Fast Concurrent Lock-Free Binary Search Trees

Aravind Natarajan Neeraj Mittal
Erik Jonsson School of Engineering and Computer Science
The University of Texas at Dallas
{aravindn, neerajm}@utdallas.edu

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. In addition to read and write instructions, our algorithm uses (single-word) compare-and-swap (CAS) and bit-test-and-set (BTS) atomic instructions, both of which are commonly supported by many modern processors including Intel 64 and AMD64. In contrast to existing lock-free algorithms for a binary search tree, our algorithm is based on marking *edges* rather than nodes. As a result, when compared to other lock-free algorithms, modify (insert and delete) operations in our algorithm (a) work on a smaller portion of the tree, thereby reducing conflicts, and (b) execute fewer atomic instructions (one for insert and three for delete). Our experiments indicate that our lock-free algorithm significantly outperforms all other algorithms for a concurrent binary search tree in many cases, especially when contention is high, by as much as 100%.

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 [20].

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)* [18] and *compare-and-swap (CAS)* [20]. Non-blocking implementations of common data structures such as queues, stacks, linked lists, hash tables, search trees and tries have been proposed (*e.g.*, [1, 2, 12, 14, 15, 17, 18, 20, 22, 23, 25, 27–29, 29–31]).

Non-blocking algorithms may provide varying degrees of progress guarantees. Three widely accepted progress guarantees for non-blocking algorithms are: obstruction-freedom, lock-freedom, and wait-freedom. An algorithm is *obstruction-free* if any process that executes in isolation will finish its operation in a finite number of steps [22]. An algorithm is *lock-free* if some process will complete its operation in a finite number of steps [18]. An algorithm is *wait-free* if every operation executed by every process will complete in a finite number of steps [18].

Binary search trees are one of the fundamental data structures for organizing and storing *ordered* data that support search, insert and delete operations [9].

Several universal constructions exist which can be used to derive a concurrent non-blocking data structure from its sequential version [8, 13, 18–20]. Due to the general nature of these constructions, when applied to a binary search tree, the resultant data structures are quite inefficient. This is because universal constructions involve either: (a) applying operations to the data structure in a serial manner, or (b) copying the entire data structure (or parts of it that will change and

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.

http://dx.doi.org/10.1145/2555243.2555256

any parts that directly or indirectly point to them), applying the operation to the copy and then updating the relevant part of the data structure to point to the copy. The first approach precludes any concurrency. The second approach, when applied to a tree, also precludes any concurrency since the root node of the tree indirectly points to every node in the tree.

Several customized non-blocking implementations of concurrent unbalanced search trees [4, 12, 23] and balanced search trees such as B-trees [1], B^+-trees [2], red-black trees [24, 27, 28], and tries [29, 30] have been proposed, which are more efficient than those obtained using universal constructions.

Ellen *et al.* proposed the first practical lock-free algorithm for a concurrent binary search tree in [12]. 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 [23]. 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, search operations in Howley and Jones's algorithm are generally faster than those in Ellen *et al.*'s algorithm [23]. This is because the path traversed by a search operation in an external search tree always terminates at a leaf node, whereas the one in an internal search tree may terminate at an internal node. 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, which increases the likelihood of contention among operations.

Recently, Drachsler *et al.* proposed a lock-based internal binary search tree in which each node maintains pointers to its *logical* predecessor and successor nodes (based on the key order), in addition to maintaining pointers to its left and right children in the tree [11]. 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. They have applied this idea to obtain lock-based algorithms for unbalanced binary search trees and relaxed AVL trees [11].

Brown *et al.* proposed general primitives for developing non-blocking algorithms for concurrent search trees, namely LLX, SCX and VLX [5]. These primitives, which operate on multiple words, are generalizations of the LL (load-linked), SC (store-conditional) and VL (validate) instructions, which operate on a single word. (An LL instruction reads the contents of a memory location. An SC instruction updates the contents of a memory location with a new value provided the location has not been modified since an LL instruction was last performed on it. A VL instruction returns true if a memory location has not been modified since an LL instruction was last performed on it.) The LLX primitive operates on a single multi-word record, while the SCX and VLX primitives operate on multiple records. They also described an algorithm for implementing these primitives using single word CAS instruction [5]. Using the three primitives, they have developed efficient non-blocking algorithms for multisets [5] and relaxed red-black trees [6].

Observe that the lock-based algorithms for relaxed balanced binary search trees proposed in [3, 10] can be easily extended to work for unbalanced binary search trees by simply ignoring the balancing component of the algorithms. This is possible because, in these algorithms, the execution of an operation and the balancing of the tree are done in two separate steps. However, in the resulting algorithm, a key once added to the tree may never be "physically" removed from the tree. As a result, the size of the tree may become much larger than the number of keys stored in the tree.

Contributions: 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 (single-word) compare-and-swap (CAS) and bit-test-and-set (BTS) atomic instructions, both of which are commonly supported by many modern processors including Intel 64 and AMD64. (Our algorithm can be easily modified to use only CAS atomic instructions.) Like Ellen *et al.*'s algorithm, our algorithm is also based on external representation of a search tree. However, we use several ideas to reduce the contention among modify (insert and delete) operations as well as lower the overhead of a modify operation. They include: (i) marking edges rather than nodes for deletion, (ii) not using a separate explicit object for enabling coordination among conflicting operations, and (iii) allowing multiple keys being deleted to be removed from the tree in a single step. As a result, modify operations have a smaller contention window, allocate fewer objects and execute fewer atomic instructions than their counterparts in other lock-free algorithms [12, 23].

We have experimentally compared the performance of our concurrent algorithm with other concurrent algorithms for a binary search tree under a variety of conditions (different tree sizes, workload distributions and contention degrees). Our experiments indicate that our algorithm significantly outperforms all other algorithms in many cases by as much as 100%, especially when contention is high (tree size is small or workload is write-dominated).

Organization: The rest of the paper is organized as follows. We describe our system model in Section 2. Our lock-free 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 search tree in Section 4. We discuss the qualitative differences between our

lock-free algorithm and the lock-free algorithms proposed by Ellen *et al.* [12] and Howley and Jones [23] in more detail in Section 5. Finally, Section 6 concludes the paper and outlines directions for future research.

