Combining HTM and RCU to Implement Highly Efficient Balanced Binary Search Trees

Dimitrios Siakavaras, Konstantinos Nikas, Georgios Goumas and Nectarios Koziris

National Technical University of Athens (NTUA)
School of Electrical and Computer Engineering (ECE)
Computing Systems Laboratory (CSLab)
{jimsiak,knikas,goumas,nkoziris}@cslab.ece.ntua.gr
http://research.cslab.ece.ntua.gr

Transact/WTTM 2017

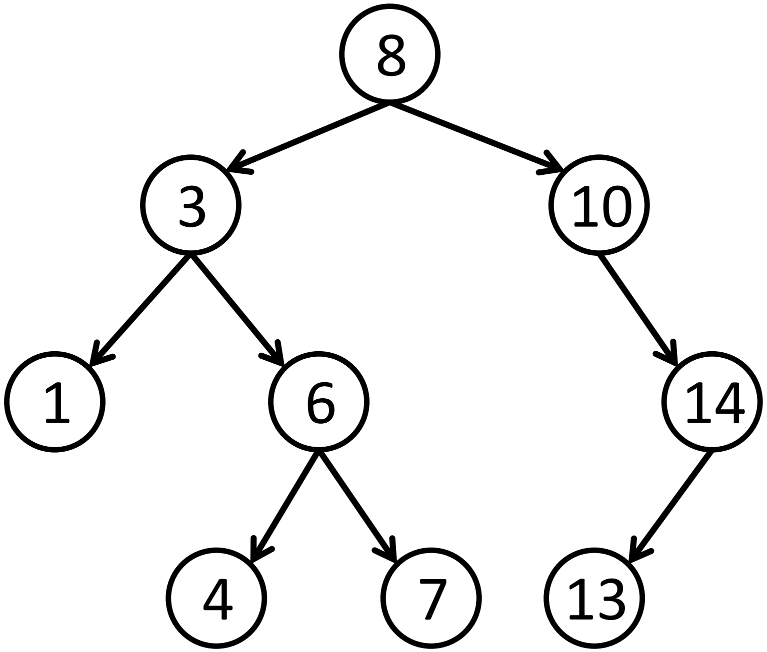
National Technical University of Athens
CSLab

Outline

- Binary Search Trees (BSTs)
- Concurrent BSTs
- RCU-HTM
- Experimental results
- Conclusions & Future work

BINARY SEARCH TREES

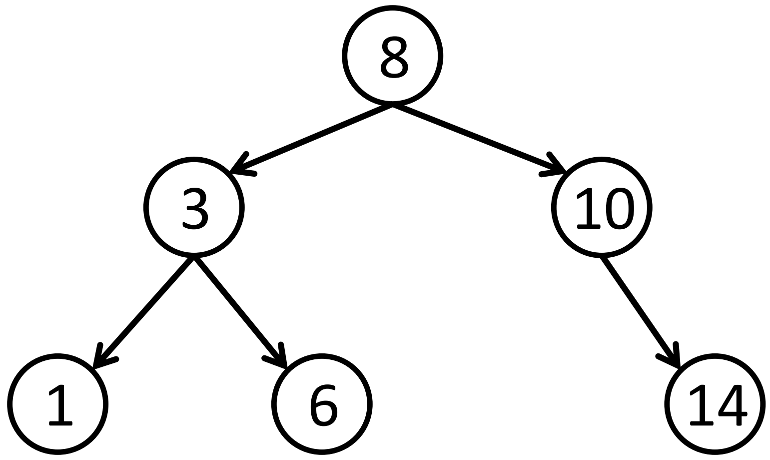
Binary Search Trees (BSTs)



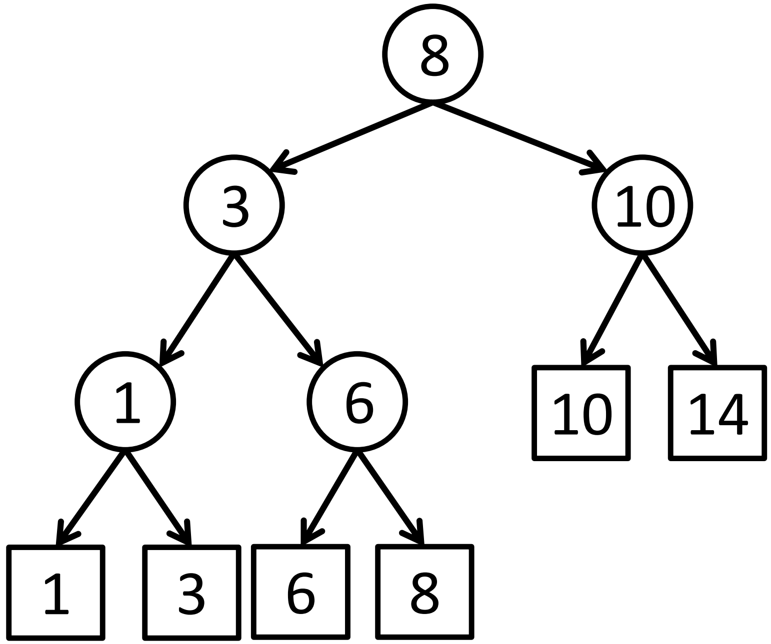
- A classic binary tree with an additional property:
  - Nodes in left subtree have keys less than the key of the root, nodes in right subtree have keys greater than the root.

- Most commonly used to implement *dictionaries:*
  - <key,value> pairs
  - 3 operations: *lookup(key), insert(key, value) delete(key)*

Internal vs. External BSTs



Internal



External

Internal: <key,value> pairs in every node
External: values only in leaves, internal nodes only contain keys.

- External trees simplify the *delete()* operation
- They require twice as much memory
- Longer traversal paths

National Technical University of Athens CSLab

Deletion in an Internal BST

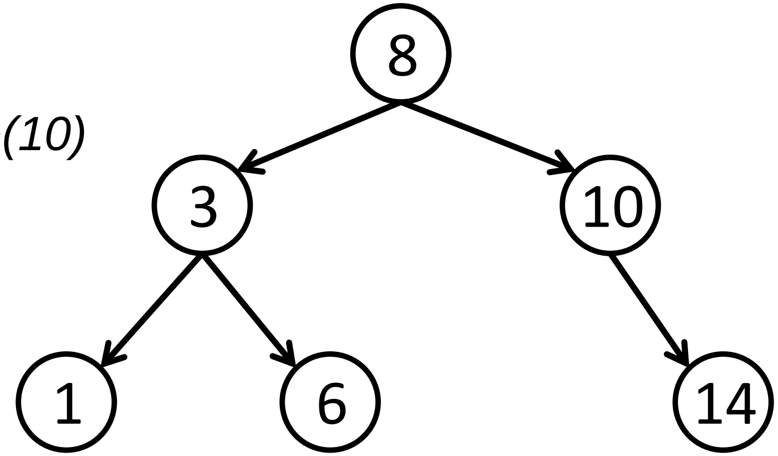
Deletion in an Internal BST

- Deleting a node with one or zero children is easy
  - Just change parent’s child pointer

Deletion in an Internal BST

- Deleting a node with one or zero children is easy
  - Just change parent's child pointer

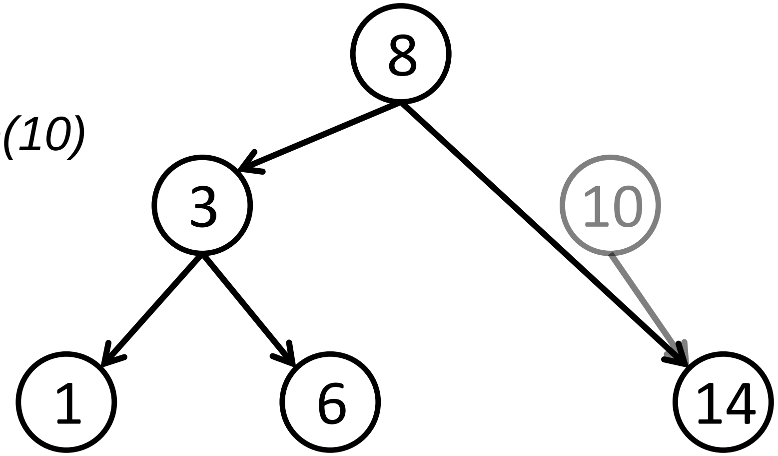
Example: *delete(10)*



Deletion in an Internal BST

- Deleting a node with one or zero children is easy
  - Just change parent's child pointer

