CASTLE: Fast Concurrent Internal Binary Search Tree using Edge-Based Locking

Arunmoezhi Ramachandran Neeraj Mittal

⟨arunmoezhi, neerajm⟩@utdallas.edu

Department of Computer Science

The University of Texas at Dallas

Abstract

We present a new *lock-based* algorithm for concurrent manipulation of a binary search tree in an asynchronous shared memory system that supports search, insert and delete operations. Some of the desirable characteristics of our algorithm are: (i) a search operation uses only read and write instructions, (ii) an insert operation does not acquire any locks, and (iii) a delete operation only needs to lock up to four edges in the absence of contention. Our experiments indicate that our lock-based algorithm outperforms existing algorithms for a concurrent binary search tree—blocking as well as non-blocking—for medium-sized and larger trees, achieving up to 59% higher throughput than the next best algorithm.

Keywords Concurrent Data Structure, Binary Search Tree, Internal Representation, Lock-Based Algorithm, Edge-Based Locking

1. Introduction

With the growing prevalence of multi-core, multi-processor systems, concurrent data structures are becoming increasingly important. In such a data structure, multiple processes may need to operate on the data structure at the same time. Contention between different processes must be managed in such a way that all operations complete correctly and leave the data structure in a valid state.

Concurrency is often managed using locks. A lock can be used to achieve mutual exclusion, which can then be used to ensure that any updates to the data structure or a portion of it are performed by one process at a time. This makes it easier to design a lock-based concurrent data structure and reason about its correctness. Moreover, this also makes it easier to implement a lock-based data structure and debug it than its lock-free counterpart. Lock-based algorithms for concurrent versions of many important data structures for storing and managing shared data have been developed including linked lists, queues, priority queues, hash tables and skiplists (*e.g.*, [8, 9, 12–14, 16]).

Binary search tree is one of the fundamental data structures for organizing *ordered* data that supports search, insert and delete operations [3]. A binary search tree may be unbalanced (different leaf nodes may be at very different depths) or balanced (all leaf nodes are at roughly the same depth). A balanced binary search tree provides better worst-case guarantees about the cost of performing an operation on the tree. However, in many cases, the overhead of keeping the tree balanced, especially in a concurrent environment, may incur significant overhead. As a result, in many cases, an unbalanced binary search tree outperforms a balanced binary search tree in practice. In this work, our focus is on developing an efficient concurrent *lock-based* algorithm for an *unbalanced* binary search tree.

Concurrent algorithms for unbalanced binary search trees have been proposed in [1, 2, 4–6, 11, 17]. Algorithms in [1, 4] are blocking (or lock -based), whereas those in [2, 5, 6, 11, 17] are non-blocking (or lock-free). Also, algorithms in [1, 2, 4, 11] use internal representation of a search tree in which all nodes store data, where as those in [5, 6, 17] use an external representation of a search tree in which only leaf nodes store data (data stored in internal nodes is used for routing purposes only).

Algorithms that use internal representation of a search tree have to address the problem that arises due to a key moving from one location in the tree to another. This occurs when the key undergoing deletion resides in a binary node, which requires it to be either replaced with its predecessor (next smallest key) or its successor (next largest key). As a result, an operation traversing the tree may fail to find the target key both at its old location and at its new location, even though the target key was continuously present in the tree. Different algorithms use different approaches to handle the problem arising due to key movement. The algorithm by Drachsler *et al.* in [4] maintains a *sorted* linked list of all the keys in the tree. If the traversal of the tree fails to find a given key, then an operation traverses the linked list to look for the key. The algorithm by Arbel and Hattiya [1] uses the RCU (Read-Copy-Update) framework (first employed in Linux kernels) to allow reads to occur concurrently with updates.

Contributions: We present a new *lock-based* algorithm for concurrent manipulation of a binary search tree in an asynchronous shared memory system that supports search, insert and delete operations. Our algorithm is based on an internal representation of a search tree as in [1, 4]. However, as in [17], it operates at edge-level (locks edges) rather than at node-level (locks nodes); this minimizes the contention window of a write operation and improves the system throughput. Further, in our algorithm, (i) a search operation uses only read and write instructions, (ii) an insert operation does not acquire any locks, and (iii) a delete operation only needs to lock up to four edges in the absence of contention. Our experiments indicate that our lock-based algorithm outperforms existing algorithms for a concurrent binary search tree—blocking as well as non-blocking—for medium-sized and larger trees, achieving up to 59% higher throughput than the next best algorithm.

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Roadmap: The rest of the paper is organized as follows. We describe our system model in Section 2. Our lock-based algorithm for a binary search tree is described in Section 3. We present the results of our experimental evaluation of different concurrent algorithms for a binary search tree in Section 4. Finally, Section 5 concludes the paper and outlines directions for future research.

2. System Model

2.1 Binary Search Tree

We assume that a binary search tree (BST) implements a dictionary abstract data type and supports *search*, *insert* and *delete* operations. For convenience, we refer to insert and delete operations as *update* operations. A search operation explores the tree for a given key and returns true if the key is present in the tree and false otherwise. An insert operation adds a given key to the tree if the key is not already present in the tree. Duplicate keys are not allowed in our model. A delete operation removes a key from the tree if the key is indeed present in the tree. In both cases, a modify operation returns true if it changed the set of keys present in the tree (added or removed a key) and false otherwise.

A binary search tree satisfies the following properties: (a) the left subtree of a node contains only nodes with keys less than the node's key, (b) the right subtree of a node contains only nodes with keys greater than or equal to the node's key, and (c) the left and right subtrees of a node are also binary search trees.

2.2 Synchronization Primitives

Our algorithm uses *exclusive* locks. We assume that a lock is (i) safe: it satisfies the mutual exclusion property, *i.e.*, at most one process can hold the lock at any time, and (ii) live: it satisfies the deadlock freedom property, *i.e.*, if the lock is free and one or more processes attempt to acquire the lock, then some process is eventually able to acquire the lock.

In addition to locks, we also use *compare-and-swap (CAS)* atomic instruction. which takes three arguments: $address$, old and new. It compares the contents of a memory location ($address$) to a given value (old) and, only if they are the same, modifies the contents of that location to a given new value (new).

2.3 Correctness Properties

To demonstrate the correctness of our algorithm, we use *linearizability* [10] for the safety property and *deadlock-freedom* [9] for the liveness property. Broadly speaking, linearizability requires that an operation should appear to take effect instantaneously at some point during its execution. Deadlock-freedom requires that some process with a pending operation be able to complete its operation eventually.

