Efficient Lock-free Binary Search Trees

Bapi Chatterjee, Nhan Nguyen and Philippas Tsigas
Chalmers University of Technology, Gothenburg, Sweden
{bapic, nhann, tsigas}@chalmers.se

ABSTRACT

In this paper we present a novel algorithm for concurrent lock-free internal binary search trees (BST) and implement a Set abstract data type (ADT) based on that. We show that in the presented lock-free BST algorithm the amortized step complexity of each set operation - ADD, REMOVE and CONTAINS - is $O(H(n)+c)$, where $H(n)$ is the height of the BST with n number of nodes and c is the contention during the execution. Our algorithm adapts to contention measures according to read-write load. If the situation is read-heavy, the operations avoid helping the concurrent REMOVE operations during traversal, and adapt to interval contention. However, for the write-heavy situations we let an operation help a concurrent REMOVE, even though it is not obstructed. In that case, an operation adapts to point contention. It uses single-word compare-and-swap (`CAS`) operations. We show that our algorithm has improved disjoint-access-parallelism compared to similar existing algorithms. We prove that the presented algorithm is linearizable. To the best of our knowledge, this is the first algorithm for any concurrent tree data-structure in which the modify operations are performed with an additive term of contention measure.

Categories and Subject Descriptors

E.1 [**data-structures**]: Distributed data-structures—*algorithm, complexity analysis*; D.1.3 [**Programming Techniques**]: Concurrent Programming - Parallel programming

Keywords

concurrent data-structures; binary search tree; amortized analysis; shared memory; lock-free; CAS

1. INTRODUCTION

With the wide and ever-growing availability of multi-core processors it is evermore compelling to design and develop more efficient concurrent data-structures. Being immune to deadlocks due to various fault-causing factors beyond the control of the data-structure designers, the non-blocking concurrent data-structures are more attractive than their blocking counterparts.

In literature, there are lock-free as well as wait-free singly linked-lists [11, 23], lock-free doubly linked-list [22], lock-free hash-tables [17] and lock-free skip-lists [11, 21]. However, not many performance-efficient non-blocking concurrent search trees are available. A multi-word compare-and-swap (`MCAS`) based lock-free BST implementation was presented by Fraser in [12]. However, `MCAS` is not a native atomic primitive provided by available multi-core chips and is very costly to be implemented using single-word `CAS`. Bronson et al. proposed an optimistic lock-based partially-external BST with relaxed balance [4]. Ellen et al. presented lock-free external binary search tree [10] based on co-operative helping technique presented by Barnes [3]. Though their work did not include an analysis of complexity or any empirical evaluation of the algorithm, the contention window of update operations in the data-structure is large. Also, because it is an external binary search tree, REMOVE is simpler at the cost of extra memory to maintain internal nodes without data. Howley et al. presented a lock-free internal BST [15] based on similar technique. A software transactional memory based approach was presented by Crain et al. [7] to design a concurrent red-black tree. While it seems to outperform some coarse-grained locking methods, it gets easily vanquished by a carefully tailored locking scheme as in [4]. Recently, two lock-free external BSTs [18, 5] and a lock-based internal BST [8] have been proposed. All of these works lack theoretical complexity analysis.

A common predicament for the existing lock-free BST algorithms is that if multiple modify operations contend at a leaf node, and, if a REMOVE operation among them succeeds then all other operations have to restart from the root. It results in the complexity of a modify operation to be $O(cH(n))$ where $H(n)$ is the height of the BST on n nodes and c is the measure of contention. It may grow dramatically with the growth in the size of the tree and the contention. In addition to that, CONTAINS operations have to be aware of an ongoing REMOVE of a node with two children, otherwise it may return an invalid result. Hence in the existing implementations of lock-free internal BST [15], a CONTAINS operation may have to restart from the root on realizing that the return may be invalid if more nodes are not scanned. The external or partially-external BSTs remain immune to this problem at the cost of extra memory for the routing internal nodes. Our algorithm solves both these problems elegantly. The CONTAINS operations in our BST enjoy oblivion of any

PODC'14, July 15–18, 2014, Paris, France.

ACM 978-1-4503-2944-6/14/07 ...$15.00.
http://dx.doi.org/10.1145/2611462.2611500 .

kind of modify operation as long as we do not make them help a concurrent REMOVE, which may be needed only in write-heavy situations. Also, the modify operations after helping a concurrent modify operation restart not from the root rather from a level in the vicinity of failure. It ensures that all the operations in our algorithm run in $O(H(n)+c)$[1]. This is our main contribution.

We always strive to exploit maximum possible disjoint-access-parallelism [16] in a concurrent data-structure. The lock-free methods for BST [10, 15], in order to use single-word `CAS` for atomically modifying the links outgoing from a node, and yet maintain correctness, store a flag as an operation field or some version indicator in the node itself, and hence a modify operation "holds" a node. This mechanism of holding a node, specifically for a REMOVE, can reduce the progress of two concurrent operations which may be otherwise non-conflicting. In [18], a flag is stored in a link instead of a node in an external BST. We found that even in an internal BST it is indeed possible that a REMOVE operation, instead of holding the node, just holds the links connected to and from a node in a predetermined order so that maximum possible progress of concurrent operations working at disjoint memory words corresponding to the links can be ensured. The design using "storing a flag" at a link instead of a node significantly improves the disjoint-access-parallelism. This is our next contribution.

Helping mechanism which ensures non-blocking progress may prove counterproductive to the performance if not used judiciously. However, in some situations the proportion of REMOVE operations may increase and they may need help to finish their pending steps. Then it is better to help them so that the traversal path does not contain large number of "under removal" nodes. Keeping that in view, we take helping to a level of adaptivity to the read-write load. In our algorithm, one may choose whether an operation helps a concurrent REMOVE operation during its traversal. We believe that this adaptive conservative helping in internal BSTs may be very useful in some situations. This is a useful contribution of this work.

Our algorithm requires only single-word atomic `CAS` primitives along with single-word atomic read and write which practically exist in all the widely available multi-core processors in the market. Based on our design, we implement a Set ADT. We prove that our algorithm is linearizable [14]. We also present complexity analysis of our implementation which is lacking in existing concurrent BST algorithms. This is another contribution in this paper.

The body of our paper will further consist of the following sections. In section 2, we present the basic tree terminologies. In section 3, the proposed algorithm is described. Section 4 presents a discussion on the correctness and the progress of our concurrent implementation along with an amortized analysis of its time complexity. The paper is concluded in section 5.

2. PRELIMINARIES

A *binary tree* is an ordered tree in which each *node* x has a *left-child* and a *right-child* denoted as $left(x)$ and $right(x)$ respectively, either or both of which may be *external*. When both the children are external the node is called a *leaf*, with one external child a *unary* node and with no external child a *binary* node, and all the non-external nodes are called *internal* nodes. We denote the parent of a node x by $p(x)$ and there is a unique node called *root* s.t. $p(root) = null$. Each parent is connected with its children via pointers as links (often we shall be using the terms pointer and link interchangeably when the context will be understood). We are primarily interested in implementing an ordered *Set* ADT - *binary search tree* using a binary tree in which each node is associated with a unique key k selected from a totally ordered universe. A node with a key k is denoted as $x(k)$ and x if the context is otherwise understood. Determined by the total order of the keys, each node x has a *predecessor* and a *successor*, denoted as $pre(x)$ and $suc(x)$, respectively. We denote height of x by $ht(x)$, which is defined as the distance of the deepest leaf in the *subtree* rooted at x from x. Height of a BST is $ht(root)$.

