A Relaxed Balanced Non-Blocking Binary Search Tree

Manish Singh
manish.singh@weltec.ac.nz
Wellington Institute of Technology
Lower Hutt, New Zealand

Lindsay Groves
lindsay.groves@ecs.vuw.ac.nz
Victoria University of Wellington
Wellington, New Zealand

Alex Potanin
alex.potanin@ecs.vuw.ac.nz
Victoria University of Wellington
Wellington, New Zealand

ABSTRACT
We present a new relaxed balanced concurrent binary search tree in which all operations are non-blocking. We utilise the notion of separating balancing operation and update operations in a concurrent environment and design a non-blocking balancing operation in addition to the regular insert, search and relaxed delete operations. Our design uses a single-word CAS supported by most modern CPUs.

CCS CONCEPTS
• Computing methodologies → Concurrent algorithms.

KEYWORDS
Non-blocking, lock-free, Concurrent Binary Search Tree

1 INTRODUCTION
To counter Intel’s single socket 28-core processor, AMD announced a 32 core processor with 64 threads for desktops recently. This trend in embedding more and more cores in multi-core processors has necessitated the design of scalable and efficient concurrent data structures. Due to the asynchronous model of computation in multi-core systems designing of data structures which can efficiently synchronise concurrent access are considered to be hard. A considerable amount of research has been done towards making both blocking and non-blocking concurrent versions of sequential data structures. Unlike blocking, a concurrent algorithm is non-blocking if it ensures that no thread accessing the data structure is postponed indefinitely while waiting for other threads that operate on the data structure. Performance has been always an important factor that will drive the design of these data structures.

The Binary Search Tree (BST) is a fundamental data structure. In recent years, many of concurrent BST (both blocking and non-blocking versions) have been proposed [1–7]. However, only few designs include self-balancing operations. Table 1 summaries some of the state of art concurrent BSTs. Most of the published work try to emulate the sequential specification of the data structure in their concurrent version. This results in performance compromise as strict invariants must be maintained in each operation execution. In a concurrent environment effects of some operations might cancel out effects of other operations. In case of a self-balancing BST each insert or delete operation requires a balancing operation to be performed immediately to maintain the height of the tree. In such scenarios a balancing operation might cancel out effect of other balancing operations on the same node. This has led to the idea of relaxing some of the invariants of the data structure while designing their concurrent versions.

Relaxing some of the structural properties of a sequential data structure in concurrent environment is not completely a new idea. Lazy deletion is an example of relaxing the requirement that nodes are immediately removed from the concurrent linked list. In a concurrent self-balancing trees such as AVL or chromatic tree rotation operations are performed separately from insertion and deletion operations. However, most previously published designs which includes balancing operation are blocking or lock-based and therefore, not immune to the problems associated with locks. We design a non-blocking relaxed balanced BST in which balancing operations are done separately from the regular insert and delete operations.

2 ALGORITHM DESCRIPTION
We implement a set in which the underlying data structure is a BST. Our implementation supports search(k): to check if the key is in set or not, insert(k): to add the key into the abstraction, delete(k): to remove the membership of key if it is present in the dataset. A BST could behave similar to a linked-list where traversal could take O(n) time especially when the input are in increasing or decreasing order. In order to prevent that our algorithm also supports rotation operation and a abstract update operation. This design closely tries to mirror the sequential version of BST except for the delete operation, where key is first flagged as deleted and then physically removed later by a dedicated separate thread which also carries the rotation operation. The aforementioned dedicated thread traverses the tree in pre-order, carries out first physical removal of deleted nodes, adjusts local heights of the node and starts the rotation operation if the balance condition is found to be violated.
To achieve lock-freedom the any thread before carrying out the actual operation which modifies the tree either the content or structure first announces its intention. To do so the thread first collects needed information into a structure known as operation-descriptor and tries to insert a pointer to operation-descriptor into the operation field of that node using a CAS. Once the pointer is inserted, the operation would not fail as we have designed helping methods which ensure any other conflicting thread will carry out the announced operation before it carrying out its own. The idea is that the operation owns the nodes or locations to be updated not a particular thread which makes it possible for other threads to finish contending operations.

Updating multiple locations using a single word CAS while persevering atomicity of updated locations is a challenging task. Especially, in case of rotation operation three locations are needed to be updated automatically, we achieve this by careful design of our rotation operation. Whenever a balance condition is found to be violated, the dedicated thread prepares a operation descriptor then it tries to grab the node by inserting a pointer to the descriptor in the operation-field of parent of the node. If it is successful then it tries to do the same for all the nodes involved in that particular rotation operation using CAS. Failing to insert the pointer in any of the nodes means that there is another operation on progress and it should help the ongoing operation first then retry. Once the first node is grabbed any other thread can finish the announced rotation operation by first, grabbing the remaining nodes then carrying out the rest of the operation. As shown in figure 1, first a newnode having same key as the node where balance violation have been occurred is created. After that Right and left pointers of the newnode are allocated to those child which it would get after notation figure 1 (b). Next step is to insert the newnode to its position after rotation, as shown in figure 1 (c). In this case of right rotation the newnode would be the right-child of node C. The third and last step is swing pointer to connect C to the parent P directly effectively removing node N from the tree.

A notable feature of our algorithm is that the rotation operations are invisible to search operations, as a result, the later simply traverses the tree and never restarts. A thread carrying out delete operation simply marks the node to be deleted which is considered to be logically removed. The deleted node then is physically removed by the dedicated thread if it has no child or have only one child. Nodes with two child are not removed until it becomes one child node or both of its child are deleted. A Thread carrying out insert operation upon finding the insert location prepares an operation-descriptor and tries to insert a pointer to the to be parent of the newnode. Similar to the rotation operation, once the operation-descriptor is successfully inserted the operation is guaranteed to be completed. Otherwise it goes to help the ongoing operation and restarts. While doing insert if the thread finds key present in the tree and the node is flagged as deleted it simply sets off the deleted flag.

### 3 SUMMARY

In our design, all the operations are non-blocking. The search operation is free of any additional synchronization. We are evaluating performance of our algorithm on x86 _64 and SPARC multi-core machines against the concurrent BSTs shown in table 1. A mechanized proof outline has been done using the linearisability as correctness criteria. To the best of our knowledge, this is the first design which utilises decoupling of operations as well as rotations to balance the tree in a non-blocking concurrent set-up.

### REFERENCES


