A Fast Lock-Free Internal Binary Search Tree

Arunmoezhi Ramachandran
arunmoezhi@utdallas.edu

Neeraj Mittal*
neerajm@utdallas.edu

Department of Computer Science
The University of Texas at Dallas
Richardson, Texas 75080, USA

ABSTRACT

We present a new *lock-free* algorithm for concurrent manipulation of a binary search tree in an asynchronous shared memory system that supports search, insert and delete operations. It combines ideas from two recently proposed lock-free algorithms: one of them provides good performance for a read-dominated workload and the other one for a write-dominated workload. Specifically, it uses internal representation of a search tree (as in the first one) and is based on marking edges instead of nodes (as in the second one). Our experiments indicate that our new lock-free algorithm outperforms other lock-free algorithms in most cases providing up to 35% improvement in some cases over the next best algorithm.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming-Parallel Programming; E.1 [**Data Structures**]: Trees; D.3.3 [**Language Constructs and Features**]: Concurrent Programming Structures

Keywords

Concurrent Data Structure, Lock-Free Algorithm, Binary Search Tree

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 overlapping regions of 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 most often managed through locks. However, locks are blocking; while a process is holding a lock, no other process can access the portion of the data structure protected by the lock. If a process stalls while it is holding a lock, then the lock may not be released for a long time. This may cause other processes to wait on the stalled process for extended periods of time. As a result, lock-based implementations of concurrent data structures are vulnerable to problems such as deadlock, priority inversion and convoying [8].

Non-blocking algorithms avoid the pitfalls of locks by using special (hardware-supported) *read-modify-write* instructions such as *load-link/store-conditional (LL/SC)* and *compare-and-swap (CAS)* [8]. Non-blocking implementations of many common data structures such as queues, stacks, linked lists, hash tables and search trees have been proposed (*e.g.*, [2–13]).

Binary search trees are one of the fundamental data structures for organizing and storing *ordered* data that support search, insert and delete operations.

Ellen *et al.* proposed the first practical lock-free algorithm for a concurrent binary search tree in [5]. Their algorithm uses an external (or leaf-oriented) search tree in which only the leaf nodes store the actual keys; keys stored at internal nodes are used for routing purposes only.

Howley and Jones proposed another lock-free algorithm for a concurrent binary search tree in [9]. Their algorithm uses an internal search tree in which both the leaf nodes as well as the internal nodes store the actual keys. As a result, the search tree in Howley and Jones' algorithm has a smaller memory footprint. However, delete operations in an internal search tree are generally slower than those in an external search tree. This is because a delete operation in the former may involve replacing the key being deleted with the largest key in the left sub-tree (or the smallest key in the right sub-tree), which increases the likelihood of contention among operations.

Natarajan and Mittal proposed a lock-free algorithm for an external binary search tree [12], which uses several ideas to reduce the contention among modify (insert and delete) operations such as: (a) marking edges rather than nodes for deletion, (b) not using a separate explicit object for enabling coordination among conflicting operations, and (c) allowing multiple keys being deleted to be removed from the tree in a single step. As a result, modify operations in their algorithm have a smaller contention window, allocate fewer objects and execute fewer atomic instructions than their counterparts in other lock-free algorithms.

Drachsler *et al.* proposed a lock-based internal binary search tree in which each node maintains pointers to its

*This work was supported, in part, by the National Science Foundation (NSF) under grant number CNS-1115733.

ICDCN '15, January 04 - 07 2015, Goa, India

http://dx.doi.org/10.1145/2684464.2684472.

logical predecessor and successor nodes (based on the key order), in addition to maintaining pointers to its left and right children in the tree [4]. Modify operations update this logical information each time they add or remove keys from the tree. Further, search operations that do not find the key after traversing the tree from the root node to a leaf node, traverse the logical chain induced by predecessor and successor pointers to handle the case in which the key may have "moved" to another node during the tree traversal.

More recently, Arbel and Attiya have proposed a lock-based internal binary search tree using RCU (Read-Copy-Update) framework [1]. In their algorithm, a delete operation that moves a key from one node to another is stalled until all the traversals of the tree currently in-progress have completed.

In most lock-free algorithms, if a modify operation encounters a conflicting operation in its "window", it restarts from the root of the tree after helping. Recently, Ellen *et al.* [6] have proposed a lock-free algorithm for a binary search tree in which an operation does not need to restart from the root of the tree. This helps in reducing the *amortized* time complexity of a modify operation in the presence of conflicts.

Our Contributions. In this work, we present a new *lock-free* algorithm for concurrent manipulation of a binary search tree in an asynchronous shared memory system that supports search, insert and delete operations. In addition to read and write instructions, our algorithm uses a (single-word) compare-and-swap (CAS) atomic instruction, which is commonly supported by many modern processors including Intel 64 and AMD64. It combines ideas from two existing lock-free algorithms, namely those by Howley and Jones [9] and Natarajan and Mittal [12], and is especially *optimized for the conflict-free scenario*. Like Howley and Jones' algorithm, it uses internal representation of a search tree in which all nodes store keys. Also, like Natarajan and Mittal's algorithm, it operates at edge-level rather than node-level and does not use a separate explicit object for enabling coordination among conflicting operations. As a result, it inherits benefits of both the lock-free algorithms. Specifically, when compared to modify operations of Howley and Jones' internal binary search tree, its modify operations (a) have a smaller contention window, (b) allocate fewer objects, (c) execute fewer atomic instructions, and (d) have a smaller memory footprint. Our experiments indicate that our new lock-free algorithm outperforms other lock-free algorithms in most cases, providing up to 35% improvement in some cases over the next best algorithm.