2. System Model

We assume an asynchronous shared memory system that, in addition to read and write instructions, also supports compare-and-swap (CAS) and bit-test-and-set (BTS) atomic instructions. 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). A bit-test-and-set instruction takes two arguments $address$ and $position$; it sets a given bit ($position$) in the contents of a memory location ($address$) to 1. Both instructions are commonly available in many modern processors such as Intel 64 and AMD64.

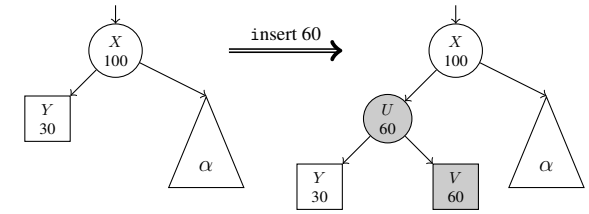
We assume that a binary search tree (BST) implements a dictionary abstract data type and supports *search*, *insert* and *delete* operations [12]. For convenience, we refer to the insert and delete operations as *modify* operations. A search operation explores the tree for a given key and returns true 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 [12], we use an *external* BST in our algorithm in which only the leaf nodes store the actual keys; keys stored at internal nodes are used for routing purposes only. Furthermore, every internal node has exactly two children.

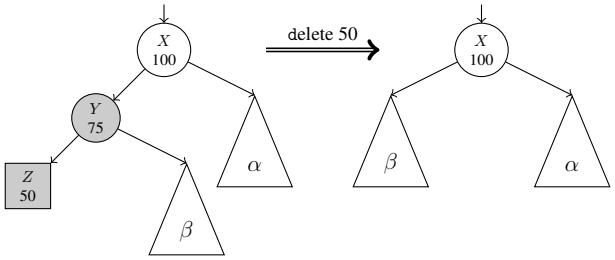
To demonstrate the correctness of our algorithm, we use *linearizability* [21] as the safety property and *lock-freedom* [18] as the liveness property. Broadly speaking, linearizability requires that an operation should appear to take effect instantaneously at some point during its execution. Lock-freedom requires that some process be able to complete its operation in a finite number of its own steps (even if one or more processes have failed).

3. A Lock-Free Algorithm for Binary Search Tree

We first present the main idea behind our algorithm. We then present the details of the algorithm along with its pseudo-code, followed by a brief proof of correctness.



(a) An illustration of the insert operation. Nodes that are added to the tree have been shaded.



(b) An illustration of the delete operation. Nodes that are removed from the tree have been shaded.

Figure 1: Illustration of insert and delete operations. Numbers denote keys.

3.1 Overview of the Algorithm

Every operation in our algorithm starts with a *seek* phase in which the operation traverses the search tree starting from the root node until it reaches a leaf node. We refer to the path traversed by the operation in the seek phase as the *access path*. For convenience, we refer to the last three nodes on the access path as the grandparent, parent and leaf nodes, respectively. The operation then compares the given key with the key stored in the leaf node on the access path. Depending on the result of the comparison and the type of the operation, the operation either terminates or moves to the next phase. We now describe the next steps for each type of operation one-by-one.

For a search operation, if the two keys match, then the operation returns true (key was found); otherwise it returns false (key was not found).

For an insert operation, if the two keys match, then the operation returns false (the key already exists in the tree); otherwise it moves to the *execution* phase (the key does not exist in the tree). In the execution phase, the insert operation adds the key to the tree by adding two new nodes to the tree: an internal node and a leaf node. The new leaf node contains the key being inserted into the tree. The two children of the new internal node are the leaf node on the access path and the new leaf node. The insert operation then switches the pointer at the parent node that is pointing to the leaf node (on the access path) to point to the new internal node. As an illustration, consider Figure 1(a). In the example, to insert the key 60 into the tree, the operation creates an internal node U and a leaf node V, with Y and V as children of U. It then

switches the pointer at node X, which is currently pointing to node Y, to point to node U.

For a delete operation, if the two keys do not match, then the operation returns false (the key does not exist in the tree); otherwise it moves to the *execution* phase (the key does exist in the tree). In the execution phase, the delete operation removes the key from the tree by removing the leaf and parent nodes on the access path from the tree. Specifically, it switches the pointer at the grandparent[1] node that is pointing to the parent node to point to the sibling node of the leaf node. As an illustration, consider Figure 1(b). In the example, to delete the key 50 from the tree, the operation switches the pointer at node X, which is currently pointing to node Y, to point to the sibling node of node Z (the root node of the subtree β).

For convenience, in the execution phase, we refer to the parent node on the access path as the *injection point* of the modify operation.

3.2 Details of the Algorithm

For ease of exposition, we assume that the memory allocated to nodes that are no longer part of the tree is not reclaimed. This allows us to assume that all new nodes have unique addresses and ignore the ABA problem that can occur otherwise [20]. A lock-free algorithm to reclaim memory allocated to objects that are no longer accessible by any process can be derived using the well-known notion of *hazard pointers* [26].

A tree node in our algorithm consists of three fields: (i) key, which contains the key stored in the node, (ii) $left$, which contains the address of the left child of the node, and (iii) $right$, which contains the address of the right child of the node. The $left$ and $right$ fields of a node are referred to as $child$ fields.

An important feature of our algorithm is that it operates at the *edge* level, rather than at the node level. In contrast to [12, 23] in which a modify operation obtains "ownership" of the nodes it needs to work on, in our algorithm, a modify operation (specifically, a delete operation) obtains "ownership" of the edges it needs to work on. A delete operation obtains "ownership" of an edge by *marking* it. Note that every edge has a *tail* node and a *head* node. The marking of an edge indicates that either both its tail and head nodes or only its tail node will be removed from the tree. To distinguish between the two cases, we refer to the first type of marking as *flagging* and to the second type as *tagging*. As an illustration, consider the delete operation shown in Figure 1(b). Let U denote the root node of the subtree β. In the example, the edge (Y, Z) will be flagged whereas the edge (Y, U) will be tagged. Once an edge has been marked, it cannot be changed, that is, it cannot point to another node. So, once both the child edges of a node have been marked, it cannot be used as an injection point of another modify operation. This helps to coordinate between conflicting modify operations.

[1] It may be an ancestor node in general if multiple delete operations are in progress along the access path.

