A relaxed balanced lock-free binary search tree

Manish Singh[1], Lindsay Groves[2], and Alex Potanin[2]

[1] Wellington Institute of Technology
[2] Victoria University of Wellington
manish.singh@weltec.ac.nz
{lindsay.groves, alex.potanin}@ecs.vuw.ac.nz

Abstract. This paper presents a new relaxed balanced concurrent binary search tree using a single word compare and swap primitive, in which all operations are lock-free. Our design separates balancing actions from update operations and includes a lock-free balancing mechanism in addition to the insert, search, and relaxed delete operations. Search in our design is not affected by ongoing concurrent update operations or by the movement of nodes by tree restructuring operations. Our experiments show that our algorithm performs better than other state-of-the-art concurrent BSTs.

Keywords: Concurrent Data Structure · Lock-Free· Binary Search Tree.

1 Introduction

Recently, the chip manufacturing company, AMD, has released the Threadripper CPU with 64 cores and 128 threads for desktops. As CPU manufacturers embed more and more cores in their multi-core processors, the need to design scalable and efficient concurrent data structures has also increased. Considerable research has been done towards making both blocking and non-blocking concurrent versions of sequential data structures. Unlike blocking a concurrent data structure design is non-blocking, or lock-free, if it ensures at any time at least one thread can complete its operation. Performance has always been an important factor and will drive the design of these data structures.

The Binary Search Tree (BST) is a fundamental data structure. Many concurrent BST, both blocking and non-blocking, have been proposed [1,4–6,14,15,17]. However, only a few designs include self-balancing operations. Most of the published work tries to emulate a sequential implementation. This results in performance compromise as strict sequential invariants must be maintained. In a concurrent environment the effect of some operations might cancel out the effects of other operations. In the case of a self-balancing AVL tree each insert or delete operation requires a balancing operation to be performed immediately to preserve the height-balanced property. A balancing operation might cancel out the effects of other balancing operations on the same set of nodes.

We have designed a lock-free relaxed balanced BST, hereafter referred as RBLFBST, in which balancing operations are decoupled from the regular insert and delete operations[1]. The idea of relaxing some structural properties of a sequential data structure in their concurrent version has been tried in several previous works. Lazy deletion [9, 10] is an example of relaxing the requirement that nodes are immediately removed from the data structure. In a relaxed self-balancing such as AVL [11], chromatic trees [16], rotation operations, are performed separately from the insertion and deletion operation.

2 Related Work

The first lock-free BST dates back to the 90's [19] where the author suggested a design based on a threaded binary tree representation, but did not discuss the implementation of his design. Fraser at el [7] described a lock-free implementation of an internal BST using multiple CAS (MCAS) to update multiple parts of the tree structure atomically. This algorithm uses a threaded representation of a BST. A delete operation required up to 8 memory locations to be updated atomically which added an appreciable overhead to the design. Based on the cooperative technique of [2], Ellen et al. [6] developed the first specific lock-free external BST algorithm, where internal nodes only act as routing nodes, which simplifies the deletion operation.

Using a helping technique similar to [6], Howley and Jones [17] presented a non-blocking algorithm for internal BSTs. They added an extra field *operation* to each node of the BST, in which all the information related to a particular update operation can be stored. A delete operation in this work could use as many as 9 CAS instructions. However, the paper gives experimental results in which this design outperforms [6, 15]. Aravind et el. [14] presented a lock-free external BST similar to [6], but using edge-marking. Being a leaf-oriented tree this design also has a smaller contention window for insert and delete operations and uses fewer auxiliary nodes to coordinate among conflicting operations. Another contemporary paper [18] describes a general template for non-blocking trees. This template uses multi-word versions of LL/SC and VL primitives, making it easier to update multiple parts of the tree. The paper also presented a fine-grained synchronized version of a leaf-oriented chromatic tree based on their template.

The concurrent AVL tree of Bronson et al. [15] uses lock coupling which uses per-node locks to remove conflicts between concurrent updates, and a relaxed balancing property which does not enforce the strict AVL property. This design uses a partially external tree such that by default nodes behave as per an internal tree but in order to simplify the removal of nodes with two children, the removed node objects are allowed to remain in the structure and act as routing nodes. This avoids the problem of locking large parts of the tree if a delete operation was implemented exactly as in a sequential BST. These deleted (logically) nodes can then be either removed during a later re-balancing operation or, if the node's key is reinserted, the key can be made part of the set again. This partially external tree design was experimentally shown to have a small increase (0-20%) on the

[1] A poster describing the design of RBLFBST was presented in ICPP 2019, Kyoto, Japan

total number of nodes required to contain the key set. Crain at el. [4] present a lock-based tree in which balancing operations are decoupled from the insert and delete, and are done by a separate dedicated thread. Keys that are deleted are not immediately removed from the tree but are only marked as deleted as in [15]. Later, a dedicated thread can remove nodes with deleted keys that have single or no child. The balancing mechanism used closely mirrors that in [3]. Despite claiming performance improvement by more than double to that of [15] another lock-based design this tree still is lock-based. More recently, Drachsler et al. [5] proposed a lock-based internal BST supporting wait-free search operations and lock-based update operations via logical ordering. Their design uses a similar threaded binary tree representation as in [19].