2. SYSTEM MODEL

We assume an asynchronous shared memory system that, in addition to read and write instructions, also supports compare-and-swap (CAS) atomic instruction. A compare-and-swap instruction 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*). The CAS instruction is commonly available in many modern processors such as Intel 64 and AMD64.

We assume that a binary search tree (BST) implements a dictionary abstract data type and supports *search*, *insert* and *delete* operations [5]. For convenience, we refer to the insert and delete operations as *modify* 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. As in [9], we use an *internal* BST in our algorithm in which all nodes (internal as well as leaf) store the keys.

3. THE LOCK-FREE ALGORITHM

For ease of exposition, we describe our algorithm assuming no memory reclamation.

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 operations one-by-one.

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 ((shown in Figure 2)) 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 3), the terminal

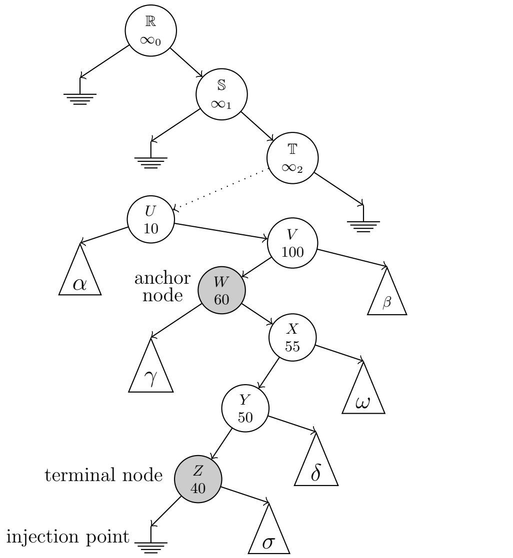

Figure 1: Nodes in the access path of seek along with sentinel keys and nodes ($\infty_0 < \infty_1 < \infty_2$)

node is removed by changing the pointer at the parent node of the terminal node. In the case of complex delete (shown in Figure 4), 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, we use sentinel keys and three sentinel nodes to handle the boundary cases easily. The structure of an empty tree with only sentinel keys (denoted by ∞_0, ∞_1 and ∞_2 with $\infty_0 < \infty_1 < \infty_2$) and sentinel nodes (denoted by $\mathbb{R}$, $\mathbb{S}$ and $\mathbb{T}$) is shown in Figure 1.

Our algorithm, like the one in [12], operates at edge level. A delete operation obtains ownership of the edges it needs to work on by marking them. To enable marking, we steal bits from the child addresses of a node. Specifically, we steal *three* bits from each child address to distinguish between three types of marking: (i) marking for *intent*, (ii) marking for *deletion* and (iii) marking for *promotion*. The three bits are referred to as *intent-flag*, *delete-flag* and *promote-flag*. To avoid the ABA problem, as in Howley and Jones [9], we use *unique* null pointers. To that end, we steal another bit from the child address, referred to as *null-flag*, and use it to indicate whether the address field contains a null or a non-null value. 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 field are not otherwise modified. This ensures that all null pointers are unique.

Finally, we also steal a bit from the key field to indicate whether the key stored in a node is the original key or the replacement key. This information is used in a complex delete operation to coordinate helping among processes.

We next describe the details of the seek function, which is used by all operations (search as well as modify) to traverse the tree after which we describe the details of the execution phase of insert and delete operations.

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 1. 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. 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 removal (either delete or promote flag set) by examining its 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.

(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.

The seek function also keeps track of the *second-to-last* edge in the access-path (whose endpoints are the parent and grandparent nodes of the terminal node), which is used for helping, if there is a conflict. 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 determines if it failed because of a *conflicting* delete operation in progress. If there is no conflicting delete

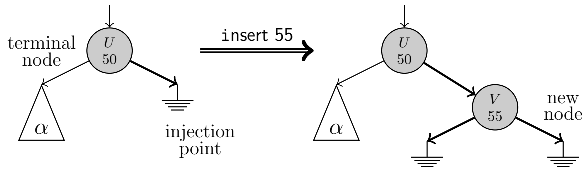

Figure 2: An illustration of an insert operation.

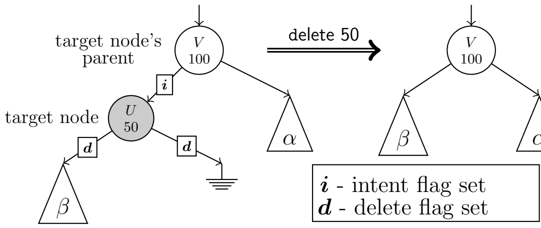

Figure 3: An illustration of a simple delete operation.

operation in progress, then the operation restarts from the seek phase; otherwise it performs helping and then restarts from the seek phase.

3.2.3 The Execution Phase of a Delete Operation

The execution of a delete operation starts in *injection mode*. Once the operation has been injected into the tree, it advances to either *discovery mode* or *cleanup mode* depending on the type of the delete operation.

Injection Mode. In the injection mode, the delete operation marks the three edges involving the target node as follows: (i) It first sets the intent-flag on the edge from the parent of the target node to the target node using a CAS instruction. (ii) It then sets the delete-flag on the left edge of the target node using a CAS instruction. (iii) Finally, it sets the delete-flag on the right edge of the target node using a CAS instruction. If the CAS instruction fails at any step, the delete operation performs helping, and either repeats the same step or restarts from the seek phase. Specifically, the delete operation repeats the same step when setting the delete-flag as long as the target node has not been claimed as the successor node by another delete operation. In all other cases, it restarts from the seek phase.