Example: *delete(10)*



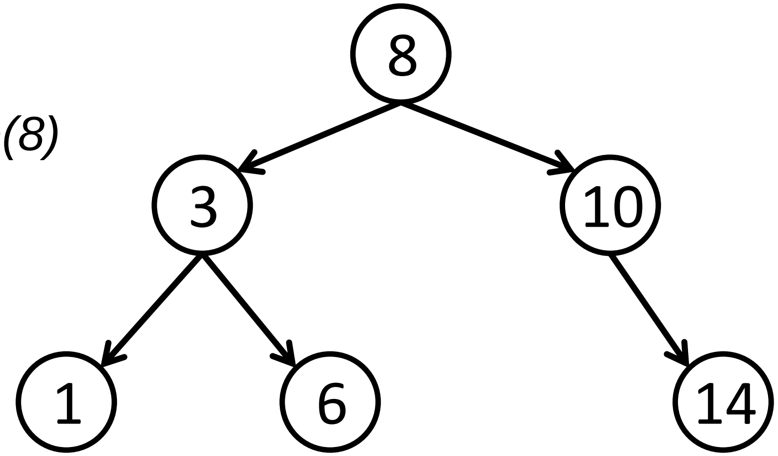
Deletion in an Internal BST

- Deleting a node with one or zero children is easy
  - Just change parent's child pointer
- Deleting a node with two children is more complicated
  - Need to find successor, swap keys and remove successor node
  - Successor may be many links away

Deletion in an Internal BST

- Deleting a node with one or zero children is easy
  - Just change parent's child pointer
- Deleting a node with two children is more complicated
  - Need to find successor, swap keys and remove successor node
  - Successor may be many links away

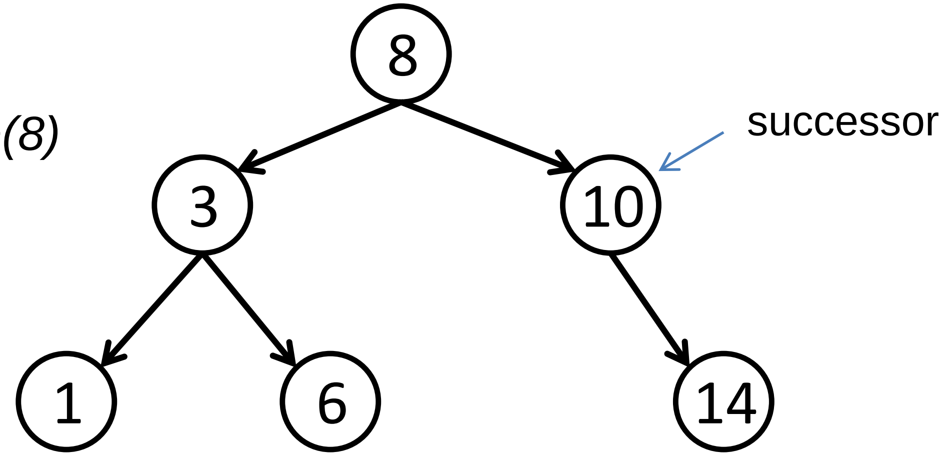
Example: *delete(8)*



Deletion in an Internal BST

- Deleting a node with one or zero children is easy
  - Just change parent's child pointer
- Deleting a node with two children is more complicated
  - Need to find successor, swap keys and remove successor node
  - Successor may be many links away

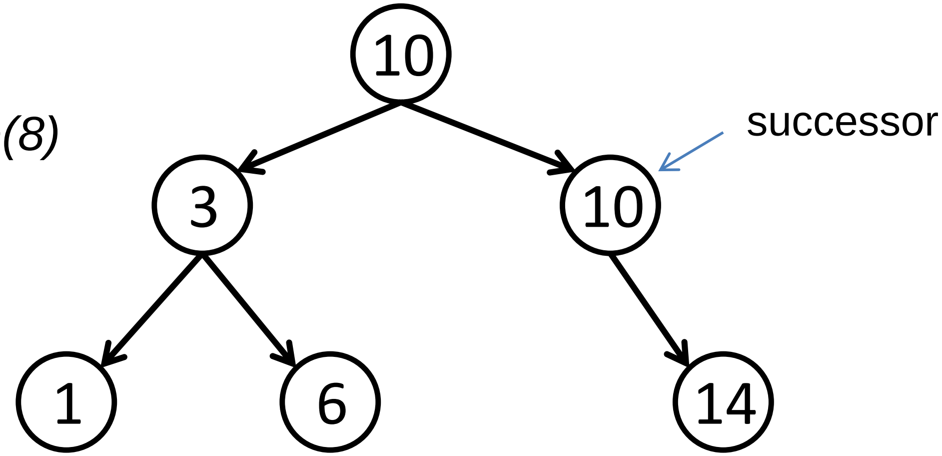
Example: *delete(8)*



Deletion in an Internal BST

- Deleting a node with one or zero children is easy
  - Just change parent's child pointer
- Deleting a node with two children is more complicated
  - Need to find successor, swap keys and remove successor node
  - Successor may be many links away

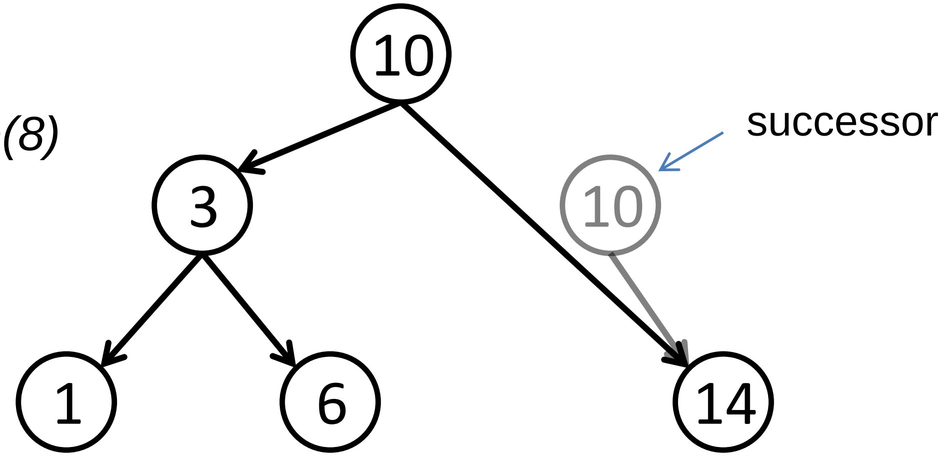
Example: *delete(8)*



Deletion in an Internal BST

- Deleting a node with one or zero children is easy
  - Just change parent's child pointer
- Deleting a node with two children is more complicated
  - Need to find successor, swap keys and remove successor node
  - Successor may be many links away