3. The Lock-Based Algorithm

We first provide an overview of our algorithm. We then describe the algorithm in more detail and also give its pseudo-code. For ease of exposition, we describe our algorithm assuming no memory reclamation, which can be performed using the well-known technique of hazard pointers [15].

3.1 Overview of the Algorithm

Every operation in our algorithm uses *seek* function as a subroutine. The seek function traverses the tree from the root node until it either finds the target key or reaches a non-binary node whose next edge to be followed points to a null node. We refer to the path traversed by the operation during the seek as the *access-path*, and the last node in the access-path as the *terminal node*. The operation then compares the target key with the stored key (the key present in the terminal node). Depending on the result of the comparison and the type of the operation, the operation either terminates or moves to the execution phase. In certain cases in which a key may have moved upward along the access-path, the seek function may have to restart and traverse the tree again; details about restarting are provided later. We now describe the next steps for each of the type of operation one-by-one.

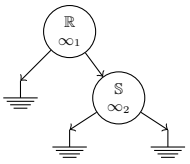

Figure 1: Sentinel keys and nodes ($\infty_1 < \infty_2$)

Search: A search operation starts by invoking seek operation. It returns true if the stored key matches the target key and false otherwise.

Insert: An insert operation starts by invoking seek operation. It returns false if the target key matches the stored key; otherwise, it moves to the execution phase. In the execution phase, it attempts to insert the key into the tree as a child node of the last node in the access-path using a CAS instruction. If the instruction succeeds, then the operation returns true; otherwise, it restarts by invoking the seek function again.

Delete: A delete operation starts by invoking seek function. It returns false if the stored key does not match the target key; otherwise, it moves to the execution phase. In the execution phase, it attempts to remove the key stored in the terminal node of the access-path. There are two cases depending on whether the terminal node is a binary node (has two children) or not (has at most one child). In the first case, the operation is referred to as *complex delete operation*. In the second case, it is referred to as *simple delete operation*. In the case of simple delete (shown in Figure 4), the terminal node is removed by changing the pointer at the parent node of the terminal node. In the case of complex delete (shown in Figure 5), the key to be deleted is replaced with the *next largest* key in the tree, which will be stored in the *leftmost node* of the *right subtree* of the terminal node.

3.2 Details of the Algorithm

As in most algorithms, to make it easier to handle special cases, we use sentinel keys and sentinel nodes. The structure of an empty tree with only sentinel keys (denoted by ∞_1 and ∞_2 with $\infty_1 < \infty_2$) and sentinel nodes (denoted by $\mathbb{R}$ and $\mathbb{S}$) is shown in Figure 1.

Our algorithm, like the one in [17], operates at edge level. A delete operation obtains ownership of the edges it needs to work on by locking them. To enable locking of an edge, we steal a bit from the child addresses of a node referred to as *lock-flag*. We also steal another bit from the child addresses of a node to indicate that the node is undergoing deletion and will be removed from the tree. We denote this bit by *mark-flag*. Finally, to avoid the ABA problem, as in Howley and Jones [11], we use *unique* null pointers. To that end, we steal yet another bit from the child address, referred to as *null-flag*, and use it to indicate whether the address points to a null or a non-null address. So, when an address changes from a non-null value to a null value, we only set the null-flag and the contents of the address are not otherwise modified. This ensures that all null pointers are unique.

We next describe the details of the seek operation, which is executed by all operations (search as well as modify) after which we describe the details of the execution phase of insert and delete operations.

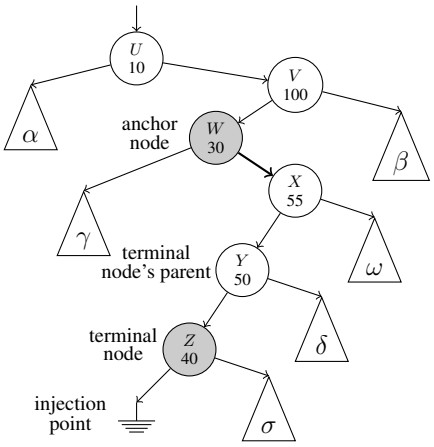

Figure 2: Nodes in the access path of seek.

3.2.1 The Seek Phase

A seek function keeps track of the node in the access-path at which it took the last "right turn" (*i.e.*, it last followed a right edge). Let this "right turn" node be referred to as *anchor node* when the traversal reaches the terminal node. Note that the terminal node is the node whose key matched the target key or whose next child edge is set to a null address. For an illustration, please see Figure 2. In the latter case (stored key does not match the target key), it is possible that the key may have moved up in the tree. To ascertain that the seek function did not miss the key because it may have moved up during the traversal, we use the following set of conditions that are *sufficient* (but not necessary) to guarantee that the seek function did not miss the key. First, the anchor node is still part of the tree. (For an illustration, see Figure 6.) Second, the key stored in the anchor node has not changed since it first encountered the anchor node during the (current) traversal. To check for the above two conditions, we determine whether the anchor node is undergoing deletion by examining it right child edge. We discuss the two cases one-by-one.

(a) *Right child edge not marked:* In this case, the anchor node is still part of the tree. We next check whether the key stored in the anchor node has changed. If the key has not changed, then the seek function returns the results of the (current) traversal, which consists of three addresses: (i) the address of the terminal node, (ii) the address of its parent, and (iii) the null address stored in the child field of the terminal node that caused the traversal to terminate. The last address is required to ensure that an insert operation works correctly (specifically to ascertain that the child field of the terminal node has not undergone any change since the completion of the traversal). We refer to it as the *injection point* of the insert operation. On the other hand, if the key has changed, then the seek function restarts from the root of the tree. A possible optimization is that the seek function restarts only if the target key is now less than the anchor node's key.

(b) *Right child edge marked:* In this case, we compare the information gathered in the current traversal about the anchor node with that in the previous traversal, if one exists. Specifically, if the anchor node of the previous traversal is same as that of the current traversal and the keys found in the anchor node in the two traversals also match, then the seek function terminates, but returns the results of the previous traversal (instead of that of the current traversal). This is because the anchor node was definitely part of the tree during the previous traversal since it was reachable from the root of the tree at the beginning of the current traversal. Otherwise, the seek function restarts from the root of the tree.

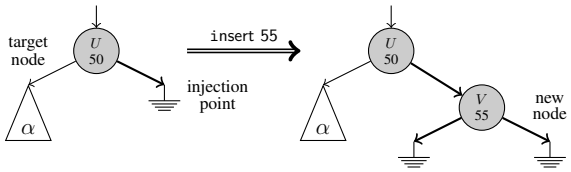

Figure 3: An illustration of an insert operation.

