Improving Efficacy of Concurrent Internal Binary Search Trees using Local Recovery*

Arunmoezhi Ramachandran
The University of Texas at Dallas
Richardson, Texas
arunmoezhi@gmail.com

Neeraj Mittal†
The University of Texas at Dallas
Richardson, Texas
neerajm@utdallas.edu

ABSTRACT

Binary Search Tree (BST) is an important data structure for managing ordered data. Many algorithms have been proposed for concurrent manipulation of a binary search tree in an asynchronous shared memory system that supports search, insert and delete operations based on both external and internal representations of a search tree.

An important step in executing an operation on a tree is to traverse the tree from top-to-down in order to locate the operation's window. A process may need to perform this traversal several times to handle failures occurring due to other processes performing concurrent updates on the tree. Most concurrent algorithms that have been proposed so far use a naïve approach and simply restart the traversal from the root of the tree.

In this work, we present a new approach to recover from such failures more efficiently for certain types of concurrent binary search tree algorithms using *local recovery* by restarting the traversal from the “middle” of the tree in order to locate an operation's window (tree section of interest). Our approach is sufficiently general in the sense that it can be applied to most of the existing concurrent binary search trees using *internal representation* based on both blocking as well as non-blocking approaches.

Using experimental evaluation, we demonstrate that our local recovery approach can significantly improve the performance of a concurrent internal BST by up to 124.7% when the contention is high.

CCS CONCEPTS

• **Theory of computation** → **Concurrent algorithms;**

KEYWORDS

Concurrent Data Structure, Binary Search Tree, Internal Representation, Local Recovery, Wait-Free Search

*Part of this work has appeared as a poster in the *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016.

†This work was supported, in part, by the National Science Foundation (NSF) under grant numbers CNS-1115733 and CNS-1619197.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](https://permissions.acm.org).

ICDCN '19, January 4–7, 2019, Bangalore, India

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6094-4/19/01...\$15.00

<https://doi.org/10.1145/3288599.3288615>

ACM Reference Format:

Arunmoezhi Ramachandran and Neeraj Mittal. 2019. Improving Efficacy of Concurrent Internal Binary Search Trees using Local Recovery. In *International Conference on Distributed Computing and Networking (ICDCN '19)*, January 4–7, 2019, Bangalore, India. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3288599.3288615>

1 INTRODUCTION

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

Binary search tree (BST) is one of the fundamental data structures for organizing *ordered* data (a key-value pair) that supports search, insert and delete operations. Concurrent algorithms for (unbalanced) BSTs have been proposed in [2, 3, 6–10, 13, 14, 16, 17]. Algorithms in [2, 3, 7, 8, 17] are blocking, whereas those in [6, 9, 10, 13, 14, 16] are non-blocking. Also, algorithms in [2, 3, 6, 8, 13, 16, 17] use internal representation of a BST in which all nodes store data, whereas those in [9, 10, 14] use an external representation of a BST in which only leaf nodes store data (data stored in internal nodes is used for routing purposes only).

An important step in executing an operation on a tree is to traverse the tree from top-to-down in order to locate the operation's window, which is basically a subset of nodes in the tree relevant to the operation. A process may need to perform this traversal several times to handle failures caused by other processes concurrently performing conflicting actions on the tree which can cause, for example, the data of interest to move from its original location or an atomic read-modify-write instruction to fail due to a concurrent change made to the tree. Most of the concurrent algorithms (for BSTs) that have been proposed so far use a naïve approach and simply restart the traversal from the root of the tree [2, 8, 9, 13, 14]. This is especially undesirable since the overhead of repeatedly traversing the tree may dominate all other overheads of performing an operation.

Recently, a few algorithms have been proposed in which an operation attempts to recover from a failure *locally* [6, 10]. The local recovery algorithm in [10], which is based on external representation and builds upon the concurrent BST algorithm in [9], maintains a stack of the nodes visited during the traversal of the tree and simply restarts from the last unmarked node. A node is marked before it is removed from the tree, so an unmarked node is

guaranteed to be part of the tree. Intuitively, this works because, in an external BST, keys *do not* move from one location in the tree to another. However, in a BST based on internal representation, keys may move from one location (in the tree) to another. This occurs when the key undergoing deletion resides in a binary node, which requires it to be either replaced with its predecessor (next smallest key) or its successor (next largest key). This causes two problems. First, an operation traversing the tree may fail to find the target key both at its old location and at its new location, even though the target key was continuously present in the tree. Second, it is not always safe to simply restart from the last unmarked node in the stack since the key may have moved to an ancestor of such a node. The restart approach in [10] is, however, sufficiently general that it can be applied to other concurrent BSTs based on external representation [14].

The algorithm in [6], which is based on internal representation, uses *backlink pointers* to find its way and recover from a failure while executing an operation. Their restart approach appears to be customized for their concurrent BST and is not clear how it can be extended to other concurrent BSTs based on internal representation.

Our Contributions: We present a general approach for local recovery that enables a process to recover from a failure while performing an operation by restarting the traversal from a point “close” to the operation’s window rather than the root of the tree. Our approach can be applied to many existing concurrent algorithms for maintaining BSTs using internal representation—blocking as well as non-blocking—such as those in [2, 8, 13, 17]. Further, our approach uses only local variables and does not require modifying a tree node (of the original algorithm) to store any additional information.

Using experimental evaluation, we demonstrate that our local recovery approach can yield significant speed-ups for some of these concurrent BSTs especially when the contention is high (and hence the likelihood of restarts is higher). Specifically, in our experiments, our local recovery approach increased the system throughput of the concurrent BST in [13], referred to as HJ-BST, by up to 124.7% and in [17], referred to as CASTLE, by up to 24.5%.