Example: *delete(8)*



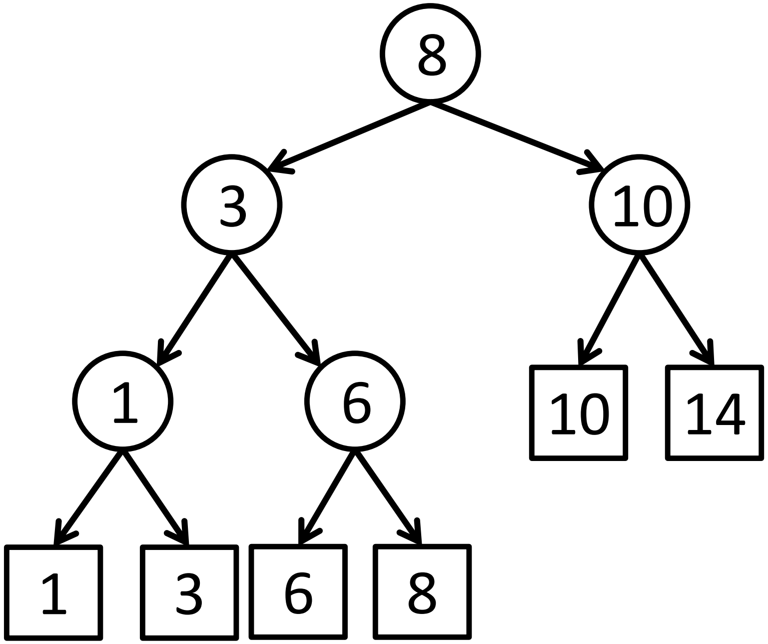
Deletion in an External BST

- Deletion is always simple

Deletion in an External BST

- Deletion is always simple

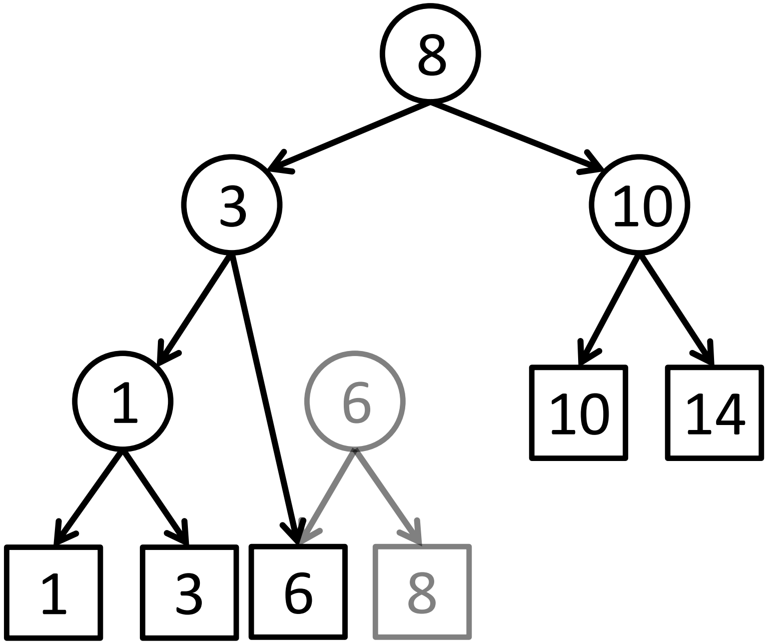
Example: *delete(8)*



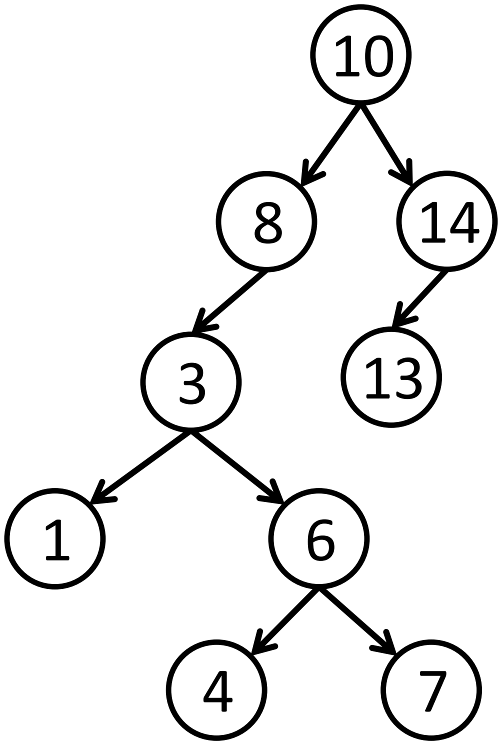
Deletion in an External BST

- Deletion is always simple

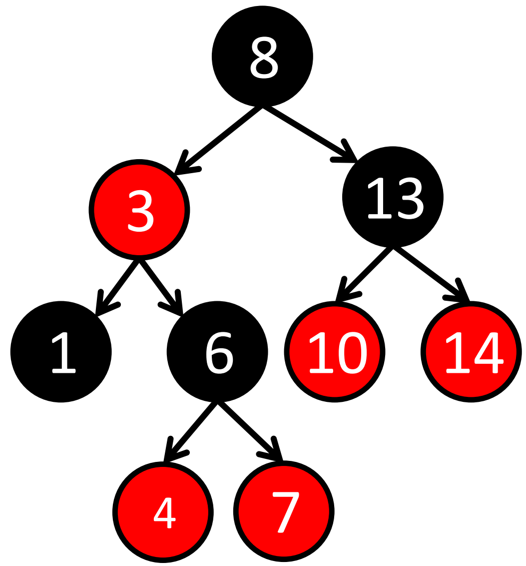
Example: *delete(8)*



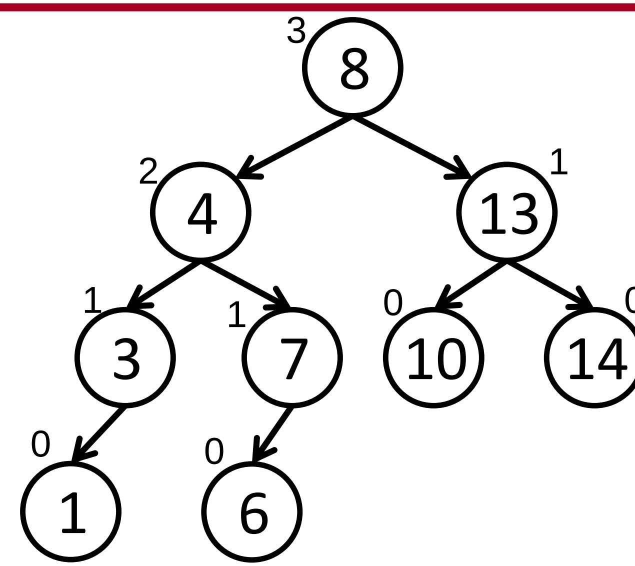
Unbalanced vs. Balanced BSTs



Unbalanced Tree



Red-Black Tree



AVL Tree

- \+ Balanced trees limit the height of the tree (i.e., the length of maximum path) to provide bounded and predictable traversal times
- \- Rebalancing requires additional effort after insertions/deletions

Insertion in an Unbalanced BST

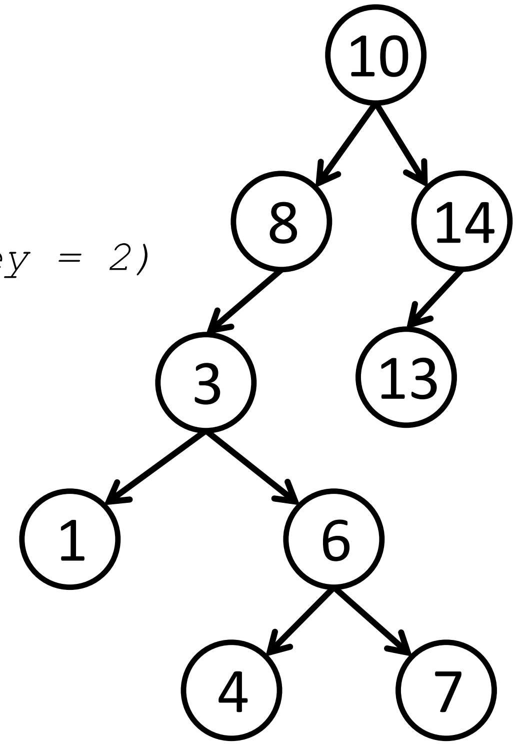
```
int bst_insert(bst_t *bst, int key, void *value)
{
  traverse_bst(bst, key);
  if (key was found) return 0;
  insert_node(bst, key, value);
  return 1;
}
```


National Technical University of Athens CSLab

Insertion in an Unbalanced BST

```
int bst_insert(bst_t *bst, int key, void *value)
{
  traverse_bst(bst, key);
  if (key was found) return 0;
  insert_node(bst, key, value);
  return 1;
}
```

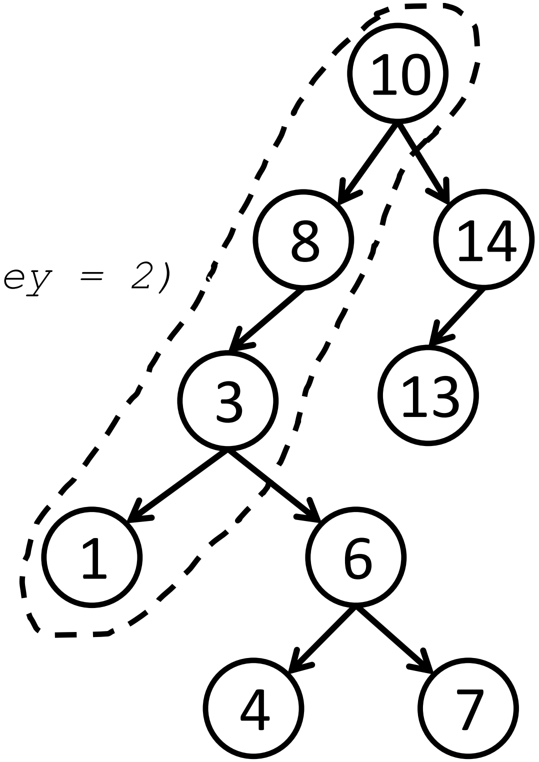
Example:
bst_insert(key = 2)



Insertion in an Unbalanced BST

```
int bst_insert(bst_t *bst, int key, void *value)
{
  traverse_bst(bst, key);
  if (key was found) return 0;
  insert_node(bst, key, value);
  return 1;
}
```

Example:
bst_insert(key = 2);

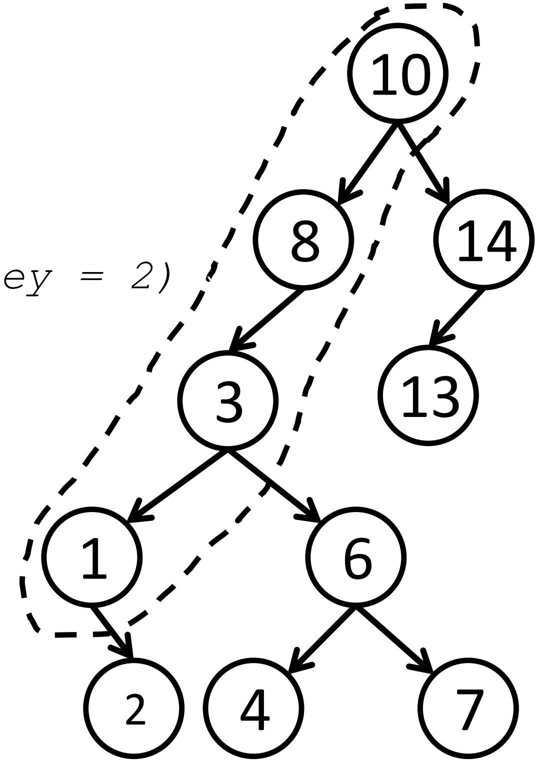


National Technical University of Athens CSLab

Insertion in an Unbalanced BST

```
int bst_insert(bst_t *bst, int key, void *value)
{
  traverse_bst(bst, key);
  if (key was found) return 0;
  insert_node(bst, key, value);
  return 1;
}
```