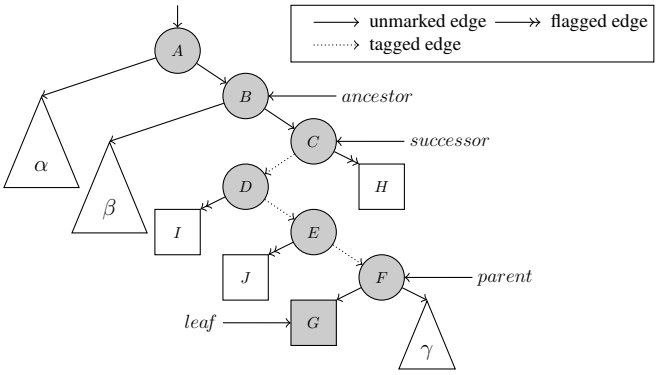


Figure 2: An illustration of the information returned by a seek phase. The figure shows an example of a suffix of an access path in the tree (consisting of shaded nodes).

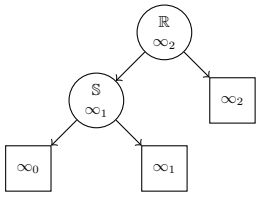


Figure 3: Sentinel nodes and keys used in our algorithm ($\infty_0 < \infty_1 < \infty_2$). The figure shows an empty tree. The two internal nodes are denoted $\mathbb{R}$ and $\mathbb{S}$.

To enable the flagging or tagging of an edge, we steal two bits from each child address stored at a node; we denote the two bits by $flag$ and tag. If $flag$ (respectively, tag) bit in a child address has value 1, then we say that the (corresponding outgoing) edge has been flagged (respectively, tagged); otherwise we say that the edge is unflagged (respectively, untagged).

The pseudo-code of our algorithm is shown in Algorithm 1-Algorithm 4. In the pseudocode, we use the $addr()$ function to obtain the address of a field and $read()$ function to read the contents of a memory location. Also, if multiple subfields are packed into a single field (or word), then we use the symbol $\rightsquigarrow$ to refer to a specific subfield (*e.g.*, $left \rightsquigarrow flag$, $left \rightsquigarrow address$).

The structure of a tree node is shown in lines 1-5.

We next describe the details of the *seek* phase, which is executed by all operations after which we describe the search, insert and delete operations in detail.

3.2.1 Seek Phase

As explained earlier, in a seek phase, an operation traverses the tree along a simple path from the root node to a leaf node, which is referred to as the access path. At each internal node, the operation either follows the left pointer or the right

pointer depending on how the given key compares with the node's key. The seek phase returns a *seek record*, which contains the addresses of *four* nodes encountered on the access path. Two of the nodes are the *leaf* node, which is the last node on the access path, and its *parent* node, which is the second-to-last node on the access path. Let (X, Y) denote the *last* untagged edge encountered by the operation on the access path *before* visiting the parent node. The other two nodes whose addresses are stored in the seek record are nodes X, referred to as the *ancestor* node, and Y, referred to as the *successor* node. The four fields of the seek record are named $leaf$, $parent$, $ancestor$ and $successor$. For an illustration, please refer to Figure 2. Note that, if there is no conflicting delete operation in progress, then the successor node is the same as the parent node and the ancestor node is the same as the grandparent node (of the leaf node on the access path). Also note that all nodes on the access path starting from the successor node to the parent node (not inclusive) are in the process of being removed from the tree.

To ensure that all the addresses of a seek record are always well-defined, we assume the presence of three *sentinel* keys, ∞_0, ∞_1 and ∞_2, where $\infty_0 < \infty_1 < \infty_2$. The sentinel keys are greater than all other keys, and are never removed from the tree. For an illustration of a tree containing the sentinel keys, please refer to Figure 3. In the figure, $\mathbb{R}$ and $\mathbb{S}$ are two internal nodes, which are also never removed from the tree. As a result, the parent and grandparent nodes of the leaf node on the access path always exist. Also, note that none of the outgoing edges of $\mathbb{R}$ and $\mathbb{S}$ can ever be marked.

The structure of a seek record is shown in lines 6-11. The pseudo-code of the seek phase is given in lines 13-33. First, the seek record is initialized using the two internal sentinel nodes (lines 15-21). The $ancestor$ pointer is initialized to the root node of the tree, $\mathbb{R}$, the $successor$ and $parent$ pointers to its left child, $\mathbb{S}$, and the $leaf$ pointer to the left child of $\mathbb{S}$. Next, the tree is traversed using a loop (lines 22-32). In each iteration of the loop, if the access path can be extended (line 22), then the edge between the current $parent$ and $leaf$ nodes is examined. If it is untagged, the $ancestor$ and $successor$ pointers are advanced to point to the $parent$ and $leaf$ nodes, respectively (lines 24 & 25). Otherwise, the $ancestor$ and $successor$ pointers are not modified. Finally, the $parent$ and $leaf$ pointers are advanced (lines 26 & 27), thereby extending the access path. The loop terminates when the $leaf$ pointer points to a leaf node, after which the seek function returns (line 33).

3.2.2 Search Operation

A search operation first executes the seek phase at the end of which it compares the given key with the key stored in the leaf node of the seek record. If the two keys match, then the operation returns true; otherwise, it returns false; The pseudo-code of the search operation is given in lines 34-39.

3.2.3 Insert Operation

An insert operation starts by executing the seek phase at the end of which it compares the given key with the key stored in the leaf node of the seek record. If the two keys match, then the operation simply returns false. Otherwise, it advances to the execution phase.

Let k denote the key to be inserted into the tree and $\mathcal{R}$ denote the seek record returned by the seek phase. Also, let k' denote the key stored in $\mathcal{R} \rightarrow leaf$. The insert operation first creates two new nodes: an internal node $newInternal$ and a leaf node $newLeaf$. The key field in $newLeaf$ is initialized to k. The key field in $newInternal$ is initialized to $\max(k, k')$. If $k < k'$, then $newLeaf$ becomes the left child of $newInternal$; otherwise it becomes the right child of $newInternal$. The sibling pointer of $newLeaf$ in $newInternal$ is set to $\mathcal{R} \rightarrow leaf$. Finally, the insert operation tries to replace the edge $(\mathcal{R} \rightarrow parent, \mathcal{R} \rightarrow leaf)$ with the edge $(\mathcal{R} \rightarrow parent, newInternal)$ using a CAS instruction on the appropriate $child$ field of $\mathcal{R} \rightarrow parent$. If the CAS instruction succeeds, then the insert operation has completed. Otherwise, the insert operation performs helping if needed and then tries again by restarting from the seek phase.