All the designs of concurrent BST with balancing operations described above use locks to synchronize concurrent updates and therefore are not immune to problems that are associated with locking in general. Based on techniques used in previous research we present a concurrent BST in which all the update operations are lock-free. To our knowledge, our design is the first AVL tree based lock-free partially external BST which includes balancing operation for all the update operations.

3 Algorithm Description

3.1 Overview

We implement a set using a BST. Our implementation supports concurrent executions of *search(k)*: to determine whether the key k is in the set, *insert(k)*: to add the key k to the set, *delete(k)*: to remove the key k from the set. To ensure that the tree does not become unbalanced, causing the operations to have linear cost, our design supports a relaxed tree balancing mechanism.

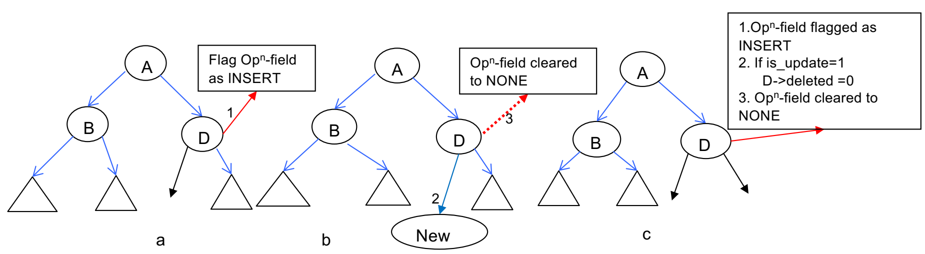

Fig. 1: left child pointer of Node D is identified as an insertion point for a 'new' key. (a). An *operation-descriptor* flagged as INSERT is inserted in to the *operation* field of node D (1). (b). After successful insertion of *operation-descriptor*, the node having key 'new' as a child of node D is physically inserted tiny(2) and then the *operation* field of node D is cleared by setting its flag to NONE (3). (c) Shows the case when the key to be inserted is found in the tree but is deleted. In this case, the *deleted* field is set to false.

Any thread attempting to insert a *key*, upon finding the insertion point first announces its intention. The thread first collects the information required in an *operation-descriptor* and tries to insert it in the *operation* field of the node identified as the insertion point (the parent of the new node). Once the *operation-descriptor* is inserted into the node the *key* is considered to be part of the set.

Then the new node is inserted using a CAS to the appropriate child pointer of the parent node as shown in figure 1(a) and (b).

To remove a key, first the node having the key to be removed is flagged as deleted (*deleted* bit is set to true) and then physically removed later. Once the *deleted* bit is set the key is considered to be removed (logically) from the tree. The physical removal is started by a separate maintenance thread by marking the deleted nodes having at most one child. Once marked any other thread can physically remove the node from the tree. Deleted nodes with two children are not physically removed from the tree until they have less than two children. This relaxed deletion is done to reduce contention in the tree as the delete algorithm does not need to locate and update the node to be deleted and the replacement node, which might involve several restarts. This relaxed deletion approach also means that a thread doing an insert operation may find the intended key already in the tree but the node is flagged as deleted. In this case, the key can be made part of the set again by setting the *deleted* bit off.

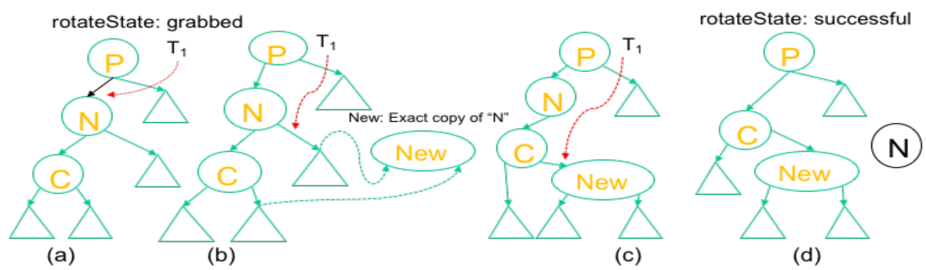

Fig. 2: Right rotation example (similarly for the left). (a) Shows the case where the maintenance thread has found a balance condition violation at node N and has successfully inserted a rotation *operation-descriptor* in the *operation* field of the parent of N, P, N, and child of N, C. After that a *new* node having key and other fields exactly as the node N is created. (b) Then the right and the left pointer of the *new node* are allocated to those children which the node N would have got after the rotation. (c) The next step is to insert the *new* node to its position after *rotation*. In this case, the *new* node would be the right-child of node C. (d) The third and last step is to connect node the parent P to child C effectively removing node N from the tree. A thread T1 carrying out search is oblivious to the movement of nodes by rotations

