaying requires two traversals, one is from root to the node to be splayed, and second is rotating back to the root. The *top-down* splaying can be used to perform splay operation in one traversal on the way down the access path. This approach uses three sets of nodes left, right and middle tree. Left tree and right tree contain all items of original tree known to be less than or greater than current item respectively and middle tree consists of the sub-tree rooted at the current node. These three sets are updated down the access path while keeping the splay operations in check. Top-down splaying uses only 2 cases: zig and zig-zig. zig-zag is reduced to a zig, and either a second zig, or a zig-zig.

Number of restructuring in splaying can be reduced with *semi-splaying* preserving the properties of splay tree. In Semi-splaying an element is splayed only partway towards the root. It reduces the depth of every node on the access path to at most about half of its previous value. Only one rotation is performed in the zig-zag case, but two steps are taken up the tree.

Another way to reduce restructuring is to do *full splaying*, but only in some of the access operations, when the access path is longer than a threshold.

Chapter 4

Methodology

In this chapter, we discuss about background idea and concept of our solution. We mainly discuss about synchronization mechanism used: Read-Copy-Update (RCU) and Relativistic Programming (RP) synchronization mechanisms, its fundamental properties, ordering and correctness criteria and primitives.

4.1 RCU Fundamental

We discussed about Read-Copy-Update as one of synchronization mechanisms in section 2.4.6, now we will describe it in details.

The essential property of RCU is that, readers can access the data structure when it is being updated. That means, RCU supports concurrency between a single updater and multiple readers. This property is achieved by following three fundamental mechanisms of RCU [25]:

1. *Publish-Subscribe Mechanism* :

This mechanism is used during concurrent insertion process. The *rcu_assign_pointer()* primitive is used to publish the new structure. The *rcu_dereference()* primitive is used by readers in read-side critical section as subscribing to a given value of the specified pointer, guaranteeing that subsequent dereference operations will see any initialization that occurred before the corresponding publish (rcu_assign_pointer()) operation. This process can be well described by the fig 4.1 [30] below. There are four-state states in insertion procedure. The red color indicates that concurrent readers may be accessing it and so updaters should take care in that situations. The green color indicates that the is inaccessible to the readers. The first state shows a global pointer named ”gptr” that is initially NULL, In second state memory is allocated for a new structure. This structure has indeterminate state but is inaccessible to readers. Because the structure is inaccessible to readers, the updater may carry out any desired operation without fear of disrupting concurrent readers. In the third state the new structure is initialized. In final state, this new structure is assigned a reference to gptr using rcu_assign_pointer(). In this state, the structure is accessible to readers. This assignment is atomic so concurrent readers will either see a NULL pointer or a valid pointer to the new structure, but not some mash-up of the two values.

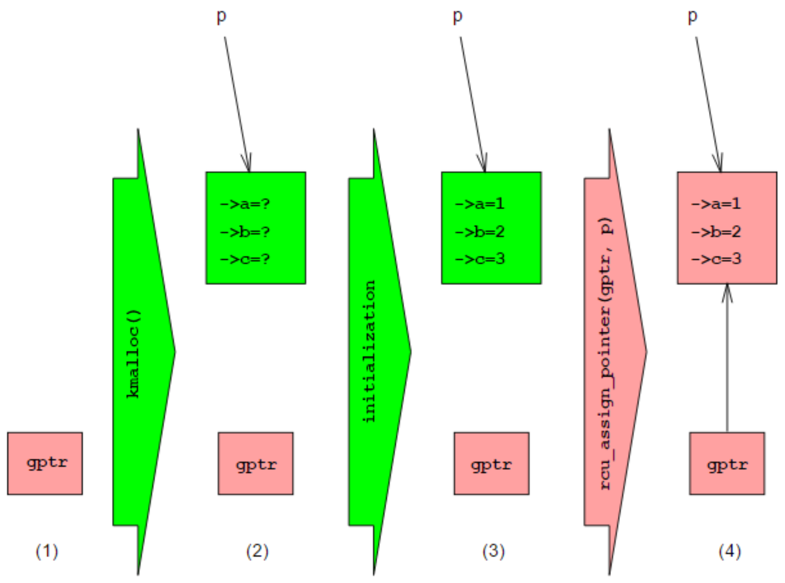

Figure 4.1: RCU insertion procedure

2. *Wait For Pre-Existing RCU Readers to Complete*:

This mechanism is used during concurrent delete process to wait for all pre-existing RCU reads (RCU reads are carried out between *rcu_read_lock* and *rcu_read_unlock* primitives called as *RCU read-side critical section*) to completely finish by using the *synchronize_rcu()* or *wait_for_readers* primitives. Considering this mechanism, McKenney describes RCU as a way of waiting for things to finish [12].

This RCU deletion process can be well described by the fig 4.2 [30] below.

The yellow color indicates that pre-existing readers might still have a reference to the data. The first state shows a linked list containing elements A, B, and C. In second state, element B is removed using *list_del_rcu()* primitive. The link from element B to C is left intact in order to allow readers currently referencing element B to traverse the remainder of the list. Readers accessing the link from element A will either obtain a reference to element B or element C, but either way, each reader will see a valid and correctly formatted linked list. Preexisting readers may still have a reference to element B, new readers have no way to obtain a reference. A *waitforreaders* operation transitions to the third state. The waitfor-readers need only wait for preexisting readers, but not new readers. Therefore, it is now safe for the updater to free element B, and so freed using free() in the final state.