We maintain the invariant that an edge, once marked, cannot be unmarked. After marking both the edges of the target node, the operation checks whether 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. Note that the type of the delete operation cannot change once all the three edges have been marked as described above. If the delete operation is complex, then it advances to the discovery mode after which it will advance to the cleanup mode. On the other hand, if it is simple, then it directly advances to the cleanup mode (and skips the discovery mode). Eventually, the target node is either removed from the tree (if simple delete) or replaced with a "new" node containing the next largest key (if complex delete).

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.

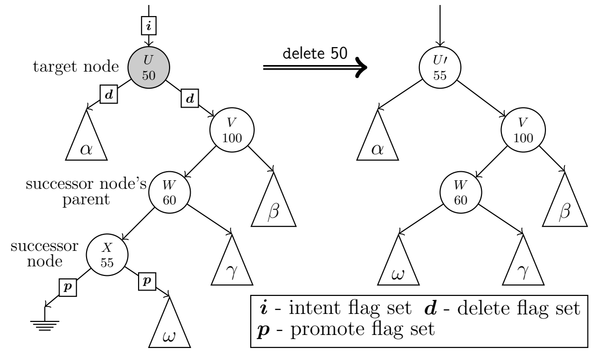

Figure 4: An illustration of a complex delete operation.

Discovery Mode. In the discovery mode, a complex delete operation performs the following steps:

1. **Find Successor Key:** 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.
2. **Mark Child Edges of Successor Node:** The operation sets the promote-flag on both the child edges of S using a CAS instruction. Note that the left child edge of S will be null. As part of marking the left child edge, we also store the address of T (the target node) in the edge. This is done to enable helping in case the successor node is obstructing the progress of another operation. In case the CAS instruction fails while marking the left child edge, the operation repeats from step 1 after performing helping if needed. On the other hand, if the CAS instruction fails while marking the right child edge, then the marking step is repeated after performing helping if needed.
3. **Promote Successor Key:** The operation replaces the target node's original key with the successor key. At the same time, it also sets the mark bit in the key to indicate that the current key stored in the target node is the replacement key and not the original key.
4. **Remove Successor Node:** The operation removes S (the successor node) by changing the child pointer at $S.parent$ that is pointing to S to point to the right child of S using a CAS instruction. If the CAS instruction succeeds, then the operation advances to the cleanup mode. Otherwise, it performs helping if needed. It then finds S again by performing another traversal of the tree starting from the right child of T. If the traversal fails to find S (recall that the left edge of S is marked for promotion and contains the address of T), then S has already been removed from the tree by another operation as part of helping, and the delete operation advances to the cleanup mode. On advancing to the cleanup mode, the operation sets a flag in T indicating that S has been removed from the tree (and T can now be replaced with a new node) so that other operations trying to help it know not to look for S.

Cleanup Mode. There are two cases depending on whether the delete operation is simple or complex.

```
struct Node {
    {Boolean, Key} mKey;
    {Boolean, Boolean, Boolean, Boolean, NodePtr} child[2];
    Boolean readyToReplace;
};
struct Edge {
    NodePtr parent, child;
    enum which { LEFT, RIGHT };
};
struct SeekRecord {
    Edge lastEdge, pLastEdge, injectionEdge;
};
struct AnchorRecord {
    NodePtr node;
    Key key;
};
struct StateRecord {
    Edge targetEdge, pTargetEdge;
    Key targetKey, currentKey;
    enum mode { INJECTION, DISCOVERY, CLEANUP };
    enum type { SIMPLE, COMPLEX } ;
    // the next field stores pointer to a seek record; it is used
       for finding the successor if the delete operation is complex
    SeekRecordPtr successorRecord;
};
// object to store information about the tree traversal when looking
   for a given key (used by the seek function)
SeekRecordPtr targetRecord := new seek record;
// object to store information about process' own delete operation
StateRecordPtr myState := new state;
```

Algorithm 1: Data Structures Used

(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 (which in turn will imply that T is a leaf node). Without loss of generality, assume that *T.right* is a null node. The removal of T involves changing the child pointer at *T.parent* that is pointing to T to point to *T.left* using a CAS instruction. If the CAS instruction succeeds, then the delete operation terminates; otherwise, it performs another seek on the tree. If the seek function either fails to find the target key or returns a terminal node different from T, then T has been already removed from the tree (by another operation as part of helping) and the delete operation terminates; otherwise, it attempts to remove T from the tree again using possibly the new parent information returned by seek. This process may be repeated multiple times.

