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

Arunmoezhi Ramachandran Neeraj Mittal

Department of Computer Science
The University of Texas at Dallas
Richardson, Texas 75080, USA
{arunmoezhi, neerajm}@utdallas.edu

Abstract

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

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

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

1. Introduction

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

Concurrency is often managed using locks. A lock can be used to achieve mutual exclusion, which can then be used to ensure that any updates to the data structure or a portion of it are performed by one process at a time. This makes it easier to design a lock-based concurrent data structure and reason about its correctness.

Binary search tree (BST) which implements a dictionary abstract data type is one of the fundamental data structures for organizing *ordered* data that supports search, insert and delete operations.

Concurrent algorithms for unbalanced binary search trees have been proposed in [1–3]. Algorithm in [2] use internal representation of a search tree in which all nodes store data, where as those in [1, 3] use an external representation of a search tree in which only leaf nodes store data (data stored in internal nodes is used for routing purposes only).

In this work, our focus is on developing an efficient concurrent *lock-based* algorithm for an *unbalanced* (different leaf nodes may be at very different depths) binary search tree (BST). It uses an internal representation of a BST. Unlike most other concurrent algorithms for a BST, our algorithm is edge-oriented rather than node-oriented. Further, it uses an optimistic approach to deal with movement of keys from location to another due to concurrent delete operations.

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the author/owner(s).

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA
ACM 978-1-4503-3205-7/15/02
http://dx.doi.org/10.1145/2688500.2688551

2. The Lock-Based Algorithm

We refer to our algorithm as CASTLE (Concurrent Algorithm for Binary Search Tree by Locking Edges). CASTLE uses compare-and-swap (CAS) atomic instruction to implement a lock.

Seek: Every operation in our algorithm uses *seek* function as a subroutine. The seek function traverses the tree from the root node until it either finds the target key or reaches a non-binary node whose next edge to be followed points to a null node. We refer to the path traversed by the operation during the seek as the *access-path*, and the last node in the access-path as the *terminal node*. In the access-path, it also keeps track of the node at which it took the last "right turn" (*i.e.*, it last followed a right edge). Let this "right turn" node be referred to as *anchor node* when the traversal reaches the terminal node. In certain cases a key may have moved upward along the access-path. To guarantee that the seek function did not miss this key, we check if the anchor node is still part of the tree and if the key stored in the anchor node has not changed since it first encountered the anchor node during the (current) traversal.

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

Insert: An insert operation starts by invoking seek operation. It returns false if the target key matches the stored key; otherwise, it creates a new node containing the target key. It then adds the new node to the tree at the injection point using a CAS instruction. If the CAS instruction succeeds, then (the new node becomes a part of the tree and) the operation terminates; otherwise, the operation restarts by invoking seek. Note that the insert operations are lock-free.

Delete: Our algorithm, like the one in [3], operates at edge level. A delete operation obtains ownership of the edges it needs to work on by locking them. To enable locking of an edge, we steal a bit from the child addresses of a node referred to as *lock-flag*. We also steal another bit from the child addresses of a node to indicate that the node is undergoing deletion and will be removed from the tree. We denote this bit by *mark-flag*. Finally, to avoid the ABA problem, as in [2], we use *unique* null pointers, which can be implemented using local sequence numbers. A delete operation starts by invoking seek function. It returns false if the stored key does not match the target key; otherwise, it checks if the terminal node is a binary node or not. If it is a binary node, then the delete operation is classified as complex; otherwise it is classified as simple.

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

(a) *Simple Delete:* In this case, either $T.left$ or $T.right$ is pointing to a null node. Note that both $T.left$ and $T.right$ may be pointing to null nodes in which case T will be a leaf node. Without loss of generality, assume that $T.right$ is a null node. The removal of T involves locking the following three edges: $\langle T.parent, T\rangle$, $\langle T, T.left\rangle$ and $\langle T, T.right\rangle$.
A lock on an edge is obtained by setting the lock-flag in the appropriate child field of the parent node using a CAS instruction. If all the edges are locked successfully, then the operation validates that the key stored in the terminal node still matches the target key. If the validation succeeds, then the operation marks both the children edges of T to indicate that T is going to be removed from the tree. Next, it changes the child pointer at $T.parent$ that is pointing to T to point to $T.left$ using a simple write instruction. Finally, the operation releases all the locks and returns true.