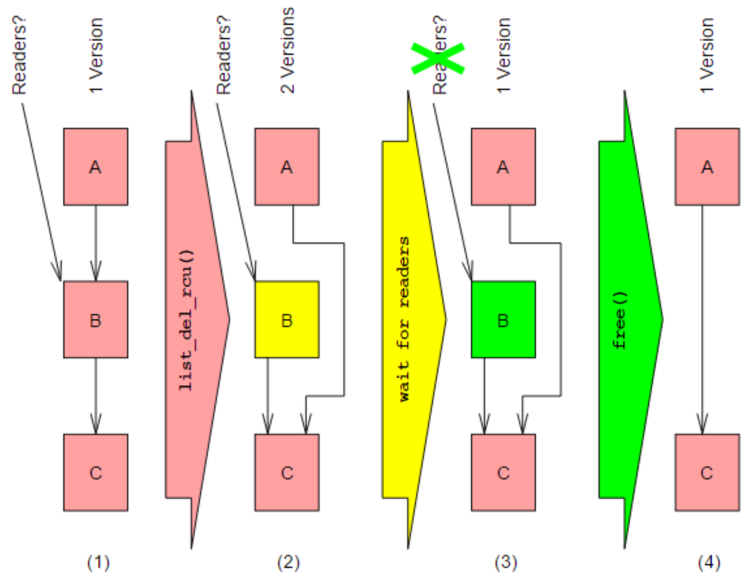

Figure 4.2: RCU delete procedure

Any statement that is not within an RCU read-side critical section is said to be in a quiescent state, and such statements are not permitted to hold references to RCU-protected data structures, nor is the wait-for-readers operation required to wait for threads in quiescent states. Any time period during which each thread resides at least once in a quiescent state is called a grace period. The wait-for-readers operation must wait for at least one grace period to elapse. Fig 4.3 [25] below depicts the way of waiting for pre-existing RCU read-side critical sections to completely finish.

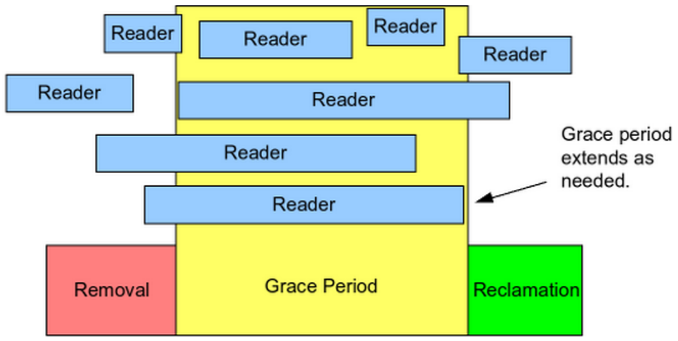

Figure 4.3: RCU way of waiting for pre-existing readers to complete

3. *Maintain Multiple Versions of Recently Updated Objects*:

This mechanism is used by readers to maintain multiple versions of data while deleting or replacing the data concurrently by the updaters. As readers do not synchronize directly with updaters, readers might be concurrently scanning while removing data. These concurrent readers might or might not see the newly removed element, depending on timing. However, readers that were delayed just after fetching a pointer to the newly removed element might see the old version of the data for quite some time after the removal. Therefore, we now have two versions of the data. Different versions (with or without B) that different readers may see can be seen in fig 4.2 [30] in state 2.

The most common use of RCU is to replace the reader-writer lock, but are used in a number of other ways too. In reader-writer locking, any reader that begins after the writer starts executing is guaranteed to see new values, and readers that attempt to start while the writer is spinning might or might not see new values, depending on the reader/writer preference of the rwlock implementation in question. In contrast, in RCU, any reader that begins after the updater completes is guaranteed to see new values, and readers that end after the updater begins might or might not see new values, depending on timing. [22] Fig 4.4 below shows how the RCU readers well see the change more quickly than reader-writer-locking readers.