(b) **Complex Delete:** Note that, at this point, the key stored in the target node is the replacement key (the successor key of the target key). Further, the key as well as both the child edges of the target node are marked. The delete operation attempts to replace target node with a *new* node, which is basically a copy of target node except that all its fields are unmarked. This replacement of T involves changing the child pointer at *T.parent* that is pointing to T to point to the new node. If the CAS instruction succeeds, then the delete operation terminates; otherwise, as in the case of simple delete, it performs another seek on the tree, this time looking for the successor key. If the seek function either fails to find the successor key or returns a terminal node different from T, then T has been already replaced (by another operation as part of helping) and the delete operation terminates. Otherwise, it attempts to replace T again using possibly the new parent information returned by seek. This process may be repeated multiple times.

```
SEEK( key, seekRecord )
begin
    pAnchorRecord := {S, ∞1};
    while true do
        // initialize all variables used in traversal
        pLastEdge := {R, S, RIGHT};
        lastEdge := {S, T, RIGHT};
        curr := T ;
        anchorRecord := {S, ∞1};
        while true do
            // read the key stored in the current node
            ⟨∗, cKey⟩ := curr→mKey;
            // find the next edge to follow
            which := key < cKey ? LEFT: RIGHT;
            ⟨n, ∗, d, p, next⟩ := curr→child[which];
            // check for the completion of the traversal
            if key = cKey or n then
                // either key found or no next edge to follow;
                   stop the traversal
                seekRecord→pLastEdge := pLastEdge;
                seekRecord→lastEdge := lastEdge;
                seekRecord→injectionEdge :=
                                        {curr, next, which};
                if key = cKey then // keys match
                    return;
                else break;
            if which = RIGHT then
                // next edge to be traversed is a right edge;
                   keep track of the current node and its key
                anchorRecord := ⟨curr, cKey⟩;
            // traverse the next edge
            pLastEdge := lastEdge;
            lastEdge := {curr, next, which};
            curr := next;
        // key was not found; check if can stop
        ⟨∗, ∗, d, p, ∗⟩ := anchorRecord.node→child[RIGHT];
        if not (d) and not (p) then
            // the anchor node is still part of the tree; check if
               the anchor node's key has changed
            ⟨∗, aKey⟩ := anchorRecord.node→mKey;
            if anchorRecord.key = aKey then return;
        else
            // check if the anchor record (the node and its key)
               matches that of the previous traversal
            if pAnchorRecord = anchorRecord then
                // return the results of the previous traversal
                seekRecord := pSeekRecord;
                return;
        // store the results of the traversal and restart
        pSeekRecord := seekRecord;
        pAnchorRecord := anchorRecord;
```

Algorithm 2: Seek Function

```
Boolean SEARCH( key )
begin
    SEEK( key, mySeekRecord );
    node := mySeekRecord→lastEdge.child;
    ⟨∗, nKey⟩ := node→mKey;
    if nKey = key then return true;
    else return false;
```

Algorithm 3: Search Operation

Discussion. It can be verified that, in the absence of conflict, a delete operation performs three atomic instructions in the injection mode, three in the discovery mode (if delete is complex), and one in the cleanup mode.

3.2.4 Helping

To enable helping, as mentioned earlier, whenever traversing the tree to locate either a target key or a successor key, we keep track of the *last two* edges encountered in the traversal. When a CAS instruction fails, depending on the reason for failure, helping is either performed along the last edge or the second-to-last edge.

3.3 Formal Description

A pseudo-code of our algorithm is given in Algorithms 1-

```
Boolean Insert( key )
begin
    while true do
        Seek( key, targetRecord );
        targetEdge := targetRecord → lastEdge;
        node := targetEdge.child;
        ⟨*, nKey⟩ := node → mKey;
        if key = nKey then return false;
        // create a new node and initialize its fields
        newNode := create a new node;
        newNode → mKey := ⟨0_m, key⟩;
        newNode → child[LEFT] := ⟨1_n, 0_i, 0_d, 0_p, null⟩;
        newNode → child[RIGHT] := ⟨1_n, 0_i, 0_d, 0_p, null⟩;
        newNode → readyToReplace := false;
        which := targetRecord → injectionEdge.which;
        address := targetRecord → injectionEdge.child;
        result := CAS( node → child[which],
                       ⟨1_n, 0_i, 0_d, 0_p, address⟩,
                       ⟨0_n, 0_i, 0_d, 0_p, newNode⟩);
        if result then return true;
        // help if needed
        ⟨*, *, d, p, *⟩ := node → child[which];
        if d then  HelpTargetNode( targetEdge ) ;
        else if p then  HelpSuccessorNode( targetEdge ) ;
```

Algorithm 4: Insert Operation

```
Boolean Delete( key )
begin
    // initialize the state record
    myState → targetKey := key;
    myState → currentKey := key;
    myState → mode := INJECTION;
    while true do
        Seek( myState → currentKey, targetRecord );
        targetEdge := targetRecord → lastEdge;
        pTargetEdge := targetRecord → pLastEdge;
        ⟨*, nKey⟩ := targetEdge.child → mKey;
        if myState → currentKey ≠ nKey then
            // the key does not exist in the tree
            if myState → mode = INJECTION then
                return false;
            else return true;
        // perform appropriate action depending on the mode
        if myState → mode = INJECTION then
            // store a reference to the target edge
            myState → targetEdge := targetEdge;
            myState → pTargetEdge := pTargetEdge;
            // attempt to inject the operation at the node
            Inject( myState );
        // mode would have changed if injection was successful
        if myState → mode ≠ INJECTION then
            // check if the target node found by the seek function
               matches the one stored in the state record
            if ( myState → targetEdge.child ≠
                                 targetEdge.child ) then
                return true;
            // update the target edge information using the most
               recent seek
            myState → targetEdge := targetEdge;
        if myState → mode = DISCOVERY then
            // complex delete operation; locate the successor node
               and mark its child edges with promote flag
            FindAndMarkSuccessor( myState );
        if myState → mode = DISCOVERY then
            // complex delete operation; promote the successor
               node's key and remove the successor node
            RemoveSuccessor( myState );
        if myState → mode = CLEANUP then
            // either remove the target node (simple delete) or
               replace it with a new node with all fields unmarked
               (complex delete)
            result := Cleanup( myState );
            if result then return true;
            else
                ⟨*, nKey⟩ := targetEdge.child → mKey;
                myState → currentKey := nKey;
```

