A Fast Lock-Free Internal Binary Search Tree

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-free* algorithm for concurrent manipulation of a binary search tree in an asynchronous shared memory system that supports search, insert and delete operations. Our algorithm uses (single-word) compare-and-swap (CAS) and bit-test-and-set (BTS) atomic instructions, both of which are commonly supported by many modern processors including Intel 64 and AMD64. We minimize conflicts by marking *edges* rather than *nodes*. We also reduce memory footprint by a factor of 2 by using an internal representation of a binary search tree. Compared to existing *lock-free* algorithms, our algorithm uses fewer atomic instructions in the absence of conflicts. Our experiments indicate that our *lock-free* algorithm significantly outperforms all other algorithms for a concurrent binary search tree in many cases, by as much as 70%.

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

2.1 Overview

Every operation in our algorithm begins with a seek phase.

Seek: The operation traverses the search tree from the root node until it finds the target key or if it reaches a non-binary node. We refer to the path traversed by the operation in the seek phase as the *access-path* and the last node in the *access-path* is referred to as *terminal node*. The operation then compares the target key with the value stored in the *terminal node*. Depending on the result of the comparison and the type of the operation, the operation either terminates or moves to the next phase. In certain cases in which a key may have moved upward along the *access-path*, 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 key in the *terminal node* 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 *terminal node* of the *access-path*. There are two cases depending on if the *terminal 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. 3), the *terminal node* is removed by changing the pointer at the parent of the *terminal node*. In the case of *complex* delete (as shown in Fig. 4), 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 *terminal node*).

2.2 Details of the Algorithm

Algorithm 1: structures used

```
struct Node{
    {Boolean,key} markAndKey;
    {Boolean,Boolean,Boolean,NodePtr} child[2];
    Boolean readyToReplace;
};
struct seekRecord{
    NodePtr node; NodePtr parent;
    NodePtr lastUParent; NodePtr lastUNode;
    NodePtr injectionPoint;
};
struct State{
    NodePtr node; NodePtr parent;
    Key key;
    enum mode{ INJECTION, DISCOVERY, CLEANUP };
    enum type{ SIMPLE, COMPLEX };
    seekRecPtr seekRec;
};
```

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], 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

Fig. 1. Sentinel nodes and keys ($\infty_0 < \infty_1$)

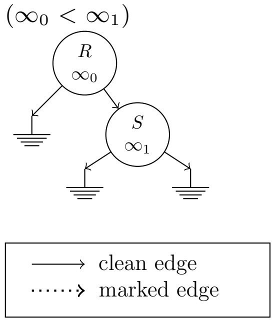
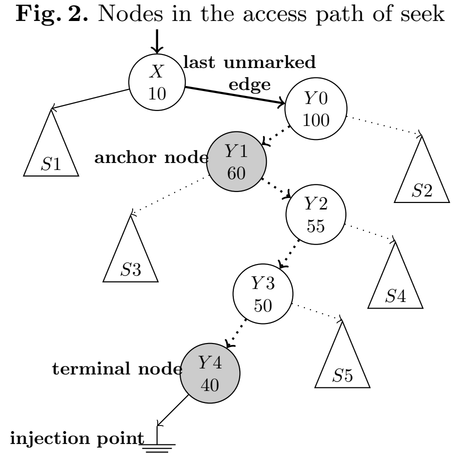


Fig. 2. Nodes in the access path of seek



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 *anchor node*. 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 *anchor node* 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 *anchor node* 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 *anchor node* of the current traversal matches with the *anchor node* of the previous traversal. If they match, then the seek operation terminates by returning the results of the previous traversal; otherwise it restarts.