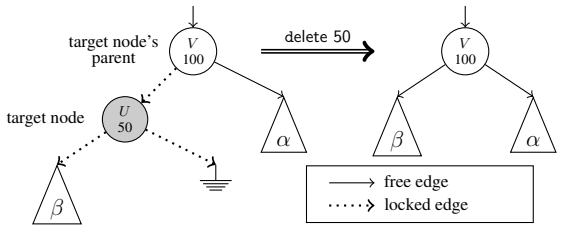

Figure 4: An illustration of a simple delete operation.

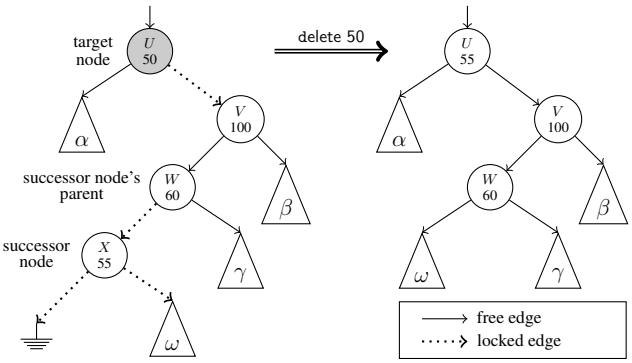

Figure 5: An illustration of a complex delete operation.

For insert and delete operations, we refer to the terminal node as the *target node*.

3.2.2 The Execution Phase of an Insert Operation

In the execution phase, an insert operation creates a new node containing the target key. It then adds the new node to the tree at the injection point using a CAS instruction. If the CAS instruction succeeds, then (the new node becomes a part of the tree and) the operation terminates; otherwise, the operation restarts from the seek phase. Note that the insert operations are lock-free.

3.2.3 The Execution Phase of a Delete Operation

The execution of a delete operation starts by checking if the target node is a binary node or not. If it is a binary node, then the delete operation is classified as complex; otherwise it is classified as simple.

For a tree node X, let $X.parent$ denote its parent node, and $X.left$ and $X.right$ denote its left and right child node, respectively. Also, hereafter in this section, let T denote the target node of the delete operation under consideration.

(a) *Simple Delete:* In this case, either $T.left$ or $T.right$ is pointing to a null node. Note that both $T.left$ and $T.right$ may be

pointing to null nodes in which case T will be a leaf node. Without loss of generality, assume that $T.right$ is a null node. The removal of T involves locking the following three edges: $\langle T.parent, T\rangle$, $\langle T, T.left\rangle$ and $\langle T, T.right\rangle$. For an illustration, see Figure 4.

A lock on an edge is obtained by setting the lock-flag in the appropriate child field of the parent node using a CAS instruction. For example, to lock the edge $\langle X, Y\rangle$, where Y is the left child of X, the lock-flag in the left child of X is set to one. If all the edges are locked successfully, then the operation validates that the key stored in the target node still matches the target key. If the validation succeeds, then the operation marks both the children edges of T to indicate that T is going to be removed from the tree. Next, it changes the child pointer at $T.parent$ that is pointing to T to point to $T.left$ using a simple write instruction. Finally, the operation releases all the locks and returns true.

(b) *Complex Delete:* In this case, both $T.left$ and $T.right$ are pointing to non-null nodes. The operation locates the next largest key in the tree, which is the smallest key in the subtree rooted at the right child of T. We refer to this key as the *successor key* and the node storing this key as the *successor node*. Hereafter in this section, let S denote the successor node. Deletion of the key stored in T involves copying the key stored in S to T and then removing S from the tree. To that end, the following edges are locked by setting the lock-flag on the edge using a CAS instruction: $\langle T, T.right\rangle$, $\langle S.parent, S\rangle$, $\langle S, S.left\rangle$ and $\langle S, S.right\rangle$. For an illustration, see Figure 5. Note that the first two edges may be same which happens if the successor node is the right child of the target node. Also, since we do not lock the left edge of the target node, the left edge may change and may possibly start pointing to a null address. But, that does not impact the correctness of the complex delete operation.

If all the edges are locked successfully, then the operation validates that the key stored in the target node still matches the target key. If the validation succeeds, then the operation copies the key stored in S to T, and marks both the children edges of S to indicate that S is going to be removed from the tree. Next, it changes the child pointer at $S.parent$ that is pointing to S to point to $S.right$ using a simple write instruction. Finally, the operation releases all the locks and returns true.

In both cases (simple as well as complex delete), if the operation fails to obtain any of the locks, then it releases all the locks it was able to acquire up to that point, and restarts from the seek phase. Also, after obtaining all the locks, if the key validation fails, then it implies that some other delete operation has removed the key from the tree while the current execution phase was in progress. In that case, the given delete operation releases all the locks, and simply returns false. Note that using a CAS instruction for setting the lock-flag also enables us to *validate that the child pointer has not changed* since it was last observed in a single step.

3.3 Formal Description

We refer to our algorithm as CASTLE (Concurrent Algorithm for Binary Search Tree by Locking Edges).

A pseudo-code of our algorithm is given in Algorithms 1-7. Different data structures used in our algorithm are shown in Algorithm 1. Besides tree node, we use three additional records: (a) *seek record:* to store the outcome of a tree traversal both when looking for the target key and the successor key, (b) *anchor record:* to store information about the anchor node during the seek phase, and (c) *lock record:* to store information about a tree edge that needs to be locked.

The pseudo-code for the seek function is shown in Algorithm 2. The pseudo-codes for search, insert and delete operations are shown in Algorithm 3, Algorithm 4 and Algorithm 5, respectively. Algorithm 6 contains the pseudo-code for locking and unlocking a set of tree edges, as specified in an array. Finally, Algorithm 7 contains the pseudo-codes for three helper functions used by a delete operation, namely: (a) CLEARFLAGS: to clear lock and mark flags from a child field, (b) FINDSMALLEST: to locate the smallest key in a subtree, and (c) REMOVECHILD: to remove a given child of a node.

In the pseudo-code, to improve clarity, we sometimes use subscripts l, m and n to denote lock, mark and null flags, respectively.