In an *internal BSTs*, all the internal nodes are *data-nodes* and the external nodes are usually denoted by `null`. There is a *symmetric order* of arranging the data - all the nodes in the *left subtree* of $x(k)$ have keys less than k and those in its *right subtree* have keys greater than k, and so no two nodes can have the same key. To query whether a BST CONTAINS a data with key k, at every *search-step* we utilize this order to look for the desired node either in the left or in the right subtree of the *current node* if the key not matched at it, unless we reach an external node. If the key matches at a node then we return `true` or address of the node if needed, otherwise `false`. To ADD data we query by its key k. If the query reaches an external node we replace this node with a new leaf node $x(k)$. To REMOVE a data-node corresponding to key k we check whether $x(k)$ is in the BST. If the BST does not contain $x(k)$, `false` is returned. On finding $x(k)$, we perform delete as following. If it is a leaf then we just replace it with an external node. In case of a unary node its only child is connected to its parent. For a binary node x, it is first replaced with $pre(x)$ or $suc(x)$ which may happen to be a leaf or a unary node, and then the replacer is removed.

In an alternate form - an *external BST*, all the internal nodes are *routing-nodes* and the external nodes are data-nodes. In this paper we focus on internal BSTs, and hence forward by a BST we shall mean an internal BST.

3. OUR ALGORITHM

3.1 The Efficient Lock-free BST Design

To implement a lock-free BST, we represent it in a threaded format [20]. In this format, if the left(right) child pointer at a node x is null and so corresponds to an external node, it is instead connected to $pre(x)(suc(x))$. An indicator is stored to indicate whether a child-link is used for such a connection. This is called *threading* of the child-links. In our design, we use the null child pointers at the leaf and unary nodes as following - right-child pointer, if null, is threaded and is used to point to the successor node, whereas a similar left-child pointer is threaded and pointed to the node itself, see Fig. 1(a). In this representation a binary tree can be viewed as an ordered list with exactly two outgoing and two incoming pointers per node, as shown in Fig. 1(b). Also among the two incoming pointers, exactly one is threaded and the other is not. In this representation, if $x(k_i)$ and $x(k_j)$ are

[1]A concurrent work by Ellen et al. [9] improving the external BST by Ellen et al. [10] achieves similar complexity as ours. This is to appear in the same proceedings and is independent from our work.

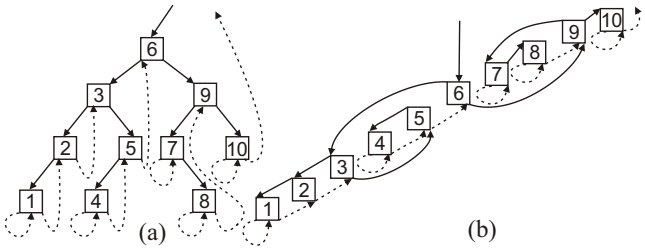

Figure 1: (a) Threaded BST (b) Equivalent ordered list.

two nodes in the BST and there is no node $x(k)$ such that $k_i \leq k \leq k_j$ then we call the interval $[k_i,\ k_j]$ *associated* with the threaded link incoming at k_j.

We exploit this symmetry of the equal number of incoming and outgoing pointers. A usual traversal in the BST following its symmetric order for a predecessor query, is equivalent to a traversal over a *subsequence* of the ordered-list produced by an in-order traversal of the BST, which is exactly the one shown in Fig. 1(b). This is made possible by the threaded right-links at leaf or unary nodes. Though in this representation, there are two pointers in both incoming and outgoing directions at each node, a single pointer needs to be modified to ADD a node in the list. To REMOVE a node we may have to modify up to four pointers. Therefore, ADD can be as simple as that in a lock-free single linked-list [11], and REMOVE is no more complex than that in a lock-free double linked-list [22]. A traversal in a lock-free linked-list may enjoy oblivion from a concurrent REMOVE of a node. In our design of internal BST also, a traversal can remain undeterred by any ongoing modification, unlike that in the existing lock-free implementations of internal BSTs [15, 12].

In all the existing designs of lock-free BSTs, when an operation fails at a link connected to an external node because of a concurrent modify operation, it retries from the scratch i.e. it restarts the operation from the root of the tree, after helping the obstructing concurrent operation. In our design an operation restarts from a node at the vicinity of the link where it fails, after the required helping. To achieve that, we need to get hold of the appropriate node(s) to restart at. For that, we use a *backlink* per node and ensure that it points to a node *present* in the tree from where the failure spot is a single link away. It should be noted that a backlink is not used for the tree traversal, see Fig. 2(a).

In designing an internal BST in a concurrent setup, the most difficult part is to perform an error-free REMOVE of a binary node. To remove a binary node we replace it with its predecessor, and hence, the incoming and outgoing links of the predecessor also need to be modified in addition to the incoming links of the node itself. According to the number of links needed to be modified in order to remove a node, unlike traditional categorization of nodes of a BST into leaf, unary and binary, we categorize them into three categories as shown in Fig. 2(b). The categorization characteristic is the origin of the threaded incoming link into the node. We call this link the *order-link*. Nodes belonging to category 1 are those whose order-link emanates from themselves; for a category 2 node, it emanates from its left-child; and for a category 3 node the incoming order-link emanates from the rightmost node in its left-subtree. We call the node where the order-link emanates from, the *order-node*. Note that, the order-node of a category 1 node is the node itself, whereas for category 2 and category 3 nodes it is its predecessor.

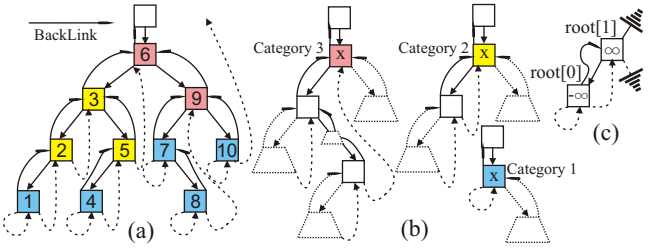

Figure 2: (a) Threaded BST with backlinks (b) Categorization of nodes for REMOVE (c) An empty tree.

To remove a node of category 1, only the incoming parent-link needs to be modified to connect to the node pointed by the right-link. For a category 2 node, the parent-link is updated to connect to the node pointed by the left-link and the order-link is modified to point to the node which the right-link was pointing to. In order to remove a category 3 node, its predecessor replaces it and the incoming and outgoing links of the predecessor are updated to take the values of that of the removed node. Parent-link of the predecessor is connected to the node which its left-link was pointing to before it got shifted. Also, when a link is updated the thread indicator value of the link, which it updates to, is copied to it. Please note that in this categorization, a unary or a binary node whose left-child is left-unary (i.e. whose right-child is null) or a leaf, gets classified in to category 2. Category 1 includes conventional leaf and right-unary nodes (whose left-child is null).