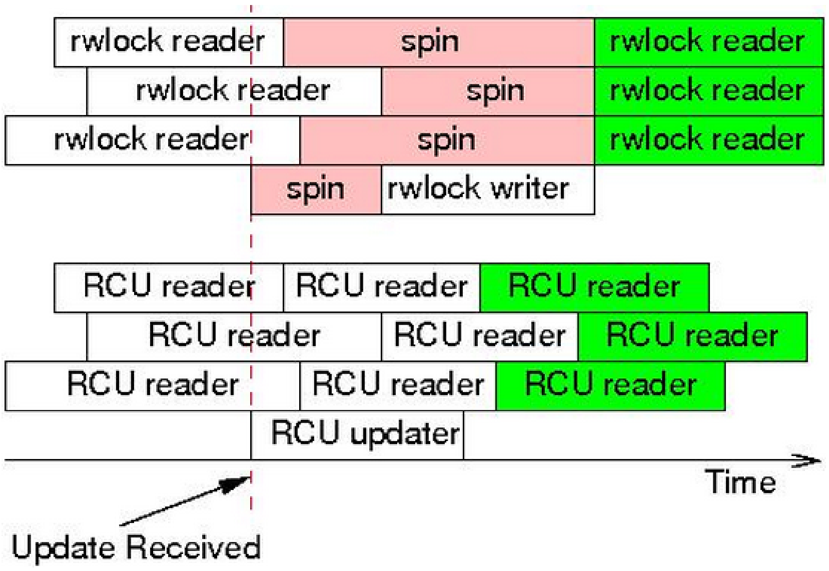

Figure 4.4: RCU and Reader-writer Lock

4.2 Relativistic Programming

In section 2.4.7, we discussed briefly about Relativistic Programming as a generalization of Read-Copy-Update synchronization mechanism, now we will discuss about the correctness and ordering criteria of Relativistic Programming, which is different than other Non-blocking synchronizations as it allows read and write to have joint access parallelism. Howard [14] presented a new analysis of the ordering requirements of relativistic programs and the primitives that support them along with the correctness criteria that can be applied to relativistic programs and Triplett [35] presented new memory model called *Relativistic Causal Ordering*,a memory model for scalable concurrent data structures in their Phd dissertations.

4.2.1 Ordering in Relativistic Programming

Concurrent implementations may be erroneous due to the results of reordering by compilers and hardware as they are capable of reordering the execution of a program. So concurrent programs must be written to prevent the erroneous results by reorderings.

Relativistic programming provides ordering primitives and rules for their placement which is used for the operations that need to be ordered. If these primitives work correctly and are used correctly, they will only allow correct orderings of execution. The effects of concurrency in Relativistic Programming are visible but the primitives and the methodology abstract away the details making it much easier to manage concurrency.
Following are the ordering relationships in concurrent implementations:

Program order is the order defined by the source code of the program. Erroneous executions by reordering should be resolved by inserting primitives to preserve program order.

Occurred Before is used to show the outcome of a race for two particular instances of threads. A *Occurred Before* B is written as A $\rightarrow$ B.

Required Before is used for the condition that two instances should occur in the desired order. A *Required Before* B is written as A $\Rightarrow$ B.

Following are the ordering primitives in Relativistic programming:
rp-publish, *rp-read*, *start-read*, *end-read*, and *wait-for-readers.* The *rp-publish* and *rp-read* primitives work together to implement the dependency ordering mechanism. Following ordering relationship is guaranteed by these primitives [14]:

$\forall$ readers and any *wait-for-readers*
if *start-read* $\rightarrow$ (the start of *wait-for-readers*)
then *end-read* $\Rightarrow$ (the end of *wait-for-readers*)

Relativistic programming does not guarantee a total order (neither totally chaotic) on all events that helps to have higher performance. There are some correct but non-linearizable solutions in RP, but NOT all operations supported and implemented by RP is non-linearizable.

4.2.2 Correctness in Relativistic Programming

As there is no isolation between reads and writes, the use of invariants enforced at the end of a write-side critical section is not adequate for correctness in relativistic prigramming. It allows non-linearizable solutions as each reader is allowed to have their own view of the order of updates. So Linearizability is also inadequate.
Howard [14] lists down following correctness criteria for relativistic implementations of ADTs:
1. Updates leave the ADT in an always-valid state meaning a read can access the data structure at any time without the need for synchronization.
2. Read operations on the ADT see the effects of all previous non-concurrent updates.
3. Read operations do not see any of the effects of later non-concurrent updates.
4. For a read that is concurrent with an update, the read sees either the state of the ADT prior to the update or after the update, but the read is not allowed to see any other state.

Chapter 5

Proposed Solution

In this chapter, we will propose the solution for the relativistic reader-writer synchronization for the splay tree operations using the RP primitives. The read-side primitives are wait-free and the implementation is focused on read-side performance and scalability so the updaters has to do extra work for synchronization so that concurrent readers always read valid data. RP primitives does not have the primitives to be used among writers, we have used fine grained locking while accessing the node for updating so that other updaters don't have access to the locked nodes.

We first specify the data structure, its attributes and operations used in the implementations followed by synchronization technique used. Then we will write algorithms to all the operations and helper functions.

5.1 Data Structures

The data structure we are using is the binary search tree(BST). BST implements the *find*, *insert* and *delete* operations. To move the frequently accessed items (by find, insert or delete operations) towards the root so that future access will be faster, additional splay operation has been implemented. The splay is performed in lazy manner without making the root a bottleneck.

We implement internal binary search tree with keys stored in all nodes. Searching for a key, either in a find operation or at the beginning of an update (insert,delete), is done in a wait-free manner inside an RCU read-side critical section. Each node contains the following fields: a *key*, *child[]* pointer with *dir* value *left* or *right* pointing left right child respectively, a *lock* field for fine grained locking used among updaters. The other fields used for the count based lazy splay operations [1, 31] are *selfCnt* which in an estimate on the total number of operations that has been performed on the node, *rightCnt* and *leftCnt* which are an estimate on the total number of operations that have been performed on items in the right and left sub-trees of the node. The *marked* field [2, 10] is used for validation purpose to see if the node was deleted concurrently by other thread before applying lock. The Tag field *tag[]* associated with *left* or *right* child is added in order to avoid an ABA problem [2]. A tag field is initialized to zero, and incremented every time the corresponding child field is set to $\perp$.
The BST tree used has the structure with the two dummy node -1 and ∞ is shown in Fig 5.1 similar to the structure tree used in [2]. The root of the tree always points to a node with key -1, this node has a right child with key ∞. All other nodes are in the left sub-tree of ∞.