Our balancing mechanism is based on heights of nodes and rotation operations, as in sequential AVL trees. However, balancing is carried out separately to the update operations, similar to that of [4, 12]. The balancing adjustments are performed by the maintenance thread. If the balancing condition at any node is found to be violated then the maintenance thread collects all the information required to do rotation operation in to an *operation-descriptor* and tries to insert it in to the *operation* field of the parent of the node. After successfully inserting the operation descriptor, the maintenance thread (or any other thread) tries to do the same for the node and the appropriate child of the node involved in the rotation. Figure 2 shows the remaining steps for the rotation operation (right rotation) once the *operation-descriptor* is inserted in all the three nodes involved. Due to the ongoing concurrent updates, it is difficult to determine the exact height of a node at any point in the execution. Thus, the balancing condition in RBLFBST is based on apparent local heights of the node and its children. It should also be noted that the rotation technique shown in figure 2 makes ongoing concurrent search operations oblivious to any node moving up or down the tree.

```
struct node_t {
    int key
    node_ t* left, right
    operation_t* op
    int local_height, lh, rh
    bool deleted,removed
}
union operation_t {
    insert_op_t insert_op
    rotate_op_t rotate_op
}
struct insert_op_t {
    bool_t is_left
    bool_t is_update=FALSE
    node_t* expected
    node_t* new
} insert_op_t
struct rotate_op_t {
    int /*volatile*/ state =
      UNDECIDED
    node_t* parent
    node_t* node
    node_t* child
    operation_t* pOp
    operation_t* nOp
    operation_t* cOp
    bool rightR
    bool dir
} rotate_op_t
//operation marking status
NONE 0, MARK 1, ROTATE 2,
  INSERT 3
```

Algorithm 1: Basic structures used

```
Input: int k, node_t* root
Output: true, if the key is found in
        the set otherwise false
node_t* node, next
operation_t* node_op
bool result = FALSE
node = root
node_op = node→op
int node_key
next = node→right
while !ISNULL(next) do
    node = next
    node_op = node→op
    node_key = node→key
    if k < node_key then
        next = node→left
    else if k > node_key then
        next = node→right
    else
        result = TRUE
        break
if result && node→deleted then
    if GETFLAG(node→op) ==
      INSERT then
        if
          node_op→insert_op.new→key
          == k then
            return TRUE
    return FALSE
return result
```

Algorithm 2: Search

All the operations in RBLFBST are lock-free and linearisable. Once an operation is flagged with the corresponding FLAG and the *operation-descriptor* is inserted, it is considered to be logically completed. Any other thread can complete the announced operation using the information available in the *operation-descriptor*. Updating multiple locations using a single word CAS while preserving atomicity is a challenging task. Especially, in case of a rotation operation, three locations are needed to be updated atomically. We achieve this by careful design of our operation, described in detail in later sections. Another notable feature of this design is that we managed to keep the most frequently used operation in BSTs, *search*, free of additional coordination.

3.2 Detailed Algorithm

Structures: Algorithm 1 shows various structures used in the implementation. As in a sequential BST, a node has key, right, and left pointers to its corresponding child which are set to NULL for every new node. To synchronize various concurrent operations, each node has an operation pointer field, *op*, that is used to announce the intended operation and contains a pointer to that particular *operation-descriptor*. A node in RBLFBST has 3 heights, *local-height*: updated by adding one to the maximum *local-height* of its children, *right-height*: *local-height* of right child, and *left-height*: *local-height* of left child, which are stored in fields *local-heights*, *lh* and *rh* respectively. In addition to that it has two boolean fields *deleted* and *removed* which are initialized to false. If a node has its *deleted* field set that means it is deleted from the set but could still be in the tree, while the *removed* field is set when the node is removed from the tree.

```
int seek (int key, node_t** parent,
  operation_t** parent_op,
  node_t** node, operation_t**
  node_op, node_t* aux_root,
  node_t* root)
int result, nodekey
node_t* next
retry:
result = NOT_FOUND_L
*node = aux_root
*node_op = *node→op
if GETFLAG(*node_op) ≠ NONE
  then
    if auxroot == root then
        bst_help_insert((operationt*)
          UNFLAG(*node_op), *node)
        goto retry
next = (nodet*) (*node)→right
while !ISNULL(next) do
    *parent = *node
    *parent_op = *node_op
    *node = next
    *node_op = (*node)→op
    nodekey = (*node)→key
    if key < nodekey then
        result = NOT_FOUND_L
        next = (*node)→left
    else if key > node_key then
        result = NOT_FOUND_R
        next = (*node)→right
    else
        result = FOUND
        break
if GETFLAG(*node_op)≠ NONE
  then
    help(*parent, *parent_op, *node,
      *node_op)
    goto retry
return result
```