Having categorized the nodes as above we describe the modifications of links associated with a node undergoing REMOVE. Storing an indicator bit in a pointer has been used in many previous papers [13, 11, 22, 18]. We use a similar technique as in [11]. Depending on the category, before swapping the pointers associated with a node we flag the incoming pointers to the node and its predecessor (for category 3 only) and mark the outgoing pointers from the same. Note that because a backlink is neither used for traversal nor for injecting a modify operation, we do not need to mark/flag it. Flagging ensures that an operation does not have to travel a long *chain* of backlinks for recovery from failure. Once a link is flagged or marked it can not be a *point of injection* of a *new* ADD or REMOVE. However, in the conflict between a REMOVE of a category 3 node, and therefore shifting its predecessor, and a concurrent REMOVE of the predecessor node itself, we give priority to the former. Also to guarantee a single pointer travelling for recovery from failure due to a concurrent modify operation, we use a *prelink* at each node. It connects a node to its order-node before any of the outgoing pointers are marked. The flagging and marking are performed in such a predetermined order so that a malformed structure of the BST is avoided and the required priorities between conflicting operations are ensured. The flag-mark order of links for a category 3 node is as following - (I) flag the incoming order-link, (II) set the prelink, (III) mark the outgoing right-link, (IV) flag the parent-link of the predecessor incoming to that, (V) flag the incoming parent-link, (VI) mark the outgoing left-link and finally (VII) mark the outgoing left-link of the predecessor. For a node belonging to category 1 or 2, because there is no node between its order-node and itself, steps (IV), (VI) and (VII) do not happen. Having performed the flagging and marking we update

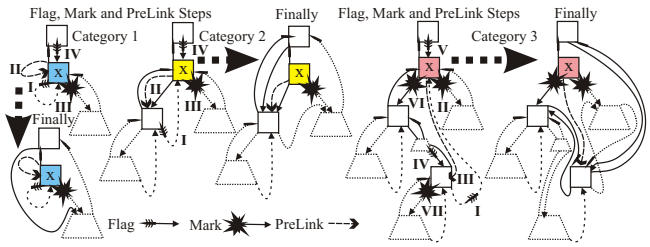

Figure 3: Remove steps of nodes

the flagged links according to the category of the node as described before. See Fig. 3.

Because we follow orderly modifications of the links, it never allows a node to be missed by a traversal in the BST unless both its incoming links are pointed away. However, because a node may shift "upward" to replace its successor, the interval associated with the order-link of its successor may shift "rightward" i.e. to the right subtree of the node after the successor is removed. Therefore, in order to terminate a traversal correctly, we use the stopping criterion given in Condition 1. It follows from the fact that the threaded left-link of a node is connected to the node itself and a traversal in the BST uses the order in the equivalent list.

CONDITION **1.** *Let k be the search key and k_{curr} be the key of the current node in the search path. If $(k = k_{curr})$ then stop. Else, if the next link is a threaded left-link then stop. Else, if the next link is a threaded right-link and the next node has key k_{next} then check if $k < k_{next}$. If true then stop, else continue.*

This stopping criterion not only solves the problem of synchronization between a concurrent REMOVE and a traversal for a predecessor query but also enables to achieve bound on the length of the traversal path. We shall explain that in section 4. A similar stopping criterion is used in [8] for conventional doubly-threaded BSTs.

We can observe that in our BST design, two modify operations that need to change two disjoint memory-words have significantly improved conditions for progress. For example, to REMOVE a category 2 node the left-link of its predecessor is never marked or flagged, therefore when such a node goes under REMOVE, a concurrent injection of ADD or REMOVE at the predecessor is possible. Also, because of the orderly flagging and marking of the links in more than one atomic steps, even after REMOVE of the node has commenced, a link at it that comes late in the order of flagging-marking can be modified for a concurrent ADD or REMOVE of another node. These progress conditions are not possible in the existing algorithms that use "node holding" [10, 15]. It shows that our algorithm has improved disjoint-access-parallelism.

3.2 The Implementation

We consider our concurrent setup as a shared memory machine in which processes are fully asynchronous and arbitrary failure halting is allowed. The read and write of a single memory-word is guaranteed to be atomic. The system provides atomic single-word `CAS` primitives. `CAS`(R, old, new) returns `true`, if ($old = R$), after updating R to new, else it returns `false` without any update. We steal three bits from a pointer, for (a) the thread indicator, (b) the mark-bit and (c) the flag-bit, see line 3. This is easily possible in high level languages such as C/C++. To prevent ABA problem in the implementation the standard technique of embedding a version tag using few bits in a pointer can be used. We also assume that the algorithm is implemented with a non-blocking safe memory reclamation scheme.

A typical node x in our BST implementation is represented by a structure consisting of five memory-words corresponding to - (a) a key k, (b) array $child[2]$ containing two child pointers : $child[0] := left(x)$ and $child[1] := right(x)$, (c) a $backLink$ and (d) a $preLink$. The bit sequence corresponding to boolean variables overlapping three stolen bits from a link is represented as $(f,\ m,\ t)$. One or more bits together can be set or unset using a single atomic `CAS` operation over the pointer containing it. The node structure is shown in lines 1 to 6. We use two dummy nodes as global variables represented by a two member array of node-pointers called $root$. The keys $-\infty$ and ∞ are stored in the two members of $root$ and they can never be deleted. Node $root[0]$ is left-child and predecessor of the node $root[1]$, see line 7 and figure 2 (c).

3.2.1 Locating a Node

The set operations - CONTAINS, ADD and REMOVE, need to perform a predecessor query using a given key k to locate an interval $[k_i,\ k_j]$, s.t. either $\{k_i \leq k < k_j\}$ or $\{k_i \geq k > k_j\}$, where $x(k_i)$ and $x(k_j)$ are two nodes in the BST. The function LOCATE is used for that, which starts from a specified set of two consecutive nodes {prev, curr} and follows the symmetric order of the internal BST to terminate at such a location $\{x(k_{prev}),\ x(k_{curr})\}$. The return value of LOCATE can be 0, 1 or 2 depending on whether the key k is less than, greater than or equal to the key k_{curr} at the termination point, line 10. If $k_{curr} \neq k$ then the desired interval is associated with the threaded outgoing link from $x(k_{curr})$ in the direction indicated by return value of LOCATE - 0 denotes left and 1 denotes right. The termination criterion for LOCATE implements Condition 1.

As we mentioned before, we can make a traversal eagerly help pending REMOVE operations, even though it is not obstructed by them, in the situations in which proportion of REMOVE increases. If that is done then a traversal cleans a node whose marked right-link it encounters during execution by calling the function CLEANMARK. This functionality can be enabled using a boolean variable as an input argument to every Set operation which is further passed to the LOCATE that it performs, see lines 14 to 20. To perform a CONTAINS operation for the key k, we start from the location $\{x(\infty),\ x(-\infty)\}$ represented by the global variables. Having performed LOCATE, the return value itself indicates whether the key k was present in the BST at the point of termination of LOCATE. If LOCATE returns 2 then CONTAINS returns `true` otherwise `false`, line 26 to 30.