An additional benefit of our local recovery algorithm is that it can be used to make search operations for many concurrent internal BSTs such as those in [2, 8, 13, 17] *wait-free*. Note that a search operation in [2] is *not wait-free* even for a bounded key space in spite of the claim made otherwise in the work. The counter example given in [9] to show that its search operation is not wait-free also applies to the concurrent BST in [2].

Roadmap: The rest of the text is organized as follows. We describe our system model in Section 2. In Section 3, we present our local recovery framework and, in Section 4, we present the results of our experimental evaluation. Finally, Section 5 concludes the paper and outlines directions for future research.

2 SYSTEM MODEL

2.1 Binary Search Tree and Internal Representation

We assume that a binary search tree (BST) implements a dictionary abstract data type and supports *search*, *insert* and *delete* operations. We refer to insert and delete operations as *modify* operations. For

Figure 1: An illustration of an insert operation in an internal BST.

(a) Simple delete

(b) Complex delete

Figure 2: An illustration of a delete operation in an internal BST.

ease of exposition, we assume that a BST only stores keys instead of key-data pairs. But our work can be easily extended to the case when a BST stores key-data pairs. 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.

We assume an *internal BST* in which all nodes—leaf as well as non-leaf—store keys. We briefly describe the working of insert and delete operations in an internal BST in the absence of concurrency. Every operation first traverses the tree starting from the root node. At each traversal step, it compares the key stored in the node with the key associated with the operation, which we refer to as *target key*. If the two keys match, then the operation terminates the traversal. Otherwise, it attempts to follow one of the child pointers (left if the target key is smaller and right otherwise). If the child pointer is null, then the operation terminates the traversal; otherwise, it moves to the next node. We refer to the path traversed by an operation until the traversal terminates as *access-path* and the last node in this access-path as *terminal node*.

An insert operation creates a new node, initializes it to contain the target key, and adds it to the tree by attaching it to the terminal node (see Figure 1).

A delete operation is further categorized into two types depending on whether the terminal node is binary (complex delete) or not (simple delete). In a simple delete operation, the terminal node is removed from the tree by changing the child pointer of its parent

node to point to its non-null child node, if any (see Figure 2(a)). In a complex delete operation, the terminal node's key is replaced with its successor—the next largest key in the tree (see Figure 2(b)). Note that the next largest key is stored in the leftmost node in the right subtree of the terminal node. The node containing the successor key is then removed from the tree in the same way as in simple delete (since the left child pointer of this successor node is guaranteed to be null).

We now describe the properties of a concurrent internal BST that our local recovery algorithm uses in order to: (i) minimize the number of restarts, and (ii) find a good restart point. A node is said to be *active* if it is reachable from the root of the tree. An active node becomes *passive* once it is removed from the tree (and hence no longer reachable from the root of the tree). For example, in Figure 2, node U is active before the delete operation modifies the tree and becomes passive afterward.

Property 1 (two-step removal). A node in the tree is marked before it is removed from the tree.

Property 2 (immutability). The value of a node's field can change only if the node is not marked.

Property 3 (step-up). The value of an active node's key is monotonically non-decreasing.

The next two properties capture the way in which the search tree can undergo structural changes. A node V is said to be a *left-descendant* of a node U if V is contained in the left subtree of U . Likewise, we can define the notion of *right-descendant*.

Property 4 (persistence). Consider the tree at some time t and let U and V be two nodes that are both active at t . Consider a time t' with $t' \geq t$ such that both U and V are still active at t' . If V is a left-descendant of U at t , then V is also a left-descendant of U at t' . Likewise, if V is a right-descendant of U at t , then V is also a right-descendant of U at t' .

Given a node U that is active at time t , we use $\Gamma_t(U)$ to denote the path that exists from the root of the tree to U at t . We omit the suffix t when t refers to the current time.

Property 5 (containment). Consider an operation traversing the tree and let U denote one of the nodes encountered in the traversal path. Let $\Pi(U)$ denote the prefix of the traversed path up to and including node U , and let t_v denote the time when U was visited. Further, let t_r denote the time when U is removed from the tree. If U is never removed from the tree, then t_r is set to ∞ . Then, for all times t with $\min(t_v, t_r) \leq t \leq t_r$, the path $\Gamma_t(U)$ satisfies the following: (i) $\Gamma_t(U)$ is a subsequence of $\Pi(U)$, and (ii) all nodes in $\Pi(U) \setminus \Gamma_t(U)$ are passive at t .

Note that the concurrent BST in [2] does not satisfy the containment property since, in its complex delete operation, the terminal node is replaced with a *new* node containing the successor key (which we treat as a separate node). But this is not necessary and the algorithm still works if the terminal node is reused and only its key is replaced.

2.2 Correctness Property

We use *linearizability* [12] as the safety property. Broadly speaking, linearizability requires that an operation should appear to take

effect instantaneously at *some* point during its execution. Our local recovery algorithm preserves the liveness property of the original algorithm—whether it be deadlock-freedom or lock-freedom.

3 THE LOCAL RECOVERY ALGORITHM

We first describe the main idea behind the algorithm and then provide its formal description.

3.1 Overview of the Algorithm

As mentioned earlier, every operation on a BST involves first traversing the tree from top to down starting from the root node. Depending on the outcome of the traversal and the type of operation, the tree may then need to be modified to actually perform the operation. We refer to the period during which the tree is being traversed as *seek phase*. Further, we refer to the period during which the tree is being modified as *update phase*.

During the seek phase, the target key may move from its current location to a new location up the tree. As a result, the traversal may miss the key both at its old location as well as at its new location. For an illustration, see Figure 3. In the illustration, key 50 has moved up by five nodes. In most concurrent BST algorithms, if it is suspected that the key may have moved up the tree, then the traversal is simply restarted from the root node. Different algorithms use different approaches to detect possible key movement. For instance, in [13], the traversal is restarted if the *last right-turn* node is detected to have undergone some change. As an example, in Figure 3(a), the last right-turn node for the operation $op(50)$ is node X . On reaching the terminal node in Figure 3(d), after resuming running, $op(50)$ needs to restart since X has since been removed from the tree.