Algorithm 3: Seek

```
Input: int key, node_t* root
Output: true: if the key is added to
        the set, false otherwise
node_t* parent
node_t* curr
node_t* new_node = NULL
operation_t* parent_op
operation_t* node_op
operation_t* cas_op
int result
while TRUE do
    result = seek(key, &parent,
      &parent_op, &node, &node_op,
      root, root)
    if result == FOUND && !
      node→deleted then
        return FALSE
    if new_node == NULL then
        new_node = alloc_node(k,v,0)
    bool_t is_left = (result ==
      NOT_FOUND_L)
    node_t* old
    if is_left then
        old = node→left
    else old = node→right

    cas_op = alloc_op()
    if result == FOUND &&
      node→deleted then
        cas_op→insert_op.is_update =
          TRUE
    cas_op→insert_op.is_left = is_left
    cas_op→insert_op.expected = old
    cas_op→insert_op.new = new_node
    if CAS(&node→op, node_op,
      FLAG(cas_op, INSERT)) ==
      node_op then
        help_insert(cas_op, node)
        return TRUE
```

Algorithm 4: Insert

The *operation-descriptor* is either an, *insert_op* containing information for an intended insert, or a *rotate_op*: containing information for a rotation. An *insert_op* contains, *new*: pointer to the new node to be inserted, *expected*: previous node, *isLeft*: indicating whether the new node would be inserted as a left-child or right-child, and *is_update*: indicating whether it is a new node to be inserted or an existing deleted node to be made part of the set again. A *rotate_op* descriptor contains in addition to the nodes involved in the corresponding operation, a *state* field which can have three values, UNDECIDED: all nodes involved in the operation have not been grabbed yet, GRABBED: all the nodes that are needed have been grabbed, ROTATED: indicating that the rotation has been completed.

As in [9, 17], we use two least significant bits of a pointer address to store the operation status. The operation pointer, *op*, of a node could be in one of following statuses, NONE: meaning no operation is currently intended, MARK: this node can be physically removed, ROTATE: a rotate operation is intended, and INSERT: an insert operation is intended. The following macros are used to modify *op*'s status, *FLAG(op, status)*: sets the operation pointer to one of the above status, *GETFLAG(op)*: returns the status of *op*, *UNFLAG(op)*: resets the status to NONE.

Search: The *search* algorithm, outlined in algorithm 2, is mostly similar to the sequential search with additional checks due to other concurrent inserts. It traverses from the *root*[2] to a leaf node looking for the *key*. If found, it breaks out of the loop as shown at lines 17-18. A further check is needed to see if the *key* is deleted from the set. In this case, if the corresponding node's *deleted* field is set and its *operation* field is flagged as INSERT then the new *key* which is to be inserted, is compared with the *key* that the *search* is looking for at lines 19-21. If a match is found then the algorithm returns *true* line 22. If *deleted* bit is not set and the node is found then the algorithm simply returns the true at line 24. If the *key* is not found the *search* algorithm returns false line 24. Synchronization of any form is not needed in this *search* algorithm. Also, the *search* algorithm never restarts.

seek: Algorithm 3 outlines the *seek* method which is used by both insert and delete algorithms to locate the position or potential position of a *key*, starting from a specified point *aux_root* in the tree. The position is returned in the variables pointed to by arguments *parent* and *node* and the values of their *operation* fields are returned in the variables pointed to by arguments *parent_op* and *node_op*. The result of the seek can be one of three values, FOUND: if the key was found, NOT_FOUND_L: if the *key* was not found but would be positioned at the left child of the *node* if it were inserted, NOT_FOUND_R: similar to NOT_FOUND_L but for the right child. The variable *next* is used to point to the next node along the seek path, the *parent* and *parent_op* are used to record previous node. The check at lines 7-10 handles the case when the root has an ongoing operation to add to the empty tree. The seek loop traverses nodes one at a time until either the *key* is found or a null link is reached. Line 27 checks whether the node that is found does has an ongoing operation. If the node has an ongoing operation, the appropriate helping is done and seek restarts at lines 28-29.

Insert: The *Insert* algorithm, algorithm 4, begins by calling the *seek* method at line 9. Depending on the result of the *seek* method : case 1: the *key* is found

[2] For implementation purpose, the first *node* to be inserted in the empty tree is kept as the right child of a fixed node root which has *key* assumed to be infinity

and the corresponding *node* is not *deleted*; case 2: the *key* is found and the corresponding *node* is *deleted*; case 3: the key is not found in the tree. For case 1, the *insert* method returns false at line 11. For both cases 2 and 3, a *new-node* and an *insert_op operation-descriptor* are allocated with all the necessary information at lines 12-13 and 20-25 respectively. Furthermore, for case 2 the *is_update* field of *insert_op* descriptor is set to true at lines 21-22. Then a CAS is used to flag the *operation* field of the node to INSERT. If successful, the *help_Insert* method is called to complete the physical insertion, otherwise it retries. If the *help_insert* method finds *is_update* field set it simply sets off the *deleted* bit of the node to make it part of the set again otherwise it inserts the *new-node*.