3.2.2 Remove operation

To REMOVE a node $x(k)$, starting from $\{x(\infty),\ x(-\infty)\}$ we locate the link that the interval containing the key $(k-\epsilon)$ associates with, see line 33. If the LOCATE terminates at $\{x(k_{prev}),\ x(k_{curr})\}$ then $suc(x(k_{curr}))$ is the desired node to remove if k matches with its key. $suc(x(k_{curr}))$ is the node pointed by the threaded right-link of $x(k_{curr})$ and this link is indeed the order-link of $x(k)$. If $x(k)$ is located then we try to flag its order-link using TRYFLAG, line 37, in order to perform the step (I) of REMOVE.

```
1  struct Node {
2    KType k;
     // Three booleans overlap with three unused bits of the
     pointer ref.  left=child[0], right=child[1].
3    {NPtr ref, bool flag, bool mark, bool thread} child[2];
4    NPtr backLink;
5    NPtr preLink;
6  } *NPtr;
   // Global variable with the fixed value of members.
7  Node root[2] =
   {{−∞, {(&root[0], 0, 0, 1), (&root[1], 0, 0, 1)}, &root[1], null},
   {∞, {(&root[0], 0, 0, 0), (null, 0, 0, 1)}, null, null}};

   // LOCATE returns 2 if key is found otherwise 0 or 1 if the
   last visited link is left or right respectively.
8  int LOCATE(NPtr& prev, NPtr& curr, KType k)
9  while true do
       // Function cmp returns 2 := equal, 1 := greater than
       and 0 := less than
10     dir = cmp(k, curr→e);
11     if (dir = 2) then return dir;
12     else
13       (R, ∗, m, t) = curr→child[dir];
         /* If eager helping is required then help cleaning
         the current node if its right child pointer has been
         marked.  */
14       if ((m = 1) and (dir = 1)) then
15         newPrev = prev→backLink;
16         CLEANMARKED(prev, curr, dir);
17         prev = newPrev;
18         pDir =cmp(k, prev→k);
19         curr = (prev→child[pDir])·ref;
20         continue;
21       if t then
           // Check the stopping criterion.
22         nextE = R→k;
23         if ((dir = 0) or (k < nextE)) then
24           return dir;
25         else prev = curr; curr = R;

26 bool CONTAINS(KType k)
   // Initialize the location variables with the global
   variables.
27 prev = &root[1]; curr = &root[0];
28 dir = LOCATE(prev, curr, k);
29 if (dir = 2) then return true;
30 else return false;

31 bool REMOVE(KType k)
   // Initialize the location variables as before.
32 prev = &root[1]; curr = &root[0];
33 dir = LOCATE(prev, curr, k− ϵ);          // locate order-link
34 (next, f, ∗, t) = curr→child[dir];
35 if (k ≠ next→k) then return false;
36 else
     // flag the order-link
37   result = TRYFLAG(curr, next, prev, true);
38   if (prev→child[dir].ref = curr) then
39     CLEANFLAG(curr, next, prev, true);
40 return result;
```

```
41 bool TRYFLAG(NPtr& prev, NPtr& curr, NPtr& back, bool
   isThread)
42 while true do
43   pDir = cmp(curr→k, prev→k) & 1;  // If cmp returns 2 then
     curr is the left link of prev; so pDir is changed to 0.
44   t = isThread;
45   result = CAS(prev→child[pDir], (curr, 0, 0, t), (curr, 1, 0, t));
46   if result then return true;
47   else
       /* The CAS fails, check if the link has been marked,
       flagged or the curr node got deleted.  If flagged,
       return false; if marked, first clean then proceed */
48     (newR, f, m, t) = prev→child[pDir];
49     if (newR = curr) then
50       if f then return false;
51       else if m then
52         CLEANMARKED(prev, pDir);
53       prev = back;
54       pDir =cmp(curr→k, prev→k);
55       newCurr = (prev→child[newPDir])·ref;
         LOCATE(prev, newCurr, curr→k);
56       if (newCurr ≠curr ) then
57         return false;
58       back = prev→backLink;

59 void TRYMARK(NPtr& curr, int dir)
60 while true do
61   back = curr→backLink;
62   (next, f, m, t) = curr→child[dir];
63   if (m == 1) then break;
64   else if (f == 1) then
65     if (t == 0)  then
66       CLEANFLAG(curr, next, back, false); continue;
67     else if ((t == 1) and (dir == 1)) then
68       CLEANFLAG(curr, next, back, true); continue;
69   result = CAS(curr→child[dir], (next, 0, 0, t), (next,
     0, 1, t));        // Try atomically marking the child link.
70   if result then break;
```

TRYFLAG, line 41 to 58, returns true only if the operation performing it could successfully flag the desired link at line 45. If the CAS step to atomically flag the link fails then it checks the reason of failure. If it fails because some other thread already had successfully flagged the link, false is returned, line 50. If it fails because the link was marked then it first helps the operation that marked the link, line 51. It could also fail to flag a threaded link because of an ADD of a new node. In both the latter cases it moves a step back and locates the node whose parent-link was desired to be flagged. The address for starting the recovery is saved in the variable back. If the node is not located then it implies that some other thread already removed the target node and so TRYFLAG just returns false, line 57.

Having performed the TRYFLAG, REMOVE checks whether the target node is still there, line 38. If the target node is found then REMOVE goes to clean the flag at the order-link of the target node, using the function CLEANFLAG, lines 71 to 117. The function CLEANFLAG is also used to clean the flag of the parent-link of a node, and therefore a boolean variable is passed to it to inform about the thread-bit of the flagged link. If the link is an order-link then for all categories of nodes, the next step is to set the prelink and then mark the right-link, see lines 72 to 87. If the CAS to mark the right-link fails because of its flagging, and if the right-link is threaded then it indicates that the node itself is being shifted to replace its successor by a concurrent REMOVE operation. We give priority to the shifting operation and therefore before proceeding it is helped. As before, we need to recover from failures and so the node back stores the address to restart the recovery from. Here one has to be careful that if back was pointed at the node under REMOVE then we need to change it before going to help, see lines 77 and 81. Having marked the right-link, REMOVE proceeds to take further steps in the function CLEANMARK. Also, before marking the right-link, the prelink gets pointed to the correct order-node, line 85.