Example:
bst_insert(key = 2);



Insertion in a Balanced BST

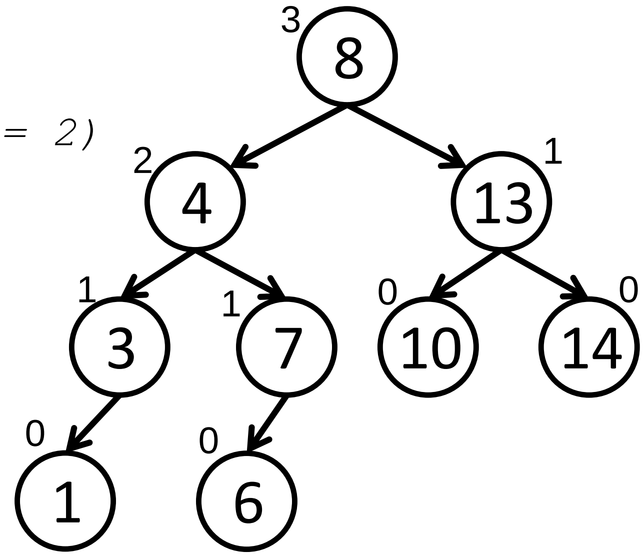
```
int bbst_insert(bbst_t *bst, int key, void *value)
{
  traverse_bbst(bbst, key);
  if (key was found) return 0;
  insert_node_and_rebalance(bbst, key, value);
  return 1;
}
```

Insertion in a Balanced BST

```
int bbst_insert(bbst_t *bst, int key, void *value)
{
  traverse_bbst(bbst, key);
  if (key was found) return 0;
  insert_node_and_rebalance(bbst, key, value);
  return 1;
}
```

Example:

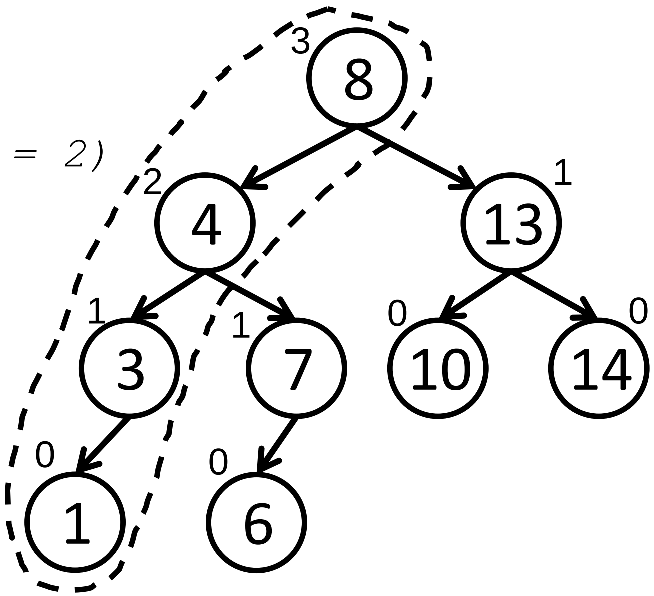
bst_insert(key = 2)



Insertion in a Balanced BST

```
int bbst_insert(bbst_t *bst, int key, void *value)
{
  traverse_bbst(bbst, key);
  if (key was found) return 0;
  insert_node_and_rebalance(bbst, key, value);
  return 1;
}
```

Example:
bst_insert(key = 2)

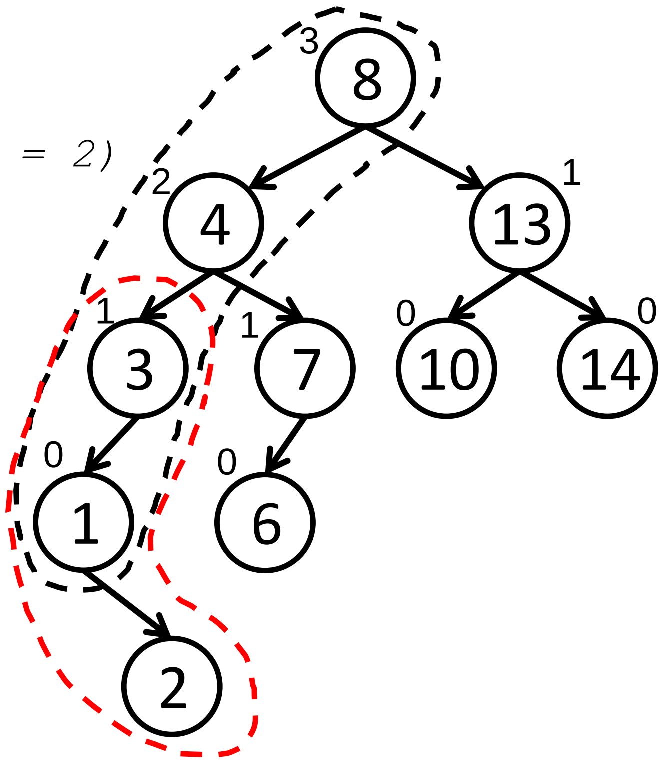


Insertion in a Balanced BST

```
int bbst_insert(bbst_t *bst, int key, void *value)
{
  traverse_bbst(bbst, key);
  if (key was found) return 0;
  insert_node_and_rebalance(bbst, key, value);
  return 1;
}
```

Example:

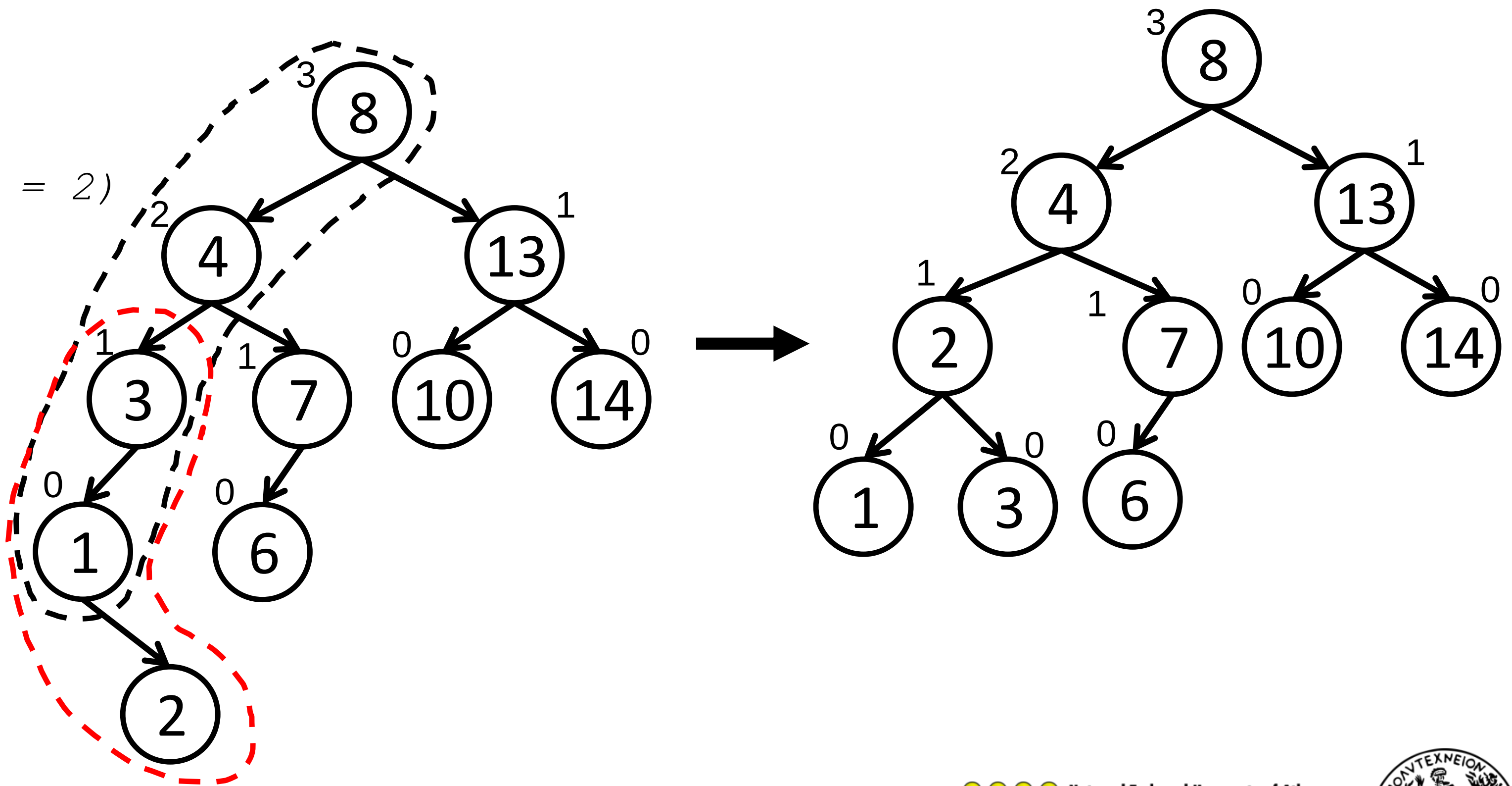
bst_insert(key = 2)



Insertion in a Balanced BST

```
int bbst_insert(bbst_t *bst, int key, void *value)
{
  traverse_bbst(bbst, key);
  if (key was found) return 0;
  insert_node_and_rebalance(bbst, key, value);
  return 1;
}
```

Example:
bst_insert(key = 2)



CONCURRENT BINARY SEARCH TREES

Concurrent BSTs

There are 2 challenges for concurrent *internal* and *balanced* BSTs:

1. The deletion of a node with 2 children requires exclusive access to the whole path from the node to the successor.
2. Rebalancing requires several modifications that need to be performed in a single atomic step.

