On Concurrent Lock-Free Binary Search Trees

Arunmoezhi Ramachandran, Neeraj Mittal

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

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

1 Introduction

A Binary Search Tree (BST) implements the dictionary abstract data type. In a BST, all the values in a node's left subtree are less than the node value and all the values in the node's right subtree are greater than the node value. Duplicates are not allowed. A BST supports three main operations, viz.: search, insert and delete. Search(x) determines if x is present in the tree. Insert(x) adds key x to the tree if it is not already present. Delete(x) removes the key x from the tree if it is present.

Several algorithms have been proposed for non-blocking binary search trees. Ellen *et al.* proposed the first practical lock-free algorithm for a concurrent binary search tree in [1]. 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 have proposed another lock-free algorithm for a concurrent binary search tree in [2] which uses an internal search tree in which both the leaf nodes as well as the internal nodes store the actual keys. One advantage of using an internal search tree is that its memory footprint is half that of an external search tree. For large size trees, search time dominates and our experimental results shows that internal BST tend to perform better than external BSTs. Natarajan and Mittal have proposed another lock-free external BST in [3]. The key change in this algorithm is that it operates at edge level and the ones described by Ellen *et al.* [1] and Howley and Jones [2] operates on node level. A node (or edge) level operation mean that nodes (or edges) are marked for deletion. Operating at edge level blocks fewer operations than operating at node level. So edge level operation provides more concurrency when there is high contention. Hence the algorithm described by Natarajan and Mittal [3] performs well for smaller trees with high contention. We have designed our algorithm to get the benefits of both the worlds. Our algorithm like the one proposed by Howley and Jones [2] is an internal BST and like the one proposed by Natarajan and Mittal [3] operate on edge level.

2 Lock-Free Algorithm

Every operation in our algorithm begins with a seek phase.

2.1 Overview

Seek: The operation traverses the search tree from the root node until it finds the target key or it reaches a leaf node. We refer to the path traversed by the operation in the seek phase as the access path. The operation then compares the target key with the value stored in node. Depending on the result of the comparison and the type of the operation, the operation either terminates or moves to the next phase. For certain cases where some key may have moved up the tree, the seek operation may have to restart (will be discussed later). We now describe the next steps for each of the type of operation one-by-one.

Search: A search operation invokes seek and returns *true* if the stored key matches the target key; otherwise it returns *false*.

Insert: An insert operation invokes seek and returns *false* if the stored key matches the target key; otherwise it moves to the execution phase. In the execution phase, it attempts to insert the key into the tree as a child node of the last node in the access-path using a CAS instruction. If the instruction succeeds, then the operation returns *true*; otherwise, it restarts from the seek phase after possibly helping.

Delete: A delete operation invokes seek and returns *false* if the stored key does not match the target key; otherwise it moves to the execution phase. In the execution phase, it attempts to remove the key stored in the last node in the access path. let the last node be referred to as *target node*. There are two cases depending on if the *target node* is a binary node (has two children) or not (at most one child). In the first case, the operation is referred to as complex delete. In the second case, it is referred to as simple delete. In the case of simple delete (as shown in Fig. 1), the target node is removed by changing the pointer at the parent of the *target node*. In the case of complex delete (as shown in Fig. 2), the key to be deleted is replaced with the next largest key in the tree (which will be stored in the leftmost node of the right subtree of the *target node*).

2.2 Details of the Algorithm

We use sentinel keys and nodes to handle the boundary cases easily. A tree node in our algorithm consists of three fields: (i) $markAndKey$ which contains the key stored in the node, (ii) *child* array, which contains the addresses of the left and right children and (iii) $readyToReplace$, which is a boolean flag used by complex delete operation to indicate if a node can be replaced with a fresh copy of it.

This algorithm like the algorithm described by Natarajan and Mittal [3], operates on edge level. A delete operation obtains ownership of the edges it needs to work on by marking them. To enable marking we steal two bits from the child addresses of a node. To avoid ABA problem, as in Howley and Jones [2],

Fig. 1. An illustration of a simple delete operation.