A *re-traversal* of the tree may also be required if the operation encounters any failure during the update phase. For example, in [17], which is a lock-based algorithm, update phase is aborted if, after locking the relevant edges, the validation step fails. This happens if the section of the tree that lies within the “operation's window” has undergone some change since it was last observed. In that case, the operation moves to the seek phase again.

In most concurrent BST algorithms, (a single instance of) the update phase of an operation typically tends to have constant time complexity. The seek phase is where an operation may end up spending most of its time especially if the tree is large. Hence, it is desirable to make the seek phase of an operation more efficient by: (i) reducing the number of restarts due to “suspected” key movement, and (ii) restarting the traversal from a point “close” to the operation's window. Note that local recovery may also improve the temporal reference locality of a concurrent BST thereby boosting its cache performance. This leads to two separate but related questions that any local recovery framework needs to address. First, “If a key is not found, then does the traversal need to restart?”. Second, “If the traversal needs to be restarted, then from which node should the traversal restart?”

Note that local recovery can also be performed when looking for a successor key in the case of a complex delete operation. This local recovery is somewhat easier (simply restart from the last unmarked node) since the traversal is not key-based and thus not impacted by key movements.

Figure 3: An illustration of a key moving up the tree while operations $op(40)$ and $op(50)$ are suspended (runnable but waiting for their turn) during their traversal. The access-path with respect to $op(40)$ (as well as $op(50)$) is shown with thick edges. Shaded nodes denote the anchor nodes on this path. Nodes with dotted border denote the marked nodes.

Consider an operation α , with the target key denoted by $\kappa(\alpha)$, currently traversing the tree; let $\Pi(\alpha)$ denote the path taken by α so far. For example, in Figure 3(a), considering only the subtree shown in the figure, $\Pi(op(50)) = \langle R, S, T, U, V, W, X, Y \rangle$. At each non-terminal node in the path, α either followed the left or the right child pointer.

Definition 3.1 (anchor node). A node in $\Pi(\alpha)$ is said to be an *anchor node* if α followed the node's *right* child pointer during the traversal; otherwise it is said to be a *non-anchor node*.

For example, in the path $\Pi(op(50))$, nodes S, U and X are anchor nodes, whereas nodes R, T, V, W and Y are non-anchor nodes. We assume that the first anchor node in a traversal path is always a *sentinel* node that is never marked (not shown in the figure). Of course, as an operation is traversing the tree, the tree may undergo changes as a result of which the path taken by the operation may no longer be correct. For example, in Figure 3(d), the new access-path of $op(50)$ in the subtree, which is obtained from the subtree in Figure 3(a) after applying multiple delete operations, is now given by $\langle R, S, T, U \rangle$.

Due to the step-up property, the action of an operation at a node, if it is still part of the access-path, may change (e.g., follow the left child pointer instead of the right) for an anchor node but cannot change for a non-anchor node. For example, as shown in Figure 3(a), the node U is an anchor node for $op(40)$. But due to the changes made to the tree, the key at U has now become 50, as shown in Figure 3(d). Hence, the pointer that $op(40)$ now needs to follow at U is left and not right.

For the remainder of the section, let α be an operation with key $\kappa(\alpha)$ traversing the tree and let $\Pi(\alpha)$ denote the path it has traversed so far. To explain our local recovery framework, we first define some notions. Strictly speaking, these notions are a function of time which is implicitly assumed to be current time unless specified otherwise. Due to space constraints, some of the proofs have been omitted.

Definition 3.2 (consistent node). An anchor node is said to be *consistent* with respect to α if its (current) key is less than $\kappa(\alpha)$; otherwise it is said to be *inconsistent* with respect to α . Moreover, it is said to be *strictly inconsistent* with respect to α if its key is strictly greater than $\kappa(\alpha)$.

For example, in Figure 3(d), anchor nodes S and X are still consistent with respect to $op(40)$ but node U has become inconsistent with respect to $op(40)$.

Definition 3.3 (legal node). An active node U in $\Pi(\alpha)$ is said to be *legal* with respect to α if every anchor node preceding U in $\Pi(\alpha)$ is either consistent with respect to α or passive.

LEMMA 3.4. An active node in $\Pi(\alpha)$ is legal with respect to α if and only if it is on the access-path of α in the tree.

We now describe an efficient method to test if a node in $\Pi(\alpha)$ is legal. The method is safe but not sound in the sense that it may not always be able to detect that a node is legal (and thus cause unnecessary restarts), but not *vice versa*.

Definition 3.5 (critical node). Consider two nodes U and V in $\Pi(\alpha)$. Then, V is said to be *critical* with respect to U if the following

conditions hold: (i) V precedes U in $\Pi(\alpha)$, (ii) V is an unmarked anchor node, and (iii) all anchor nodes between U and V (if any) are marked.

For example, in Figure 3(c), consider the path $\langle R, S, T, U, V, W, X, Y \rangle$. In the path, the critical node with respect to node Y is node U and not node X since X is marked. Note that critical node is guaranteed to exist assuming appropriate sentinel nodes in the tree at the top.

LEMMA 3.6 (CONSISTENCY IS PREFIX-CLOSED). *Consider two anchor nodes U and V in $\Pi(\alpha)$ such that U precedes V . Assume that both U and V are active nodes. Then, if V is consistent with respect to α , then so is U .*

Definition 3.7 (safe node). Consider a node U in $\Pi(\alpha)$ and let V denote its critical node. Then, U is said to be *safe* with respect to α if V and all anchor nodes between U and V (if any) are consistent with respect to α .

For example, in Figure 3(c), consider the path $\langle R, S, T, U, V, W, X, Y \rangle$. Node Y is safe with respect to $op(40)$. However, in Figure 3(d), node Y is no longer safe with respect to $op(40)$. The next lemma relates the notions of legal and safety.