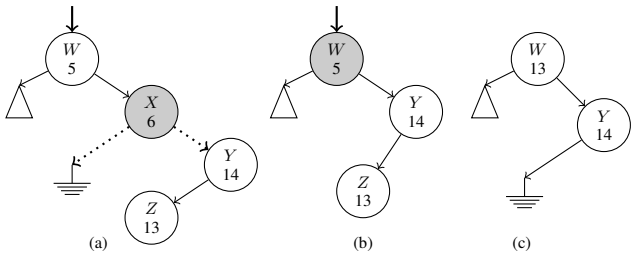

Figure 6: A scenario in which the last right turn node is no longer part of the tree. Search (13) gets stalled at Y in (a). Its last right turn node is X. Delete (6) removes X from the tree in (b). The key stored in X is still 6. Delete (5) results in 13 moving up the tree from Z to W in (c). When search (13) wakes up, it will miss 13.

```
   // a tree node
1  struct Node {
2      Key key;
3      { boolean, boolean, boolean, NodePtr } child[2];
       // each child field contains four subfields: lFlag,
          mFlag, nFlag and address
4  };
   // used to store the results of a tree traversal
5  struct SeekRecord {
6      NodePtr node;
7      NodePtr parent;
8      NodePtr nullAddress;
9  };
   // used to store information about an anchor node
10 struct AnchorRecord {
11     NodePtr node;
12     Key key;
13 };
   // used to store information about an edge to lock
14 struct LockRecord {
15     NodePtr node;
16     enum { LEFT, RIGHT } which;
17     { boolean, NodePtr } addressSeen;
       // addressSeen contains two subfields: nFlag and
          address
18 };
   // local seek record used when looking for a node
19 SeekRecordPtr seekTargetKey, seekSuccessorKey;
   // local array used to store the set of edges to lock
20 LockRecord lockArray[4];
```

Algorithm 1: Data Structures Used

3.4 Correctness Proof

It is convenient to treat insert and delete operations that do not change the tree as search operations. We call a tree node *active* if it is reachable from the root of the tree. We call a tree node *passive* if it was active earlier but is not active any more. Note that,

```
21 SEEK( key, seekRecord )
22 begin
23     while true do
           // initialize the variables used in the traversal
24         pNode := ℝ;      cNode := 𝕊;
25         address := 𝕊 → child[RIGHT].address;
26         anchorRecord := {ℝ, ∞1};
27         while true do
               // reached terminal node; read the key stored in
                  the current node
28             cKey := cNode → key;
29             if key = cKey then
30                 seekRecord := {cNode, pNode, address};
31                 return;
32             which := key < cKey ? LEFT : RIGHT;
               // read the next address to dereference along
                  with mark and null flags
33             ⟨∗, ∗, nFlag, address⟩ := cNode → child[which];
34             if nFlag then
                   // the null flag is set; reached terminal
                      node
35                 aNode := anchorRecord → node;
36                 if aNode → child[RIGHT].mFlag then
                       // the anchor node is marked; it may no
                          longer be part of the tree
37                     if anchorRecord = pAnchorRecord then
                           // the anchor record of the current
                              traversal matches that of the
                              previous traversal
38                         seekRecord := pSeekRecord;
39                         return;
40                     else break;
41                 else
                       // the anchor node is definitely part of
                          the tree
42                     if aNode → key < key then
                           // seek can terminate now
43                         seekRecord := {cNode, pNode, address};
44                         return;
45                     else break;
               // update the anchor record if needed
46             if which = RIGHT then
                   // the next edge to be traversed is a right
                      edge; keep track of current node and its
                      key
47                 anchorRecord := {cNode, cKey};
               // traverse the next edge
48             pNode := cNode;      cNode : =address;
```

Algorithm 2: Seek Function

```
49 boolean SEARCH( key )
50 begin
51     SEEK( key, seekTargetKey );
52     node := seekTargetKey → node;
53     if node → key = key then
54         return true;                                  // key found
55     else
56         return false;                             // key not found
```

Algorithm 3: Search Operation

before an active node is made passive by a delete operation, both its children edges are *marked*. Also, a CAS instruction performed on an edge (by either an insert operation or a delete operation as part of locking) is successful only if the edge is unmarked. As a result,

```
57 boolean INSERT( key )
58 begin
59     while true do
60         SEEK( key, seekTargetKey );
61         node := seekTargetKey → node;
62         if node → key = key then
63             return false;                                 // key found
64         else
               // key not found; add the key to the tree
65             newNode := create a new node;
               // initialize its fields
66             newNode → key := key;
67             newNode → child[LEFT] := ⟨0l, 0m, 1n, null⟩;
68             newNode → child[RIGHT] := ⟨0l, 0m, 1n, null⟩;
               // determine which child field (left or right)
                  needs to be modified
69             which := key < node → key ? LEFT : RIGHT;
               // fetch the address observed by the seek
                  function in that field
70             address := seekTargetKey → nullAddress;
71             result := CAS( node → child[which],
                              ⟨0l, 0m, 1n, address⟩,
                              ⟨0l, 0m, 0n, newNode⟩ );
72             if result then
                   // new key successfully added to the tree
73                 return true;
```

Algorithm 4: Insert Operation

clearly, if an insert operation completes successfully, then its target node was active when its edge was modified to make the new node (containing the target key) a part of the tree. Likewise, if a delete operation completes successfully, then all the nodes involved in the operation (up to three nodes) were active when their edges were locked.

3.4.1 All Executions are Linearizable

We show that an arbitrary execution of our algorithm is linearizable by specifying the *linearization point* of each operation. Note that the linearization point of an operation is the point during its execution at which the operation appeared to have taken effect. Our algorithm supports three types of operations: search, insert and delete. We now specify the linearization point of each operation.

1. *Insert operation:* The operation is linearized at the point at which it performed the successful CAS instruction that resulted in its target key becoming part of the tree.
2. *Delete operation:* There are two cases depending on whether the delete operation is simple or complex. If the operation is simple delete, then the operation is linearized at the point at which a successful write step was performed at the parent of the target node that resulted in the target node becoming passive. Otherwise, it is linearized at the point at which the original key of the target node was replaced with its successor key.
3. *Search operation:* There are two cases depending on whether the target node was active when the operation read the key stored in the node. If the target node was not active, then the operation is linearized at the point at which the target node became passive. Otherwise, it is linearized at the point at which the read step was performed.