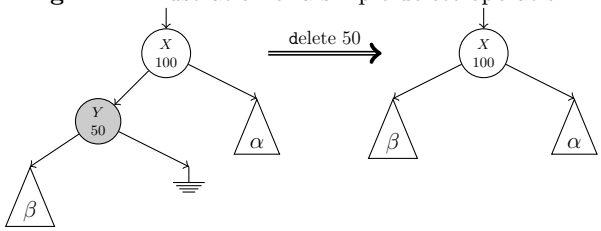
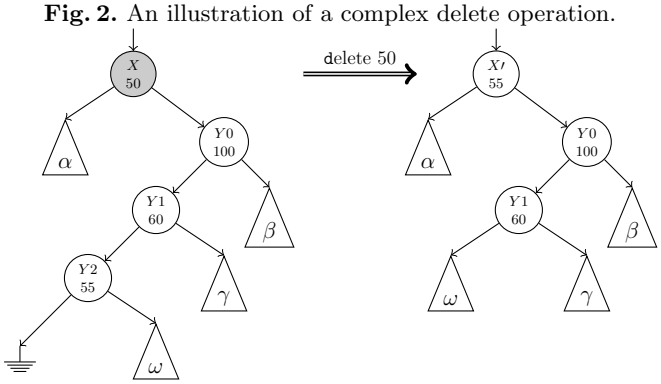


Fig. 2. An illustration of a complex delete operation.



we steal another bit from the child address referred to as $nullFlag$ and use it to indicate whether the address points to a null or a non-null value. So when an address changes from a non-null value to a null value, we only set the $nullFlag$ and the contents of the address are not modified. As complex delete replaces a key in a node being deleted, a flag is required to identify if the key in a node has changed. So we steal a bit from the key field and use it as a mark bit. If the mark bit is set, it denotes that the key in the node has changed.

We next describe the details of the seek phase, which is executed by all operations (search as well as modify) after which we describe the details of the execution phases of insert and delete operations.

The Seek Phase

A seek operation keeps track of the node in the access path at which the last "right turn" was taken (i.e., right edge was followed). Let us call this node as $anchorNode$. Upon reaching the last node, it compares the stored key with the target key. If they do not match, then it is possible that the key may have moved up in the tree. So key stored in the $anchorNode$ is checked to see if has changed. If the key has changed then the seek operation restarts. If the key has not changed, then the key stored in the $anchorNode$ is checked to see if it is undergoing deletion (by checking if its left child edge is marked). If the key is not undergoing deletion, then the seek operation terminates. If the key is

undergoing deletion, then it checks if the *anchorNode* of the current traversal matches with the *anchorNode* of the previous traversal. If they match, then the seek operation terminates by returning the results of the previous traversal; otherwise it restarts.

In the case of insert operation, the seek operation also returns the injection point which is the current contents of the child location at which the insert will occur.

Execution Phase of an Insert Operation

In the execution phase, an insert operation creates a new node containing the target key. It then adds the new node to the tree at the injection point using a CAS instruction. If the CAS succeeds, then (the new node becomes a part of the tree and) the operation terminates; otherwise, the operation determines if it failed because of a conflicting delete operation in progress. If there is no conflicting delete operation in progress then the operation restarts from the seek phase; otherwise it performs helping (which will be described later) and then restarts from the seek phase.

Execution Phase of a Delete Operation

The execution of a delete operation starts in the *injection* mode. Once the operation has been injected into the tree, then it advances to the *cleanup* mode.

Injection Mode: In the injection mode, the delete operation marks the left child edge of the target node using a CAS instruction. If the CAS instruction succeeds, then the delete operation has been injected into the tree and is guaranteed to complete. Then the operation moves on to the *cleanup* mode. But if the CAS instruction fails, the operation performs helping and restarts from the seek phase (and stays in the injection mode).

Cleanup Mode: In the *cleanup* mode, the operation begins by marking the right child edge of the target node using a BTS (Bit Test and Set) instruction (this can also be done using a CAS instruction as well). Note that we maintain an invariant that edges which are once marked cannot be unmarked. Eventually the node is either removed from the tree (by simple delete) or replaced with a "new" node containing the next largest key (by complex delete). Further, a marked edge is changed only under a specific situation by a delete operation as part of helping (described later).

After marking both the edges, an operation checks whether the node is a binary node or not. If it is a binary node, then the delete operation is classified as a complex delete; otherwise it is classified as a simple delete. Let **A** be the target node, B be its parent node and C and D be the left and right child node respectively, of A. We now discuss the two types of delete operation.