Proposed non-blocking and lock-based concurrent BSTs are either:

- Unbalanced [Natarajan PPoPP'14, Howley SPAA'12, Ellen PODC'10]
- Relaxed balanced [Bronson PPoPP'10, Drachsler PPoPP'14, Brown PPoPP'14]
- External [Natarajan PPoPP'14, Ellen PODC'10]
- Partially external [Bronson PPoPP'10]

Concurrent RCU-based BSTs

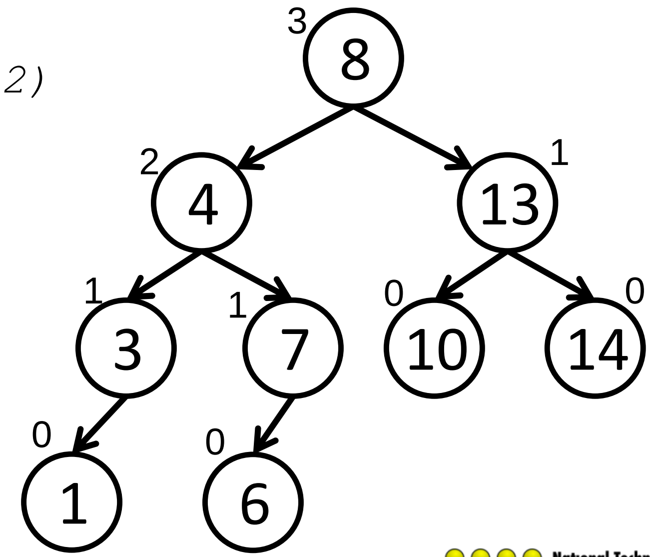
- Read-Copy-Update (RCU)
  - Modifications are performed in copies and not in place. Copies are atomically *installed* in the shared data structure.
  - Readers may proceed without any synchronization and without restarting
  - Updaters need to be explicitly synchronized (most commonly only a single updater is allowed to operate)

Concurrent RCU-based BSTs

- Read-Copy-Update (RCU)
  - Modifications are performed in copies and not in place. Copies are atomically *installed* in the shared data structure.
  - Readers may proceed without any synchronization and without restarting
  - Updaters need to be explicitly synchronized (most commonly only a single updater is allowed to operate)

Example:

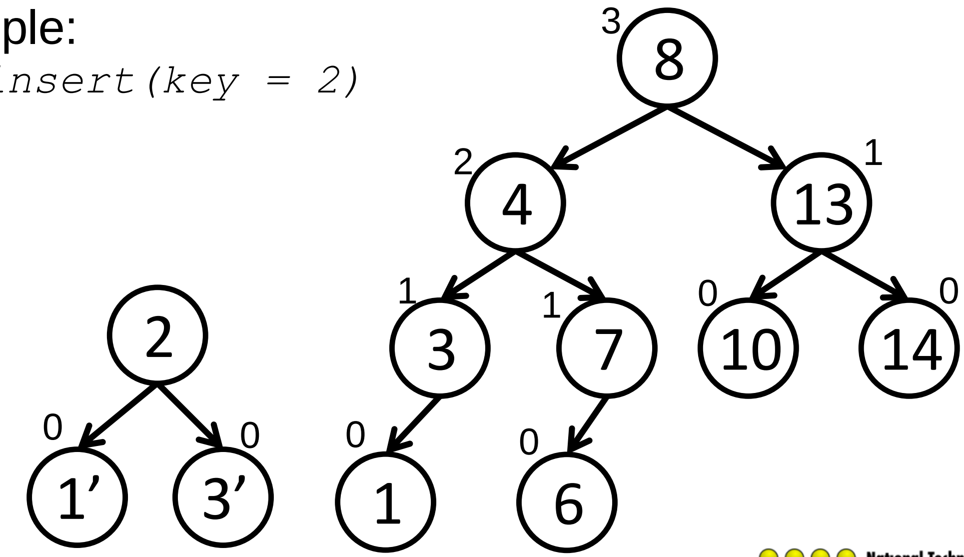
bst_insert(key = 2)



Concurrent RCU-based BSTs

- Read-Copy-Update (RCU)
  - Modifications are performed in copies and not in place. Copies are atomically *installed* in the shared data structure.
  - Readers may proceed without any synchronization and without restarting
  - Updaters need to be explicitly synchronized (most commonly only a single updater is allowed to operate)

Example:
bst_insert(key = 2)

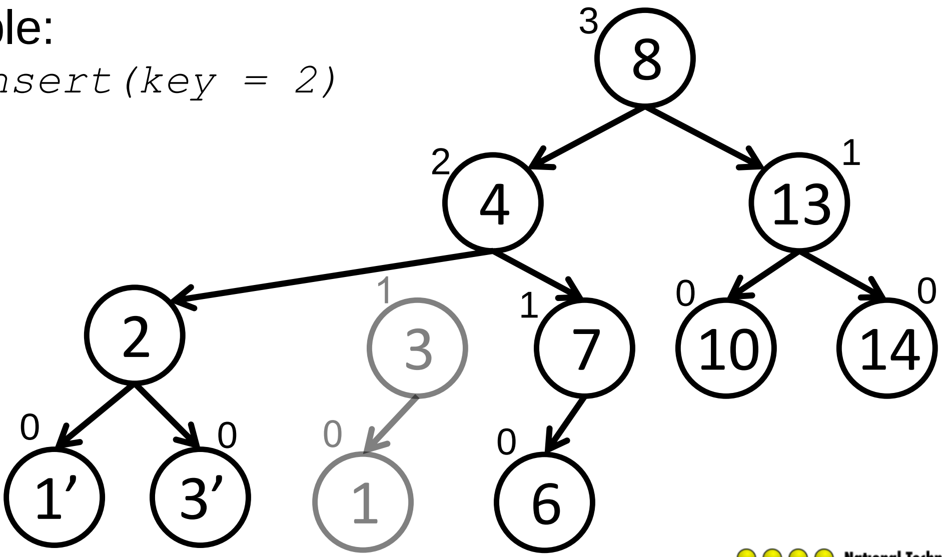


National Technical University of Athens CSLab

Concurrent RCU-based BSTs

- Read-Copy-Update (RCU)
  - Modifications are performed in copies and not in place. Copies are atomically *installed* in the shared data structure.
  - Readers may proceed without any synchronization and without restarting
  - Updaters need to be explicitly synchronized (most commonly only a single updater is allowed to operate)

Example:
`bst_insert(key = 2)`



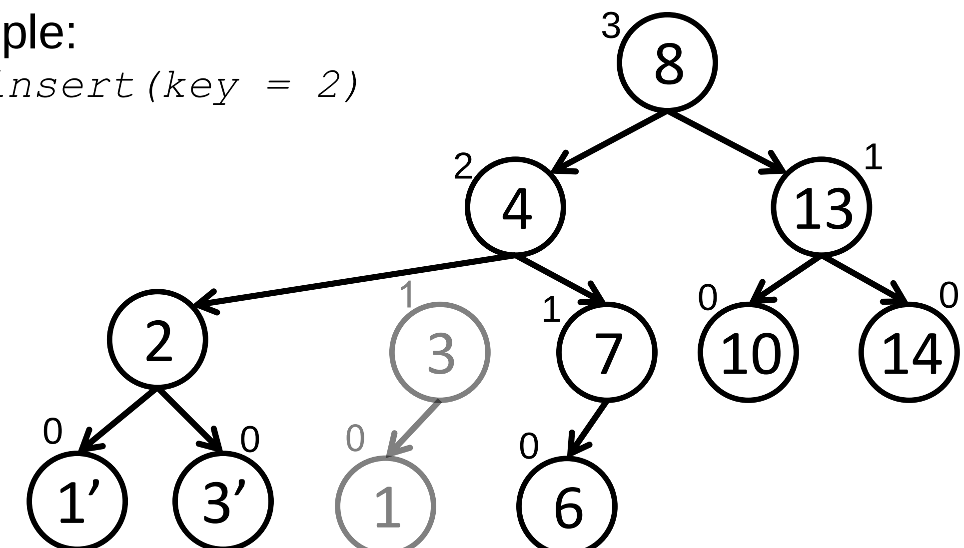
National Technical University of Athens CSLab

Concurrent RCU-based BSTs

- Read-Copy-Update (RCU)
  - Modifications are performed in copies and not in place. Copies are atomically *installed* in the shared data structure.
  - Readers may proceed without any synchronization and without restarting
  - Updaters need to be explicitly synchronized (most commonly only a single updater is allowed to operate)

Example:
`bst_insert(key = 2)`

Old readers may still traverse old versions of nodes. New readers will see the new nodes.