Delete: Similar to the *Insert*, the *Delete* method starts by calling the *seek* method at line 6. If a *node* with the desired *key* is not found, it simply returns false at lines 7-8. If the *key* is found and is *deleted*, it further checks whether the node is flagged as INSERT, since an ongoing *insert* operation might be trying to insert the same *key*. If the *node* is not flagged as INSERT, the *delete* method returns false as the *node* is already deleted at lines 9-11. If the key is found and not deleted, the *delete* method tries to set the *node*'s *deleted* field to *true* and returns *true* on successful CAS at line 12-15.

tree-maintenance: The tree maintenance actions involve checking for balance condition violations, rotation operations, and physical removal of deleted nodes having at most one child. The latter two operations are started by a maintenance-thread and can be completed by any other thread[3].

The maintenance-thread repeatedly performs a depth-first traversal in the background in which, at each node, it looks to see if the node can be removed and then adjusts heights. If it finds any node with *deleted* field set and at most one child, it then tries to flag the *operation* field of the node to MARK. If marking of the node is successful, the *help_marked* method outlined in algorithm 7 is called to physically remove the node from the tree. After adjusting heights, a check is performed to see if a balance violation has occurred. At any node, the balance condition is said to be violated if *right-height* and *left-height* of the node differ by 2 or more. If there is a balance violation at any node further checks are performed to determine whether a single rotation (left or right) or double rotations are required. Once the type (left or right) and number (single or double) of rotation operations are determined the rotation process is started as listed in the *left_rotate*, algorithm 6, for the left rotation (similarly for the right rotation).

In the *left_rotate* method checks are performed to ensure that nodes that are involved in rotation operation are still intact at lines 2-6. The check at line 7 is performed to see whether double rotations are needed. After this check, a *rotate_op* descriptor is allocated at line 10 using appropriate values.

[3] Detailed explanations and full algorithm for tree-maintenance can be found in [13]

```
Input: int k, node_t* root
Output: true: if the node is found (not
        already deleted) and deleted
        from the set; else false
node_t* parent
node_t* node
operation_t* parent_op
operation_t* node_op
while TRUE do
    int res = seek(k, &parent,
      &parent_op, &node, &node_op,
      root, root)
    if (res ≠ FOUND) then
        return FALSE
    if node→deleted then
        if GETFLAG(node→op) ≠
          INSERT then
            return FALSE
    else
        if GETFLAG(node→op) ==
          NONE then
            if CAS(&node→deleted,
              FALSE, TRUE) ==
              FALSE then
                return TRUE
```

Algorithm 5: delete

```
int leftrotate(node_t* parent, int
  dir, bool_t rotate)
node_t* node
if parent→removed then
    return 0
node = (dir == 1) ?
  (node_t*)parent→right :
  (node_t*)parent→left
if ISNULL(node) ||
  ISNULL(node→right) then
    return 0
if (node→right→lh -node→right→rh)
  > 0 && !rotate then
    return 3; //double rotation
if GETFLAG(parent→op) == NONE
  then
    operation_t* rotateOp
    rotateOp =
      alloc_rotateop(parent,node,
      node→right, parent→op,
      node→op,
      node→right→op,dir,FALSE)
    if CAS(&(parent→op),
      rotateOp→rotate_op.pOp,
      FLAG(rotateOp,ROTATE)) ==
      rotateOp→rotate_op.pOp then
        help_rotate(rotateOp, parent,
          node, node→right);
return 1;
```

Algorithm 6: left_rotate

```
void help(node_t* parent,
  operation_t* parent_op, node_t*
  node, operation_t* node_op)
if GETFLAG(node_op) == INSERT
  then
    help_insert(UNFLAG(node_op),
      node)
else if GETFLAG(parent_op) ==
  ROTATE then
    help_rotate((UNFLAG(parent_op),
      parent,node,
      parent_op→rotate_op.child)
else if GETFLAG(node_op) ==
  MARK then
    help_marked(parent, parent_op,
      node)

void help_insert(operation_t* op,
  node_t* dest)
if op→insert_op.update then
    CAS(&dest→deleted, TRUE,
      FALSE)
else
    node_t** address = NULL
    if op→insert_op.is_left then
        address = (node_t**)
          &(dest→left)
    else
        address = (node_t**)
          &(dest→right)
    CAS(address,
      op→insert_op.expected,
      op→insert_op.new)
CAS(&(dest→op), FLAG(op,
  INSERT), FLAG(op, NONE))

void help_marked(node_t* parent,
  operation_t* parent_op, node_t*
  node)
node_t* child, address
if ISNULL((node_t*) node→left) then
    if ISNULL((node_t*)
      node→right) then
        child =
          (node_t*)SETNULL(node)
    else
        child = (node_t*) node→right
else
    child= (node_t*) node→left
node→removed = TRUE
operation_t* cas_op = alloc_op()
cas_op→insert_op.is_left = (node ==
  parent→left)
cas_op→insert_op.expected = node
cas_op→insert_op.new = child
if CAS(&(parent→op), parent_op,
  FLAG(cas_op, INSERT)) ==
  parent_op then
    help_insert(cas_op, parent)
```