Algorithm 2: seek(*key*,*seekRec*)

```
while true do
    // create two local seek records: cSeek (current seek record) and
       pSeek (previous seek record) used for the traversal
    while true do
        ⟨*,cKey⟩ := curr→markAndKey; // key in the curr of cSeek
        if key= cKey then // key found; stop the traversal
            done:= true; break;
        which:= key<cKey ? LEFT: RIGHT;
        ⟨n,d,p,address⟩ := curr→child[which]; // read the next edge
        if n then // null flag is set; reached a leaf node
            if key stored in anchor node has not changed then
                done:= true; break; // use data from cSeek
            else if anchor nodeof cSeek & pSeek matches then
                done:= true; break; // use data from pSeek
            else
                break; // after copying cSeek to pSeek
        if which= RIGHT then // next edge to be traversed is a right edge
            anchor node:= curr; // keep track of curr node
            anchor key:= cKey; // and its key
        prev:= curr; curr: = address; // traverse the next edge
        if not (d or p) then // keep track the last unmarked edge
            lastUParent:= prev; lastUNode:= curr;
    if done then
        // initialize the appropriate seek record (cSeek or pSeek)
        return;
```

Algorithm 3: Search(*key*)

```
seek( key, mySeekRec);
⟨*, nKey⟩ := mySeekRec→node→markAndKey;
if key = nKey then return true;
else return false;
```

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

Algorithm 4: Insert(*key*)

```
while true do
    seek( key, mySeekRec);
    ⟨*, nKey⟩ := mySeekRec→node→markAndKey;
    if key = nKey then return false;
    newNode:= create a new node and initialize its fields;
    which := key < nKey ? LEFT: RIGHT;
    ⟨*,*,*,address⟩ := mySeekRec→injectionPoint;
    out:= CAS(node→child[which],⟨1,0,0,address⟩,⟨0,0,0,newNode⟩);
    if out then return true;
    ⟨*,d,p,address⟩ := node→child[which]; // find out why the CAS failed
    if not (d or p) then continue; // CAS failed due to another insert op
    deepHelp(mySeekRec→lastUNode,mySeekRec→lastUParent);
```

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

Algorithm 5: Delete(*key*)

```
// initialize the state record
myState→mode:= INJECTION; myState→key:= key;
while true do
    seek( key, mySeekRec);
    node:= mySeekRec→node; parent:= mySeekRec→parent;
    ⟨*, nKey⟩ := node→markAndKey;
    if myState→key≠ nKey then
        // the key does not exist in the tree
        if myState→mode= INJECTION then return false;
        else return true;
    needToHelp:=false;
    // perform appropriate action depending on the mode
    if myState→mode= INJECTION then
        myState→node:= node // store a reference to the node
        out:= inject(myState)  // attempt to inject
        if not out then needToHelp:= true;
    // mode would have changed if the op was injected
    if myState→mode≠ INJECTION then
        // if the node found by seek is different from the one stored
           in state record, then the node is already deleted
        if myState→node≠ node then return true;
        myState→parent:= parent // update parent with recent seek
    if myState→mode= DISCOVERY then
        findAndMarkSuccessor(myState);
    if myState→mode= DISCOVERY then
        removeSuccessor(myState);
    if myState→mode= CLEANUP then
        out:= cleanup(myState,0);
        if out then return true;
        else
            ⟨*, nKey⟩ := node→markAndKey; myState→key:= nKey;
            // help if helpee node is not the node of interest
            if mySeekRec→lastUNode≠ node then needToHelp:=true;
    if needToHelp then
    deepHelp(mySeekRec→lastUNode,mySeekRec→lastUParent) ;
```

Fig. 3. An illustration of a simple delete operation.