If CLEANFLAG is performed on a link that is not threaded then the link must be the incoming parent-link of a node or of its predecessor. In that case it is determined whether the link is the parent-link of a node under REMOVE or that of its predecessor by checking the flag, mark and thread bits of the right-link. The right-link of a node being removed is marked, whereas that for the predecessor of the same is

```
71 void CleanFlag(NPtr& prev, NPtr& curr, NPtr& back, bool
   isThread)
72 if (isThread ) then
       // Cleaning a flagged order-link
73     while true do
           // To mark the right-child link
74         (next, f, m, t) = curr→child[1];
75         if (m) then break;
76         else if (f) then
               // Help cleaning if right-child is flagged
77             if (back = next) then
                   // If back is getting deleted then move back.
78                 back = back→backLink;
79             backNode = curr→backLink;
80             CleanFlag(curr, next, backNode, t);
81             if (back = next) then
82                 pDir =cmp(prev→k, backNode→k);
83                 prev = back→child[pDir];
84         else
85             if (curr→preLink ≠ prev) then curr→preLink =
               prev;
86             result = CAS(curr→child[1], (next, 0, 0, t),
               (next, 0, 1, t)) ;   // Try marking the right-link.
87             if result then break;
88     CleanMark(curr, 1);
89 else                              // Cleaning a flagged parent-link
90     (right, rF, rM, rT) = curr→child[1];
91     if (rM) then                 // The node is to be deleted itself
92         (left, lF, lM, lT) = curr→child[0];
93         preNode = curr→preLink;
94         if (left ≠ preNode) then // A category 3 node
95             TryMark(curr, 0);
96             CleanMark(curr, 0);
97         else                     // This is a category 1 or 2 node
98             pDir =cmp(curr→k,prev→k);
99             if (left = curr) then          // A category 1 node
100                CAS(prev→child[pDir], (curr
                   , f, 0, 0), (right, 0, 0, rT));
101                if (!rT) then CAS(right→backLink, (curr
                   , 0, 0, 0), (prev, 0, 0, 0));
102            else                           // A category 2 node
103                CAS(preNode→child[1], (curr
                   , 1, 0, 1), (right, 0, 0, rT));
104                if (!rT) then CAS(right→backLink, (curr
                   , 0, 0, 0), (prev, 0, 0, 0));
105                CAS(prev→child[pDir], (curr
                   , 1, 0, 0), (preNode, 0, 0, rT));
106                CAS(preNode→backLink, (curr, 0, 0, 0), (prev
                   , 0, 0, 0));
107    else if (rt and rF) then
           // The node is moving to replace its successor
108        delNode = right;
109        while true do
110            parent = delNode→backLink;
111            pDir =cmp(curr→k, prev→k);
112            (∗, pF, pM, pT) = parent→child[pDir];
113            if (pM) then CleanMark(parent, pDir);
114            else if (pF) then break;
115            else if (CAS(parent→child[pDir], (curr
               , 0, 0, 0), (curr, 1, 0, 0))) then
116                break;
117        backNode = parent→backLink;
118        CleanFlag(parent, curr, backNode, true);
```

always flagged and threaded. Accordingly, either step (V) (lines 107 to 117) or step (VI) (lines 94) are performed. To perform the step (VI) the outgoing left-link of a category 3 node is marked, and for that the function TryMark is used. TryMark atomically sets the mark-bit at a link specified by its direction outgoing from a node, see lines 59 to 70. Note that, if the CAS to atomically mark a link fails due to a concurrent flagging of the link then TryMark helps the concurrent operation that flagged the link, except in the

```
119 void CleanMark(NPtr& curr, int markDir)
120 (left, lF, lM, lT) = curr→child[0];
121 (right, rF, rM, rT) = curr→child[1];
122 if (markDir == 1) then
        /* The node is getting deleted itself.  if it is
        category 1 or 2 node, flag the incoming parent link; if
        it is a category 3 node, flag the incoming parent link
        of its predecessor.  */
123     while true do
            preNode = delNode→preLink;
125         if (preNode == left) then // Category 1 or 2 node.
126             parent = delNode→backLink;
127             back = parent→backLink;
128             TryFlag(parent, curr, back, true);
129             if (parent→child[pDir].ref == curr) then
130                 CleanFlag(parent, curr, back, true);
131                 break;
132         else
                // Category 3 node.
133             preParent = preNode→backLink;
134             (∗, pPF, pPM, pPT) = preParent→child[1];
135             backNode = preParent→backLink;
136             if (pM) then CleanMark(preParent, 1);
137             else if (pF) then
138                 CleanFlag(preParent, preNode, backNode, true
                    );break;
139             else if (CAS(parent→child[pDir], (curr
                    , 0, 0, 0), (curr, 1, 0, 0))) then
140                 CleanFlag(preParent, preNode, backNode, true
                    );break;
141 else
        // The node is getting deleted (†) or moved to replace
        its successor (‡).
142     if (rM) then
            // (†) clean its left marked link.
143         preNode = curr→preLink;
144         TryMark(preNode, 0);
145         CleanMark(preNode, 0);
146     else if (rt and rF) then
            // (‡) change links accordingly
147         delNode = right;
148         delNodePa = delNode→backLink;
149         preParent = curr→backLink;
150         pDir =cmp(delNode→k, delNodePa→k);
151         (delNodeL, ∗, ∗, dlT) = delNode→child[0];
152         (delNodeR, ∗, ∗, drT) = delNode→child[1];
153         CAS(preParent→child[1], (curr
            , lF, 0, 0), (left, lF, 0, lT));
154         if (!lT) then CAS(left→backLink, (curr, 0, 0, 0),
            (preParent, 0, 0, 0));
155         CAS(curr→child[0], (left, 0, 1, lT), (delNodeL, 0, 0, 0));
156         CAS(delNodeL→backLink, (delNode, 0, 0, 0), (curr
            , 0, 0, 0));
157         CAS(curr→child[1], (right, 1, 0, 1), (delNodeR, 0, 0, drT));
158         if (!drT) then CAS(delNodeR→backLink,
            (delNode, 0, 0, 0), (curr, 0, 0, 0));
159         CAS(delNodePa→child[pDir], (delNode, 1, 0, 0), (curr
            , 0, 0, 0));
160         CAS(curr→backLink, (preParent, 0, 0, 0),
            (delNodePa, 0, 0, 0));
```

case when the link is a threaded left-link. This is because a flagged and threaded left-link of a node indicates that the node is a category 1 node under Remove and we give priority to the operation that must have already flagged its right link to shift it.

Because the last step for category 1 and category 2 nodes is flagging their parent-link before the pointers are appropriately swapped, CleanFlag also includes the final swapping of pointers for such nodes. Having determined that a node belongs to category 1 or 2 by checking the equality of left-node and order-node at line 97, it performs the pointer swapping for category 1 nodes in lines 100 and 101. Pointer

swapping steps to clean a category 2 node is shown between lines 103 and 106.

The CleanMark function is used to help an operation that sets a mark-bit at a link. Depending on whether the link is a left- or a right-link, it takes different steps. If the link is a right-link, for category 1 and 2 nodes, the flagging of the parent-link is executed; otherwise for category 3 nodes the parent-link of its predecessor is flagged. On failing to flag the parent-link of predecessor of a category 3 node, because a concurrent operation already marked the link, it helps that operation. However, after helping it may be that the node under Remove changes from a category 3 to a category 2 node, so its status is checked again, see lines 123 to 139.

A set mark-bit at a left-link indicates that it is either coming out from a category 3 node under Remove or its predecessor. In the first case, the step (VII) to mark the left-link of the predecessor of the node is performed, line 142. In the second case the pointer swapping steps of the category 3 nodes are performed, see lines 147 to 160.