Algorithm 7: help,help_insert,help_marked

Then, an attempt is made to insert *rotate_op* to the *parent* of the *node* where the violation has occurred using CAS. If successful, a call is made to the *help_rotate* method.

```
Input: operation_t* op, node_t* parent, node_t* node, node_t* child
1  int seen_state = op→rotate_op.state
2  retry:
3  if seen_state== UNDECIDED then
4      if GETFLAG(node→op)== NONE ||
         GETFLAG(node→op) == ROTATE
         then
5          if GETFLAG(node→op)==NONE
             then
6              CAS(&(op→rotate_op.node→op),
                 node→op,FLAG(op,ROTATE))
7          if GETFLAG(node→op) ==
             ROTATE then
8              nodegrabbed:
9              if GETFLAG(child→op)
                 ==NONE ||
                 GETFLAG(child→op)
                 ==ROTATE then
10                 if GETFLAG(child→op)
                     ==NONE then
11                     CAS(&(op→rotate_op.
                         child→op), child→op,
                         FLAG(op,ROTATE))
12                 if GETFLAG(child→op)
13                 == ROTATE then
14                     CAS(&
                         (op→rotate_op.state),
                         UNDECIDED,
                         GRABBED)
15                     seen_state =GRABBED
16                 else goto nodegrabbed
17
18             else
19                 help(node,node→op,child,
                     child→op)
20                 goto nodegrabbed
21         else
22             goto retry
23     else
24         help(parent,parent→op,node,
             node→op)
25         goto retry
26 if seen_state==GRABBED then
27     if op→rotate_op.rightR then
28         // right rotation
29         //allocate newnode with
30         appropriate children and heights
31         //carry out rotation steps
32         using CAS
33     else
34         // left rotation
35         ///allocate newnode with
36         appropriate children and heights
37         //carry out rotation steps
38         using CAS
39     //parent pointer swing
40     if op→rotate_op.rightR then
41         CAS(&op→rotate_op.parent→left,
             op→rotate_op.node,child)
42     else
43         CAS(&op→rotate_op.parent
44         →right, op→rotate_op.node,child)
45     //adjust child and parent heights
46     CAS(&(op→rotate_op.state),
47     GRABBED, ROTATED)
48     seen_state=ROTATED;
49 if seen_state==ROTATED then
50     //Clear parent,node,child operation
51     fields from ROTATE to NONE
52 bool_t result = (seen_state == ROTATED)
53 return result;
```

Algorithm 8: help_rotate

The *help_rotate* method, outlined in algorithm 8 can be called by any thread if it finds that the *operation* field of a node is flagged as ROTATE. Any thread executing *help_rotate* tries to the grab remaining nodes namely, the *node* and the *child* by flagging their *operation* fields to ROTATE at lines 2-24. Failing to flag any node means that there is an ongoing operation. In this case, the thread helps the ongoing operation first, lines 18 and 23. Once both *node* and *child* are flagged the *state* field of *rotate_op* descriptor is updated to a value GRABBED at line 13. The rest of the operation is carried out (lines 25-40) as shown in figure 2.

It should be noted that all of the methods involved in tree balancing except the *help_rotate* are not exposed to any thread other than the maintenance-thread.

help_insert/Marked: Both these methods are called only after inserting appropriate flags to the *operation* field of the *node*. Once a *node* is flagged it will never be un-flagged until the operation is completed. The former adds a new node as a child of the node that was flagged as INSERT or simply sets off the *deleted* field depending on the *is_update* field of the node, and the later physically removes the marked node from the tree.

3.3 Correctness

Lock-freedom : The non-blocking property of the algorithm is explained by describing the interactions that can occur between the threads that are executing read and write operations. A *search* will either locate the *key* it is searching for or terminate at a leaf node. The search operation never restarts. An *Insert* or *Delete* operation retries until it gets clean nodes through the *seek* method. Then, the *insert* tries to flag node as INSERT which cannot be undone until the physical insertion is completed. The *Delete* operation will only set the *deleted* bit when there is no other *insert* operation going on concurrently. The *seek* method restarts when it finds the leaf node has an ongoing operation when called from *Insert*, or if it finds the node having the *key* to be deleted has an ongoing operation when called from *Delete*. In both cases if the *operation* field of the node contains INSERT, ROTATE or MARK, this means an operation has been applied to the tree so system-wide progress was achieved. Assuming that there is no infinite addition of nodes on its search path, the *seek* method will eventually return clean nodes. An *Insert* operation could also restart if it fails to flag the node returned by *seek* to INSERT. In this case also it has encountered an ongoing operation. Any thread executing an operation will help the ongoing operation to its completion before restarting its own operation. Similarly, a rotation starts only when the maintenance-thread successfully flags the parent of the node where balance violation has occurred. If flagging of other nodes involved rotation fails in the *help_rotate* method, this means there is an ongoing operation and the process completes that first before coming back to flagging the node to ROTATE.

