Improving Efficacy of Internal Binary Search Trees using Local Recovery *

Arunmoezhi Ramachandran Neeraj Mittal
⟨arunmoezhi, neerajm⟩@utdallas.edu
Department of Computer Science
The University of Texas at Dallas

Abstract

Binary Search Tree (BST) is an important data structure for managing ordered data. Many algorithms—blocking as well as non-blocking—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 any failures occurring due to other processes performing conflicting actions on the tree. Most concurrent algorithms that have 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 in a concurrent binary search tree based on internal representation using *local recovery* by restarting the traversal from the "middle" of the tree in order to locate an operation's window. Our approach is sufficiently general in the sense that it can be applied to a variety of concurrent binary search trees based on both blocking and non-blocking approaches.

Using experimental evaluation, we demonstrate that our local recovery approach can yield significant speed-ups of up to 69% for many concurrent algorithms.

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, Binary Search Tree, Internal Representation, Local Recovery

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 that supports search, insert and delete operations. Concurrent algorithms for (unbalanced) internal binary search trees have been proposed in [1, 3, 4].

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 performing conflicting updates on the tree which can cause for example, the target key to move from its original location. Most of the concurrent algorithms (for BSTs) that have proposed so far use a naïve approach and simply restart the traversal from the root of the tree [1–3]. This is especially undesirable if the tree has large height, and the overhead of repeatedly traversing the tree may dominate all other overheads of performing an operation.

In this work, we present a general approach to recover locally in a concurrent binary search tree based on internal representation. Our algorithm enables a process to quickly 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 binary search trees using internal representation—blocking as well as non-blocking—such as those in [1, 3, 4]. Using experimental evaluation, we demonstrate that our local recovery approach can yield significant speed-ups for many concurrent algorithms.

2. The Local Recovery Algorithm

Note that 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 the operation, the tree may then need to be modified to actually realize 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 its new location. A *re*-traversal of the tree may also be required if the operation encounters any failure during the update phase.

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

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

PPoPP '16,, March 12-16, 2016, Barcelona, Spain.

http://dx.doi.org/10.1145/2851141.2851173

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. This leads to two separate but related questions that any local recovery algorithm 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?"

We assume that, when the key stored in a binary node is deleted, it is replaced with its successor key. Thus, the value of key stored in a node can only increase. Further, in most concurrent BST algorithms, a node is marked before being removed from the tree. Thus, an unmarked node is guaranteed to be a part of the tree.

To achieve local recovery, we maintain a *log* of all the nodes visited on the traversal path. Note that, at each non-terminal node in the path, an operation either follows the left or the right child pointer. As a process is traversing the tree, other operations may be making changes to the tree concurrently due to which a turn taken by the process earlier may no longer be valid. Specifically, a right-turn node may no longer be right-turn node (a left-turn node however remains a left-turn node).

We say that a node in the traversal path is an *anchor* node if the operation follows its *right* child pointer; otherwise we say that it is a *non-anchor* node. An anchor node is said to be *consistent* if its key is still less than the operation's key; otherwise it is said to be *inconsistent*.

Consider two nodes U and V in the traversal path (log). We say that V is *critical* with respect to U if the following conditions hold: (i) V precedes U in the traversal path, (ii) V is an unmarked anchor node, and (iii) all anchor nodes between U and V in the traversal path are marked.

We say that a node U is *safe* if the following holds: (a) its critical anchor node, say V, is consistent and (b) all anchor nodes between U and V are also consistent.

We are now ready to answer the two questions posed earlier. First, the traversal does not need to be restarted if the terminal node of the access-path is safe. Second, to identify a restart point, the algorithm examines the traversal log to find the latest node in the log that is unmarked and safe. For search and delete operations, further optimizations are possible that avoid the need to restart traversal during the seek phase [5].

3. Experimental Evaluation

To evaluate our local recovery algorithm, we implemented it for three different concurrent BSTs based on internal representation, namely those based on: (i) the lock-free BST by Howley and Jones [3], denoted by LF-IBST, (ii) the lock-based BST by Ramachandran and Mittal [4], denoted by CASTLE and (iii) the RCU (Read-Copy-Update) framework-based BST by Arbel and Attiya [1], denoted by CITRUS.

Experiments were performed on an Intel Xeon Phi Coprocessor having 61 cores with 4 hardware threads per core. We measured system throughput, which is defined as the total number of operations (in millions) completed per second. The number of threads that can concurrently operate on the tree was varied from 1 to 244 in suitable increments. We considered two different key ranges (2,000 (2k) and 200,000 (200K) keys) and considered two workloads: *read-dominated:* (90% search, 5% insert and 5% delete) and *write-dominated:* (0% search, 50% insert and 50% delete).

Usually uniform key distribution have been used to evaluate concurrent BSTs. But, in many of the real world workloads, keys have skewed distribution where some keys are more popular than others. Zipfian distribution, a type of power-law distribution simulates this behavior. In our experiments, we used both uniform and Zipfian distributions to evaluate the local recovery algorithm.

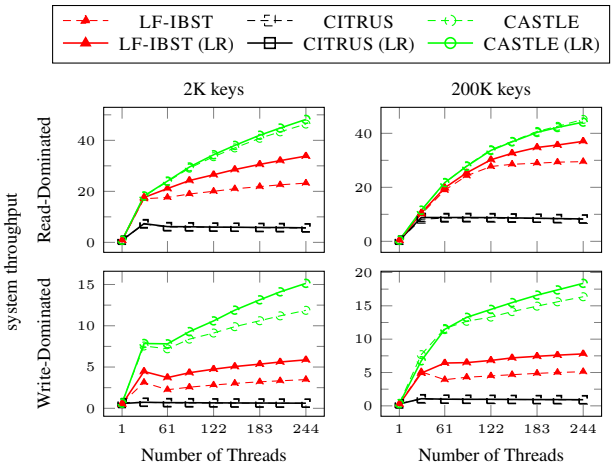

Figure 1: Comparison of throughput of different concurrent BST implementations with (solid lines) and without (dotted lines) local recovery for Zipfian distribution with α=1. Higher is better.

For uniform distribution, the performance gain was marginal and, in many cases, was actually negative due to the overhead of log maintenance. This is not surprising because, for small trees, even though contention is higher, seek time is small to begin with and any benefit of local recovery is nullified by additional overhead of log maintenance. For larger trees, even though seek time is larger, contention is low as key accesses are spread evenly.

Figure 1 shows the behavior for Zipfian distribution. In general, Zipfian distribution causes more contention than uniform distribution. So, even for smaller trees for which seek times are lower, we still see performance gains for write-dominated workload. In particular, we see up to 69%, 28% and 8% improvement in system throughput for LF-IBST, CASTLE and CITRUS respectively. In LF-IBST, if a process sees another pending operating while traversing the tree, it helps the pending operation and then restarts the traversal. This results in frequent restarts and hence local recovery improves performance by a larger margin. We see smaller improvements for CASTLE and CITRUS as they are lock-based algorithms with no helping performed during tree traversal. More details of the experimental evaluation can be found in [5].

References

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

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

[3] 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.

[4] A. Ramachandran and N. Mittal. 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)*, pages 281–282, Feb. 2015.

[5] A. Ramachandran and N. Mittal. Improving Efficacy of Internal Binary Search Trees using Local Recovery. Technical Report UTDCS-13-15, Department of Computer Science, The University of Texas at Dallas, Dec. 2015.