(b) *Complex Delete:* In this case, both $T.left$ and $T.right$ are pointing to non-null nodes. The operation locates the next largest key in the tree, which is the smallest key in the subtree rooted at the right child of T. We refer to this key as the *successor key* and the node storing this key as the *successor node* denoted by S. Deletion of the key stored in T involves copying the key stored in S to T and then removing S from the tree. To that end, the following edges are locked by setting the lock-flag on the edge using a CAS instruction: $\langle T, T.right\rangle$, $\langle S.parent, S\rangle$, $\langle S, S.left\rangle$ and $\langle S, S.right\rangle$. If all the edges are locked successfully, then the operation validates that the key stored in the terminal node still matches the target key. If the validation succeeds, then the operation copies the key stored in S to T, and marks both the children edges of S to indicate that S is going to be removed from the tree. Next, it changes the child pointer at $S.parent$ that is pointing to S to point to $S.right$ using a simple write instruction. Finally, the operation releases all the locks and returns true.

In both cases, if the operation fails to obtain any of the locks, then it releases all the locks it was able to acquire up to that point, and restarts by invoking seek. Also, after obtaining all the locks, if the key validation fails, then it implies that some other delete operation has removed the key from the tree while the current delete operation was in progress. In that case, the given delete operation releases all the locks, and simply returns false. Note that using a CAS instruction for setting the lock-flag also enables us to *validate that the child pointer has not changed* since it was last observed in a single step. Due to space constraints, details have been omitted and can be found in [4].

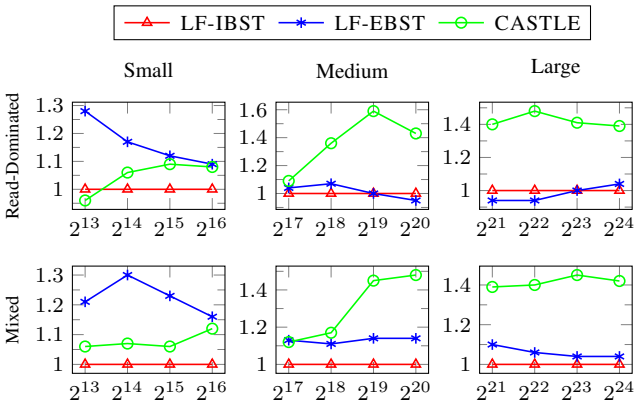

Figure 1: Comparison of system throughput of different algorithms *relative to that of* LF-IBST at 32 threads. Each row represents a workload type. Each column represents a range of key space size. Higher the ratio, better the performance of the algorithm.

3. Experimental Evaluation

Besides CASTLE, we considered two other implementations of a concurrent BST for comparative evaluation, namely those based on: (i) the lock-free internal BST [2], denoted by LF-IBST and (ii) the lock-free external BST [3], denoted by LF-EBST

We compared different implementations with respect to system throughput, which is defined as the total number of operations (in millions) completed per second. Experiments were performed on a 32 cores processor. The number of threads was set to 32 and the key space size was varied from 2^{13} to 2^{24}. We used two workloads: *read-dominated:* 90% search, 9% insert and 1% delete and *mixed*: 70% search, 20% insert and 10% delete.

The results of our experiments are shown in Figure 1. As Figure 1 show, for medium and large key space sizes, CASTLE achieves the best system throughput for both the workload types in all the cases. The maximum gap between CASTLE and the next best performer is about 59% which occurs at around 512Ki key space size, read-dominated workload and 32 threads.

Some of the reasons for the better performance of CASTLE over the other two concurrent algorithms, especially when the contention is relatively low, are as follows. First, operating at edge-level rather than at node-level reduces the contention among modify (insert and delete) operations and improves overall concurrency. Second, we observed in our experiments that CASTLE has a smaller memory footprint than the other two algorithm. As a result, it is able to benefit from caching to a larger extent than other algorithms.

References

[1] 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.

[2] 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.

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

[4] A. Ramachandran and N. Mittal. CASTLE: Fast Concurrent Internal Binary Search Tree using Edge-Based Locking. Technical Report UTDCS-17-14, Department of Computer Science, The University of Texas at Dallas, 2014.