We now describe how the insert operation determines if it needs to perform helping. The insert operation ascertains that the edge from $\mathcal{R} \rightarrow parent$ to $\mathcal{R} \rightarrow leaf$ still exists and has been marked (flagged or tagged). If yes, then clearly it implies that a concurrent delete operation is trying to remove $\mathcal{R} \rightarrow parent$ from the tree (assuming that $\mathcal{R} \rightarrow parent$ has not already been removed from the tree). In this case, the insert operation performs helping by executing the last two steps of a delete operation, which will result in $\mathcal{R} \rightarrow parent$ and one of its children being removed from the tree. Note that, if the edge from $\mathcal{R} \rightarrow parent$ to $\mathcal{R} \rightarrow leaf$ does not exist, then the injection point of the insert operation has changed and no helping is performed. After performing the helping if needed as described above, the insert operation restarts from the seek phase.

To summarize, the execution of an insert operation consists of an alternating sequence of seek and execution phases until the operation terminates.

The pseudo-code of the insert operation is given in lines 40-59. The pseudo-code of its execution phase is given in lines 44-59. The steps for creating and installing a new subtree are given in lines 50-51. In case the CAS instruction fails, the steps for helping a delete operation are given in lines 56-57. Helping, if needed, is performed by invoking a cleanup routine, which is given in lines 88-109 (explained in the next section).

3.2.4 Delete Operation

A delete operation has two modes: *injection* and *cleanup*. In the injection mode, its objective is to locate the leaf node that contains the given key and mark it by flagging its incoming

edge. In the cleanup mode, its objective is to physically remove the leaf node that it flagged in the injection mode (and its parent) from the tree.

A delete operation starts by executing the seek phase at the end of which it compares its key with the key stored in the leaf node of the seek record. If the two keys do not match, then the operation simply returns false. Otherwise, it advances to the execution phase.

Let $\mathcal{R}$ denote the seek record returned by the seek phase. In the injection mode, the delete operation tries to flag the edge from $\mathcal{R} \rightarrow parent$ to $\mathcal{R} \rightarrow leaf$ using a CAS instruction; flagging the edge involves setting the $flag$ bit in the $child$ ($left$ or $right$) field of $\mathcal{R} \rightarrow parent$ that points to $\mathcal{R} \rightarrow leaf$ to 1. If the CAS instruction succeeds, then the delete operation is guaranteed to complete eventually, and the operation enters the cleanup mode. However, if the CAS instruction fails, then the delete operation performs helping in the same way as discussed in the case of insert operation by invoking the cleanup routine if needed after which it tries again by restarting from the seek phase.

In the cleanup mode, the delete operation tries to remove nodes $\mathcal{R} \rightarrow parent$ and $\mathcal{R} \rightarrow leaf$ from the tree using the following two steps. For ease of exposition, without loss of generality, assume that $\mathcal{R} \rightarrow leaf$ is the left child of $\mathcal{R} \rightarrow parent$. The other case when $\mathcal{R} \rightarrow leaf$ is the right child of $\mathcal{R} \rightarrow parent$ is symmetric and can be easily derived. In the first step, the delete operation tags the sibling edge of $\mathcal{R} \rightarrow parent$, which points to the right child of $\mathcal{R} \rightarrow parent$, using a BTS instruction; tagging the edge involves setting the tag bit in the $right$ field of $\mathcal{R} \rightarrow parent$ to 1. Note that this step is guaranteed to succeed. Also, none of the children edges of $\mathcal{R} \rightarrow parent$ can now change (point to different nodes). This also implies that $\mathcal{R} \rightarrow parent$ can no longer act as the injection point of any modify operation from now on. Let S denote the right child of $\mathcal{R} \rightarrow parent$. In the second step, the delete operation tries to replace the edge $(\mathcal{R} \rightarrow ancestor, \mathcal{R} \rightarrow successor)$ with the edge $(\mathcal{R} \rightarrow ancestor, S)$ using a CAS instruction on the appropriate $child$ field of $\mathcal{R} \rightarrow ancestor$. Note that the edge $(\mathcal{R} \rightarrow parent, S)$ may have been concurrently flagged by another delete operation which is trying to remove the key stored in S from the tree (in this case S must be a leaf node and the edge must have been flagged before it was tagged). The value of the $flag$ bit in the $right$ field of $\mathcal{R} \rightarrow parent$ is copied to the new edge being created from $\mathcal{R} \rightarrow ancestor$ to S. This ensures that if the edge $(\mathcal{R} \rightarrow parent, S)$ is flagged, then the new edge $(\mathcal{R} \rightarrow ancestor, S)$ is also flagged. Note that after an edge has been tagged, it cannot be flagged. So the value of its $flag$ bit of the sibling edge cannot change after it has been tagged. If the CAS instruction succeeds, then the delete operation has completed. Otherwise, the delete operation executes the seek phase again.

Let $\mathcal{S}$ denote the seek record returned by the new seek phase. If $\mathcal{R} \rightarrow leaf$ and $\mathcal{S} \rightarrow leaf$ do not match, then the leaf node flagged for deletion has already been removed by another modify operation (via helping) and the delete operation terminates. Otherwise, it repeats the two steps described above (tag the sibling edge and switch the appropriate child pointer at the ancestor node) using the new seek record $\mathcal{S}$. Again, if the second step of the clean up fails (since it involves performing a CAS instruction), then the operation repeats the seek phase to obtain a new seek record with a fresh set of the four node addresses. Note that the two steps for removing the nodes to clean up the tree may need to be repeated multiple times but the number of repetitions is guaranteed to be finite. This is because, once the edge to a leaf node has been flagged, the access path leading up to that leaf node cannot gain any new internal nodes. So every time, the clean up fails, it can be verified that either the access path has lost one or more internal nodes or the last untagged edge on the access path has moved closer to the root node.

To summarize, the execution of a delete operation consists of an alternating sequence of seek and execution phases until the operation terminates. It first executes the two phases in injection mode and then in cleanup mode.

As an illustration, consider Figure 2. Let K denote the root node of the subtree γ. In the example, to delete the key stored at G, a delete operation first flags the edge (F, G), then tags the edge (G, K) and finally replaces the edge (B, C) with the edge (B, K). Note that this will cause multiple leaf nodes (H, I and J all which are in the process of being deleted) to be removed from the tree in a single step.