LEMMA 3.8 (SAFETY $\implies$ LEGAL). *If an active node in $\Pi(\alpha)$ is safe with respect to α , then it is also legal with respect to α .*

LEMMA 3.9. *Let U denote the last node of the path $\Pi(\alpha)$ such that: (i) the child pointer that α needs to follow at U is null, and (ii) U 's key does not match $\kappa(\alpha)$. If U is safe with respect to α , then there exists a time during α 's traversal when $\kappa(\alpha)$ was not present in the tree.*

The main idea is to show that U is (or was at some point if no longer active) on the access-path of α in the tree. Based on the above discussion, we now describe an efficient procedure to test whether or not a node is safe.

3.1.1 Procedure to Test for Safety. To test whether or not a node U in $\Pi(\alpha)$ is safe with respect to α , we can examine the anchor nodes (preceding U) in $\Pi(\alpha)$ in the reverse order in which they were encountered during the traversal, starting from the one closest to U . We keep going as long as we find the anchor node to be consistent and marked. Note that such an anchor node does not need to be examined again when testing for safety in the future since its key value cannot change any more. We stop as soon as we encounter an anchor node, say V , that is either inconsistent or unmarked. If V is an inconsistent anchor node, then we *discard the suffix of $\Pi(\alpha)$ after V* and return false. This is because no node after V in $\Pi(\alpha)$ can act as a restart point in general. Moreover, if α is a search or delete operation, then the next lemma suggests that α can complete by announcing that $\kappa(\alpha)$ was not found in the tree.

LEMMA 3.10. *If an anchor node in $\Pi(\alpha)$ is strictly inconsistent with respect to α , then there exists a time during α 's traversal when $\kappa(\alpha)$ was not present in the tree.*

Intuitively, if $\kappa(\alpha)$ was present in the tree when α started but was not present in the tree at some point during α 's traversal, then α can be linearized after the delete operation that removed $\kappa(\alpha)$ from the tree in case α is a search or delete operation. On the other hand, if V is a consistent and unmarked node, then we return true. Note that safety is a co-stable property: if a node is safe now, then

it was safe earlier as well. We next describe how each operation (search, insert and delete) achieves local recovery.

3.1.2 Details of an Operation. Assume that α 's traversal ends in a terminal node, say U , whose key does not match $\kappa(\alpha)$. As a first step, α invokes the procedure to test whether or not U is a safe node. Note that, if the procedure returns false, then it would have truncated the path such that the last node in the truncated path is an inconsistent node, say V .

Search or Delete Operation: If the procedure returns true, then α announces that the key was not found and terminates (Lemma 3.9). On the other hand, if the procedure returns false, then α compares V 's key with $\kappa(\alpha)$. If V is found to be strictly inconsistent, then again α announces that the key was not found and terminates (Lemma 3.10). Otherwise, it implies that V 's key matches $\kappa(\alpha)$. In this case, α either announces that the key was found and terminates (search operation) or moves to the update phase to try to remove the key from the tree (delete operation). Note that a search operation does not need to restart. A delete operation does not need to restart due to key movements; it only needs to restart if a failure occurs in the update phase in which case it attempts to find a suitable node in the truncated path from where to restart the traversal (explained later).

Insert Operation: If the procedure returns true, then α moves to the update phase to try to add $\kappa(\alpha)$ to the tree (Lemma 3.9). In this case, the new node will be attached as a child node of U . On the other hand, if the procedure returns false, then α compares V 's key with $\kappa(\alpha)$. If V 's key matches $\kappa(\alpha)$, then α announces that the key was found and terminates. Otherwise, it implies that V is a strictly inconsistent node. In that case, α attempts to find a suitable node in the truncated path from where to restart the traversal (explained later).

3.1.3 Procedure to Find a Restart Point. To find a restart point, α first rolls back to an unmarked node on the path, say U , and invokes the procedure to test whether or not U is a safe node. If the procedure returns true, then U is returned as a valid restart point (Lemmas 3.4 and 3.8). Otherwise, as mentioned earlier, the procedure would have truncated the path such that the last node in the truncated path is an inconsistent node, say V . Next, α compares V 's key with $\kappa(\alpha)$. There are two possible cases depending on the outcome of the comparison.

Case 1 (V 's key matches $\kappa(\alpha)$): In this case, α either moves to the update phase (if α is a delete operation) or announces that the key was found and terminates (otherwise).

Case 2 (V 's key is greater than $\kappa(\alpha)$): In this case, α either recursively invokes the procedure to find a restart point (if α is an insert operation) or announces that the key was not found and terminates (otherwise).

3.2 Details of the Algorithm

A pseudo-code of the local recovery algorithm is given in Pseudo-codes 1-4. The pseudo-code only shows the seek phase of an algorithm and not its update phase since the update phase is algorithm-specific and is not affected by local recovery.

```

// Used to store information about a node visited during tree traversal
1 struct StackEntry
2 {
3     NodePtr node;
4     enum Direction which;
5     integer anchor;
6 };

// Used to store the path from the root node to the current node in the
// tree
7 struct State
8 {
9     StackEntry[] stack;
10    integer top;
11 };

// Used to store information about the operation currently in progress
12 struct OpRecord
13 {
14     enum Type type;
15     Key key;
16     State targetStack;
17     NodePtr injectionPoint;
18 };

// algorithm-specific fields
19 };

// Used to store the outcome of a tree traversal
20 struct SeekRecord
21 {
22     // algorithm-specific fields (e.g., target node and its parent)
23 };

```

Pseudo-code 1: Data structures used.