Simple Delete: In this case, either C or D is a null node. Note that both C and D may be null nodes in which case A will be a leaf node. Without loss of generality, assume that D is a null node. Delete operation attempts to change the pointer at B that is pointing to A to point to C using a CAS instruction. If the CAS instruction succeeds, then the operation terminates; otherwise, the delete operation performs another seek operation. If the seek operation either fails to find the target key or return a target node different from A, then A has been already removed from the tree (by another operation as part of helping) and the operation terminates; otherwise, it attempts to remove A from the tree again. Note that the new seek operation may return a different parent node. This process may be repeated multiple times.

Complex Delete: In this case, both C and D are non-null nodes. The operation now performs the following steps:

1. Locate the next largest key in the tree, which is the smallest key in the subtree rooted at the right child of the target node A. We refer to this key as the successor key and the node storing this key as the successor node. Let S denote the successor node and T denote its parent node.
2. Claim the successor node. This involves marking both the child edges of S. Note that the left edge of S will be null. To distinguish between marking a target node (for deletion) and marking a successor node (for promotion), we steal two bits from the address and refer to them as $deleteFlag$ and $promoteFlag$. The left edge of S is marked (i.e., $promoteFlag$ is set) using a CAS instruction. As part of marking the left edge, we also store the address of the target node A in the left edge. This is done to enable helping in case the successor node is obstructing the progress of another operation. In case if the CAS instruction fails, the operation repeats from step 1. The right edge of S is marked using a BTS instruction.
3. Promote the successor Key. The successor key is copied into the target node. At the same time, the mark bit in the key is set to indicate that the key currently stored in the target node is the successor key and not the target key.
4. The successor node S is deleted by changing the pointer at T that is pointing to S to point to the right child of S using a CAS instruction. If the CAS instruction fails, then the operation performs helping if needed. It then finds the successor node again by performing a traversal starting from the right child of the target node A and repeats step 4. If the successor node is not found in the traversal, then it has been already removed from the tree (by another operation as part of helping) and the operation moves to step 5.
5. Note that, at this point, the original key in the target node has been replaced with the successor key. Further, the key as well as both its edges are marked. The target node is now replaced with a new node whose contents are same as that of the target node expect that all the fields are unmarked. The target node is then replaced with a new node using a CAS instruction at the parent node. If the CAS instruction succeeds, then the operation terminates;

otherwise, as in the case of simple delete, it performs another seek operation, this time looking for the successor key. If the seek operation either fails to find the successor key or returns a target node different from A, then A has been already replaced (by another operation as part of helping) and the operation terminates. On the other hand if the seek operation finds the target node A with the successor key, it attempts to replace A again. Note that the new seek operation may return a different parent node. This process may be repeated multiple times.

Helping

To enable helping, whenever traversing the tree to locate either a target key or a successor key, we keep track of the last unmarked edge encountered in the traversal. Whenever an operation fails while executing a CAS instruction, it helps the operation in progress at the child end of the unmarked edge if different from its own operation. Note that, when traversing the tree looking for a target key, the last unmarked edge will always be found because of the sentinel keys. However, when traversing the tree looking for a successor key, the last unmarked edge may not always exist since the traversal starts from the middle of the tree from the right child of a target node. Recall that A denotes a target node and D its right child node.If no unmarked edge is found during the traversal from D, then helping is performed along the edge (A,D) under the following conditions (i) delete operation at D is of type simple delete, or (ii) delete operation at D is of type complex delete and is currently at its last step (replacing the target node with a new node).

3 Pseudo code

4 Experimental Evaluation

Experimental Setup: We conducted our experiments on an X86_64 AMD Opteron 6276 machine running GNU/Linux operating system. We used gcc 4.6.3 compiler with optimization flag set to $O3$. The Table I shows the hardware features of this machine. All implementations were written in C++. 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 five different key ranges: 1000(1K), 10,000 (10K), 100,000 (100K), 1 million (1M) and 10 million (10M) keys. To capture only the steady state behaviour we *pre-populated* the tree to 50% of its maximum size, prior to starting the simulation run.
2. **Relative Distribution of Various Operations:** We consider three different workload distributions: (a) *read-dominated* workload : 90% search, 9% insert and 1% delete, (b) *mixed* workload : 70% search, 20% insert and 10%

delete and (c) *write-dominated* workload : 0% search, 50% insert and 50% delete

3. **Maximum Degree of Contention:** This depends on number of threads concurrently operating on the tree. We varied the number of threads from 1 to 128 in increments in powers of 2.