The pseudo-code of the delete operation is given in lines 60-87. The pseudo-code of its execution phase is given in lines 73-87. The pseudo-code for flagging the edge to the leaf node that contains the given key is given in line 73. The pseudo-code for cleaning up the tree is given in lines 82-87. Since the code for cleaning up the tree may also be executed by another modify operation trying to the help the delete operation complete, its core steps are presented as a separate routine in lines 88-109, which may be invoked from line 57, line 76 or line 81.

3.3 Correctness Proof

To prove that our algorithm is correct, we show that it generates only linearizable executions and guarantees that some operation eventually completes. Due to space constraints, we only provide a proof sketch.

Linearizability: We start by making a few observations about our algorithm. First, the key stored at a node, once initialized, never changes. Second, an internal node always stays an internal node and a leaf node always stays a leaf node. Third, before an internal node is removed from the tree, both its child edges (or addresses) are marked after which the edges cannot change (point to another node). This implies that the first CAS instruction of a modify operation only succeeds if its injection point is still part of the tree at the time the CAS instruction is performed.

```
1  struct Node {
2      Key key;
       // each of the next two fields contains three
          sub-fields packed into a single word: one
          bit for flagging (flag), one bit for tagging
          (tag) and the remaining bits for a node
          address (address)
3      {Boolean, Boolean, NodePtr} left;
4      {Boolean, Boolean, NodePtr} right;
5  };
6  struct SeekRecord {
7      NodePtr ancestor;
8      NodePtr successor;
9      NodePtr parent;
10     NodePtr leaf;
11 };
   // create a new seek record, which is used by all
      operations
12 struct SeekRecordPtr seekRecord := allocate a new seek record;
13 seek( key, seekRecord )
14 begin
       // initialize the seek record using the
          sentinel nodes
15     seekRecord→ancestor := ℝ;
16     seekRecord→successor := 𝕊;
17     seekRecord→parent := 𝕊;
18     seekRecord→leaf := (𝕊→left)⇝address;
       // initialize other variables used in the
          traversal
19     parentField := (seekRecord→parent)→left;
20     currentField := (seekRecord→leaf)→left;
21     current := currentField⇝address;
22     while current ≠ null do
           // move down the tree
           // check if the edge from the (current)
              parent node in the access path is tagged
23         if not (parentField⇝tag) then
               // found an untagged edge in the access
                  path; advance ancestor and successor
                  pointers
24             seekRecord→ancestor :=
                                  seekRecord→parent;
25             seekRecord→successor :=
                                  seekRecord→leaf;
           // advance parent and leaf pointers
26         seekRecord→parent := seekRecord→leaf;
27         seekRecord→leaf := current;
           // update other variables used in traversal
28         parentField := currentField;
29         if key < current→key then
30             currentField := current→left;
31         else currentField := current→right ;
32         current := currentField⇝address;
       // traversal complete
33     return;
```

Algorithm 1: The definitions of the node structure and the seek record along with the pseudo-code of the seek routine.

It can be shown that the traversal of the tree during a seek phase never takes a "wrong turn". This means that the injection point of a modify operation as returned by a seek phase is always a "correct" node if the operation eventually completes. This, in turn, can be used to prove that our algorithm always maintains a legal binary search tree.

```
34 Boolean search( key )
35 begin
36     seek( key, seekRecord );
37     if (seekRecord→leaf)→key = key then
38         return true;          // key present in the tree
39     else return false;    // key not present in the tree
40 Boolean insert( key )
41 begin
42     while true do
43         seek( key, seekRecord );
44         if (seekRecord→leaf)→key ≠ key then
               // key not present in the tree
45             parent := seekRecord→parent;
46             leaf := seekRecord→leaf;
               // obtain the address of the child field
                  that needs to be modified
47             if key < parent→key then
48                 childAddr := addr( parent→left );
49             else childAddr := addr( parent→right ) ;
50             create two nodes newInternal and
                   newLeaf and initialize them appropriately;
               // try to add the new nodes to the tree
51             result := CAS( childAddr, {0, 0, leaf},
                                 {0, 0, newInternal});
52             if result then
53                 return true;     // insertion successful
54             else
                   // insertion failed; help the
                      conflicting delete operation
55                 {flag, tag, address} := read( childAddr );
56                 if (address = leaf) and (flag or tag) then
                       // address of the child has not
                          changed and either the leaf
                          node or its sibling has been
                          flagged for deletion
57                     cleanup( key, seekRecord );
58         else
               // key already present in the tree
59             return false;
```

Algorithm 2: The pseudo-code of the search and insert operations.

Now, to establish the linearizability property, we specify the *linearization point* of every operation. Note that modify operations that do not change the tree are treated as search operations. Also, search operations are partitioned into two types: those that find the key in the tree (search-hit) and those that do not (search-miss). The linearization point of an insert operation is defined to be the point at which it performs its CAS instruction successfully. The linearization point of a delete operation is defined to be the point at which the CAS instruction that removes its target key from the tree is performed. Note that this instruction may be performed by another operation. Also, if multiple keys are removed from the key by a single CAS instruction, then the linearization points of the corresponding delete operations can be ordered arbitrarily since all delete operations must have distinct target keys. For a search-hit operation, we consider two cases. If the leaf node returned by the seek phase of the operation is still part of the tree when the seek phase completes, then its

```
60 Boolean delete( key )
61 begin
       // start the operation in the injection mode
62     mode := INJECTION;
63     while true do
64         seek( key, seekRecord );
65         parent := seekRecord→parent;
           // obtain the address of the child field
              that needs to be modified
66         if key < parent→key then
67             childAddr := addr( parent→left );
68         else childAddr := addr( parent→right ) ;
69         if mode = INJECTION then
               // injection mode: check if the key is
                  present in the tree
70             leaf := seekRecord→leaf;
71             if leaf→key ≠ key then
                   // key not present in the tree
72                 return false;
               // inject the delete operation into the
                  tree
73             result := CAS( childAddr, {0, 0, leaf},
                                {1, 0, leaf});
74             if result then
                   // advance to the cleanup mode and
                      try to remove the leaf node from
                      the tree
75                 mode := CLEANUP;
76                 done := cleanup( key, seekRecord );
77                 if done then return true;
78             else
79                 {flag, tag, address} := read( childAddr );
80                 if (address = leaf) and (flag or tag) then
                       // address of the child has not
                          changed and either the leaf
                          node or its sibling has been
                          flagged for deletion
81                     cleanup( key, seekRecord );
82         else
               // cleanup mode: check if the leaf node
                  that was flagged in the injection
                  mode is still present in the tree
83             if seekRecord→leaf ≠ leaf then
                   // leaf node no longer present in
                      the tree
84                 return true;
85             else
                   // the leaf node is still present in
                      the tree; remove it
86                 done := cleanup( key, seekRecord );
87                 if done then return true;
```