Algorithm 5: Delete Operation

```
Inject( state )
begin
    targetEdge := state → targetEdge;
    // try to set the intent flag on the target edge
    // retrieve attributes of the target edge
    parent := targetEdge.parent;
    node := targetEdge.child;
    which := targetEdge.which;
    result := CAS( parent → child[which],
                   ⟨0_n, 0_i, 0_d, 0_p, node⟩, ⟨0_n, 1_i, 0_d, 0_p, node⟩ );
    if not (result) then
        // unable to set the intent flag; help if needed
        ⟨*, i, d, p, address⟩ := parent → child[which];
        if i then  HelpTargetNode( targetEdge ) ;
        else if d then
            HelpTargetNode( state → pTargetEdge );
        else if p then
            HelpSuccessorNode( state → pTargetEdge );
        return;
    // mark the left edge for deletion
    result := MarkChildEdge( state, LEFT );
    if not (result) then  return;
    // mark the right edge for deletion; cannot fail
    MarkChildEdge( state, RIGHT );
    // initialize the type and mode of the operation
    InitializeTypeAndUpdateMode( state );
```

Algorithm 6: Injecting a Deletion Operation

```
FindAndMarkSuccessor( state )
begin
    // retrieve the addresses from the state record
    node := state → targetEdge.child;
    seekRecord := state → successorRecord;
    while true do
        // read the mark flag of the key in the target node
        ⟨m, *⟩ := node → mKey;
        // find the node with the smallest key in the right
           subtree
        result := FindSmallest( state );
        if m  or not (result) then
            // successor node had already been selected before
               the traversal or the right subtree is empty
            break;
        // retrieve the information from the seek record
        successorEdge := seekRecord → lastEdge;
        left := seekRecord → injectionEdge.child;
        // read the mark flag of the key under deletion
        ⟨m, *⟩ := node → mKey;
        if m then // successor node has already been selected
            continue;
        // try to set the promote flag on the left edge
        result := CAS( successorEdge.child → child[LEFT],
                       ⟨1_n, 0_i, 0_d, 0_p, left⟩,
                       ⟨1_n, 0_i, 0_d, 1_p, node⟩ );
        if result then break;
        // attempt to mark the edge failed; recover from the
           failure and retry if needed
        ⟨n, *, d, *, *⟩ := successorEdge.child → child[LEFT];
        if n  and d then
            // the node found is undergoing deletion; need to help
            HelpTargetNode( successorEdge );
    // update the operation mode
    UpdateMode( state );
```

Algorithm 7: Locating the Successor Node

12.

Algorithm 1 describes the data structures used in our algorithm. Besides Node, three important data types in our algorithm are: Edge, SeekRecord and StateRecord. The data type Edge is a structure consisting of three fields: the two endpoints and the direction (left or right). The data type SeekRecord is a structure used to store the results of a tree traversal. The data type StateRecord is a structure used to store information about a delete operation (*e.g.*, target edge, type, current mode, etc.). Note that only objects of type Node are shared between processes; objects of all other types (*e.g.*, SeekRecord, StateRecord) are *local* to a process and not shared with other processes.

The pseudo-code of the seek function is described in Al-

```
RemoveSuccessor( state )
begin
    // retrieve addresses from the state record
    node := state → targetEdge.child;
    seekRecord := state → successorRecord;
    // extract information about the successor node
    successorEdge := seekRecord → lastEdge;
    // ascertain that the seek record for the successor node
       contains valid information
    ⟨*, *, *, p, address⟩ := successorEdge.child → child[LEFT];
    if not (p) or (address ≠ node) then
        node → readyToReplace := true;
        UpdateMode( state );
        return;
    // mark the right edge for promotion if unmarked
    MarkChildEdge( state, RIGHT );
    // promote the key
    node → mKey := ⟨1_m, successorEdge.child → mKey⟩;
    while true do
        // check if the successor is the right child of the target
           node itself
        if successorEdge.parent = node then
            // need to modify the right edge of the target node
               whose delete flag is set
            dFlag := 1;        which := RIGHT;
        else
            dFlag := 0;        which := LEFT;
        ⟨*, i, *, *, *⟩ := successorEdge.parent → child[which];
        ⟨n, *, *, *, right⟩ := successorEdge.child → child[RIGHT];
        oldValue := ⟨0_n, i, dFlag, 0_p, successorEdge.child⟩;
        if n then
            // only set the null flag; do not change the address
            newValue :=
                ⟨1_n, 0_i, dFlag, 0_p, successorEdge.child⟩;
        else
            // switch the pointer to point to the grand child
            newValue := ⟨0_n, 0_i, dFlag, 0_p, right⟩ ;
        result := CAS( successorEdge.parent → child[which],
                       oldValue, newValue );
        if result or dFlag then break;
        ⟨*, *, d, *, *⟩ := successorEdge.parent → child[which];
        pLastEdge := seekRecord → pLastEdge;
        if d and (pLastEdge.parent ≠ null) then
            HelpTargetNode( pLastEdge );
        result := FindSmallest( state );
        lastEdge := seekRecord → lastEdge;
        if ( not (result) or
             lastEdge.child ≠ successorEdge.child ) then
            // the successor node has already been removed
            break;
        else successorEdge := seekRecord → lastEdge ;
    node → readyToReplace := true;
    UpdateMode( state );
```