The local recovery algorithm assumes that the original algorithm supports the following functions: (a) `GETKEY()`, `ISMARKED()` and `GETCHILD()` return the various attributes of a tree node, (b) `ISNULL()` returns true if a reference is null and false otherwise, (c) `GETADDRESS()` returns the node address stored in a reference, if non-null, (d) `MOVE()` enables the original algorithm to move along an edge, which may invoke helping and cause traversal to restart as in [13], (e) `NEEDTOMODIFYPARENTNODE()` returns true if the operation needs to modify a child pointer field of the parent node (true for a simple delete operation), and (f) `POPULATESEEKRECORD()` copies the relevant information from the traversal state required by the algorithm into a seek record (e.g., addresses of the terminal node and its parent).

The data structures used by the local recovery algorithm are shown in lines 1-21 (see Pseudo-code 1). Note that all the data structures shown are *local* to a process and not shared among processes. A process uses three main data structures, namely `State`, `OpRecord` and `SeekRecord`. A `State` (lines 8-11) is essentially a stack used to store the nodes visited during tree traversal when looking for a key (target or successor). Note that the traversal stack satisfies the last-in-first-out (LIFO) semantics but our algorithm sometimes uses it in a non-traditional way by accessing entries in the middle of the stack. One way to implement such an “augmented” stack is to use an auto-resizing vector provided as part of C++ STL library or Java package. Each entry in a traversal stack (lines 2-6) stores the address of the node, the location of its closest anchor node (within the stack) and whether the node is a left or right child of its parent. An `OpRecord` (lines 13-18) stores information about the operation such as type and key as well two stacks: one used when looking for the target key (all operations) and one used when looking for the successor key (only complex delete operations). Finally, a `SeekRecord` (lines 20-21) is used to return the outcome of a tree traversal to the original algorithm. Its fields are algorithm-specific. For example, for CASTLE, `SeekRecord` contains three fields: (a) two addresses, namely those of the target node and its parent,

```

// Returns the number of elements in the stack
22 integer SIZE( state )
23 begin
24     return state → top + 1;
25 end

// Adds the given node to the stack along with its anchor node
26 AddToTop( state, node, which )
27 begin
28     { stack, top } := state;
29     // find the anchor node
30     if which = RIGHT then anchor := top;
31     else anchor := stack[top] → anchor;
32     // push the node into the stack
33     stack[top + 1] := { node, which, anchor };
34     state → top := top + 1;
35 end

// Removes the topmost node from the stack
36 RemoveFromTopAndRememberCritical( state )
37 begin
38     { stack, top } := state;
39     // update the anchor node of the penultimate entry if needed
40     anchor := stack[top - 1] → anchor;
41     if stack[top] → anchor < stack[anchor] → anchor then
42         stack[anchor] → anchor := stack[top] → anchor;
43     // pop the node from the stack
44     state → top := top - 1;
45 end

// Pops the stack until a given entry
46 RemoveUntil( state, index )
47 begin
48     state → top := index;
49 end

// Remember the critical node (to avoid locating it again)
50 RememberCritical( state, critical )
51 begin
52     { stack, top } := state;
53     anchor := stack[top] → anchor;
54     if critical < stack[anchor] → anchor then
55         stack[anchor] → anchor := critical;
56 end

// Returns the topmost node in the stack
57 NodePtr GetTop( state )
58 begin
59     { stack, top } := state;
60     return stack[top] → node;
61 end

// Returns the second topmost node in the stack
62 NodePtr GetSecondToTop( state )
63 begin
64     { stack, top } := state;
65     return stack[top - 1] → node;
66 end

// Returns a given entry in the stack
67 { NodePtr, enum Direction, integer }
68 GetFullEntry( state, index )
69 begin
70     { stack, top } := state;
71     if index = τ then return stack[top];
72     else return stack[index];
73 end

// initializes the traversal stack
74 InitializeTraversalState( state, type )
75 begin
76     // initialize the stack using sentinel nodes
77     // sentinel nodes are never removed from the stack
78     // a sentinel node is always a safe starting point for the traversal
79     // the actual initialization will depend on the algorithm
80 end

```

Pseudo-code 2: Some useful functions to manipulate the traversal stack.

and (b) the contents of the injection point where an insert operation needs to attach the new node.

The functions used to manipulate a traversal stack are shown in lines 22-62 (see Pseudo-code 2). The function `SIZE` (lines 22-24) returns the number of entries in the stack. The functions `GetTop` (lines 48-51) and `GetSecondToTop` (lines 52-55) return the address of the node stored in the topmost entry and the entry below it, respectively. The function `AddToTop` (lines 25-31) adds an entry to the top of the stack while `RemoveFromTopAndRememberCritical` (lines 32-38) removes an entry from the top of the stack. The function `RemoveUntil` (lines 39-41) removes the entries from the

```

// Looks for a given key in the tree (invoked by every operation)
63 boolean SEEK( opRecord, seekRecord )
64 begin
65   state := opRecord → targetStack;
66   status := CONTINUE;
67   while status = CONTINUE do
68     // find a suitable restart point in the path
69     result := FINDARESTARTPOINT( opRecord, state );
70     if not (result) then
71       // examine the last node in the path
72       status := EXAMINETOP( opRecord );
73       continue;
74     // traverse the tree starting from the topmost node in the stack
75     current := GETTOP( state );
76     while true do
77       key := GETKEY( current );
78       which := opRecord → key < key ? LEFT : RIGHT;
79       // read the next address to de-reference
80       reference := GETCHILD( current, which );
81       if ( (opRecord → key = key) or ) then
82         if ( ISNULL( reference ) ) then
83           key := GETKEY( current );
84           // either stop or backtrack & restart
85           if opRecord → key ≠ key then
86             // if an insert operation, store the injection point
87             if opRecord → type = INSERT then
88               opRecord → injectionPoint :=
89                 GETADDRESS( reference );
90             // test if the terminal node is safe
91             result := ISNODESAFE( opRecord, state );
92             if result then
93               // terminal node is a safe node
94               status := NOT_FOUND;
95             else
96               // examine the last node in the path
97               status := EXAMINETOP( opRecord );
98             else
99               // terminal node contains the matching key
100              status := FOUND;
101              // terminate the current traversal
102              break;
103             // traverse the next edge
104             address := GETADDRESS( reference );
105             if opRecord → type ∈ { INSERT, DELETE } then
106               restart := MOVE( current, address, which );
107               if restart then
108                 // the algorithm wants to restart the traversal
109                 break;
110             // push the node visited into the stack
111             ADDTOTOP( state, address, which );
112             // advance the traversal
113             current := address;
114           // initialize the seek record using the traversal results
115           POPULATESEEKRECORD( seekRecord, opRecord );
116           // return the outcome
117           return (status = FOUND ? true : false);