Algorithm 3: The pseudo-code of the delete operation.

```
88 Boolean cleanup( key, seekRecord )
89 begin
       // retrieve all addresses stored in the seek
          record for easy access
90     ancestor := seekRecord→ancestor;
91     successor := seekRecord→successor;
92     parent := seekRecord→parent;
93     leaf := seekRecord→leaf;
       // obtain the addresses on which atomic
          instructions will be executed
       // first obtain the address of the field of the
          ancestor node that will be modified
94     if key < ancestor→key then
95         successorAddr := addr( ancestor→left );
96     else successorAddr := addr( ancestor→right ) ;
       // now obtain the addresses of the child fields
          of the parent node
97     if key < parent→key then
98         childAddr := addr( parent→left );
99         siblingAddr := addr( parent→right );
100    else
101        childAddr := addr( parent→right );
102        siblingAddr := addr( parent→left );
103    {flag, _, _} := read( childAddr );
104    if not (flag) then
           // the leaf node is not flagged for
              deletion; thus the sibling node must be
              flagged for deletion
           // switch the sibling address
105        siblingAddr := childAddr;
       // tag the sibling edge; no modify operation
          can occur at this edge now
106    BTS( siblingAddr, TAG );
       // read the flag and address fields
107    {flag, _, address} := read( siblingAddr );
       // the flag field will be copied to the new
          edge that will be created
       // make the sibling node a direct child of the
          ancestor node
108    result := CAS( successorAddr, {0, 0, successor},
                      {flag, 0, address} );
109    return result;
```

Algorithm 4: The pseudo-code of the cleanup routine.

linearization point is defined to be the point at which the seek phase ends. Otherwise, its linearization point is defined to be the point just before the leaf node is removed from the tree. For a search-miss operation, we again consider two cases. If there is one or more delete operation on the same key whose linearization point overlaps with the search-miss operation, then the point just after the *last* such linearization point is taken to be the linearization point of the search-miss operation. Otherwise, the linearization point of the search-miss operation is defined to be the point at which the operation starts. It can be proved that, when the operations are ordered according to their linearization points, then the resulting sequence of operations is legal.

Lock-Freedom: Note that, if the system reaches a state after which no modify operation completes, then every search operation is guaranteed to complete after that because the tree will not undergo any further structural changes. Thus it suffices to prove that modify operations are lock-free. Assume, on the contrary, that the system can reach a state in which there is a process, say p, with a pending modify operation, say α, that takes an infinite number of steps but still no modify operation completes after that. This implies that there is a time, say t, after which the tree stops changing. This, in turn, means that every instance of the seek phase of α that starts after t traverses the *same* access path and returns the *same* node as the injection point for α. Assume that α is an insert operation. Since the tree does not change after t, the repeated failure of α is because α repeatedly finds

the last edge along its access path to be flagged or tagged, thereby implying that there is a delete operation β in progress that conflicts with α. Clearly, in our algorithm, every time p fails to inject α into the tree, it will try to help the conflicting delete operation complete. Thus, it can be verified that, after t, every time p fails to help β complete, the depth of the $ancestor$ pointer returned by the subsequent seek phase of α will strictly decrease. Since the depth of the $ancestor$ pointer can only decrease a finite number of times, eventually, β is guaranteed to complete. A similar argument can be given if α is a delete operation.

4. Experimental Evaluation

In this section, we describe the results of the comparative evaluation of different implementations of a concurrent BST, including that based on the algorithm described in this paper.

Other Concurrent Binary Search Tree Implementations: For our experiments, we considered three other implementations of concurrent BST besides our algorithm, denoted by NM-BST. They are: (i) the lock-based implementation based on Bronson *et al.*'s algorithm [3], denoted by BCCO-BST, and (ii) two lock-free implementations: one based on Ellen *et al.*'s algorithm [12], denoted by EFRB-BST, and one based on Howley and Jones's algorithm [23], denoted by HJ-BST, For a fair comparison, no memory reclamation is performed in any of the implementations.

Experimental Setup: We conducted our experiments on a 4-processor AMD Opteron 6276 2.3 GHz machine, with 16 cores per processor (yielding 64 cores in total), 256 GB of RAM, running the x86_64 version of Fedora release 16.

All implementations were written in C. The code for HJ-BST was obtained directly from the authors [23]. The code for EFRB-BST and BCCO-BST were derived from the C++ implementations of Wicht [7]. All implementations were compiled using g++ version 4.6.3 with optimization level set to O3. We used Google's TCMalloc library [16] for dynamic memory allocation because it has significantly lower overhead for dynamically allocating objects in a multithreaded program.

Our experimental setup is similar to the one used by Bronson *et al.* [3] and Howley and Jones [23]. 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 consider four different key ranges: 1000 (1K), 10,000 (10K), 100,000 (100K) and 1 million (1M) keys. To ensure consistent results, as in [6], rather than starting with an empty tree, we *pre-populated* the tree prior to starting the simulation run.
2. **Relative Distribution of Various Operations:** As in [23], we considered three different workload distributions: (a) *write-dominated workload:* 0% search, 50% insert and 50% delete. (b) *mixed workload:* 70% search, 20% insert and 10% delete, and (c) *read-dominated workload:* 90% search, 9% insert and 1% delete,
3. **Maximum Degree of Contention:** This depends on the number of threads. We conducted our experiments for 1, 2, 4, 8, 16, 32, 64, 128 and 256 threads.

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

Simulation Results: Each simulation run was carried out for 30 seconds and the results were averaged over multiple runs. Figure 4 shows the results of our experiments.