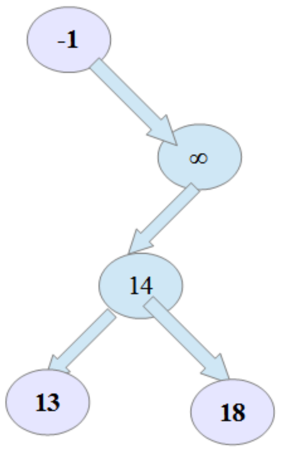

Figure 5.1: BST with two dummy nodes

5.2 Operations

Following are the operation including the helper functions that are implemented to support basic BST operations and Splay operation.

1. **get**

   The helper function *get* is used by find/insert/delete operations to access or the check if the node exists to avoid. This helps avoiding lock when searching for a node. It starts from the root and searches down the tree. It is performed inside a read-side critical section, wrapped with relativistic programming primitives rp_start_read and rp_end_read. Corresponding RCU primitives rcu_read_lock and rcu_read_unlock can also be used.

2. **validate**

   The validate function is used for validating the nodes that may have been changed by other overlapping updater after returning from get function but before locking the node. Parent-child relation is used by checking the child pointer of the parent. Validation that the node was not removed is used using marked field.

3. **incrementTag**

   This helper function receives a node and a direction to the child, if the child of node in the provided direction is $\perp$, it increments the tag associated with this direction. The purpose of this function is to avoid ABA problem.

4. **find**

   The find operation invokes get to find the key. If the key is found, it returns the node. If not found, it returns false. It calls *splayNode* function and updates the count values of the corresponding node after accessing it.

5. **insert**

   This function is used to insert a node as a leaf in a way that preserves the BST property. It requires little synchronization as it does not have to do with internal nodes. It invokes get, and returns false if get finds the key. If it does not find the key, a new node with the key is inserted as a leaf, added to the tree as the child of the parent of ⊥ returned by get. It validates before inserting the node taking help from validate function. The insert is performed in a relativistic programming way, using RP primitive. It updates the count values of the corresponding node and calls *splayNode* function after inserting it.

6. **delete**

   The delete function also invokes get helper function, and returns false if get does not find the key. There are two cases of deletion. First case is, if the node has at most one child, the node is removed by redirecting the child field of parent to point to its child. The second case is, it has two children and it is replaced with its successor in the tree. The successor of a node is the node with the smallest key among the nodes with keys larger than or equal to the node's key, which is stored in the leftmost node in the right sub-tree of the node to be deleted. These two cases are shown in the figure. The delete operation requires coordination with the concurrent get operation searching for the successor. So the replacement is done in relativistic way using RCU/RP primitive. Searches that start before wait_for_readers starts, find the successor in its previous location. Searches that start after wait_for_readers starts, find the new copy of the successor. Fig 5.2 and 5.3 shows the diffenrent case of delete operation performed.

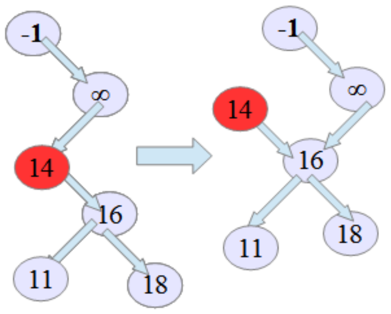

Figure 5.2: Deletition of node with one child

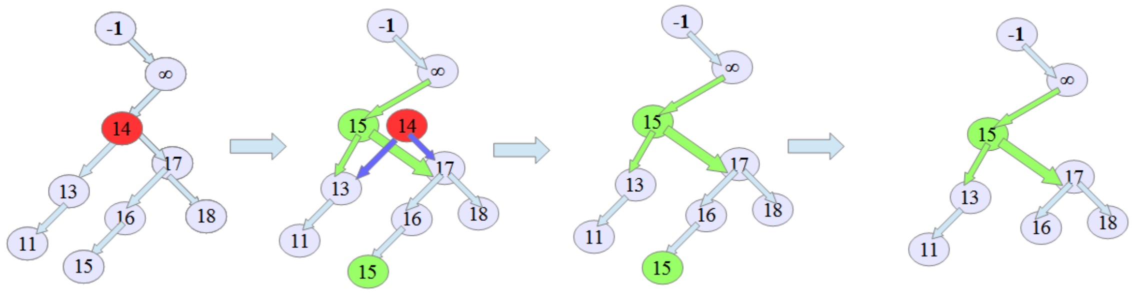

Figure 5.3: Deletition of node with two children

There might be two copies of the successor in the original and in the new locations. So, the BST here is following weak BST (WBST) property. The WBST property allows multiple nodes with the same key. If all nodes with the same key hold the same value, preserving the WBST property ensures that contains is correct, as it may return the value of some duplicate node, and ignore the others [2].