Linearisability : To prove the linearisability of RBLFBST, we first define the linearisation points of the *Search*, *Insert*, and *Delete* operations. The *search* operation can have two possible outcomes: a *key* is found in the set or not. The linearisation point for finding a *key* is the point at which an atomic read of the *key* has occurred at line 11. As our design allows a deleted *key* to be present in the tree it has to pass check at line 19. If the search does not find the *key*, it will linearise reading a NULL link at either line 13 or 15. If the tree is empty the search will linearise at reading the null link at line 7. A successful *Insert* operation will linearise when the *operation-descriptor* is successfully inserted to the *node* returned by the seek algorithm at line 26. Failure of insert operation will have linearisation point at line 17 of the *seek* algorithm where it reads the *key* to be inserted already present in the tree. However, it has to verify if the *key* is logically deleted by failing the test at line 11 of the *Insert* algorithm. Similarly, the successful *Delete* operation will linearise at the successful CAS at line 14 of the delete algorithm. Linearisation point of the failed *Delete* will occur at line 7 of the delete algorithm where it is verified that the *key* is not present in the tree or is logically deleted by passing the checks at lines 10-12. An elaborate correctness discussion can be found in [13].

4 Experimental Results

To evaluate performance of RBLFBST we compared it with following recently published implementations : (i) non-blocking internal BST denoted by bst-howley [17]. (ii) lock-free external binary BST denoted by , bst-Aravind [14]. (iii) lock-based partially internal BST with balancing operations denoted by bst-bronson [15]. (iv) lock-based internal BST with balancing operations denoted by bst-drachsler [5].

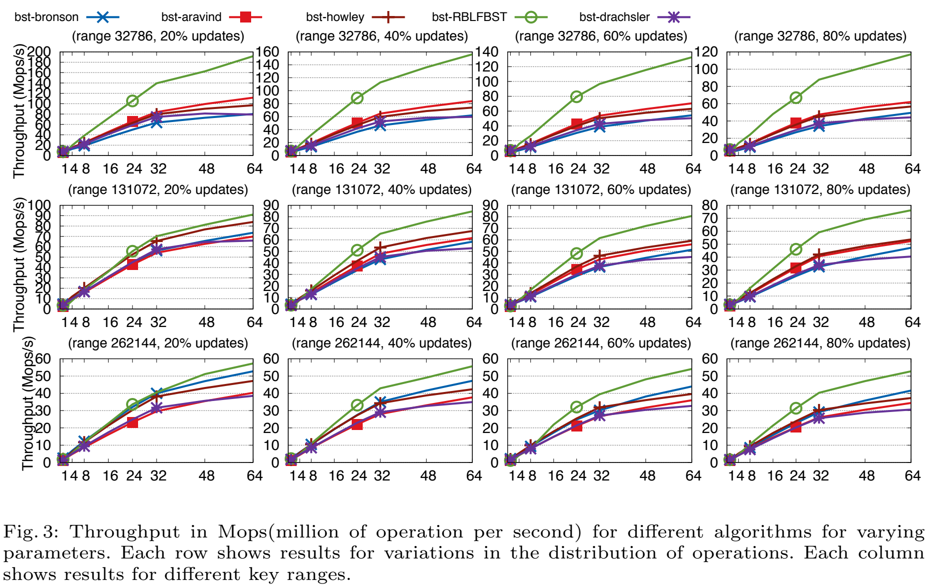

Fig. 3: Throughput in Mops(million of operation per second) for different algorithms for varying parameters. Each row shows results for variations in the distribution of operations. Each column shows results for different key ranges.

All experiments were conducted on a machine with 32 core AMD Ryzen Threadripper 2990WX processor, 64 hardware threads, and 32 GB RAM running x86_64 version of Ubuntu 18.04. All codes were implemented in C and compiled using gcc version 7.5.0 using optimization level O3. The source codes for other implementations were obtained from ASCYLIB [8].

The comparison of RBLFBST with other implementation was done varying three parameters: the key range ([0, 32786], [0, 131072] and [0, 262144]), the distribution of operations, and the number of threads concurrently executing on the data structures (1 to 64). Operations distribution considered were: (a) Write dominated workload: 80% updates, 20% search (b) mixed workload: 60% updates, 40% search (c) mixed workload: 40% updates, 60% search (d) read dominated workload: 20% updates, 80% search. Update operations had an equal number of insert and delete operations. All the operation types and the keys were generated using a pseudo-random number generator which were seeded with different values for each run. Each run was carried out for 5 seconds, and the results were collected averaging throughput over 20 runs. To mimic steady state the tree was filled with half the key range for each run.