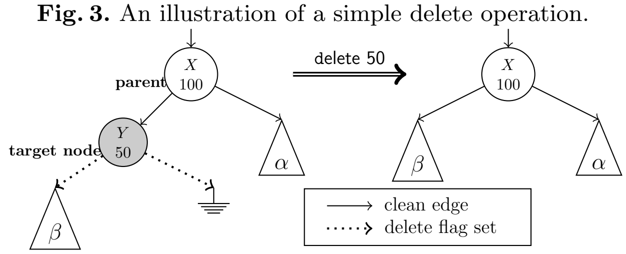
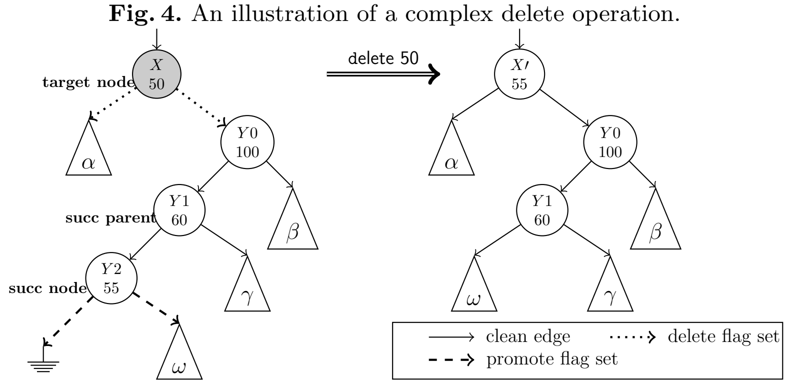


Fig. 4. An illustration of a complex delete operation.



the CAS instruction fails, the operation performs helping and restarts from the seek phase (and stays in the *injection* mode).

Algorithm 6: Inject(*state*)

```
node:= state→node // try to set the delete flag on the left edge
while true do
    ⟨n,d,p,left⟩ := node→child[LEFT];
    if d or p then return false; // edge is already marked
    out:= CAS(node→child[LEFT],⟨n,0,0,left⟩,⟨n,1,0,left⟩);
    if out then break; // retry from beginning of while loop
    updateModeAndType(state) // mark right edge, update mode and type
    return true;
```

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).

Algorithm 7: cleanup(*state*,*dFlg*)

```
// retrieve the addresses from the state record
pWhich:= edge of the parent which needs to be switched;
if state→type= COMPLEX then
    newNode:= a new copy of the node in which all the fields are unmarked;
    // try to switch the edge at the parent
    out:= CAS(parent→child[pWhich],⟨0,dFlg,0,node⟩,⟨0,dFlg,0,newNode⟩);
else
    nWhich:= non-Null child of the node being deleted;
    ⟨n,*,*,address⟩ := node→child[nWhich];
    if n then  // set only the null flag; do not change the address
        out:= CAS(parent→child[pWhich],⟨0,dFlg,0,node⟩,⟨1,dFlg,0,node⟩);
    else  // change the address here by switching the pointer
        out:= CAS(parent→child[pWhich],⟨0,dFlg,0,node⟩,⟨0,dFlg,0,address⟩);
return out;
```

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 T be the *terminal node*, $T.parent$ be its parent node and $T.left$ and $T.right$ be the left and right child node respectively, of T. We now discuss the two types of delete operation.

Simple Delete: In this case, either $T.left$ or $T.right$ is a null node. Note that both $T.left$ and $T.right$ may be null nodes in which case T will be a leaf node. Without loss of generality, assume that $T.right$ is a null node. Delete operation attempts to change the pointer at $T.parent$ that is pointing to T to point to $T.left$ 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 returns a *terminal node* different from T, then T has been already removed from the tree (by another operation as part of helping) and the operation terminates; otherwise, it attempts to remove T from the tree again. Note that the new seek operation may return a different parent node. This process may be repeated multiple times.