```

Pseudo-code 3: Seek function for target key.

top of the stack until a given point. The function REMEMBERCRITICAL (lines 42-47) updates the anchor field of the anchor node of the topmost entry in the stack. The function GETFULLENTRY (lines 56-60) returns all the three fields of a given entry in the stack. The function INITIALIZETRAVERSALSTATE (lines 61-62) initializes the traversal stack of an operation using sentinel nodes.

The function SEEK used to look for a key in the tree is shown in lines 63-99 (see Pseudo-code 3). The function first backtracks to a safe node in the stack (line 68). Initially, the start point of the traversal will be a sentinel node, which is always a safe node. The function then traverses the tree from top to down by following either the left or the right child pointer (line 75) until it either finds the key or encounters a null pointer (lines 77-90). In case

```

// Determines if the last node in the path is safe
100 boolean ISNODESAFE( opRecord, state )
101 begin
102   // examine the anchor nodes in the path one-by-one starting from the
103   // closest one
104   {*,*,critical} := GETFULLENTRY( state, T );
105   while true do
106     // retrieve the node and its anchor from the stack
107     {node,*,anchor} :=
108       GETFULLENTRY( state, critical );
109     // read the attributes of the node
110     marked := ISMARKED( node );
111     key := GETKEY( node );
112     if opRecord → key > key then
113       // the anchor node is still consistent
114       if not (marked) then
115         // the last node is safe
116         REMEMBERCRITICAL( state, critical );
117         return true;
118       else
119         // examine the previous anchor node
120         critical := anchor;
121       else
122         // the anchor node is inconsistent
123         // discard the suffix and return
124         REMOVEUNTIL( state, critical );
125         return false;
126   // Find a suitable node in the path from where to restart
127   boolean FINDARESTARTPOINT( opRecord, state )
128   begin
129     while true do
130       // backtrack until an unmarked node
131       current := GETTOP( state );
132       while ISMARKED( current ) do
133         REMOVEFROMTOPANDREMEMBERCRITICAL( state );
134         current := GETTOP( state );
135       // check if the operation needs to act on the parent node
136       if NEEDTOMODIFYPARENTNODE( opRecord, current ) then
137         parent := GETSECONDTOTOP( state );
138         if ISMARKED( parent ) then
139           // need to backtrack even further
140           REMOVEFROMTOPANDREMEMBERCRITICAL( state );
141           continue;
142       // check if the last node in the path is a suitable restart point
143       result := ISNODESAFE( opRecord, state );
144       if result then return true;
145       // the path has been truncated and its last node is inconsistent
146       status := EXAMINETOP( opRecord );
147       if status ∈ { FOUND, NOT_FOUND } then
148         return false;
149   // Invoked after a node in the path was deemed to be not safe. In this
150   // case, the path would have been truncated such that its last node is
151   // inconsistent
152   // Returns one of the following three values: FOUND, NOT_FOUND or
153   // CONTINUE
154   enum Outcome EXAMINETOP( opRecord )
155   begin
156     state := opRecord → targetStack;
157     // retrieve the topmost node from the stack which must be
158     // inconsistent
159     current := GETTOP( state );
160     key := GETKEY( current );
161     if opRecord → key < key then
162       // the last node is strictly inconsistent
163       if opRecord → type = INSERT then
164         return CONTINUE;
165       else return NOT_FOUND;
166     else // the last node contains the matching key
167       return FOUND;

```

Pseudo-code 4: Functions used to achieve local recovery.

the terminal node of the traversal does not contain the matching key, the function tests whether or not the terminal node is safe by invoking ISNODESAFE function (line 83). If the node is found to be safe, then the traversal is ended (line 85). Otherwise, the traversal path would have been truncated such that the last node in the path is an inconsistent node. In this case, the function determines if the

traversal can still stop by invoking `EXAMINETOP` function that looks at the last node in the path (line 87). As the traversal moves down the tree, the function also populates the traversal stack (line 97).

The helper functions used by the function `SEEK` are shown in lines 100-143 (see Pseudo-code 4). The function `ISNODESAFE` (lines 100-115) implements the method described in Section 3.1.1. The function `FINDARESTARTPOINT` (lines 116-132) implements the method described in Section 3.1.3. The function `EXAMINETOP` (lines 133-143) examines the topmost node in the stack (or the last node in the path). If the node is found to be a strictly inconsistent node, then the traversal needs to be restarted for an insert operation (line 140) but can stop for a search and delete operation (line 141). If the node's key matches the operation's key, then the traversal can stop (line 142).

Due to lack of space, the pseudo-code for local recovery when looking for successor key has been omitted.

3.3 Correctness Proof

We show that our algorithm is safe and live.

Safety Property: 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 adds the key to the tree (the key becomes reachable from the root of the tree).

The linearization point of a delete operation is defined to be the point at which it removes the key from the tree (the key is no longer reachable from the root of the tree).

For a search-hit operation, we consider two cases. If the target key is still reachable from the root of the tree when a node containing the matching key is found, then its linearization point is defined to be the point at which the comparison is made. Otherwise, its linearization point is defined to be the point just before the target key 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 verified that, if the operations are ordered according to their linearization points, then the resulting sequence of operations is legal.