Return value of Remove is that of the absolutely first TryFlag that it performs if the node was located.

3.2.3 Add operation

To Add a new data-node with key k in the BST, we Locate the target interval $[k_i,\ k_j]$ that k belongs to, associated with a threaded link. If Locate returns 2 then key is present in the BST and therefore Add returns false. If it returns 0 or 1 then we create a new node containing the key k. Its left-link is threaded and connected to itself and right-link takes the value of the link to which it needs to be connected, line 171. Note that when a new node is added, both its children links are threaded. Also, its backlink is pointed to the node $x(k_i)$. The link which it needs to connect to is modified in one atomic step to point to the new node using a `CAS`. If the `CAS` succeeds then true is returned. On failure, it is checked whether the target link was flagged, marked or another Add operation succeeded to insert a new node after we read the link. In case a new node is inserted, we start locating a new appropriate link starting with the location comprising nodes at the two ends of the changed link. On failure due to marking or flagging of the current link, the concurrent Remove operation is helped. Recovery from failure due to a flagging or marking by a concurrent Remove operation makes it to go one link back following the backlink of prev. After locating the new link, Add is retried.

```
161 bool Add(KType k)
162 prev = &root[1]; curr = &root[0];
    /* Initializing a new node with supplied key and left-link
    threaded and pointing to itself. */
163 node = new Node(k);
164 node→child[0] = (node, 0, 0, 1);
165 while true do
166     dir = Locate(prev, curr, k);
167     if (dir = 2) then // key exists in the BST
168         return false;
169     else
170         (R, *, *, *) = curr→child[dir];
            /* The located link is threaded. Set the right-link
            of the adding node to copy this value */
171         node→child[1] = (R, 0, 0, 1);
172         node→backLink = curr;
173         result = CAS(curr→child[dir], (R, 0, 0, 1),
            (node, 0, 0, 0));        // Try inserting the new node.
174         if result then return true;
175         else
                /* If the CAS fails, check if the link has been
                marked, flagged or a new node has been inserted.
                If marked or flagged, first help. */
176             (newR, f, m, t) = curr→child[dir];
177             if (newR = R) then
178                 newCurr = prev;
179                 if m then CleanMarked(curr, dir);
180                 else if f then
181                     CleanFlagged(curr, R, prev, true);
182                 curr = newCurr;
183                 prev = newCurr→backLink;
```

4. CORRECTNESS AND COMPLEXITY

4.1 Correctness

Here we present a proof-sketch of correctness and defer a detailed proof to [6] due to space constraints. First, we give classification of nodes in a BST Υ implemented by our algorithm.

Definition **1.** *A node $x \in \Upsilon$ is called logically removed if its right-link is marked and $\exists\ y \in \Upsilon$ s.t. either $left(y) = x$ or $right(y) = x$.*

Definition **2.** *A node $x \in \Upsilon$ is called physically removed if $\nexists\ y \in \Upsilon$ s.t. either $left(y) = x$ or $right(y) = x$.*

Definition **3.** *A node $x \in \Upsilon$ is called regular if it is neither logically removed nor physically removed.*

A node ever inserted in Υ has to fit in one of the above categories. At a point in the history of an execution, the BST formed by our algorithm contains elements stored in regular nodes or logically removed nodes. Before we show that the return values of the set operations are consistent with this definition according to their linearizability, we have to show that the operations work as intended and the structure formed by the nodes operated with these operations maintains a valid BST. We present some invariants maintained by the operations and the nodes in lemmas 1 to 9.

Lemma **1.** *If a* Locate(prev, curr, k) *returns dir and terminates at $[x(k_{prev}),\ x(k_{curr})]$ then*

(a) If $k_{curr} \neq k$ then the link $x(k_{curr})$→child[dir] is threaded.

(b) $x(k_{prev})$ and $x(k_{curr})$ are not physically deleted.

Lemma **2.** *A* Contains *operation returns true if and only if the key is located at a non-physically removed node.*

Lemma **3.** *An* Add *always happens at an unmarked and unflagged threaded link.*

Lemma **4.** *A* Remove *always starts by flagging the incoming order-link to a node.*

Lemma **5.** *When a node is added, both its left- and right-links are threaded.*

Lemma **6.** *Before a node is logically removed its incoming order-link is flagged and its prelink points to its correct order-node.*

Lemma **7.** *Backlink of a node always points to a node which is a single link away.*

Lemma **8.** *An unthreaded left-link and a right-link can not be both flagged and marked.*

Lemma **9.** *If a node gets logically removed then eventually it will be physically removed.*

Lemma 1 follows from the lines 11 and 21. Contains returns true only if Locate returns 2 and that happens only

if curr is non-physically removed at line 10 during its execution, this proves lemma 2. In case the key of a node matches with the query key, ADD returns false, otherwise it tries adding the new node using an atomic CAS. If the CAS fails, it always uses LOCATE to find the desired link before retrying the CAS to add the new node. From this observation and using 1(a), lemma 3 follows. By lemma 1 if $x(k)$ is present in the tree then LOCATE(prev, curr, $(k - \epsilon)$) will always terminate at a location such that curr is order-node of $x(k)$ and that establishes lemma 4. Lemma 5 follows from lines 164 and 171. Line 153 ensures that even if the function CLEANFLAG helps a pending REMOVE, before it could successfully mark the right-link at line 86, the flag that was put on the order-link at line 45 is copied to the new order-link. Also, the line 85 inside the while loop ensures that prelink is always set to the order-node. That proves the correctness of lemma 6. When a node is added its backlink is pointed to curr at line 172. Before a REMOVE operation returns, the backlinks of the predecessor, left-child and right-child, if present for the node under REMOVE, are updated at lines 101, 104, 106, 154, 156, 158 and 160 only after the incoming parent-links of those nodes are updated. Hence, lemma 7 is proved. In our algorithm we always use an atomic CAS to set a flag or mark-bit in a pointer. Whenever a CAS fails we check the possible reason. The function TRYMARK on failing to mark a link because it is flagged, helps cleaning the flag in all cases except when the link has direction 0 and it is threaded, line 67. These observations prove lemma 8. Lemma 8 proves that once the right-link of a node is marked, it can not be flagged. By lemmas 3 and 4, a new ADD or REMOVE will be obstructed at this link. Therefore, if the thread invoking REMOVE to mark it becomes faulty then eventually another thread invoking a possible ADD or REMOVE, which gets obstructed, will help to complete the physical removal. That proves lemma 9. Having proved the lemmas listed above, it is trivial to observe that whenever a pointer is dereferenced it is not null. And, by the initialization of the global variable to ∞ and $-\infty$, at line 7, the two starting nodes are never deleted.

Hence, we state proposition 1 whose proof will follow by the above stated lemmas and the fact that a thread always takes a correct "turn" during traversal according to the symmetric order of the internal BST.