Algorithm 8: findAndMarkSuccessor(*key*,*seekRec*)

```
// retrieve the addresses from the state record
node:= state→node; seekRec:= state→seekRec;
while true do
    right:= address of the right child;
    findSmallest(node,right,seekRec);
    succNode:= seekRec→node; // retrieve succ node from seek record
    left:= address of the left child of the succNode;
    // try to set the promote flag & copy the node address on the
       left edge using CAS
    out:= CAS(succNode→child[LEFT],⟨1,0,0,left⟩,⟨1,0,1,node⟩);
    if out then break; // promote flag set; promotion will eventually succeed
    // reread the edge to see why the attempt to mark the edge failed
    ⟨n,d,p,left⟩ := succNode→child[LEFT];
    if p then
        if left= node then
            break// successor node has already been selected
        else  // the node found is a successor node for another delete operation
            node→readyToReplace:= true
        if not n then continue; // the node found has since gained a left child
        if d then the node found is undergoing deletion. So invoke helping;
updateModeAndType(state); // update the operation mode and type
return;
```

Complex Delete: In this case, both $T.left$ and $T.right$ 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 *terminal node* T. 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 *S.parent* 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 *terminal node* (for deletion) and marking a *successor node* (for promotion),

Algorithm 9: removeSuccessor(*state*)

```
// retrieve the addresses from the state record
node:= state→node; seekRec:=state→seekRec;
succNode:= seekRec→node;
if promote flag not set on right child edge of succNode then
    BTS(succNode→child[RIGHT],PROMOTE_FLAG);
node→markAndKey:= ⟨1,succNode→markAndKey⟩; // promote the key
while true do
    succParent:= seekRec→parent; // retrieve parent of the succNode
    right:= right child address of succNode;
    out:= CAS(succParent→child[LEFT],⟨0,0,0,succNode⟩,⟨0,0,0,right⟩);
    if out then break; // successor removed successfully
    // invoke helping if needed
    findSmallest(node, right,seekRec);
    if seekRec→node≠ succNode then break; // successor already removed
node→readyToReplace:= true;
if state→parent≠ null then updateModeAndType(state);
return;
```

we steal two bits from the address and refer to them as *deleteFlag* and *promoteFlag* respectively. 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 *terminal node* T 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 *terminal node*. At the same time, the mark bit in the key is set to indicate that the key currently stored in the *terminal node* is the *successor key* and not the target key.
4. The *successor node* S is deleted by changing the pointer at *S.parent* 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 *terminal node* T 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 *terminal node* has been replaced with the *successor key*. Further, its key as well as both its edges

are marked. The *terminal node* is now replaced with a new node whose contents are same as that of the *terminal node* expect that all the fields are unmarked. The *terminal 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 *terminal node* different from T, then T 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 *terminal node* T with the *successor key*, it attempts to replace T again. Note that the new seek operation may return a different parent node. This process may be repeated multiple times.

Algorithm 10: findSmallest(*node*,*right*,*seekRec*)

```
// find the smallest key in the subtree rooted at the right child
lastUParent:= node; lastUNode:= right; prev:= node; curr:= right;
while true do
    ⟨n,d,p,left⟩ := curr→child[LEFT];
    if n then break;
    prev:= curr; curr:= left; // traverse the next edge
    if not (d or p) then  // keep track of the last unmarked edge
        lastUParent:= prev; lastUNode:= curr
    // update the seek record
    return;
```

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 *terminal node*. Recall that T denotes a *terminal node* and $T.right$ its right child node. If no unmarked edge is found during the traversal from $T.right$, then helping is performed along the edge (T,$T.right$),that is, the delete operation in progress at $T.right$ is helped. This will involve modifying the

Algorithm 11: updateModeAndType(*state*)

```
node:= state→node // retrieve the address from the state record
if node→child[RIGHT] ≠ ⟨*,1,*,*⟩ then // mark right edge if unmarked
    BTS(node→child[RIGHT], DELETE_FLAG);
⟨m,*⟩ := node→markAndKey;
⟨lN,*,*,*⟩ := node→child[LEFT]; ⟨rN,*,*,*⟩ := node→child[RIGHT];
if lN or rN then // update the op mode and type
    if m then
        state→type:= COMPLEX; node→readyToReplace:= true;
    else
        state→type:= SIMPLE; state→mode:= CLEANUP;
else
    state→type:= COMPLEX;
    if readyToReplace then state→mode:= CLEANUP;
    else state→mode:= DISCOVERY;
return;
```

edge (T,$T.right$) even though the edge is marked to either remove $T.right$ (if *simple* delete) or replace $T.right$ with a fresh node (if *complex* delete).