It can be easily verified that, for any execution of the algorithm, the sequence of operations obtained by ordering operations based on their linearization points is legal, *i.e.*, all operations in the sequence satisfy their specification.

```
74  boolean DELETE( key )
75  begin
76      while true do
77          SEEK( key, seekTargetKey );
78          node := seekTargetKey→node;
79          if node→key ≠ key then
80              return false;                          // key not found
81          else
                // key found; remove the key from the tree
                // read the contents of the target node's
                   children fields into local variables
82              lField := CLEARFLAGS( node→child[LEFT] );
83              rField := CLEARFLAGS( node→child[RIGHT] );
84              if lField.nFlag or rField.nFlag then
                    // simple delete operation
                    // determine the edges to be locked
85                  parent := seekTargetKey→parent;
86                  if key < parent→key then which := LEFT;
87                  else which := RIGHT;
88                  lockArray[0] := {parent, which, ⟨0, node⟩};
89                  lockArray[1] := {node, LEFT, lField};
90                  lockArray[2] := {node, RIGHT, rField};
91                  result := LOCKALL( lockArray, 3 );
92                  if result then
                        // all locks acquired; perform the
                           operation
93                      if node→key = key then
                            // key still matches; remove the node
94                          REMOVECHILD( parent, which );
95                          match := true;
96                      else match := false;
97                      UNLOCKALL( lockArray, 3 );
98                      return match;
99              else
                    // complex delete operation
                    // locate the successor node
100                 FINDSMALLEST( node, rField.address,
                                  seekSuccessorKey );
                    // fetch its information from the seek record
101                 sNode := seekSuccessorKey→node;
102                 sParent := seekSuccessorKey→parent;
                    // determine the edges to be locked
103                 lockArray[0] := {node, RIGHT, rField};
104                 if node ≠ sParent then
                        // the successor node is not the right
                           child of the target node
105                     lockArray[1] := {sParent, LEFT, ⟨0, sNode⟩};
106                     size := 4;
107                 else size := 3 ;
108                 lField := CLEARFLAGS( sNode→child[LEFT] );
109                 rField := CLEARFLAGS( sNode→child[RIGHT] );
110                 lockArray[size − 2] := {sNode, LEFT, lField};
111                 lockArray[size − 1] := {sNode, RIGHT, rField};
112                 result := LOCKALL( lockArray, size );
113                 if result then
                        // all locks acquired; perform the
                           operation
114                     if node→key = key then
                            // key still matches; copy the key in
                               the successor node to the target
                               node
115                         node→key := sNode→key;
116                         REMOVECHILD( sParent, LEFT );
117                         match := true;
118                     else match := false;
119                     UNLOCKALL( lockArray, size );
120                     return match;
```

Algorithm 5: Delete Operation

```
121 boolean LOCKALL( lockArray, size )
122 begin
123     for i ← 0 to size − 1 do
            // acquire lock for the i-th entry
124         node := lockArray[i].node;
125         which := lockArray[i].which;
126         lockedAddress := lockArray[i].addressSeen;
127         lockedAddress.lFlag := true;
            // set the lock flag in the child edge
128         result := CAS( node→child[which],
                           lockArray[i].addressSeen,
                           lockedAddress );
129         if not (result) then
                // release all the locks acquired so far
130             UNLOCKALL( lockArray, i − 1 );
131             return false;
132     return true;

133 UNLOCKALL( lockArray, size )
134 begin
135     for i ← size − 1 to 0 do
136         node := lockArray[i].node;
137         which := lockArray[i].which;
            // clear the lock flag in the child edge
138         node→child[which].lFlag := false;
```

Algorithm 6: Lock and Unlock Functions

Thus we have:

THEOREM 1. *Every execution of our algorithm is linearizable.*

3.4.2 All Executions are Deadlock-Free

We say that the system is in a *quiescent state* if no modify operation completes hereafter. We say that the system is in a *potent state* if it has one or more pending modify operations. Note that quiescence is a *stable property*; once the system is in a quiescent state, it stays in a quiescent state. We show that our algorithm is deadlock-free by proving that a potent state is necessarily non-quiescent.

Note that, in a quiescent state, no edges in the tree can be marked. This is because a delete operation marks edges only after it has successfully obtained all the locks, after which it is guaranteed to complete. This also implies that the tree cannot undergo any changes now because that would imply eventual completion of a modify operation. Thus, once a system has reached a quiescent state, all modify operation currently pending repeatedly alternate between seek and execution phases. We say that the system is in a *strongly-quiescent state* if all pending modify operations started their most recent seek phase *after* the system became quiescent. Note that, like quiescence, strong quiescence is also a stable property. Now, once the system has reached a strongly quiescent state, the following can be easily verified. First, for a given modify operation, every traversal of the tree in the seek phase returns the same target node. Second, for a given delete operation, the set of edges it needs to lock remains the same.

Now, assume that the system eventually reaches a state that is both potent and quiescent. Clearly, from this state, the system will eventually reach a state that is potent and strongly-quiescent. Note that a delete operation in our algorithm locks edges in a *top-down*, *left-right* manner. As a result, there cannot be a "cycle" involving delete operations. If a delete operation continues to fail in the execution phase, then it is necessarily because it tried to acquire lock on an already locked edge. (Recall that the set of edges does not change any more and there are no marked edges in the tree.) We can construct a chain of operations such that each operation in the chain tried to lock an edge already locked by the next operation in the chain. Clearly, the length of the chain is bounded. This implies

```
139 word CLEARFLAGS( word field )
140 begin
141     newField := field with lock and mark flags cleared;
142     return newField;

143 FINDSMALLEST( parent, node, seekRecord )
144 begin
        // initialize the variables used in the traversal
145     pNode := parent;      cNode := node;
146     while true do
147         ⟨∗, ∗, nFlag, address⟩ := cNode → child[LEFT];
148         if not (nFlag) then
                // visit the next node
149             pNode := cNode;      cNode := address;
150         else
                // reached the successor node
151             seekRecord := {cNode, pNode, address};
152             break;

153 REMOVECHILD( parent, which )
154 hu begin
        // determine the address of the child to be removed
155     node := parent → child[which];
        // mark both the children edges of the node to be
           removed
156     node → child[LEFT].mFlag := true;
157     node → child[RIGHT].mFlag := true;
        // determine whether both the child pointers of the
           node to be removed are null
158     if (node → child[LEFT].nFlag and
            node → child[RIGHT].nFlag) then
            // set the null flag only
159         parent → child[which].nFlag := true;
160     else
            // switch the pointer at the parent to point to its
               appropriate grandchild
161         if node → child[LEFT].nFlag then
162             address := node → child[RIGHT].address;
163         else address := node → child[LEFT].address ;
164         parent → child[which].address := address;
```

Algorithm 7: Helper Functions used by Delete Operation

Table 1: Comparison of different concurrent algorithms in the absence of contention.

Algorithm	Number of Objects Allocated		Number of Synchronization Primitives Executed	
	Insert	Delete	Insert	Delete
LF-IBST	2	1	3	simple: 4 complex: 9
LF-EBST	2	0	1	3
CASTLE	1	0	1	simple: 3 complex: 4

that the last operation in the chain is guaranteed to obtain all the locks and will eventually complete. This contradicts the fact that the system is in a quiescent state.

Thus, we have:

THEOREM 2. *Every execution of our algorithm is deadlock-free.*

4. Experimental Evaluation

We now describe the results of the comparative evaluation of different implementations of a concurrent binary search tree (BST) using simulated workloads.