The graphs in figure 3 show the comparisons of other implementations against RBLFBST. Overall, RBLFBST scales very well as the number of threads increases. For smaller key range 2^{15}, RBLFBST outperforms its nearest competitor bst-aravind by a maximum of 90% and 85% in read-dominated (80% and 60% search respectively), by 87% and 70% (60% and 80% updates respectively) in write-heavy workloads. Our tree beats the other two lock-based trees with balancing operations(bst-drachsler and bst-bronson) by 142% in 80% search workload and more than 85% in update heavy workload for the same key range. The performance of RBLFBST for 2^{17} key range again is better by 43% and 37% than its nearest lock-free competitor bst-howley in 60% and 80% update workload respectively. Similarly, it performs 56% and 61% better than the closest lock-based competitor bst-bronson in 60% and 80% update workload respectively. For the key range 2^{18}, RBLFBST beats bst-bronson by 24-38% and 20-30%, bst-howely by 10-31% and 20-30%, bst-aravind by 11-53% and 47-50% and bst-drachsler by 11-72% and 2-65% for the workload containing 80% and 60% updates respectively. The better performance of RBLFBST is due to the fact that it uses less number of expensive operations (CAS) than other lock-free implementations for insert and delete operations combined, thereby allowing more concurrency. The performance of our design goes down relatively for read-heavy workloads (80% read) particularly for the larger key ranges (2^{18}, 2^{17}). Larger key range will grow tree longer and more rotations will be performed. This is when the effect of threads helping rotation operation which can use up to 10 CAS instructions is clearly visible. However, the performance of RBLFBST in such cases is still better than all other implementations.

5 Conclusion and Future work

In this work, we presented a relaxed balanced lock-free binary search tree. In our design, all the set operations are lock-free. The search operation in our algorithm is oblivious to any structural changes done by other operations and is also free of any additional synchronization. Our results show its concurrent performance to be very good compared with other concurrent BSTs. We have discussed the correctness of RBLFBST operations. We are working on formal verification of our algorithm. We are also planning to apply the separation of tree balancing operations as well as relaxing delete operation to other lock-free balanced binary trees designs as future work.

References

1. Afek, Y., Kaplan, H., Korenfeld, B., Morrison, A., Tarjan, R.E.: Cbtree: A practical concurrent self-adjusting search tree. In: Aguilera, M.K. (ed.) Distributed Computing. pp. 1–15. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
2. Barnes, G.: A method for implementing lock-free shared-data structures. In: Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures. pp. 261–270. SPAA '93 (1993)
3. Bougé, L., Vallés, J., Peypoch, X.M., Schabanel, N.: Height-relaxed avl rebalancing: a unified, fine-grained approach to concurrent dictionaries (1998)
4. Crain, T., Gramoli, V., Raynal, M.: A contention-friendly binary search tree. In: Wolf, F., Mohr, B., an Mey, D. (eds.) Euro-Par 2013 Parallel Processing. pp. 229–240. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
5. Drachsler, D., Vechev, M., Yahav, E.: Practical concurrent binary search trees via logical ordering. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 343–356. PPoPP '14, Association for Computing Machinery, New York, NY, USA (2014)
6. Faith Ellen, Panagiota Fatourou, E.R., van Breugel, F.: Non-blocking binary search trees. In Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing, PODC '10 pp. 131–140 (2010)
7. Fraser, K.: Practical Lock freedom. Ph.D. thesis, King's College ,University of Cambridge (September 2003)
8. Guerraoui, R.: Ascylib (July 2020), https://dcl.epfl.ch/site/optik
9. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Proceedings of the 15th International Conference on Distributed Computing. pp. 300–314. DISC '01, Springer-Verlag, Berlin, Heidelberg (2001)
10. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008)
11. Kessels, J.L.W.: On-the-fly optimization of data structures. Commun. ACM **26**(11), 895–901 (Nov 1983)
12. Larsen, K.S.: Avl trees with relaxed balance. Journal of Computer and System Sciences **61**(3), 508–522 (2000)
13. Manish Singh, Lindsay Groves, A.P.: A relaxed balanced lock-free binary search tree. Tech. rep. (2020), https://ecs.wgtn.ac.nz/Main/TechnicalReportSeries
14. Natarajan, A., Mittal, N.: Fast concurrent lock-free binary search trees. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 317–328. PPoPP '14, New York, NY, USA (2014)
15. Nathan G. Bronson, Jared Casper, H.C., Olukotun, K.: A practical concurrent binary search tree. ACM SIGPLAN Symposium on Principals and Practice of Parallel Programming (2010)
16. Nurmi, O., Soisalon-Soininen, E.: Uncoupling updating and rebalancing in chromatic binary search trees. In: Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART. pp. 192–198. PODS '91, New York, NY, USA (1991)
17. Shane V. Howley, J.J.: A non blocking internal binary tree. SPAA (June 2012)
18. T. Brown, F. Ellen, E.R.: A general technique for non-blocking trees. In Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) (2014)
19. Valois, J.D.: Lock-Free Data Structures. Ph.D. thesis, Rensselaer Polytechnic Institute, Troy, NY, USA (1996)