Concurrent RCU-based BSTs

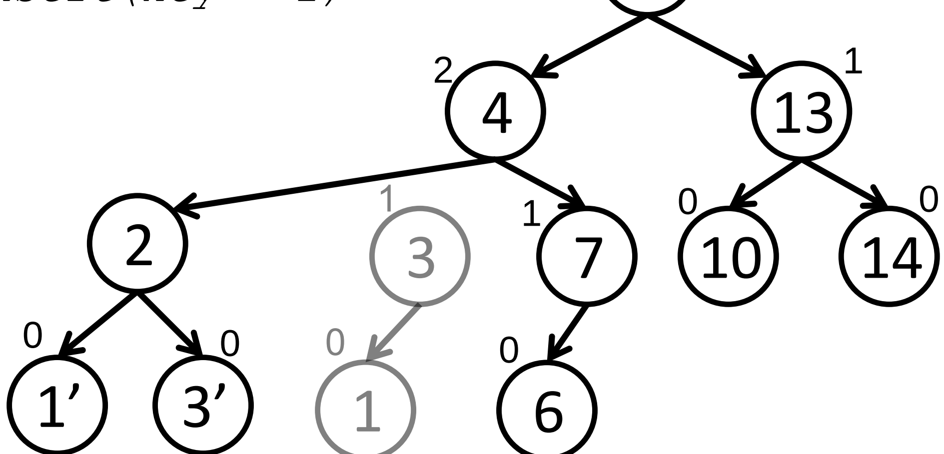
- Read-Copy-Update (RCU)
  - Modifications are performed in copies and not in place. Copies are atomically *installed* in the shared data structure.
  - Readers may proceed without any synchronization and without restarting
  - Updaters need to be explicitly synchronized (most commonly only a single updater is allowed to operate)

Example:

`bst_insert(key = 2)`

Old readers may still traverse old versions of nodes. New readers will see the new nodes.

Updaters can safely replace parts of the tree as only a single updater is allowed.



National Technical University of Athens CSLab

Concurrent RCU-based BSTs

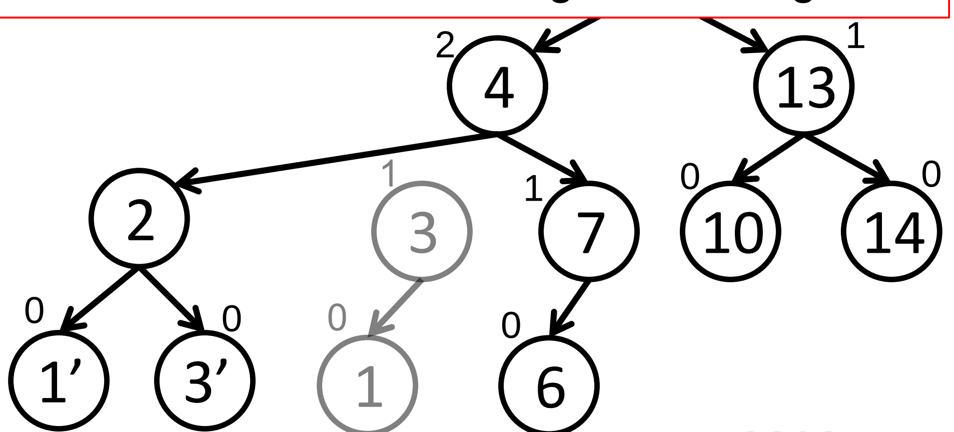
- Read-Copy-Update (RCU)
  - Modifications are performed in copies and not in place. Copies are atomically *installed* in the shared data structure.
  - Readers may proceed without any synchronization and without restart
  - Updat ... ly only a single

Single updater RCU tree:
- Multiple readers
- Single updater

Citrus RCU tree [Arbel PODC'14]:
- Multiple updaters using fine-grain locks.
- Unbalanced tree to enable fine-grain locking

Old readers may still traverse old versions of nodes. New readers will see the new nodes.

Updaters can safely replace parts of the tree as only a single updater is allowed.



National Technical University of Athens CSLab

Concurrent HTM-based BSTs

- Hardware Transactional Memory (HTM)
  - Avoids STM's huge overheads
  - Allows the modification of multiple locations atomically → good fit for the rebalancing phase in a BBST
- HTM-based BSTs:
  - Coarse-grained HTM (cg-htm):
    - Each operation enclosed in a single transaction
    - \+ Easy to implement
    - \- Large transactions (increased conflict probability)
  - Consistency-Oblivious-Programming HTM (cop-htm) [Avni TRANSACT'14]:
    - The traversal is performed outside the transaction
    - The executed transaction includes 2 steps:
      - Validate that the traversal ended at the correct node
      - Insert/Delete the node and rebalance if necessary
    - \+ Shorter transactions than cg-htm
    - \- Traversals (and consequently lookup operations) may need to restart

RCU-HTM

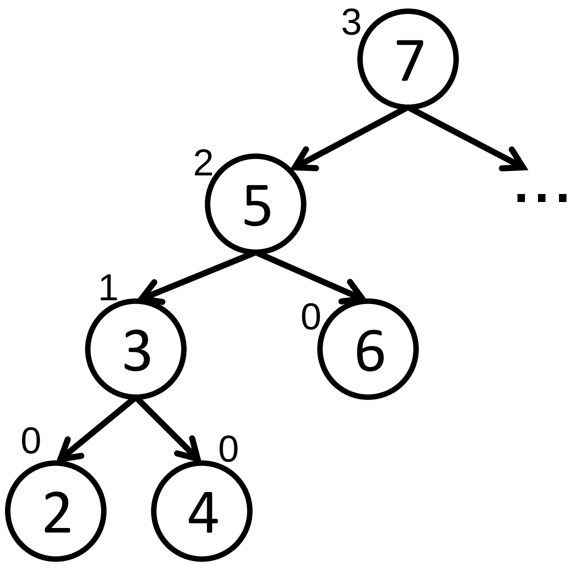
RCU-HTM

Combines **RCU** with **HTM** in an innovative way and provides trees with:

1. Asynchronized traversals (thanks to RCU)
   - Oblivious of concurrent updates in the tree
   - No locks, no transactions or any other synchronization
   - No restarts
2. Concurrent updaters (thanks to HTM)
   - All updates are performed in copies
   - Modified copies are first validated and then installed in the tree
   - An HTM transaction is used for the validation+installation phase
     - HTM transaction includes several reads but only a single write → minimized conflict probability

RCU-HTM: insert operation

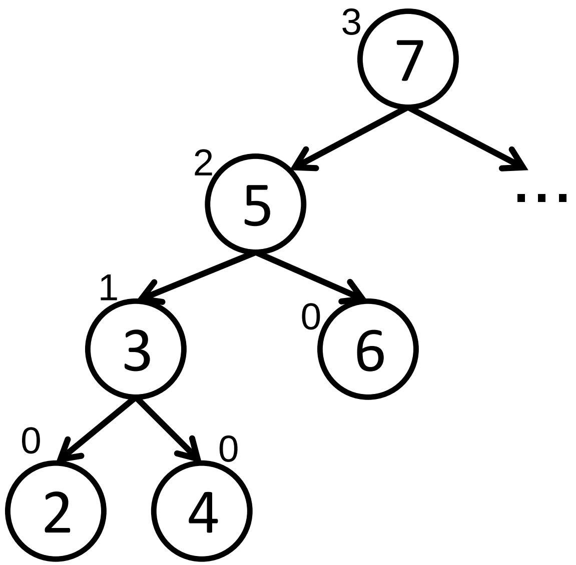
Example: *insert(key = 1)*



RCU-HTM: insert operation

1. Traverse the tree to locate the insertion point
   - During traversal we maintain a stack of pointers to the traversed nodes

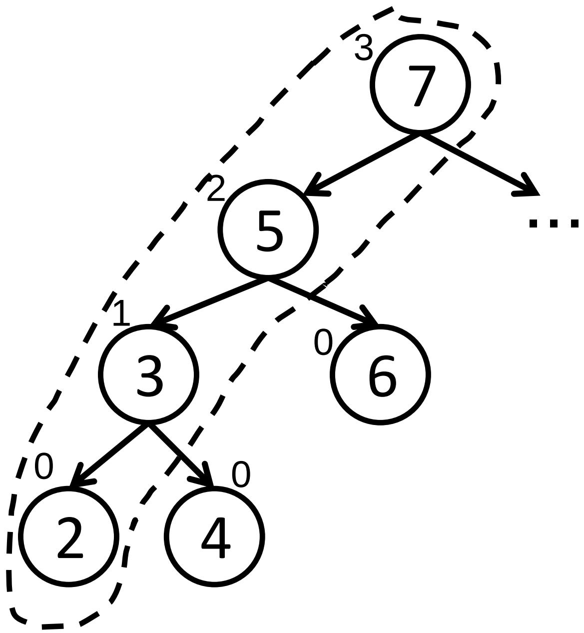
Example: *insert(key = 1)*



RCU-HTM: insert operation

1. Traverse the tree to locate the insertion point
   - During traversal we maintain a stack of pointers to the traversed nodes