7. **splayNode**

   This function is used to perform lazy splay operation on the node that is accessed either by find operation or insert operation. The splay rotation depends on the access counter values of neighboring nodes. Based on these counter values, it performs either zig or zig-zag operation. Zig-zag is carried out if the total number of accesses to the node right sub-tree is higher than the total number of accesses to the parent and its right sub-tree. If zig-zag was not performed then zig is performed if the total number of accesses to the node and its left sub-tree is larger than the total number of accesses to the node-parent and its right sub-tree.

8. **zigRightRotation**

   This splay operation is basically the rotation of the tree and depending on certain pre-condition at each node based on access count of node and its left and right sub-trees. The operation is performed as shown in fig 5.4. All the operations are done relativistic way so that the concurrent readers will not see undesired result.

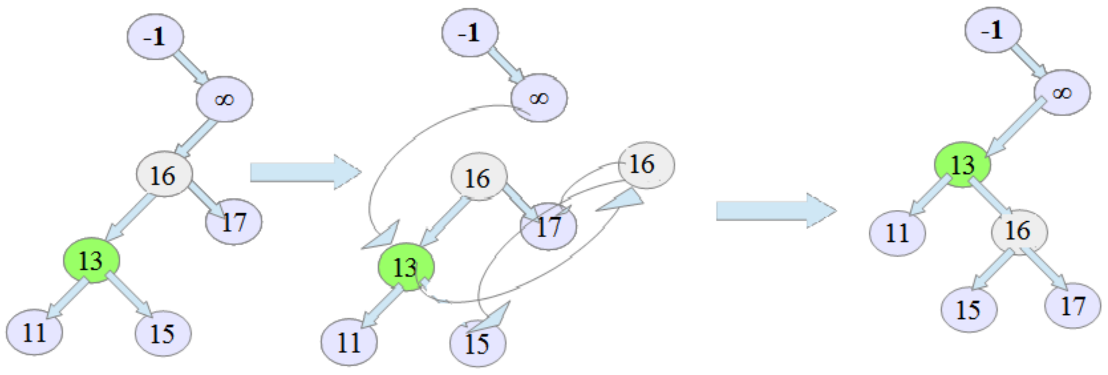

Figure 5.4: Zig rotation example

9. **zigLeftZagRightRotation**

   This splay operation is also the rotation of the tree and depending on certain pre-condition (defined in splaynode function) at each node based on access count of node and its left and right sub-trees. The operation is performed as shown in fig 5.5. All the operations are done relativistic way so that the concurrent readers will not see undesired result.

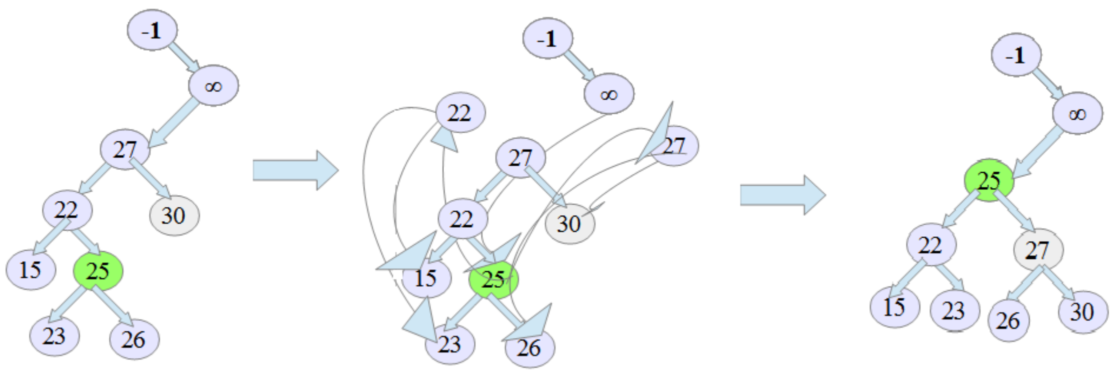

Figure 5.5: Zig-zag rotation example

The pseudo-codes for the BST and Splay operations and helper functions are presented below:

Algorithm 5.1 Helper function - get

```
function GET(key)
    $rp\_start\_read$                         ▷ read-side critical section begins,
                                  ▷ corresponding rcu primitive is rcu_read_lock
    $parent \leftarrow root$
    $curNode \leftarrow parent.child[right]$
    $curKey \leftarrow curNode.key$
    $dir \leftarrow right$
    while ($curNode \neq \perp$ and $curKey \neq key$) do
        $parent \leftarrow curNode$
        if $curKey < key$ then
            $dir \leftarrow right$
        else
            $dir \leftarrow left$
        end if
        $curNode \leftarrow parent.child[dir]$
    end while
    if $curNode \neq \perp$ then
        $curKey \leftarrow curNode.key$
    end if
    $tag \leftarrow parent.tag[dir]$
    $rp\_end\_read$                           ▷ read-side critical section begins,
                                ▷ corresponding rcu primitive is rcu_read_unlock
    return $(parent, tag, curNode, dir)$
end function
```

Algorithm 5.2 Helper function - validate

```
function VALIDATE(parent,tag,curNode,dir)
    if $parent.marked \lor parent.child[dir] \neq \perp$ then
        return $False$
    end if
    if $curNode \neq \perp$ then
        return $!curNode.marked$
    end if
    return $parent.tag[dir] = tag$
end function
```