Liveness Property: Our local recovery framework overwrites the traversal method of the underlying concurrent BST algorithm in two ways. First, it changes the condition to determine when traversal needs to restart in the seek phase. Second, it changes the restart point for the new traversal. The second change involves examining a finite stack. Hence, only the first change can violate the liveness property of the original algorithm by possibly causing infinite number of restarts. Note that, in our local recovery framework, an operation restarts its traversal in the seek phase only if

the tree has changed *during* the last traversal. Specifically, a node where right turn was taken has now become either terminal or left-turn node implying that some delete operation has performed one or more steps. Clearly, this implies that our local recovery algorithm preserves the liveness property of the original algorithm (deadlock-freedom or lock-freedom).

4 EXPERIMENTAL EVALUATION

We now describe the results of applying our local recovery framework on the performance of different concurrent internal BSTs using simulated workloads.

4.1 Concurrent Internal BST Implementations

To evaluate the efficacy of our local recovery framework, we modified three different algorithms for a concurrent internal BST to use local recovery. They are: (i) the lock-free BST by Howley and Jones [13], denoted by HJ-BST, (ii) the RCU (Read-Copy-Update) lock-based BST by Arbel and Attiya [2], denoted by CITRUS, and (iii) the edge-oriented lock-based BST by Ramachandran and Mittal [17], denoted by CASTLE.

All implementations were obtained from their respective authors and were modified to enable local recovery. All implementations are written in C/C++.

We do not reclaim memory of garbage objects in our experiments. Recently, two general approaches for memory reclamation have been proposed based on the signaling mechanism of a Unix/Linux system [1, 5]. As per the results reported, both impose only a small performance penalty on a concurrent data structure [1, 5]. We believe that our experimental results should still stand even when memory is reclaimed using these approaches.

4.2 Experimental Setup

We conducted our experiments on an Intel Xeon Phi 7250 Processor consisting of 68 1.4 GHz cores with 4 hardware threads per core and 96GB DDR4 plus 16GB high-speed MCDRAM. We used Intel C/C++ compiler (version 2017.0.4) with the optimization flag set to O3. To avoid memory allocation in a multi-threaded environment from becoming a bottleneck, we used Intel's *TBB Malloc* [18] as the dynamic memory allocator since it provides superior performance to the C/C++ default allocator in a multi-threaded environment.

To study the performance impact of local recovery on different BSTs, we considered the following parameters:

- (1) **Maximum Tree Size:** This depends on the size of the key space. We considered three different key space sizes: 2^8 (256), 2^{11} (2K) and 2^{14} (16K) keys.
- (2) **Relative Distribution of Operations:** We considered four different workload distributions: (a) *read-dominated*: 90% search, 5% insert and 5% delete, (b) *mixed*: 60% search, 20% insert and 20% delete, (c) *balanced*: 34% search, 33% insert and 33% delete, and (d) *write-dominated*: 0% search, 50% insert and 50% delete.
- (3) **Maximum Degree of Concurrency (or Contention):** This depends on number of threads that can concurrently operate on the tree. Since Xeon Phi has 68 cores with 4 hardware threads per core, we varied the number of threads from 1 to 272 in suitable increments.

Figure 4: Change in system throughput using local recovery *relative* to that of the original algorithm for Zipfian distribution with $\alpha=1$ when both optimizations are used. Each column represents a workload type. Each row represents a key space range. A positive value indicates an improvement in performance, whereas a negative value indicates a drop in performance.

In each run of the experiment, we ran an implementation for five seconds and averaged the results over twenty runs. The beginning of each run consisted of a one second “warm-up” phase whose numbers were excluded from the calculations to minimize the effect of initial caching on the computed statistics. Also, to capture only the steady state behavior, we *pre-populated* the tree to 50% of its maximum size, prior to starting a simulation run.

Key Distribution: Typically, when evaluating concurrent BSTs, keys are assumed to be uniformly distributed, that is, all keys in the key space are assumed to have the same probability of occurrence [2, 8, 13, 14]. But, many real-world workloads are better modeled using skewed distributions in which some keys are more popular than others [4]. Zipfian distribution is a type of power-law distribution that simulates this behavior [4, 11]. It is characterized by a parameter α that usually lies between 0.5 and 1 [4]. Zipfian distribution was used to evaluate the performance of balanced search trees in [11]. In our experiments, we used both uniform and Zipfian (with $\alpha = 1$) distributions to evaluate the efficacy of local recovery.

Evaluation Metrics: To evaluate the impact of local recovery, we compared the performance of different implementations with respect to *system throughput*, which is given by the number of operations executed per unit time.

4.3 Optimizations

We expect local recovery to help only when the contention is high and the traversal path is long and induce unnecessary overhead in

other situations. To that end, we used the following two optimizations. First, when finding a restart point, if the stack length is below a certain threshold, then we discard the stack and simply restart the traversal from the root of the tree. Note that the stack is still used to determine whether or not to restart. In our experiments, we used the threshold value of 8 for the stack length. Second, we maintain *exponential weighted moving average (EWMA)* of the restart count of an operation [15]. When an operation traverses the tree for the first time, it uses traversal stack only if the current value of the average restart count exceeds a certain threshold. We maintained separate counters for search and non-search operations. In our experiments, we used the weight value of 0.5 to compute the EWMA and the threshold value in the range [0.03, 0.1] for the restart count depending on the algorithm and the operation type.

4.4 Simulation Results

Due to space constraints, we only show the detailed results for Zipfian distribution.

When keys are uniformly distributed, with local recovery, HJ-BST sees up to 27.8% increase in system throughput when the contention is high to up to 9.3% decrease in the throughput when the contention is low. The corresponding numbers for CITRUS are 7.1% and 5.3% respectively, whereas for CASTLE are 10.8% and 3.7%, respectively.