It is clear that our algorithm has the best throughput for all the key space sizes under the write-dominated workload. For the 1K key space size, the NM-BST scheme performs 54%-113% better than the BCCO-BST scheme, 11%-228% better than the HJ-BST scheme, and 11%-122% better than the EFRB-BST scheme. This is due to the fact that modify operations in our algorithm allow greater concurrency and allocate fewer objects (discussed in detail in Section 5). For the 100K key space size, the NM-BST scheme outperforms the BCCO-BST scheme by 33%-67%, the HJ-BST scheme by 5%-187%, and the EFRB-BSTscheme by 9%-125% . Finally, for the 1M key space size, the NM-BST scheme outperforms the BCCO-BST scheme by 27%-74%, the HJ-BST scheme by 4%-135%, and the EFRB-BST scheme by 18%-80%. Note that our algorithm uses an external representation of the tree, which causes operations to traverse longer paths compared to the BCCO-BST and HJ-BST schemes. However, in spite of this, modify operations in our scheme are significantly faster than those in all the other schemes.

As the percentage of search operations is increased in the workload, however, the relative performance of the NM-BST scheme drops with an increase in the tree size. For the mixed workload, for the 10K key space size, the NM-BST scheme outperforms the next best HJ-BST scheme by 6%-80%, which falls to 2%-40% for the 100K key space size. For the 1M key space size, however, the HJ-BST scheme beats the NM-BST scheme by 4%-20%, for up to 16 threads. However, for this key space size, the NM-BST scheme still outperforms the BCCO-BST scheme by 6%-30%, and the EFRB-BST scheme by 3%-23%.

The impact that the external tree structure has on operations is clearly visible in the graphs for the read-dominated workload. Here, the NM-BST and EFRB-BST schemes outperform all the other schemes for the 1K and 10K key space sizes, but are beaten by the HJ-BST scheme for the 100K and 1M key space sizes. For the 1M key range, the HJ-BST scheme outperforms the next best NM-BST scheme by 2%-17%. The NM-BST scheme outperforms the EFRB-BST scheme by 12%-20% and the BCCO-BST scheme by 5%-50%.

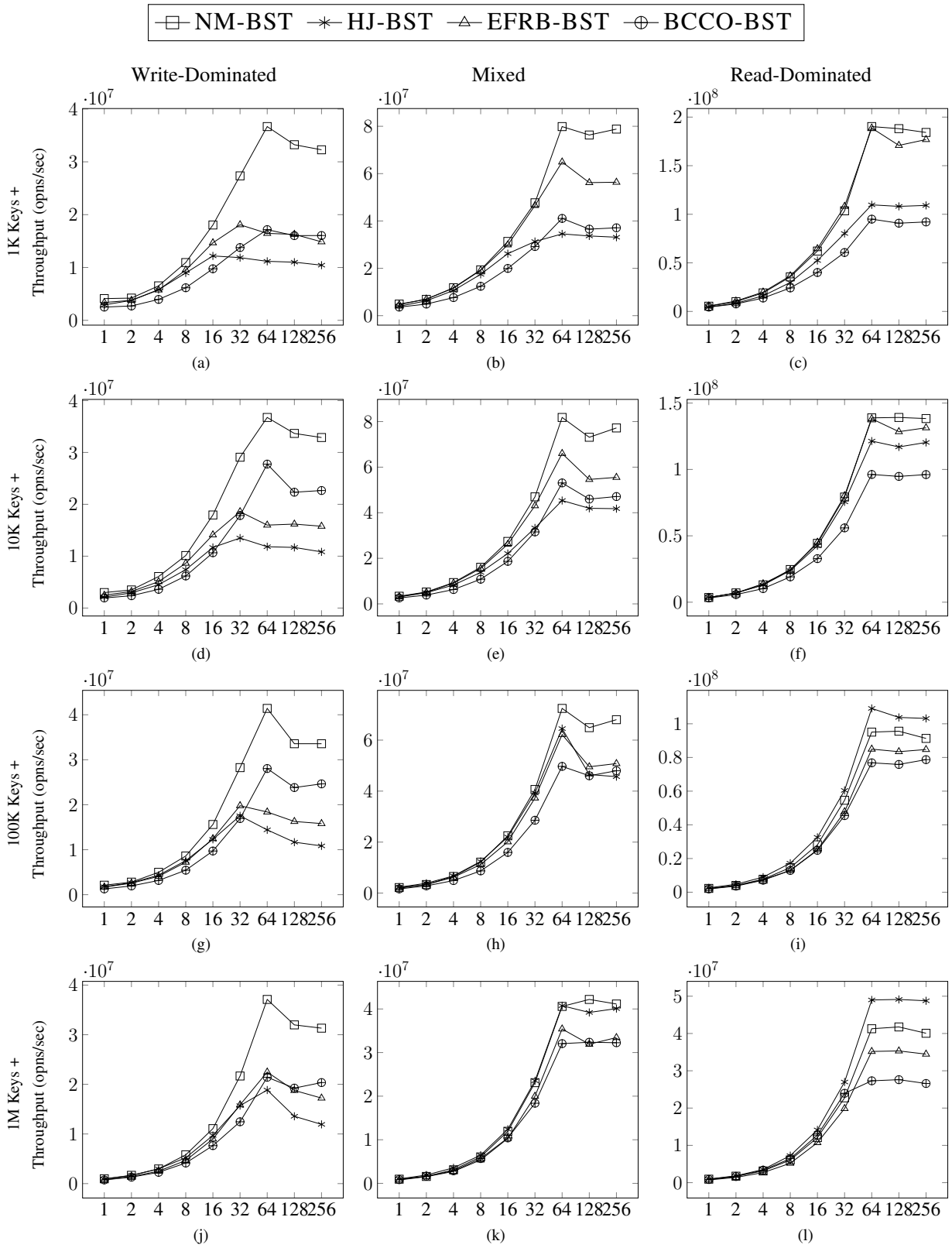


Figure 4: A comparison of the throughput of different algorithms. Each row of graphs represents a different key space size, and each column represents a different workload. Higher the throughput, better the performance of the algorithm.

Table 1: Comparison of different lock-free algorithms for a BST (in the absence of contention and with no memory reclamation).

Algorithm	Number of Objects Allocated		Number of Atomic Instructions Executed	
	Insert	Delete	Insert	Delete
Ellen *et al.*	4	1	3	4
Howley and Jones	2	1	3	up to 9
This work	2	0	1	3

5. Discussion

There are many reasons why our algorithm exhibited much better performance than other lock-free algorithms for a BST in our experiments. They are described as follows.