Algorithm 5.3 Helper function - incrementTag

```
function INCREMENTTAG(node,dir)
    if $node.child[dir] == \perp$ then
        $node.tag[dir] += 1$
    end if
end function
```

Algorithm 5.4 Find operation

```
function FIND(key)
    $(parent, -, curNode, dir) \leftarrow get(key)$
    if $curNode = \perp$ then
        return $False$
    end if
    $SplayNode(parent, dir, curNode, curNode.child[left], curNode.child[right])$
    $curNode.selfCnt{+}{+}$
    return $curNode$
end function
```

Algorithm 5.5 Insert operation

```
function INSERT(key)
    loop                                  ▷ If the key doesnot exist but the validation fails, retry
    $(parent, tag, curNode, dir) \leftarrow get(key)$
    if $curNode \neq \perp$ then
        return $False$
    end if
    $lock(parent)$
    if $validate(parent, tag, , dir)$ then
        $newNode \leftarrow new(key, value, , )$          ▷ Create new leaf node, initialization
        $rp\_publish(parent.child[dir], newNode)$          ▷ relativistic insert
        $SplayNode(parent, dir, newNode, newNode.child[left], newNode.child[right])$
        $newNode.selfCnt{+}{+}$
        $unlock(parent)$          ▷ Release lock for concurrent updaters
        return $True$
    end if
    $unlock(parent)$          ▷ validation failed, release lock and retry
end function
```

Algorithm 5.6 Delete operation

```
function DELETE(key)
    loop                                    ▷ If the key exists but the validation fails, retry
    $(parent, tag, curNode, dir) \leftarrow get(key)$
    if $curNode == \perp$ then                                        ▷ key does not exist
        return $False$
    end if
    $lock(parent)$
    $lock(curNode)$
    if $validate(parent, -, curNode, dir)$ then
        if $curNode.child[left] == \perp \vee curNode.child[left] == \perp$ then
                                                        ▷ Case 1, curNode has single child
            $curNode.marked \leftarrow true$
            if $curNode.child[left] \neq \perp$ then
                $rp\_publish(parent.child[dir], curNode.child[left])$
            else
                $rp\_publish(parent.child[dir], curNode.child[right])$
            end if
            $incrementTag(parent, dir)$
            $unlock(parent)$
            $wait\_for\_readers()$
                ▷ Wait for all pre-existing readers in read-side critical section to complete
            $rp\_free(curNode)$                                    ▷ Relativistic Deletion
            return $True$
        else                                            ▷ Case 2, curNode has two children
            $parentSucc \leftarrow curNode$                            ▷ Searching the successor
            $succ \leftarrow curNode.child[right]$
            $next \leftarrow succ.child[left]$
            while $((next \neq \perp)$ do
                $parentSucc \leftarrow succ$
                $succ \leftarrow next$
                $next \leftarrow next.child[left]$
            end while
            if $curNode == parentSucc$ then
                $succDir \leftarrow right$
            else
                $succDir \leftarrow left$
                $lock(parentSucc)$
                      ▷ parentSucc is locked only if it is not the curNode (already locked)
            end if
            $lock(succ)$
```

Algorithm 5.6 Delete operation (continued)

```
            if $validate(parentSucc, -, succ, succDir) \wedge validate(succ, succ.tag[left], , left)$ then
                $succ\_copy \leftarrow new(succ.key, succ.value, curNode.child[left],$
                        $curr.child[right])$
        ▷ Create new node of by copying the successor with children pointing to children of curNode
                $curNode.marked \leftarrow true$
                $rp\_publish(parent.child[dir], succ\_copy)$                    ▷ Relativistic insertion
                $wait\_for\_readers()$
                $rp\_free(succ)$                                          ▷ Relativistic Deletion of succ
                if $parentSucc == curNode$ then
                    $rp\_publish(succ\_copy.child[right], succ.child[right])$
                    $incrementTag(succ\_copy, right)$
                else
                    $rp\_publish(parentSucc.child[left], succ.child[right])$
                    $incrementTag(parentSucc, left)$
                end if
                $unlock(parent)$
                $unlock(parentSucc)$
                $unlock(succ)$
                $wait\_for\_readers()$
                 ▷ Wait for all pre-existing readers in read-side critical section to complete
                $rp\_free(curNode)$                                            ▷ Relativistic Deletion
                return $True$
            end if
        end if
    end if
    $unlock(parent)$                                 ▷ Validation failed, release locks and retry
    $unlock(curNode)$
end function
```