4.1 Other concurrent Binary Search Tree Implementations

Besides CASTLE, we considered three other implementations of a concurrent BST for comparative evaluation, namely those based on: (i) the lock-free internal BST by Howley and Jones [11], denoted by LF-IBST, (ii) the lock-free external BST by Natarajan and Mittal [17], denoted by LF-EBST, and (iii) the RCU-based internal BST by Arbel and Attiya [1], denoted by CITRUS. Note that CITRUS is a blocking implementation. The above three implementations were obtained from their respective authors. All implementations were written in C/C++. In our experiments, none of the implementations used garbage collection to reclaim memory. The experimental evaluation in [11, 17] showed that, in all cases, either LF-IBST or LF-EBST outperformed the concurrent BST implementation based on Ellen *et al.*'s lock-free algorithm in [5]. So we did not consider it in our experiments.

To our knowledge, there is no other implementation of a concurrent (unbalanced) BST available in C/C++. Drachsler *et al.*'s algorithm has only Java-based implementation available, whereas no implementation is currently available for Chatterjee *et al.*'s algorithm [2].

4.2 Experimental Setup

We conducted our experiments on a single large-memory node in stampede cluster at TACC (Texas Advanced Computing Center) [**?**]. This node is a Dell PowerEdge R820 server with 4 Intel E5-4650 8-core processors (32 cores in total) and 1TB of DDR3 memory [19]. Hyper-threading has been disabled on the node. It runs CentOS 6.3 operating system. We used Intel C/C++ compiler (version 2013.2.146) with the optimization flag set to O3. We used GSL (GNU Scientific Library) to generate random numbers. To avoid memory allocation in a multi-threaded environment from becoming a bottleneck, we used Intel's *TBB Malloc* [18] as the dynamic memory allocator since it provides superior performance to the C/C+ default allocator in a multi-threaded environment (we observed up to two times better throughput).

To compare the performance of different implementations, we considered the following parameters:

1. **Maximum Tree Size:** This depends on the size of the key space. We considered three different key ranges: 50,000 (50K), 500,000 (500K) and 5 million (5M) keys.
2. **Relative Distribution of Operations:** We considered three different workload distributions: (a) *read-dominated:* 90% search, 9% insert and 1% delete, (b) *mixed:* 70% search, 20% insert and 10% delete, and (c) *write-dominated:* 0% search, 50% insert and 50% delete.
3. **Maximum Degree of Contention:** This depends on number of threads that can concurrently operate on the tree. We varied the number of threads from 1 to 32 in powers of two.

We compared the performance of different implementations with respect to *system throughput*, given by the number of operations executed per unit time.

4.3 Simulation Results

In each run of the experiment, we ran each implementation for two minutes, and calculated the total number of operations completed by the end of the run to determine the system throughput. The results were averaged over five runs. To capture only the steady state behaviour, we *pre-populated* the tree to 50% of its maximum size, prior to starting a simulation run. The beginning of each run consisted of a "warm-up" phase whose numbers were excluded in the computed statistics to avoid initial caching effects. The results of our experiments are shown in Figure 7 and Figure 8. In Figure 7, the absolute value of the system throughput is plotted against the

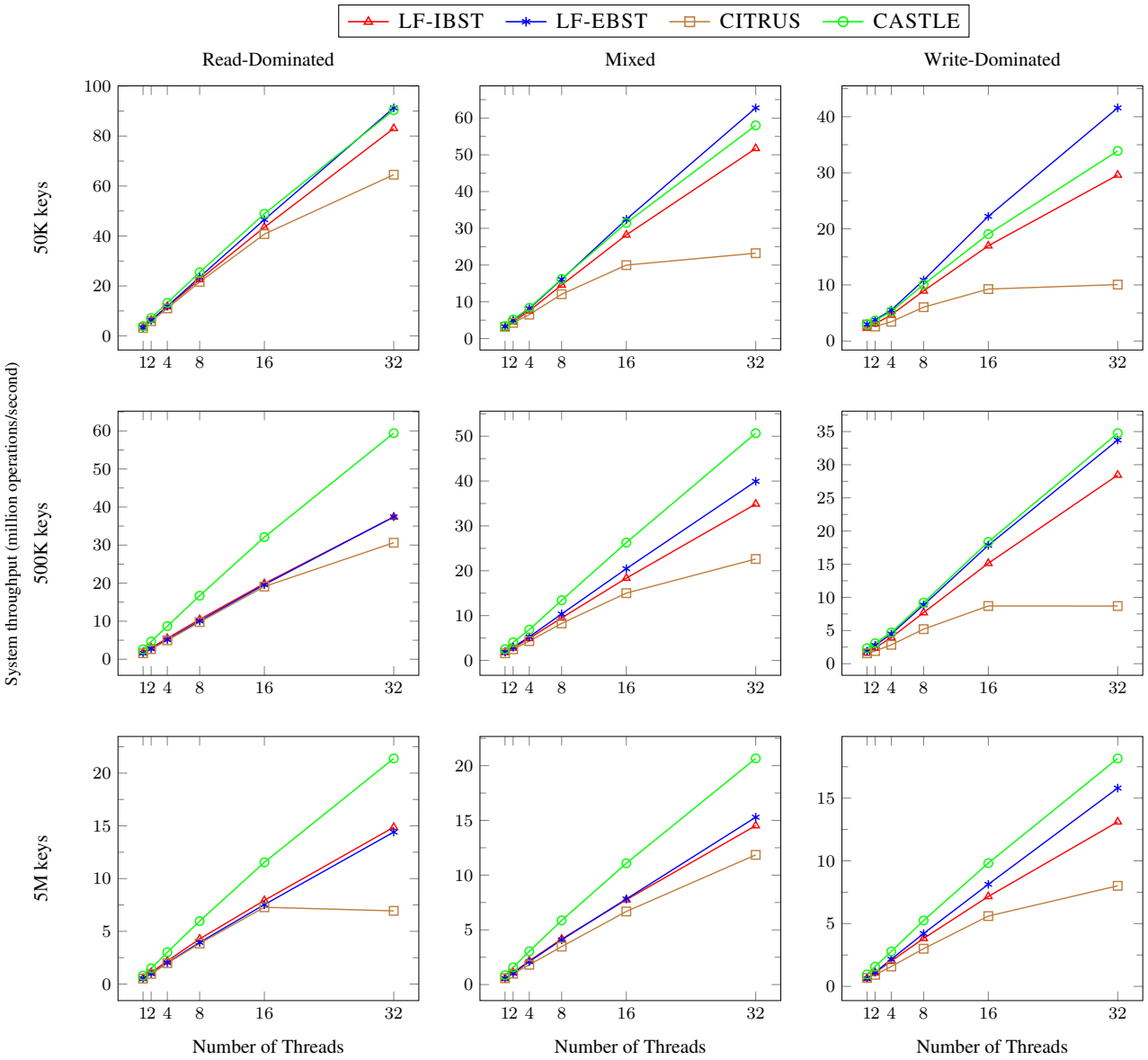

Figure 7: Comparison of system throughput of different algorithms. Each row represents a key space size and each column represents a workload type. Higher the throughput, better the performance of the algorithm.