First, our algorithm has a smaller contention window than Ellen *et al.*'s algorithm. In fact, it can be verified that any two operations that can execute concurrently in Ellen *et al.*'s algorithm can also execute concurrently in our algorithm, but *not vice versa*. (For reference, in Ellen *et al.*'s algorithm, an insert operation needs to "lock" the parent node of the leaf node, and a delete operation needs to "lock" the parent and grandparent nodes of the leaf node.) One of the reasons for smaller contention window is that our algorithm operates at edge-level (marks edges) whereas Ellen *et al.*'s algorithm operates at node-level (marks nodes). For example, consider the tree in Figure 5. Some examples of operations that can be performed concurrently in our algorithm (because they involve disjoint set of edges) but not in Ellen *et al.*'s algorithm (because they involve one or more common nodes) are: `insert(40)` and `insert(60)`, and `delete(25)` and `delete(125)`. Comparing the contention window of our algorithm with that of Howley and Jones directly is not possible because the two algorithms use different internal representations (internal vs external BST).

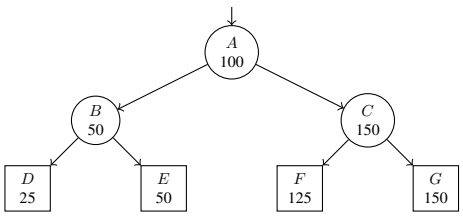


Figure 5: A sub-tree of a binary search tree.

Second, our algorithm allocates fewer objects and executes fewer atomic instructions per modify operation than the other two algorithms (see Table 1).

Third, a modify (insert as well as delete) operation has to successfully execute only a single atomic instruction to guarantee completion. (The remaining atomic instructions in a delete operation are required for "cleaning up" the tree and do not affect the outcome of the operation.) As a result, unlike as in Ellen *et al.*'s algorithm, there is no notion of aborting a delete operation if the subsequent steps could not be performed successfully.

Fourth, our algorithm does not use explicit objects for coordination between conflicting operations ($DInfo$ and $IInfo$ objects in Ellen *et al.*'s algorithm and `ChildCASOp` and `RelocateOp` objects in Howley and Jones's algorithm). These objects are used for helping to ensure the lock-freedom property. In fact, in our algorithm, helping is performed for delete operations only; no helping is performed for insert operations. This is desirable since helping increases the overhead of an operation and may cause duplication of work. Furthermore, instead of using explicit objects for coordination, our algorithm steals a small number of bits from child addresses stored in a node to enable coordination between operations.

Finally, in our algorithm, multiple leaf nodes may be removed from the tree in a single step. For example, in Figure 2, when edge (B, C) is replaced with edge (B, K) at node B, where K is the root node of the subtree γ, then it not only removes the leaf node G from the tree but also removes the leaf nodes H, I and J along with it.

6. Conclusion and Future Work

In this paper, we have presented a new lock-free algorithm for concurrent manipulation of a binary search tree using single-word compare-and-swap and bit-test-and-set atomic instructions. Using a combination of several ideas (*e.g.*, marking an edge rather than a node, not using an explicit separate object for coordination among operations), we have not only reduced the contention between modify operations but also lowered the overhead of a modify operation. As we mentioned earlier, our algorithm can be easily modified to use only compare-and-swap instructions. Our experiments indicate that, when contention is high (tree size is small or workload is write-dominated), our algorithm significantly outperforms *all other existing* algorithms for a concurrent BST that we are aware of.

Ellen *et al.*'s lock-free algorithm for a BST has been extended to develop other lock-free tree-based data structures such as k-ary search trees [4] and binary tries [30]. As a future work, we plan to use the ideas in this work to develop more efficient lock-free algorithms for k-ary search trees and binary tries than those in [4, 30]. Further, we also plan to extend our lock-free algorithm to add support for other tree operations such as replace [30] and snapshot [29].

References

[1] 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.

[2] 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.

[3] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A Practical Concurrent Binary Search Tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 257–268, Jan. 2010.

[4] T. Brown and J. Helga. Non-Blocking k-ary Search Trees. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, pages 207–221, 2011.

[5] T. Brown, F. Ellen, and E. Ruppert. Pragmatic Primitives for Non-blocking Data Structures. In *Proceedings of the33rdACM Symposium on Principles of Distributed Computing (PODC)*, pages 207–221, 2013.

[6] T. Brown, F. Ellen, and E. Ruppert. A General Technique for Non-blocking Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014.

[7] B.Wicht. Binary Trees Implementations Comparison for Multicore Programming. Technical report, Switzerland HES-SO University of Applied Science, June 2012. Code available at: `https://github.com/wichtounet/btrees/`.

[8] P. Chuong, F. Ellen, and V. Ramachandran. A Universal Construction for Wait-Free Transaction Friendly Data Structures. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 335–344, 2010.

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

[10] T. Crain, V. Gramoli, and M. Raynal. A Contention-Friendly Binary Search Tree. In *Proceedings of the European Conference on Parallel and Distributed Computing (Euro-Par)*, pages 229–240, Aachen, Germany, 2013.

[11] D. Drachsler, M. Vechev, and E. Yahav. Practical Concurrent Binary Search Trees via Logical Ordering. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Orlando, Florida, USA, Feb. 2014.

[12] 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.

[13] P. Fatourou and N. D. Kallimanis. A Highly-Efficient Wait-Free Universal Construction. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 325–334, 2011.

[14] 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.

[15] K. Fraser and T. L. Harris. Concurrent Programming Without Locks. *ACM Transactions on Computer Systems*, 25(2), May 2007.

[16] S. Ghemawat and P. Menage. TCMalloc: Thread-Caching Malloc. URL `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[17] T. Harris. A Pragmatic Implementation of Non-blocking Linked-lists. *Distributed Computing (DC)*, pages 300–314, 2001.

[18] M. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1): 124–149, Jan. 1991.

[19] M. Herlihy. A Methodology for Implementing Highly Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

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

[21] 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.

[22] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 522–529, 2003.

[23] 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.

[24] J. H. Kim, H. Cameron, and P. Graham. Lock-Free Red-Black Trees Using CAS. *Concurrency and Computation: Practice and Experience*, pages 1–40, 2006.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent Tries with Efficient Non-Blocking Snapshots. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 151–160, 2012.

[30] N. Shafiei. Non-blocking Patricia Tries with Replace Operations. In *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 216–225, JUL 2013.

[31] H. Sundell and P. Tsigas. Scalable and Lock-Free Concurrent Dictionaries. In *Proceedings of the 19th Annual Symposium on Selected Areas in Cryptography*, pages 1438–1445, Mar. 2004.