3 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 Concurrency:** 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.

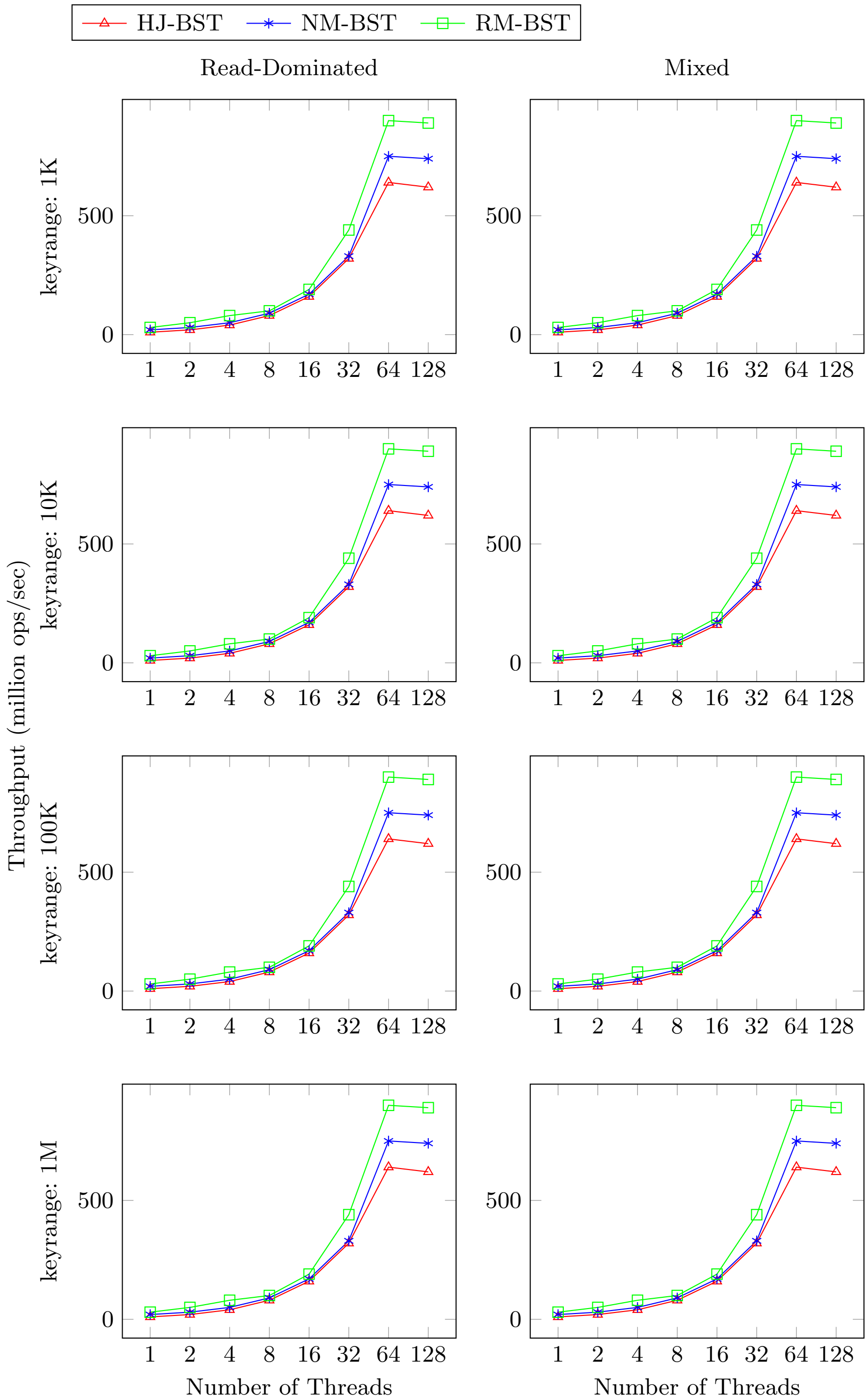
Table 1. hardware features

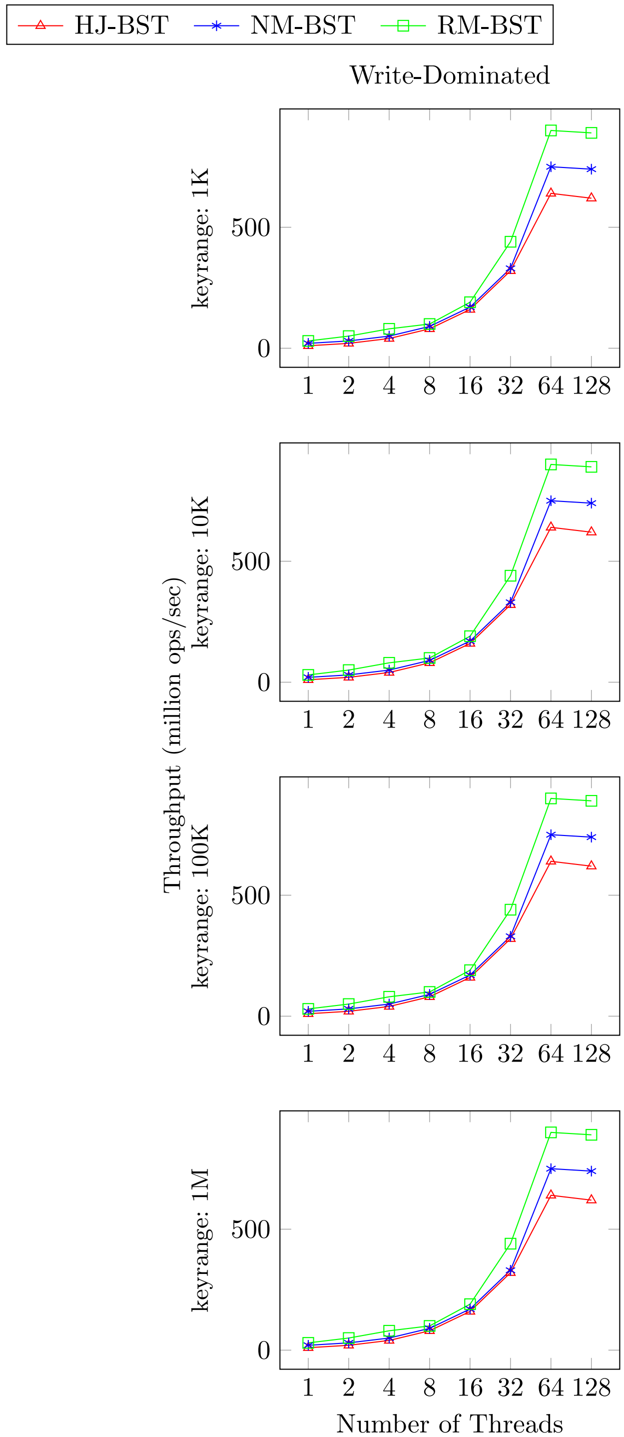
CPU sockets	4
Cores per socket	8
Threads per core	2
Clock frequency	2.3 GHz
L1 cache (I/D)	64KB/16KB
L2 cache	2 MB
L3 cache	6 MB
Memory	256 GB

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
Throughput (million ops/sec)
keyrange: 1K
keyrange: 10K
keyrange: 100K
keyrange: 1M
0
500
1
2
4
8
16
32
64
128
Number of Threads


HJ-BST
NM-BST
RM-BST
Write-Dominated
Throughput (million ops/sec)
keyrange: 1K
keyrange: 10K
keyrange: 100K
keyrange: 1M
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