Algorithm 5.7 Lazy splay operation

```
function SPLAYNODE(parent,dir, curNode,lChild,rChild)
    if $lChild \neq \perp$ then //                                                  ▷ Propagate Counter
        $curNode.leftCnt \leftarrow lChild.leftCnt + lChild.rightCnt + lChild.selfCnt$
    else
        $curNode.leftCnt \leftarrow 0$
    end if
    if $curNode.right \neq \perp$ then
        $curNode.rightCnt \leftarrow rChild.leftCnt + rChild.rightCnt + rChild.selfCnt$
    else
        $curNode.rightCnt \leftarrow 0$
    end if
▷ check for zigRight and zigLeftZagRight
    $nodePlusLeftCount \leftarrow lChild.selfCnt + lChild.leftCnt$
    $parentPlusRightCount \leftarrow curNode.selfCnt + curNode.rightCnt$
    $nodeRightCount \leftarrow lChild.rightCnt$
    if $nodeRightCount \geq parentPlusRightCount$ then                     ▷ zigzag condition
        zigLeftZagRightRotation($parent, curNode, dir$)
        $parent.leftCnt \leftarrow lChild.right.rightCnt$
        $lChild.rightCnt \leftarrow lChild.right.leftCnt$
        $lChild.child[right].rightCnt \leftarrow lChild.child[right].rightCnt +$
$parentPlusRightCount$
        $lChild.child[right].leftCnt \leftarrow lChild.right.leftCnt + nodePlusLeftCount$
    else if $nodePlustLeftCount > parentPlusRightCount$ then                ▷ zig condition
        $zigRightRotation(parent, curNode, dir)$
        $parent.leftCnt \leftarrow lChild.rightCnt$
        $lChild.rightCnt \leftarrow lChild.rightCnt + parentPlusRightCount$
    end if

end function
```

Algorithm 5.8 Zig right rotataion operation

```
function zigRightRotation(parent, curNode, dir)
    $leftNode \leftarrow CurNode.child[left]$
    if $!validate(parent, -, CurNode, dir) \vee CurNode == \perp \vee leftNode == \perp$ then
        return $false$
    end if
    lock($parent$)
    lock($CurNode$)
    lock($leftNode$)
    $CurNode\_copy \leftarrow new(CurNode.key, CurNode.value, leftNode.child[right],$
                        $CurNode.child[right])$
                                                        ▷ copy $CurNode$ to create $CurNode\_copy$
    $rp\_publish(leftNode.child[right], CurNode\_copy)$
    if $dir == left$ then
                                                        ▷ If CurNode is left child of a parent.
        $rp\_publish(parent.child[left], leftNode)$
    else
                                                        ▷ If node CurNode is right child of a parent.
        $rp\_publish( parent.child[right], leftNode)$
    end if
    $rp\_free( CurNode)$
    unlock($leftNode$)
    unlock($parent$)
    return $true$
end function
```

Algorithm 5.9 Zig left zag right rotataion operation

```
function zigLeftZagRightRotation( parent,curNode,dir )
    $x \leftarrow CurNode.child[left]$
    $r \leftarrow x.child[right]$
    if $!validate(parent, -, CurNode, dir) \vee CurNode == \perp \vee x == \perp \vee r == \perp$ then
        return $false$
    end if
    lock($parent$)
    lock($curNode$)
    lock($x$)
    lock($r$)                                   $\triangleright$ create new node $x\_copy$ to replicate $x$
    $x\_copy \leftarrow new(x.key, x.value, x.child[left], r.child[left])$
    $rp\_publish(r.child[left], x\_copy)$
    $wait\_for\_readers$
    $free(x)$                   $\triangleright$ create new node $curNode\_copy'$ to replicate $curNode$
    $curNode\_copy \leftarrow new(curNode.key, curNode.value, r.child[right],$
                    $curNode.child[right])$
    if $dir == left$ then                          $\triangleright$ If curNode is left child of a parent.
        $rp\_publish(parent.child[left], r)$
    else                                        $\triangleright$ If node parent is right child of a grand.
        $rp\_publish(parent.child[right], r)$
    end if
    $wait\_for\_readers$
    $free(curNode)$
    unlock($r$)
    unlock($parent$)
    return $true$
end function
```

Chapter 6

Conclusion and Future Work

In this thesis, we have studied the Relativistic Programming approach, generalization of RCU synchronization technique applicable for the read-mostly data structure. We borrowed the idea and concept from various implementations and proposed the relativistic solution for BST operations including lazy splay. RP primitives are used to synchronize between readers and updaters, that makes the read and update can go concurrently even when updaters making change in the same portion of the data readers are accessing. This makes read paths extremely fast and readers do not directly synchronize with RCU updaters, synchronization and data consistency is maintained by updater. The fine grained locking has been used to synchronize among the updaters. To resolve the possible ABA problem and data changes during overlapping updaters, validation on the data being modified has been applied. We conclude that, these techniques we have used in our implementation makes the algorithm scalable and efficient.

Though, the proposed solution can be claimed to have better performance because of the synchronization used, the actual implementation of the proposed solution and comparison with other implementation is the interesting future work. The RCU has been used extensively in Linux kernel and academic research and implementations are going on, yet it has not been used in concurrent data structure implementations that extensively. The Relativistic Programming is also new in the field. So using these technique in other the read-mostly data structure is another future work.

Bibliography

[1] Y. Afek, B. Korenfeld, A. Morrison, "Concurrent Search Tree by Lazy Splaying."

[2] M. Arbel and H. Attiya, "Concurrent updates with RCU: Search tree as an example", *PODC*, pp. 196-205, 2014.

[3] M. Arbel and A. Morrison, "Predicate RCU: An RCU for Scalable Concurrent Updates", *PPoPP* , pp. 21-30, 2015.

[4] H. Attiya, E. Hillel, and A. Milani, "Inherent limitations on disjoint access parallel implementations of transactional memory.", *SPAA*, pages 69-78, 2009.