PROPOSITION **1.** *The union of the regular and logically removed nodes operated under the set operations in the algorithm Efficient Lock Free BST maintains a valid internal Binary Search Tree.*

An execution history in our implementation may consist of ADD, REMOVE and CONTAINS operations. We present the linearization point of the execution of these operations. Proving that a history consisting of concurrent executions of these operations is legal will be ordering these linearization points. The linearization points of the operations are as following:

ADD - For a successful ADD operation, execution of the CAS at line 173 will be the linearization point. For an unsuccessful one the linearization point will be at line 10 where a key in the tree is found matched.

REMOVE - For a successful REMOVE operation the linearization point will be the successful CAS that swaps the flagged parent link. For an unsuccessful one there may be two cases - (a) if the node is not located then it is treated as an unsuccessful CONTAINS and its linearization point will be accordingly (b) if the node is located but its order-link got flagged by another concurrent REMOVE then its linearization point is just after the linearization point of that REMOVE.

CONTAINS - Our algorithm varies according to the read-write load situation. In case we go for eager helping by a thread performing LOCATE, a successful CONTAINS shall always return a regular node. However, if we opt otherwise then a successful CONTAINS returns any non-physically removed node. In both situations a successful CONTAINS will be linearized at line 10. An unsuccessful one, if the node never existed in the BST, is linearized at the start point. And, if the node existed in the BST when the CONTAINS was invoked but got removed during its traversal by a concurrent REMOVE then the linearization point will be just after the linearization point of that REMOVE.

Following the linearization points as described above we have proposition 2:

PROPOSITION **2.** *The Set operations in the algorithm Efficient Lock Free BST are linearizable.*

4.2 Lock-Freedom

The lemmas 6, 8 and 9 imply the following lemma.

LEMMA **10.** *If* REMOVE(x) *and* REMOVE(y) *work concurrently on nodes x and y then without loss of generality*

- *(a) If x is a child of y and the link $[y, x]$ is flagged then* REMOVE(x) *finishes before* REMOVE(y)*; otherwise if this link is marked,* REMOVE(y) *finishes before* REMOVE(x).
- *(b) If x is the predecessor of y and the order-links of both x and y have been successfully flagged then* REMOVE(y) *finishes before* REMOVE(x)*; otherwise if x has been logically deleted then* REMOVE(x) *finishes before the order-link of y could be successfully flagged.*
- *(c) If x is the left-child of the predecessor of y and the link $[pre(y), x]$ is flagged then* REMOVE(x) *finishes before* REMOVE(y)*; otherwise if this link is marked,* REMOVE(y) *finishes before* REMOVE(x).
- *(d) In all other cases* REMOVE(x) *and* REMOVE(y) *do not obstruct each other.*

By the description of our algorithm, a non-faulty thread performing CONTAINS will always return unless its search path keeps on getting longer forever. If that happens, an infinite number of ADD operations would have successfully completed adding new nodes making the implementation lock-free. So, it will suffice to prove that the modify operations are lock-free. Suppose that a thread t performs an operation *op* on a BST Υ and takes infinite steps and no other modify operation completes after that. Now, if no modify operation completes then Υ remains unchanged forcing t to retract every time it wants to execute its own modification step on Υ. This is possible only if every time *op* finds its injection point flagged or marked. This implies that a REMOVE operation is pending. It is easy to observe in the function ADD that if it gets obstructed by a concurrent REMOVE then before retrying after recovery from failure it helps the pending REMOVE by executing all the remaining steps of that. Also from lemma 10, whenever two REMOVE operations obstruct each other, one finishes before the other. It implies that whenever two modify operations obstruct each other one finishes before the other and so Υ changes. It

is contrary to our assumption. Hence, by contradiction we show that no non-faulty thread shall remain taking infinite steps if no other non-faulty thread is making progress. This proves the proposition 4.

PROPOSITION **3.** *Lock-freedom is guaranteed in the algorithm Efficient Lock Free BST.*

4.3 Complexity

Having proved that our algorithm guarantees lock-freedom, though we can not compute worst-case time complexity of an operation, we can definitely derive their amortized complexity. We derive the amortized step complexity of our implementation by the accounting method along the similar lines as in [11, 19].

In our algorithm, an ADD operation does not have to hold any pointer and so does not obstruct an operation for itself. We observe that after flagging the order-link of a node, a REMOVE operation op_r takes only a constant number of steps to flag, mark and swap pointers connected to the node and to its predecessor(if any) in addition to setting the prelink of the node under REMOVE. Also using lemma 7 and the fact that to access the predecessor of a node its prelink is used, recovery from failure due to a concurrent REMOVE operation needs only a constant number of links to traverse. Therefore, following lemma follows

LEMMA **11.** *An obstructing operation op makes an obstructed operation op' take only a constant number of extra steps for recovery from failure in order to finish its execution.*

Now for an execution E, let $\mathcal{O}$ be the set of operations and let $\mathcal{S}$ be the set of steps taken by all $op \in \mathcal{O}$. Considering the *invocation point* $t_i(op)$ of op to be the time it reads the root, and *response point* $t_r(op)$ to be the time it reads or writes at the last link before it leaves the BST, its *interval contention* c_I is defined as the total number of operations whose execution overlaps the interval $[t_i(op),\ t_r(op)]$ [1]. We define a function $f : \mathcal{S} \mapsto \mathcal{O}$ such that if $f(s) = op$ then s is charged to the account of op and

(a) In case of no contention, all the essential steps s, representing read, write and `CAS` taken by an operation op is mapped to op by f.
(b) In case of contention, any failed `CAS` by an operation op is mapped by f to the operation op' whose successful `CAS` causes the failure.
(c) If an extra read is performed by a traversal due to an added node to the set of existing nodes by a concurrent ADD operation op then it is mapped by f to op.
(d) Any read, write or `CAS` step s taken by an operation op after the first failed `CAS` and before retrying at the same link i.e. during helping and recovery from failure is mapped by f to the operation op' that performed the successful `CAS` in order to make op help it, provided op' further does not help some op'' so that op helps op'' recursively. This includes resetting of prelink, if needed.
(e) In case of recursive helping, the extra steps by all the operations helping op is mapped by f to op.

Having identified the operations to map for a step, it is easy to observe that an operation op' to which a step s by an operation op is mapped, has its execution interval overlapping $[t_i(op),\ t_r(op)]$ and therefore op is counted in $c_I(op')$.

Next we bound the length of the traversal path for a predecessor query in our implementation. Note that, all the set operations have to perform a predecessor query by key k to LOCATE an interval $[k_i,\ k_j]$ associated with a link s.t. $x(k_i)$ and $x(k_j)$ are two nodes in the BST. Let us define the *access-node* of an interval as the node that the link which it associates with, emanates from. We define *distance* of an interval from an operation op as the number of links that op traverses from its current location (root $\forall\ t \leq t_i(op)$) to read the access-node of the interval. Suppose that at $t_i(op)$ there are n nodes in the BST Υ. Clearly, distance of any interval for op at $t_i(op)$ is $O(H(n))$. Now suppose that at $t_{ref}(op) \in [t_i(op),\ t_r(op)]$, $x(k_j)$ is a category 1 node and the distance of the interval $[k_i,\ k_j]$ associated with the order-link of $x(k_j)$ from op at $t_{ref}(op)$ is d. We can observe that if at $t \in [t_{ref}(op),\ t_r(op)]$, $x(k_j)$ is still a category 1 node and it is removed then the interval associated with its order-link gets subsumed by the interval associated with the order-link of the leftmost child in its right subtree or with the order-link emanating from its parent if the right subtree is `null`. In the former case the distance of $[k_i,\ k_j]$ from op becomes $d + ht(x(k_j))$ and in the latter it decreases by 1. Also if a node $x(k_l)$ is added by an operation op' then the extra distance except d traversed by op to access $[k_i,\ k_l]$ or $[k_l,\ k_j]$ is no more than c_I. Using the above observations, an inductive proof is derived to imply that op does not traverse more than $d + ht(x(k_j)) + c_I$ links to access a subinterval of $[k_i,\ k_j]$. Hence the lemma 12 follows whose detailed proof is deferred to [6].