number of threads (varying from 1 to 32 in powers of 2). Here each column represents a specific workload (read-dominated, mixed or write-dominated) and each row represents a specific key space size (50K, 500K or 5M). In Figure 8, the relative value of the system throughput with respect to that of LF-IBST is plotted against the key space size. Here each column represents a range of key space sizes (small, medium and large) and each row represents a specific workload. As the peak performance for all the four algorithms (for all cases) occurred at 32 threads, we set the number of threads to 32 while varying the key space size from 2^{13} to 2^{24}.

As both Figure 7 and Figure 8 show, for smaller key space sizes, LF-EBST achieves the best system throughput. This is not surprising since LF-EBST is optimized for high contention scenarios. For medium and large key space sizes, CASTLE achieves the best system throughput for all three workload types in almost all the cases (except when the workload is write-dominated and the key space size is in the lower half of the medium range). The maximum gap between CASTLE and the next best performer is around 59% which occurs at 500K key space size, read-dominated workload and 32 threads.

We also ran experiments on an an AMD based machine. The results are shown in Figure 9 and are similar to those for Stampede, which is an Intel based machine.

We believe that some of the reasons for the better performance of CASTLE over the other three concurrent algorithms, especially when the contention is relatively low, are as follows. First, as explained in [17], operating at edge-level rather than at node-level reduces the contention among operations. Second, using a CAS instruction for locking an edge also validates that the edge has not undergone any change since it was last observed. Third, locking edges as late as possible minimizes the blocking effect of the locks. Table 1 shows a comparison of LF-IBST, LF-EBST and CAS-

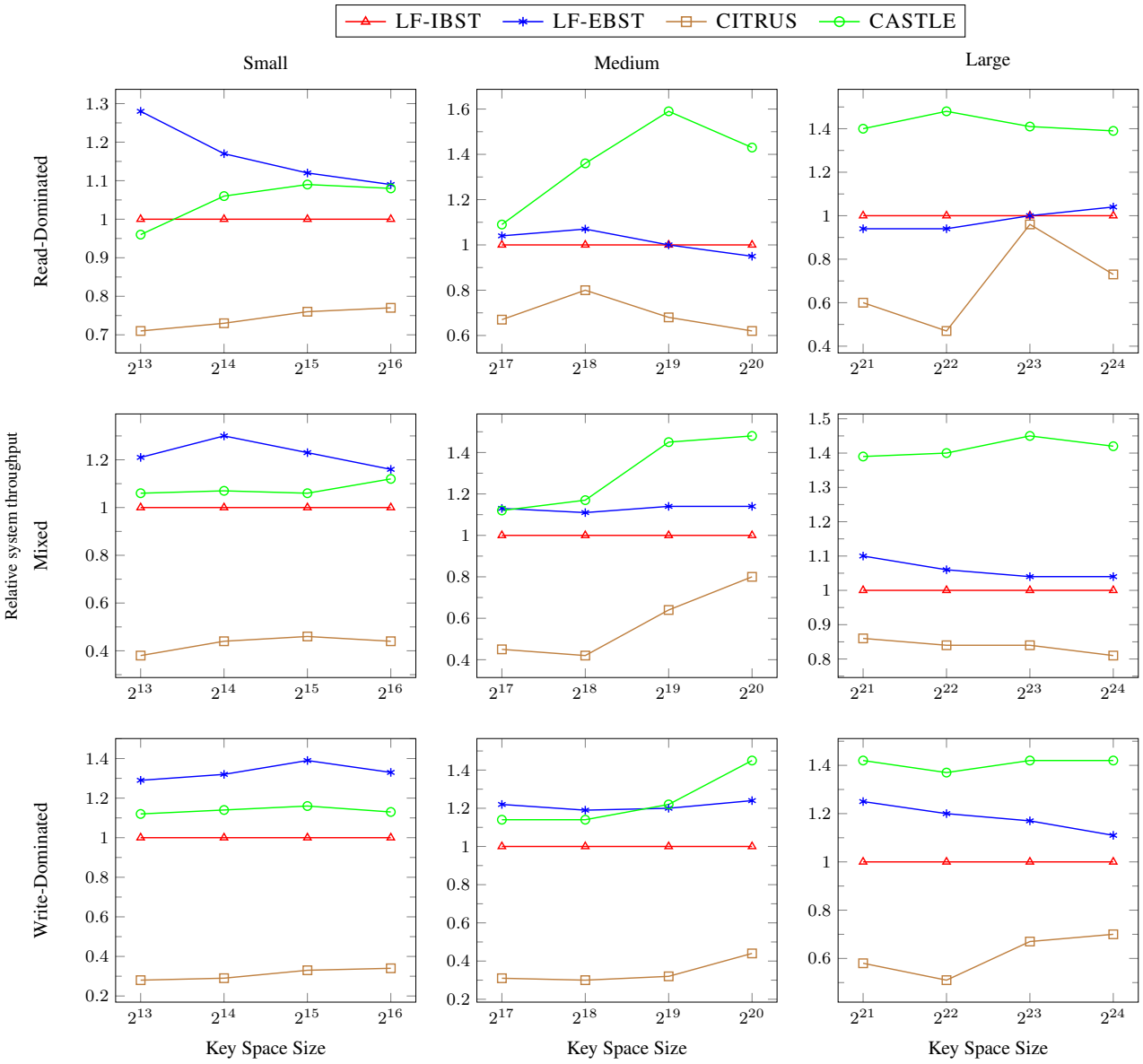

Figure 8: Comparison of system throughput of different algorithms *relative to that of* LF-IBST at 32 threads. Each row represents a workload type. Each column represents a range of key space size. Higher the ratio, better the performance of the algorithm.