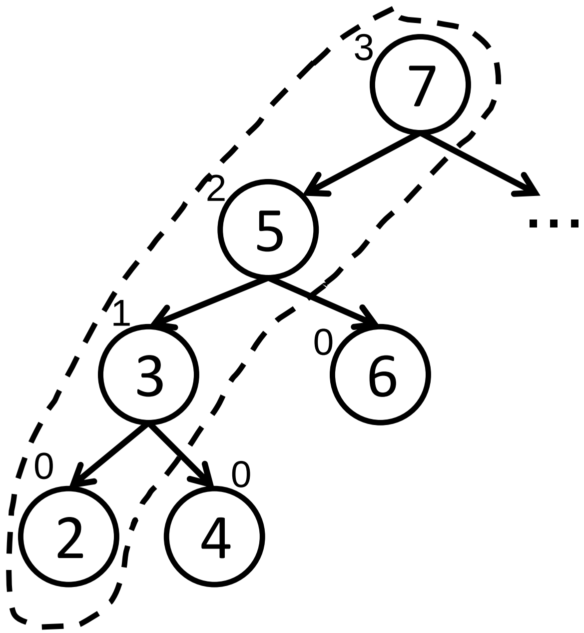
Example: *insert(key = 1)*



RCU-HTM: insert operation

1. Traverse the tree to locate the insertion point
   - During traversal we maintain a stack of pointers to the traversed nodes
2. Perform the insertion and rebalance using copies
   - The reverse traversal uses the saved stack of pointers
   - For each copied node we store the observed children pointers

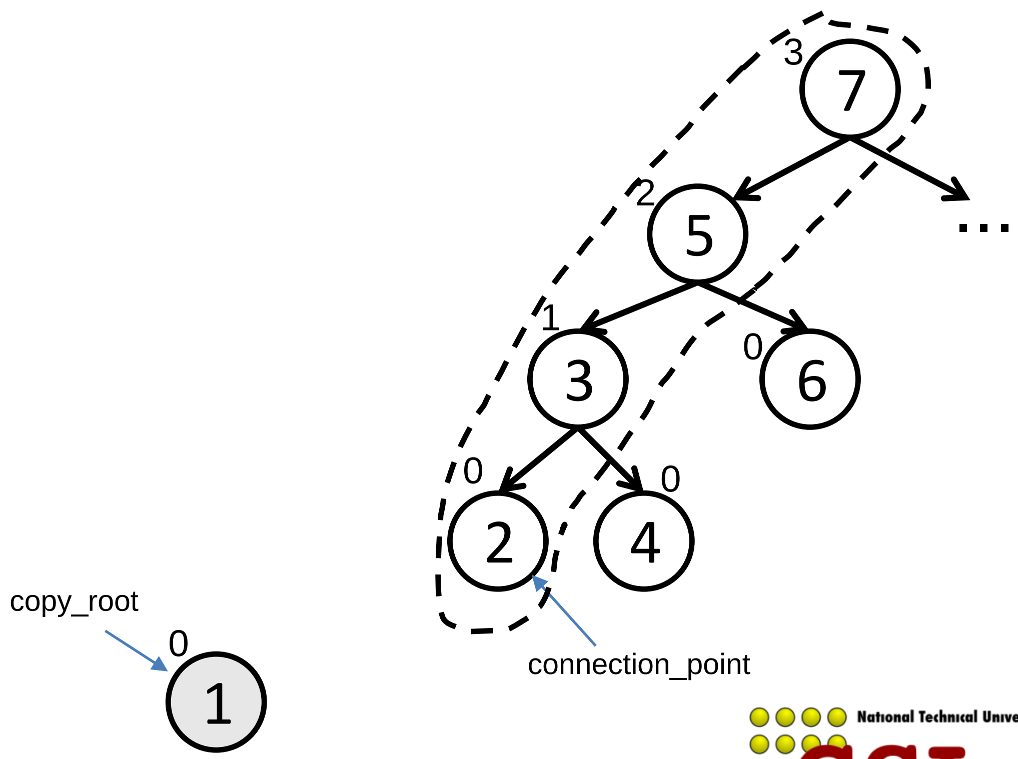
Example: *insert(key = 1)*



RCU-HTM: insert operation

1. Traverse the tree to locate the insertion point
   - During traversal we maintain a stack of pointers to the traversed nodes
2. Perform the insertion and rebalance using copies
   - The reverse traversal uses the saved stack of pointers
   - For each copied node we store the observed children pointers

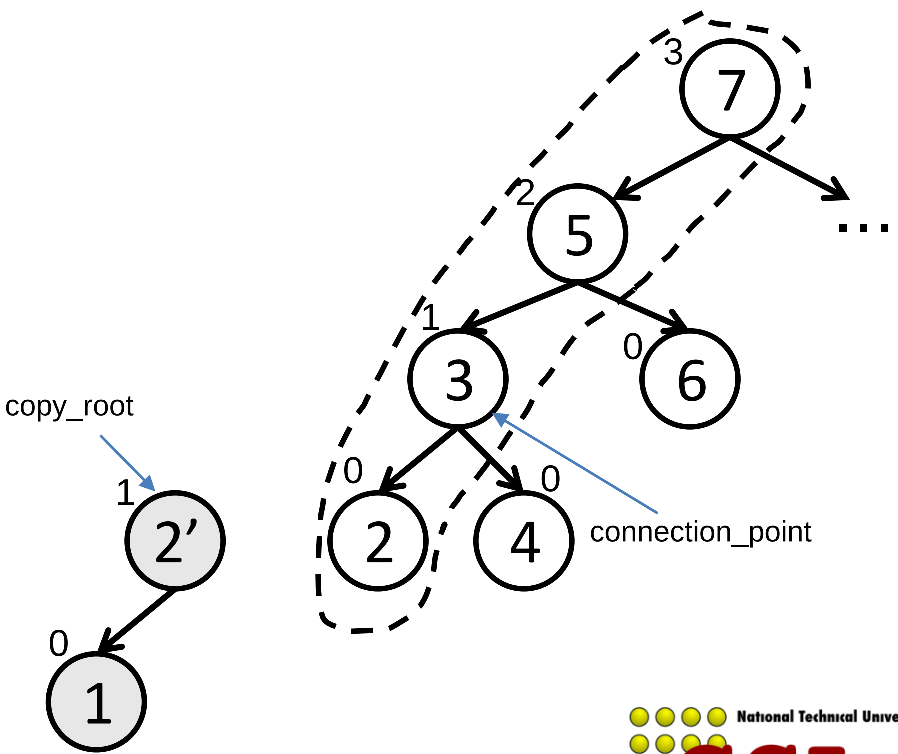
Example: *insert(key = 1)*



RCU-HTM: insert operation

1. Traverse the tree to locate the insertion point
   - During traversal we maintain a stack of pointers to the traversed nodes
2. Perform the insertion and rebalance using copies
   - The reverse traversal uses the saved stack of pointers
   - For each copied node we store the observed children pointers

Example: *insert(key = 1)*

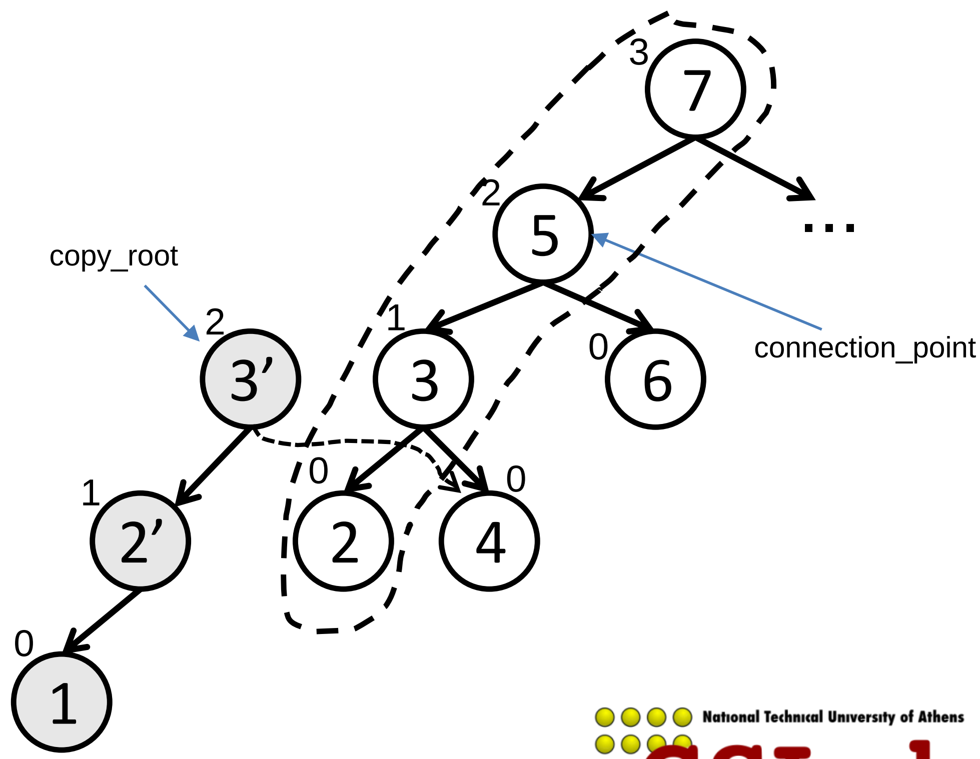


National Technical University of Athens CSLab

RCU-HTM: insert operation

1. Traverse the tree to locate the insertion point
   - During traversal we maintain a stack of pointers to the traversed nodes
2. Perform the insertion and rebalance using copies
   - The reverse traversal uses the saved stack of pointers
   - For each copied node we store the observed children pointers