The impact of local recovery on the throughput of the three concurrent BST algorithms we tested for Zipfian distribution is shown in Figure 4. It can be seen that, as the contention increased (either

the key space size became smaller or the fraction of write operations became larger), the benefit of using local recovery also became more pronounced as expected yielding performance improvements of up to 124.7%.

The biggest beneficiary of local recovery in our experiments was HJ-BST. It saw an increase in its throughput for all four workload types—up to 44.5% for read-dominated, 89.2% for mixed, 107.1% for balanced and 124.7% for write-dominated workloads. The (percentage) gain improved with contention; it increased as the number of threads became larger or the key space size became smaller. In HJ-BST, if a modify operation encounters an operation pending at a node during its traversal, it proceeds to help the operation and then restarts. As a result, the number of restarts was much higher for HJ-BST than for other BST algorithms and hence local recovery had larger impact on HJ-BST than any other algorithm.

The second biggest beneficiary of local recovery in our experiments was CASTLE. It also saw an increase in its throughput for all workload types—up to 9.1% for read-dominated, 15.7% for mixed, 20.0% for balanced and 24.5% for write-dominated workloads. As in the case of HJ-BST, the (percentage) gain increased with contention.

The local recovery had little or no impact on the performance of CITRUS. We believe that, this is because, in CITRUS, an operation only restarts on encountering a failure in the update phase while modifying the tree; no restarts occur in the seek phase while traversing the tree. Also, CITRUS is especially optimized for low contention scenarios and does not scale well as the contention increases (a complex delete operation in CITRUS has to wait until all operations currently traversing the tree have completed their traversals).

Impact of optimizations: Using local recovery without any optimizations (results not shown), on one hand, improved the performance by as much as 195.3% (HJ-BST for write-dominated workload) but, on the other hand, degraded the performance by as much as 33.1% (HJ-BST for read-dominated workload). Thus, as desired, the optimizations helped reduce the degradation in performance when the contention is low. However, they also had the unintended effect of limiting the gains when the contention is high.

5 CONCLUSION AND FUTURE WORK

In this work, we have presented a framework that enables (an operation of) certain concurrent algorithms for maintaining an internal BST to recover locally on encountering a failure due to concurrent updates being made to the tree. Our framework is sufficiently general that it can be applied to many existing concurrent BST algorithms. Using experiments, we have shown that our local recovery framework can significantly improve the performance of some concurrent BSTs by as much as 124.7% when the contention is high enough to cause a significant number of restarts.

Our local recovery framework relies on certain properties of a concurrent internal BST algorithm. These properties are not necessarily satisfied by concurrent algorithms for other search trees and even some internal BSTs (e.g., the one in [16]). Note that it is still possible to perform local recovery in such data structures. However, when compared to the local recovery framework described in Section 3, local recovery for such trees is likely to not only require

more frequent restarts but also yield a restart point that is farther away from the operation's section of interest.

As a future work, we would like to refine our local recovery framework and then theoretically analyze it so that, when applied to a *non-blocking* concurrent BST such as the one in [13], it yields a concurrent BST whose operations have provably low amortized time complexity as in [6, 10]. At present, there are scenarios in which an operation may end up traversing the same section of the tree multiple times.

REFERENCES

- [1] Dan Alistarh, W. M. Leiserson, A. Matveev, and N. Shavit. 2015. ThreadScan: Automatic and Scalable Memory Reclamation. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, New York, NY, USA, 123–132.
- [2] M. Arbel and H. Attiya. 2014. Concurrent Updates with RCU: Search Tree as an Example. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, New York, NY, USA, 196–205.
- [3] M. Arbel and H. Attiya. 2015. Predicate RCU: An RCU for Scalable Concurrent Updates. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, New York, NY, USA, 21–30.
- [4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. 1999. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of the 18th IEEE Conference on Computer Communications (INFOCOM)*. IEEE, New York, NY, USA, 126–134.
- [5] T. A. Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, New York, NY, USA, 261–270.
- [6] B. Chatterjee, N. N. Dang, and P. Tsigas. 2014. Efficient Lock-Free Binary Search Trees. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, New York, NY, USA, 321–331.
- [7] T. David, R. Guerraoui, and V. Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, USA, 631–644.
- [8] D. Drachler, M. Vechev, and E. Yahav. 2014. Practical Concurrent Binary Search Trees via Logical Ordering. In *Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, New York, NY, USA, 343–356.
- [9] F. Ellen, P. Fataourou, E. Ruppert, and F. van Breugel. 2010. Non-Blocking Binary Search Trees. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, New York, NY, USA, 131–140.
- [10] F. Ellen, P. Fatourou, J. Helga, and E. Ruppert. 2014. The Amortized Complexity of Non-Blocking Binary Search Trees. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, New York, NY, USA, 332–340.
- [11] C. Faloutsos and H. V. Jagadish. 1992. On B-tree Indices for Skewed Distributions. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann, San Francisco, CA, USA, 364–373.
- [12] M. Herlihy and J. M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (July 1990), 463–492.
- [13] S. V. Howley and J. Jones. 2012. A Non-Blocking Internal Binary Search Tree. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, New York, NY, USA, 161–171.
- [14] A. Natarajan and N. Mittal. 2014. Fast Concurrent Lock-Free Binary Search Trees. In *Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, New York, NY, USA, 317–328.
- [15] NIST. 2012. NIST/SEMATECH e-Handbook of Statistical Methods: Single Exponential Smoothing. <http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc431.htm>.
- [16] A. Ramachandran and N. Mittal. 2015. A Fast Lock-Free Internal Binary Search Tree. In *Proceedings of the 16th International Conference on Distributed Computing and Networking (ICDCN)*. ACM, New York, NY, USA, 10.
- [17] A. Ramachandran and N. Mittal. 2015. CASTLE: Fast Concurrent Internal Binary Search Tree using Edge-Based Locking. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, New York, NY, USA, 281–282.
- [18] J. Reinders. 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, Inc., Sebastopol, CA, USA.