Algorithm 8: Removing the Successor Node

```
Boolean Cleanup( state )
begin
    ⟨parent, node, pWhich⟩ := state → targetEdge;
    if state → type = COMPLEX then
        // replace the node with a new copy in which all fields
           are unmarked
        ⟨*, nKey⟩ := node → mKey;
        newNode → mKey := ⟨0_m, nKey⟩;
        // initialize left and right child pointers
        ⟨*, *, *, *, left⟩ := node → child[LEFT];
        newNode → child[LEFT] := ⟨0_n, 0_i, 0_d, 0_p, left⟩;
        ⟨n, *, *, *, right⟩ := node → child[RIGHT];
        if n then
            newNode → child[RIGHT] := ⟨1_n, 0_i, 0_d, 0_p, null⟩;
        else newNode → child[RIGHT] := ⟨0_n, 0_i, 0_d, 0_p, right⟩ ;
        // initialize the arguments of CAS instruction
        oldValue := ⟨0_n, 1_i, 0_d, 0_p, node⟩;
        newValue := ⟨0_n, 0_i, 0_d, 0_p, newNode⟩;
    else // remove the node
        // determine to which grand child will the edge at the
           parent be switched
        if node → child[LEFT] = ⟨1_n, *, *, *, *⟩ then
            nWhich := RIGHT;
        else nWhich := LEFT;
        // initialize the arguments of the CAS instruction
        oldValue := ⟨0_n, 1_i, 0_d, 0_p, node⟩;
        ⟨n, *, *, *, address⟩ := node → child[nWhich];
        if n then // set the null flag only
            newValue := ⟨1_n, 0_i, 0_d, 0_p, node⟩;
        else // change the pointer to the grand child
            newValue := ⟨0_n, 0_i, 0_d, 0_p, address⟩ ;
    result := CAS( parent → child[pWhich],
                   oldValue, newValue );
    return result;
```

Algorithm 9: Cleaning Up the Tree

gorithm 2, which is used by all the operations. The pseudo-codes of the search, insert and delete operations are given in Algorithm 3, Algorithm 4 and Algorithm 5, respectively. A delete operation executes function Inject in injection mode, functions FindAndMarkSuccessor and RemoveSuccessor in discovery mode and function Cleanup in cleanup mode. Their pseudo-codes are given in Algorithm 6, Algorithm 7, Algorithm 8 and Algorithm 9, respectively. The pseudo-codes for helper routines (used by multiple functions) are given in Algorithm 10 and Algorithm 11. Finally, the pseudo-codes of functions used to help other (conflicting) delete operations are given in Algorithm 12.

It can be shown that our algorithm satisfies linearizability and lock-freedom properties [8]. Broadly speaking, linearizability requires that an operation should appear to take effect instantaneously at some point during its execution. Lock-freedom requires that some process should be able to complete its operation in a finite number of its own steps. Due to lack of space, the proof of correctness has been omitted and can be found elsewhere [14].

4. EXPERIMENTAL EVALUATION

We now describe the results of the comparative evaluation of different implementations of a concurrent BST using simulated workloads.

4.1 Other Concurrent Binary Search Tree Implementations

We considered three other implementations of concurrent BST for comparative evaluation, namely those based on: (i) the lock-free external BST by Natarajan and Mittal [12], denoted by NM-BST, (ii) the lock-free internal BST by Howley and Jones [9], denoted by HJ-BST and (iii) the RCU-based internal BST by Arbel and Attiya [1], denoted by CITRUS. The above three implementations were obtained from their respective authors. We refer to the implementation based on our algorithm as OurBST. All implementations were written in C/C++. In our experiments, none of the implementations used garbage collection to reclaim memory. The experimental evaluation in [9,12] showed that, in all cases, either HJ-BST or NM-BST 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 the lock-free algorithm in [6].

4.2 Experimental Setup

We conducted our experiments on a single large-memory node in stampede[1] 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