LEMMA **12.** *If at $t_{ref}(op) \in [t_i(op),\ t_r(op)]$, $x(k_j)$ is a category 1 node and the distance of the interval $[k_i,\ k_j]$ associated with the order-link of $x(k_j)$ from op at $t_{ref}(op)$ is d then to access an interval $[k,\ k'] \subseteq [k_i,\ k_j]$ op traverses no more than $d + ht(x(k_j)) + c_I$ links.*

When op traverses in the left subtree of a category 3 node x and if x gets removed, the interval associated with its order-link gets subsumed by the interval associated with the order-link of the leftmost node in the right subtree of x which is a category 1 node. On removal of a category 2 node, the interval associated with its order-link is subsumed by the interval associated with the order-link of its parent which can be a category 2 or category 3 node. Because $(d + ht(x)) \leq 2H(n)\ \forall\ x \in \Upsilon$, lemma 12 along with these observations show that the path length of a traversal in our lock-free BST is bounded by $2H(n) + c_I$.

We have shown that (a) the traversal path is bounded by $O(H(n) + c_I)$, (b) a constant number of steps are needed for a modify operation after locating its target (node or link) and (c) to an operation op only a constant number of extra steps can be charged by any concurrent operation op' counted in c_I. Therefore, we can infer that the amortized step complexity for a set operation, which is the actual step complexity plus the number of steps charged by other operations minus the number of steps charged to other operations, in our lock-free BST algorithm is $O(H(n) + c_I)$. We also give choice of eager helping to operations according to the read-write load. Now if that happens, we do not need to count all the operations whose executions overlap the interval $[t_i(op),\ t_r(op)]$. We can then use a tighter notion of *point contention* c_P, which counts the maximum number of operations that execute concurrently at $t \in [t_i(op),\ t_r(op)]$ [2]. In that case, given the above discussion, along the similar lines

as presented in [11], we can show that for any execution E, the average amortized step complexity of a set operation op in our algorithm will be

$$\hat{t}_{op \in E} \in O\left(\frac{\sum_{op \in E}(H(n(op)) + c_P(op))}{|\{op \in E\}|}\right)$$

where $n(op)$ is the number of nodes in the BST at the point of invocation of op and $c_P(op)$ is its point contention during E. That concludes the amortized analysis of our algorithm with the following proposition.

PROPOSITION **4.** *In a BST with n nodes at the start of an execution, the amortized step complexity of each operation in the algorithm Efficient Lock Free BST is $O(H(n) + c)$, where c is the measure of contention during the execution.*

It is straightforward to observe that the number of memory-words used by a BST with n nodes in our design is $5n$.

5. CONCLUSION AND FUTURE WORK

In this paper we proposed a novel algorithm for the implementation of a lock-free internal BST. Using amortized analysis we proved that all the operations in our implementation run in time $O(H(n) + c)$. We solved the existing problem of "retry from scratch" for modify operations after failure caused by a concurrent modify operation, which resulted into an amortized step complexity of $O(cH(n))$. This improvement takes care of an algorithmic design issue for which the time complexity of modify operations increases dramatically with the increase in the contention and the size of the data-structure. This is an important improvement over the existing algorithms. Our algorithm also comes with improved disjoint-access-parallelism compared to similar lock-free BST algorithms. We also proposed a conservative helping technique which adapts to read-write load on the implementation. We proved the correctness showing linearizability and lock-freedom of the proposed algorithm.

We plan to thoroughly evaluate our algorithm experimentally vis-a-vis existing concurrent set implementations.

Acknowledgments

This work is partially supported by the Swedish Research Council under grant number 37252706 as part of the project SCHEME (www.scheme-project.org).

6. REFERENCES

[1] Y. Afek, G. Stupp, and D. Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002.

[2] H. Attiya and A. Fouren. Algorithms adapting to point contention. *Journal of the ACM*, 50(4):444–468, 2003.

[3] G. Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the 5th ACM SPAA*, pages 261–270, 1993.

[4] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM PPoPP*, pages 257–268, 2010.

[5] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM PPoPP*, pages 329–342, 2014.

[6] B. Chatterjee, N. Nguyen, and P. Tsigas. Efficient lock-free binary search trees. Technical Report 2014:05, ISNN 1652-926X, Department of Computer Science and Engineering, Chalmers University of Technology, 2014.

[7] T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. In *Proceedings of the 17th ACM PPoPP*, pages 161–170, 2012.

[8] D. Drachsler, M. Vechev, and E. Yahav. Practical concurrent binary search trees via logical ordering. In *Proceedings of the 19th ACM PPoPP*, pages 343–356, 2014.

[9] F. Ellen, P. Fatourou, J. Helga, and E. Rupert. The amortized complexity of non-blocking binary search trees. In *Proceedings of the 2013 ACM PODC*, 2014.

[10] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM PODC*, pages 131–140, 2010.

[11] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the 23rd ACM PODC*, pages 50–59, 2004.

[12] K. Fraser. *Practical lock-freedom*. PhD thesis, Cambridge University, Computer Laboratory, 2004.

[13] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Distributed Computing*, pages 300–314. Springer, 2001.

[14] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[15] S. V. Howley and J. Jones. A non-blocking internal binary search tree. In *Proceedinbgs of the 24th Annual ACM SPAA*, pages 161–171, 2012.

[16] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th ACM PODC*, pages 151–160, 1994.

[17] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th ACM SPAA*, pages 73–82, 2002.

[18] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM PPoPP*, pages 317–328, 2014.

[19] R. Oshman and N. Shavit. The skiptrie: low-depth concurrent search without rebalancing. In *Proceedings of the 2013 ACM PODC*, pages 23–32, 2013.

[20] A. J. Perlis and C. Thornton. Symbol manipulation by threaded lists. *Communications of the ACM*, 3(4):195–204, 1960.

[21] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5):609–627, 2005.

[22] H. Sundell and P. Tsigas. Lock-free and practical doubly linked list-based deques using single-word compare-and-swap. In *Principles of Distributed Systems*, pages 240–255. Springer, 2005.

[23] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank. Wait-free linked-lists. In R. Baldoni, P. Flocchini, and R. Binoy, editors, *Principles of Distributed Systems*, volume 7702 of *LNCS*, pages 330–344. Springer Berlin Heidelberg, 2012.