TLE with respect to the number of objects allocated dynamically and the number of synchronization primitives executed per modify operation in the absence of contention. We omitted CITRUS in this comparison since it based on a different framework. As Table 1 shows, our algorithm allocates fewer objects dynamically than the two lock-free algorithms (one for insert operations and none for a delete operations). Further, again as Table 1 shows, our algorithm executes much fewer synchronization primitives than LF-IBST. It executes the same number of synchronization primitives as LF-EBST for insert and simple delete operations and only one more for complex delete operations. This is important since a synchronization primitive is usually much more expensive to execute than a simple read or write instruction. Finally, we observed in our experiments that CASTLE had a smaller memory footprint than all the three implementations (by a factor of two or more) since it uses internal representation of a search tree and allocates fewer objects dynamically. As a result, it was likely able to benefit from caching to a larger degree than the other algorithms.

In our experiments, we observed that, for key space sizes larger than 10K, the likelihood of an operation restarting was extremely low (less than 0.1%) even for a write-dominated workload. This implies that, in at least 99.9% of the cases, an operation was able to complete without encountering any conflicts. Thus, for key space sizes larger than 10K, we expect CASTLE to outperform the implementation based on the lock-free algorithm described in [6], which is basically derived from the one in [5].

5. Conclusion and Future Work

We have proposed a new concurrent lock-based algorithm for an unbalanced binary search tree. In contrast to other lock-based algorithms, it locks edges rather than nodes. This minimizes the contention window of an operation and improves the system through-

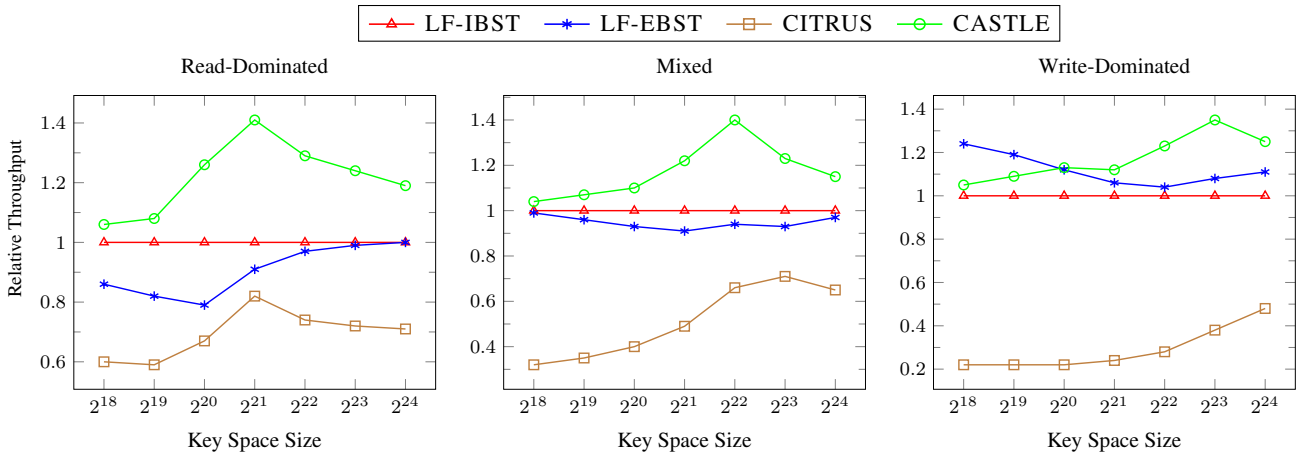

Figure 9: Comparison of system throughput of different algorithms *relative to that of* LF-IBST for various key space sizes with 64 threads on **AMD Opteron(TM) 6276 Processor** having 32 cores with 2 threads per core. We used **gcc 4.8.2 compiler** with **jemalloc** memory allocator. Each column represents a workload type. Higher the value, better the performance of the algorithm.

put. A desirable feature of our algorithm is that its search and insert operations are lock-free; they do not obtain any locks. This is important because, in most practical workloads, search and insert operations are likely to substantially outnumber delete operations.

As indicated by our experiments, our algorithm has the best performance—compared to other concurrent algorithms for a binary search—when the contention is relatively low. Specifically, it achieved the best performance for medium-sized and larger trees with mixed workloads and read-dominated workloads, beating the next best concurrent algorithm by as much as 59%. When the contention is relatively high (either the tree is small or the workload is write-dominated), the lock-free algorithm proposed by Natarajan and Mittal [17] usually achieved the best performance. Some of the reasons for the better performance of our algorithm are edge-level locking, lazy locking, fewer synchronization primitives and smaller memory footprint.

As future work, we plan to use the ideas in this work to develop efficient concurrent algorithms for other types of search trees such as k-ary search trees and suffix trees.

References

[1] M. Arbel and H. Attiya. Concurrent Updates with RCU: Search Tree as an Example. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 196–205, July 2014.

[2] B. Chatterjee, N. N. Dang, and P. Tsigas. Efficient Lock-Free Binary Search Trees. In *Proceedings of the 34th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 321–331, 2014.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1991.

[4] D. Drachsler, M. Vechev, and E. Yahav. Practical Concurrent Binary Search Trees via Logical Ordering. In *Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 343–356, Feb. 2014.

[5] F. Ellen, P. Fataourou, E. Ruppert, and F. van Breugel. Non-Blocking Binary Search Trees. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 131–140, July 2010.

[6] F. Ellen, P. Fatourou, J. Helga, and E. Ruppert. The Amortized Complexity of Non-Blocking Binary Search Trees. In *Proceedings of the 34th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 332–340, 2014.

[7] J. Evans. A Scalable Concurrent malloc(3) Implementation for FreeBSD. In *Proceedings of the BSDCan Conference*, Ottawa, Canada, 2006.

[8] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. S. III, and N. Shavit. A Lazy Concurrent List-Based Set Algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS)*, pages 3–16, Pisa, Italy, 2005.

[9] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann, 2012.

[10] M. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, July 1990.

[11] S. V. Howley and J. Jones. A Non-Blocking Internal Binary Search Tree. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 161–171, June 2012.

[12] D. Lea. Java Community Process, JSR 166, Concurrent Utilities. `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`, 2003.

[13] Y. Lev, M. Herlihy, V. Luchangco, and N. Shavit. A Simple Optimistic Skiplist Algorithm. In *Proceedings of the 14th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 124–138, Castiglioncello, Italy, June 2007.

[14] M. M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-based Sets. In *Proceedings of the 14th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 73–82, 2002.

[15] M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 15(6):491–504, 2004.

[16] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, 1996.

[17] A. Natarajan and N. Mittal. Fast Concurrent Lock-Free Binary Search Trees. In *Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 317–328, Feb. 2014.

[18] J. Reindeers. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[19] STAMPEDE. Dell PowerEdge C8220 Cluster with Intel Xeon Phi Co-Processors. `https://www.tacc.utexas.edu/resources/hpc/stampede-technical`.