[1]www.tacc.utexas.edu/resources/hpc/stampede-technical

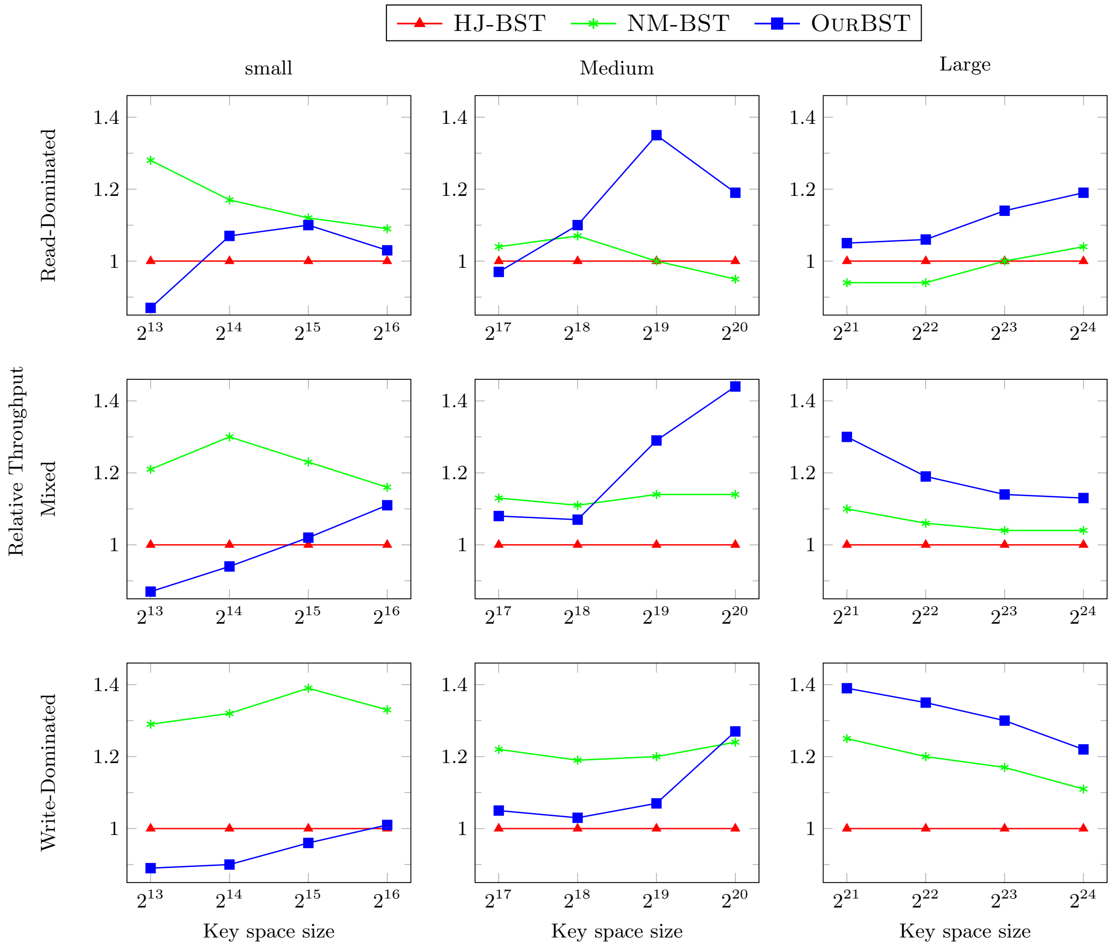

Figure 5: Comparison of system throughput of different concurrent BST implementations *relative to that of* HJ-BST at 32 threads. Each row represents a workload type. Each column represents a range of key space size. Higher the ratio, better the performance of the algorithm.

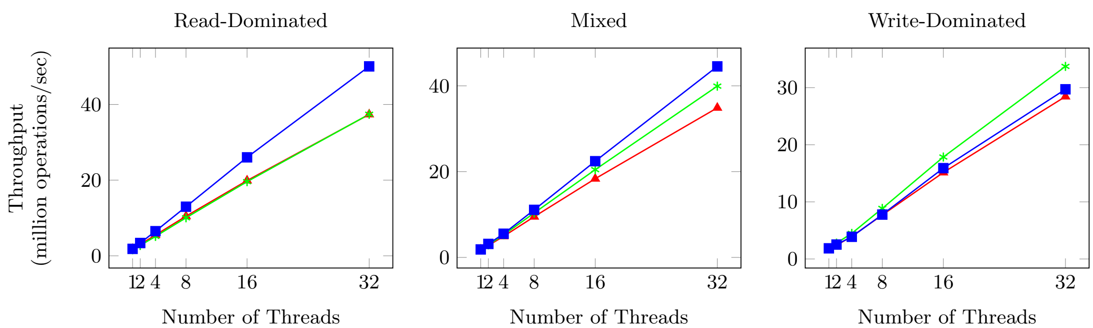

Figure 6: Comparison of system throughput (in million operations/second) of different concurrent BST implementations for key space size of 512Ki. Higher the throughput, better the performance.

1TB of DDR3 memory. 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 optimization flag set to O3. We used GNU Scientific Library to generate random numbers. We used Intel's *TBB Malloc* [15] as the dynamic memory allocator since it provided superior performance to C/C+ default allocator in a multi-threaded environment.

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 varied key space size from 2^{13} (8Ki) to 2^{24} (16Mi).
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.

```
Boolean MarkChildEdge( state, which )
begin
    if state→mode = INJECTION then
        edge := state→targetEdge;
        flag := DELETE_FLAG;
    else
        edge := (state→successorRecord)→lastEdge;
        flag := PROMOTE_FLAG;
    node := edge.child;
    while true do
        ⟨n, i, d, p, address⟩ := node→child[which];
        if i then
            helpeeEdge := {node, address, which};
            HelpTargetNode( helpeeEdge );
            continue;
        else if d then
            if flag = PROMOTE_FLAG then
                HelpTargetNode( edge );
                return false;
            else return true;
        else if p then
            if flag = DELETE_FLAG then
                HelpSuccessorNode( edge );
                return false;
            else return true;
        oldValue := ⟨n, 0_i, 0_d, 0_p, address⟩;
        newValue := oldValue | flag;
        result := CAS( node→child[which], oldValue,
                       newValue );
        if result then break;
    return true;

Boolean FindSmallest( state )
begin
    // find the node with the smallest key in the subtree rooted at
       the right child of the target node
    node := state→targetEdge.child;
    seekRecord := state→seekRecord;
    ⟨n, ∗, ∗, ∗, right⟩ := node→child[RIGHT];
    if n then // the right subtree is empty
        return false;

    // initialize the variables used in the traversal
    lastEdge := ⟨node, right, RIGHT⟩;
    pLastEdge := ⟨node, right, RIGHT⟩;
    while true do
        curr := lastEdge.child;
        ⟨n, ∗, ∗, ∗, left⟩ := curr→child[LEFT];
        if n then // reached the node with the smallest key
            injectionEdge := ⟨curr, left, LEFT⟩;
            break;

        // traverse the next edge
        pLastEdge := lastEdge;
        lastEdge := ⟨curr, left, LEFT⟩;

    // initialize seek record and return
    seekRecord→lastEdge := lastEdge;
    seekRecord→pLastEdge := pLastEdge;
    seekRecord→injectionEdge := injectionEdge;
    return true;
```

Algorithm 10: Helper Routines

```
InitializeTypeAndUpdateMode( state )
begin
    // retrieve the target node's address from the state record
    node := state→targetEdge.child;
    ⟨lN, ∗, ∗, ∗, ∗⟩ := node→child[LEFT];
    ⟨rN, ∗, ∗, ∗, ∗⟩ := node→child[RIGHT];
    if lN or rN then
        // one of the child pointers is null
        ⟨m, ∗⟩ := node→mKey;
        if m then state→type := COMPLEX;
        else state→type := SIMPLE;
    else // both child pointers are non-null
        state→type := COMPLEX;
    UpdateMode( state );

UpdateMode( state )
begin
    // update the operation mode
    if state→type = SIMPLE then // simple delete
        state→mode := CLEANUP;
    else // complex delete
        node := state→targetEdge.child;
        if node→readyToReplace then
            state→mode := CLEANUP;
        else state→mode := DISCOVERY;
```