[5] Wikipedia contributors, Binary Search Tree. Wikipedia, The Free Encyclopedia,*http://en.wikipedia.org/wiki/Binary_search_tree.*

[6] A. Binstock, "Multi-Core Processor Architecture Explained", *https://software.intel.com/en-us/articles/multi-core-processor-architecture-explained*, 2008.

[7] Wikipedia contributors, Concurrent Computing. Wikipedia, The Free Encyclopedia, *http://en.wikipedia.org/wiki/Concurrent_computing.*

[8] Wikipedia contributors, Critical Section. Wikipedia, The Free Encyclopedia, *http://en.wikipedia.org/wiki/Critical_section.*

[9] M. Desnoyers, M. R. Dagenais, P. E. McKenney, A. Stern, and J. Walpole, "User-level implementations of read-copy update", *TPDS*, pp: 375-382, 2012.

[10] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit, "A Lazy Concurrent List-Based Set Algorithm.", *OPODIS*, pp. 3-16, 2006

[11] M. Herlihy, “A Methodology for Implementing Highly Concurrent Data Objects”, *ACM Trans. Program. Lang. Syst.*, pp. 745-770, 1993.

[12] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint.* Elsevier, 2012.

[13] M. Herlihy and J. Wing. Linearizability: “A correctness condition for concurrent objects.” *ACM Transactions on Programming Languages and Systems*, 12(3), 1990.

[14] P. W. Howard, *Extending Relativistic Programming to Multiple Writers.* PhD thesis, Portland State University, 2011.

[15] P. W. Howard. and J. Walpole, “Relativistic red-black trees”, *Technical Report 10-06*, Portland State University, Computer Science Department, 2010.

[16] P. W. Howard. and J. Walpole, “A Case for Relativistic Programming”, *RACES*, pp. 33-38 , 2012.

[17] A. Hraska, *Read-Copy-Update for HelenOS.* Masters thesis, Charles University in Prague, 2013.

[18] P. E. McKenney, “A Case for Relativistic Programming”, 2014.

[19] P. E. McKenney, *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System kernels.* PhD thesis, Oregon State University, 2004.

[20] P. E. McKenney, “RCU vs. Locking Performance on Different CPUs”, 2004.

[21] P. McKenney, “Recent read-mostly research”,*http://lwn.net/Articles/619355/*, 2014.

[22] P. E. McKenney, “What is RCU? Part 2: Usage?”, *http://lwn.net/Articles/263130/*, 2007.

[23] P. E. McKenney and J. D Slingwine, “Read-copy update: Using execution history to solve concurrency problems.”, *PDCS*, pp. 509-518, 1998

[24] J. Triplett, P. E. McKenney, and J. Walpole, “Resizable,scalable, concurrent hash tables”, *USENIX Annual Technical Conference*, 2011.

[25] P. E. McKenney and J. Walpole, "What is RCU, Fundamentally?",*http://lwn.net/Articles/262464/*, 2007.

[26] M. Moir and N. Shavit, "Concurrent data structures", *Handbook of Data Structures and Applications*,pp. 47-14, 2007.

[27] Wikipedia contributors, Non-blocking Algorithm. Wikipedia, The Free Encyclopedia,*http://en.wikipedia.org/wiki/Non-blocking_algorithm.*

[28] A. Podzimek, *Read-Copy-Update for OpenSolaris.* Diploma thesis, Charles University in Prague, 2010.

[29] G. Prinslow, "Overview of Performance Measurement and Analytical Modeling Techniques for Multi-core Processors", 2011.

[30] Wikipedia contributors, Read-copy-update. Wikipedia, The Free Encyclopedia, *http://en.wikipedia.org/wiki/Read-copy-update*

[31] M. R. Regmee, *Self Adjusting Contention Friendly Concurrent Binary Search Tree by Lazy Splaying.* Masters thesis, University of Nevada Las Vegas, 2014.

[32] http://wiki.cs.pdx.edu/ Contributors, "Relativistic Programming Overview, What is RP?", *http://wiki.cs.pdx.edu/rp/.*

[33] Wikipedia contributors, Splay Tree. Wikipedia, The Free Encyclopedia, *http://en.wikipedia.org/wiki/Splay_tree.*

[34] Wikipedia contributors, Transactional Memory. Wikipedia, The Free Encyclopedia, *http://en.wikipedia.org/wiki/Transactional_memory.*

[35] J. Triplett, *Relativistic Causal Ordering A Memory Model for Scalable Concurrent Data Structures.* PhD thesis, Portland State University, 2011.

Curriculum Vitae

Graduate College

University of Nevada, Las Vegas

Jaya Ram Sedai

Degrees:

Bachelor of Engineering in Computer Engineering 2009

Tribhuvan University

Institute of Engineering, Pulchowk Campus

Thesis Title: Concurrent Lazy Splay Tree with Relativistic Programming Approach

Thesis Examination Committee:

Chairperson, Dr. Ajoy K. Datta, Ph.D.

Committee Member, Dr. John Minor, Ph.D.

Committee Member, Dr. Lawrence Larmore, Ph.D.

Graduate Faculty Representative, Dr. Venkatesan Muthukumar, Ph.D.