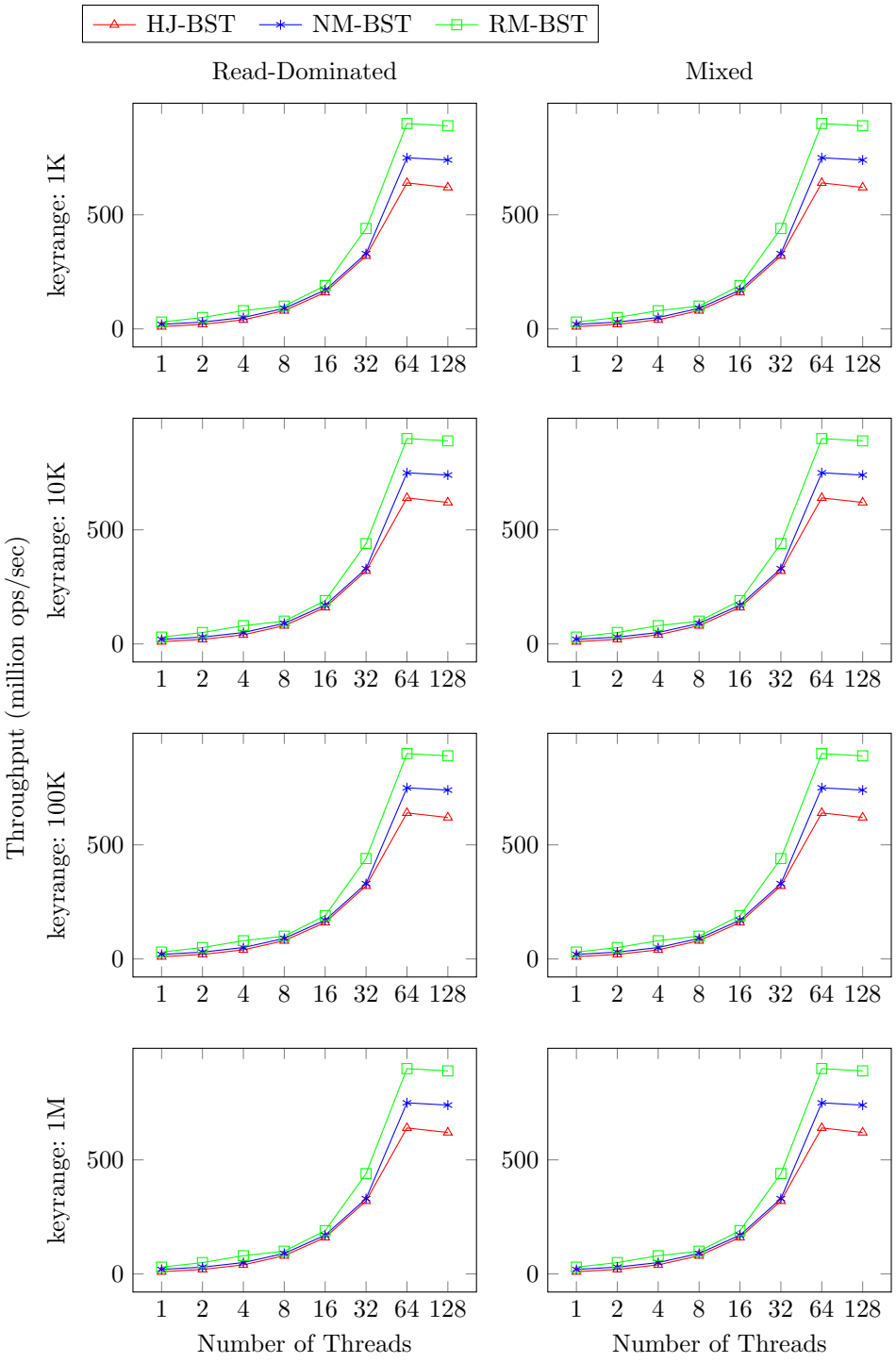
Table 1. hardware features

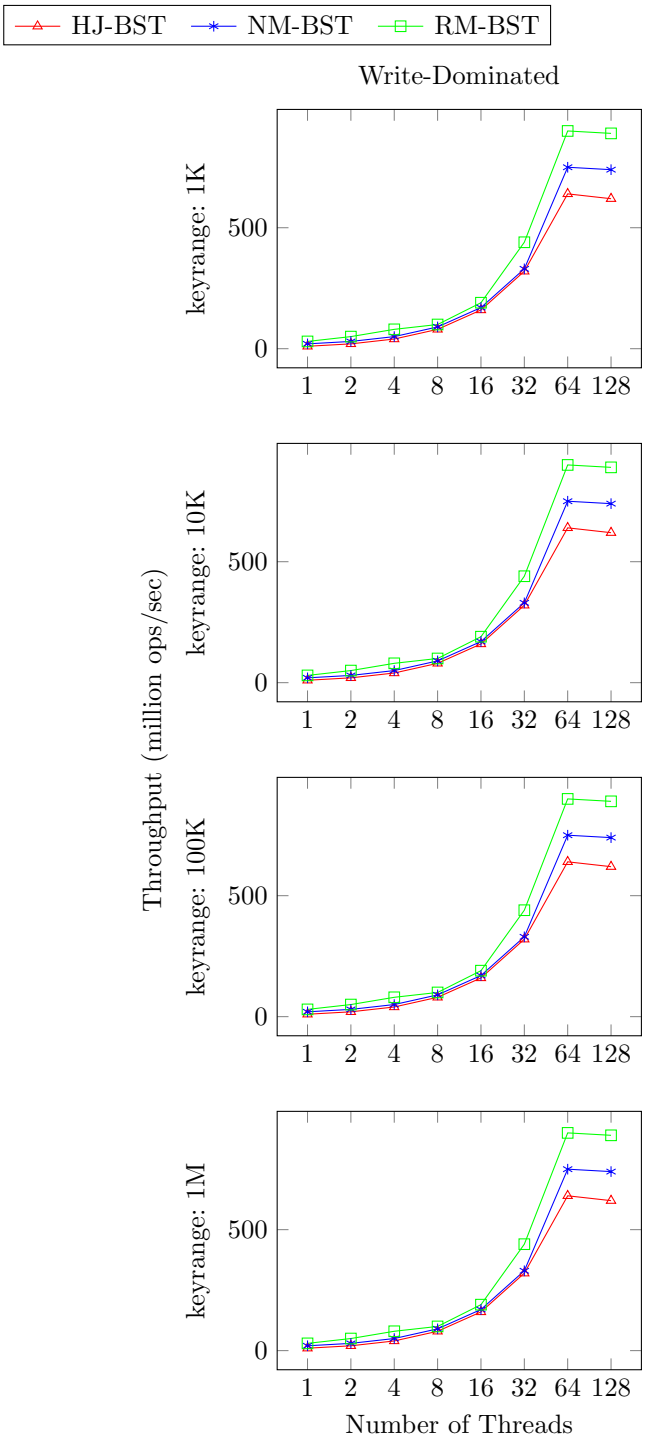
CPU packages	4
Cores per package	8
Threads per core	2
Clock frequency	2.3 GHz
L1 cache (I/D)	64KB/16KB
L2 cache	2 MB
L3 cache	6 MB
Memory	256 GB

Algorithm	**# of Objects Allocated**		**#of Atomic Instructions Executed**	
	Ins	**Del**	**Ins**	**Del**
Ellen & *et al.*	4	2	3	4
Howley & Jones	2	1	3	upto 9
Natarajan & Mittal	2	0	1	3
This work	1	1	1	upto 6

We compared the performance of different implementations with respect to two metrics:

1. **System Throughput:** it is defined as the number of operations executed per unit time.
2. **Avg Seek Length:** it is defined as the average length of the access path of a seek operation.


HJ-BST
NM-BST
RM-BST
Read-Dominated
Mixed
keyrange: 1K
keyrange: 10K
keyrange: 100K
keyrange: 1M
Throughput (million ops/sec)
0
500
1 2 4 8 16 32 64 128
Number of Threads


HJ-BST
NM-BST
RM-BST
Write-Dominated
keyrange: 1K
keyrange: 10K
keyrange: 100K
keyrange: 1M
Throughput (million ops/sec)
0
500
1 2 4 8 16 32 64 128
Number of Threads

References

1. Ellen, F., Fataourou, P., Ruppert, E., van Breugel, F.: Non-Blocking Binary Search Trees. In: Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC). pp. 131–140 (Jul 2010)
2. Howley, S.V., Jones, J.: A Non-Blocking Internal Binary Search Tree. In: Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). pp. 161–171 (Jun 2012)
3. Natarajan, A., Mittal, N.: Fast concurrent lock-free binary search trees. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 317–328. PPoPP '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2555243.2555256