Algorithm 11: Helper Routines

```
HelpTargetNode( helpeeEdge )
begin
    // intent flag must be set on the edge
    // obtain new state record and initialize it
    state→targetEdge := helpeeEdge;
    state→mode := INJECTION;
    // mark the left and right edges if unmarked
    result := MarkChildEdge( state, LEFT );
    if not (result) then return;
    MarkChildEdge( state, RIGHT );
    InitializeTypeAndUpdateMode( state );
    // perform the remaining steps of a delete operation
    if state→mode = DISCOVERY then
        FindAndMarkSuccessor( state );

    if state→mode = DISCOVERY then
        RemoveSuccessor( state );

    if state→mode = CLEANUP then Cleanup( state ) ;

HelpSuccessorNode( helpeeEdge )
begin
    // retrieve the address of the successor node
    parent := helpeeEdge.parent;
    node := helpeeEdge.child;
    // promote flat must be set on the successor node's left edge
    // retrieve the address of the target node
    ⟨∗, ∗, ∗, ∗, left⟩ := node→child[LEFT];
    // obtain new state record and initialize it
    state→targetEdge := {null, left, _};
    state→mode := DISCOVERY;
    seekRecord := state→successorRecord;
    // initialize the seek record in the state record
    seekRecord→lastEdge := helpeeEdge;
    seekRecord→pLastEdge := {null, parent, _};
    // promote the successor node's key and remove the successor
       node
    RemoveSuccessor( state );
    // no need to perform the cleanup
```

Algorithm 12: Helping Conflicting Delete Operations

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 *prepopulated* 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 5 and Figure 6. In Figure 5, each row represents a specific workload (read-dominated, mixed or write-dominated) and each column represents a specific key space size; *small* (8Ki to 64Ki), *medium* (128Ki to 1Mi) and *large* (2Mi to 16Mi). Figure 6 shows the scaling with respect to the number of threads for key space size of 2^{19} (512Ki). We do not show the numbers for CITRUS in the graphs as it had the worst performance among all implementations (slower by a factor of four in some cases). This is not surprising as CITRUS is optimized for read operations (*e.g.*, 98% reads & 2% updates) [1].

Table 1: Comparison of different lock-free algorithms in the absence of contention.

Algorithm	Number of Objects Allocated		Number of Atomic Instructions Executed	
	Insert	Delete	Insert	Delete
HJ-BST	2	simple: 1 complex: 1	3	simple: 4 complex: 9
NM-BST	2	0	1	3
OurBST	1	simple: 0 complex: 1	1	simple: 4 complex: 7

As the graphs show, OurBST achieved nearly same or higher throughput than the other two implementations for medium and large key space sizes (except for medium key space size with write-dominated workload). Specifically, at 32 threads and for a read-dominated workload, OurBST had 35% and 24% higher throughput than the next best performer for key space sizes of 512Ki and 1Mi, respectively. Also, at 32 threads and for a mixed workload, OurBST had 27% and 19% higher throughput than the next best performer for key space sizes of 1Mi and 2Mi, respectively. Overall, OurBST outperformed the next best implementation by as much as 35%; it outperformed HJ-BST by as much as 44% and NM-BST by as much as 35% (both achieved for medium key space sizes). For large key space sizes, the overhead of traversing the tree appears to dominate the overhead of actually modifying the operation's window, and the gap between various implementations becomes smaller.

There are several reasons why OurBST outperformed the other two implementations in many cases. First, as Table 1 shows, our algorithm allocates fewer objects than the two other algorithms on average considering the fact that the fraction of insert operations will generally be larger than the fraction of delete operations in any realistic workload. Further, we observed in our experiments that the number of simple delete operations outnumbered the number of complex delete operations by two to one, and our algorithm does not allocate any object for a simple delete operation. Second, again as Table 1 shows, our algorithm executes the same number of atomic instructions as in [12] for insert operations; and, in all the cases, executes same or fewer atomic instructions than in [9]. This is important since an atomic instruction is more expensive to execute than a simple read or write instruction. Third, we observed in our experiments that OurBST had a smaller memory footprint than the other two implementations (by almost a factor of two) since it uses internal representation and allocates fewer objects. As a result, it was likely able to benefit from caching to a larger degree than HJ-BST and NM-BST.

We also observed in our experiments that, for key space sizes larger than 8Ki, the likelihood of an operation restarting was extremely low (less than 0.1%) even for a write-dominated workload implying 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 8Ki, we expect OurBST to outperform any implementation based on the lock-free algorithm described in [6], which is basically derived from the one in [5].

5. 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] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent Cache-Oblivious B-Trees. In *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 228–237, July 2005.

[3] A. Braginsky and E. Petrank. A Lock-Free B+tree. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 58–67, 2012.

[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 33rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 332–340, 2014.

[7] M. Fomitchev and E. Ruppert. Lock-Free Linked Lists and Skiplists. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 50–59, July 2004.

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

[9] 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.

[10] 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.

[11] A. Natarajan and N. Mittal. Brief Announcement: A Concurrent Lock-Free Red-Black Tree. In *Proceedings of the 27th Symposium on Distributed Computing (DISC)*, Jerusalem, Israel, Oct. 2013.

[12] 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.

[13] A. Natarajan, L. H. Savoie, and N. Mittal. Concurrent Wait-Free Red-Black Trees. In *Proceedings of the 15th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 45–60, Osaka, Japan, Nov. 2013.

[14] A. Ramachandran and N. Mittal. A Fast Lock-Free Internal Binary Search Tree. Technical Report UTDCS-11-14, Department of Computer Science, The University of Texas at Dallas, Aug. 2014.

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