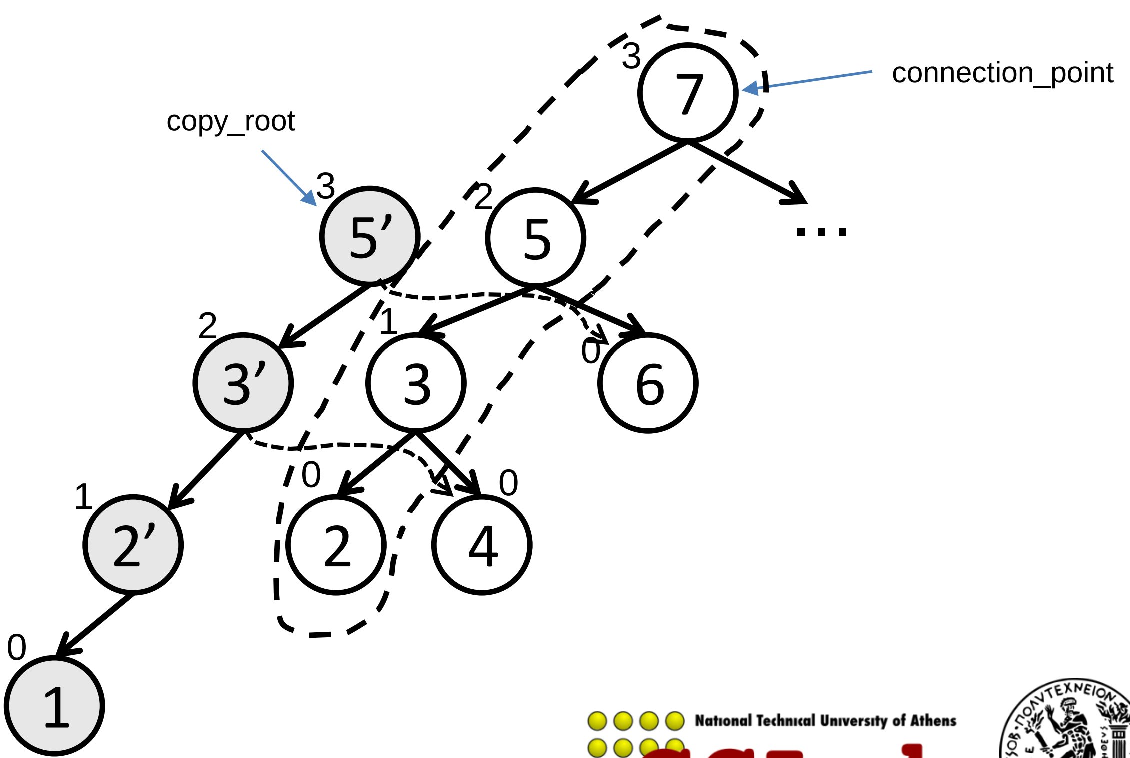
Example: *insert(key = 1)*



RCU-HTM: insert operation

1. Traverse the tree to locate the insertion point
   - During traversal we maintain a stack of pointers to the traversed nodes
2. Perform the insertion and rebalance using copies
   - The reverse traversal uses the saved stack of pointers
   - For each copied node we store the observed children pointers

Example: *insert(key = 1)*

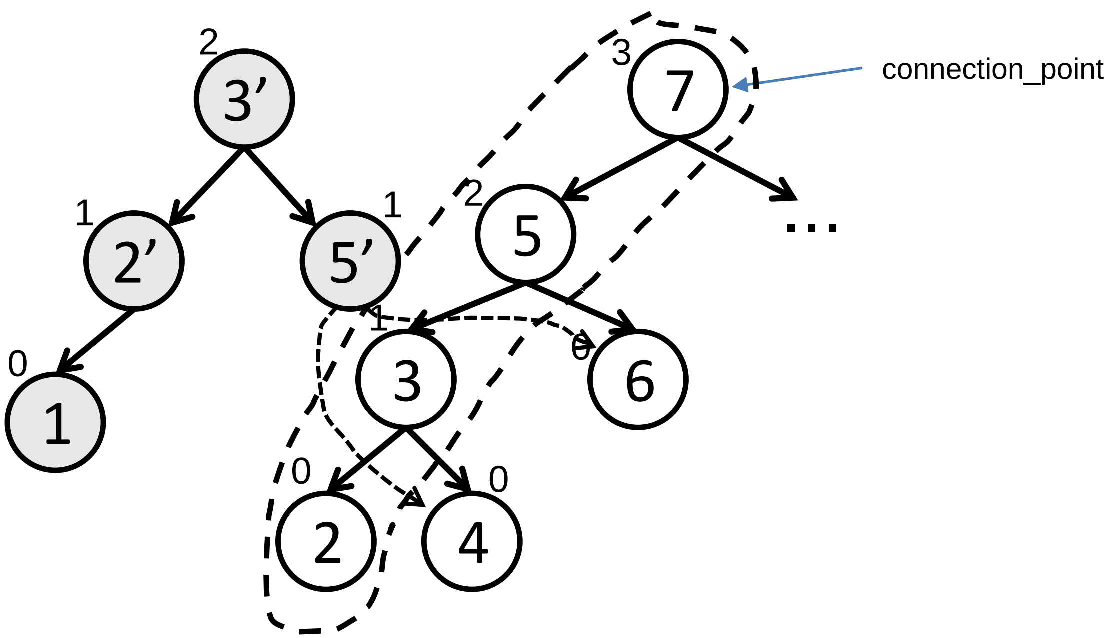


National Technical University of Athens CSLab

RCU-HTM: insert operation

1. Traverse the tree to locate the insertion point
   - During traversal we maintain a stack of pointers to the traversed nodes
2. Perform the insertion and rebalance using copies
   - The reverse traversal uses the saved stack of pointers
   - For each copied node we store the observed children pointers

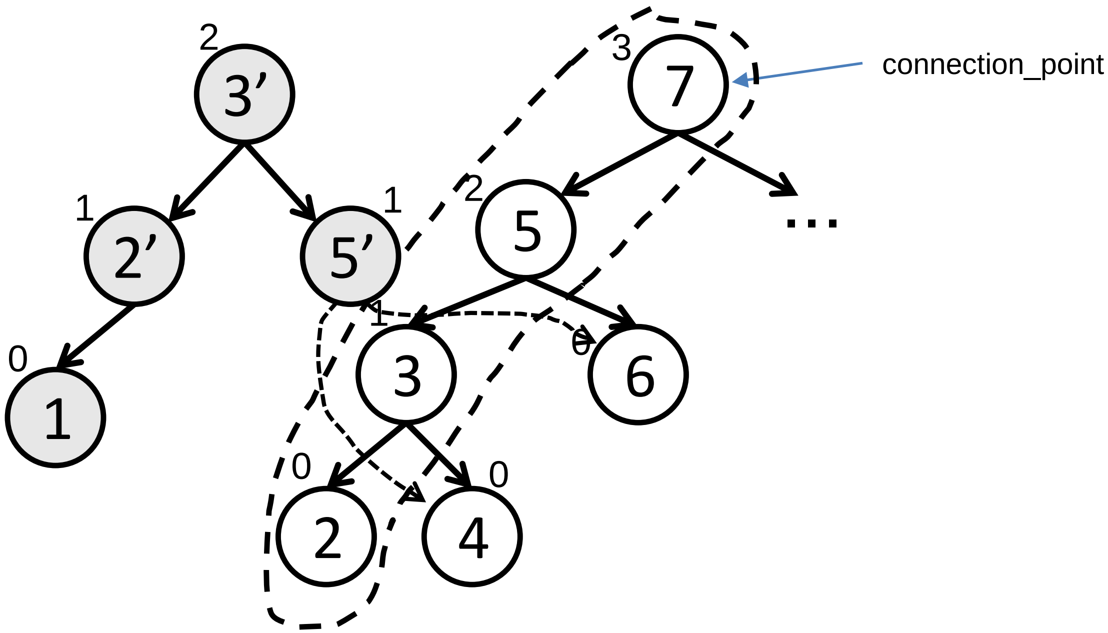
Example: *insert(key = 1)*



RCU-HTM: insert operation

1. Traverse the tree to locate the insertion point
   - During traversal we maintain a stack of pointers to the traversed nodes
2. Perform the insertion and rebalance using copies
   - The reverse traversal uses the saved stack of pointers
   - For each copied node we store the observed children pointers
3. Validate the modified copy
   - For each copied node check that children pointers haven't been modified since we copied the node
   - Also validate the access path followed during traversal

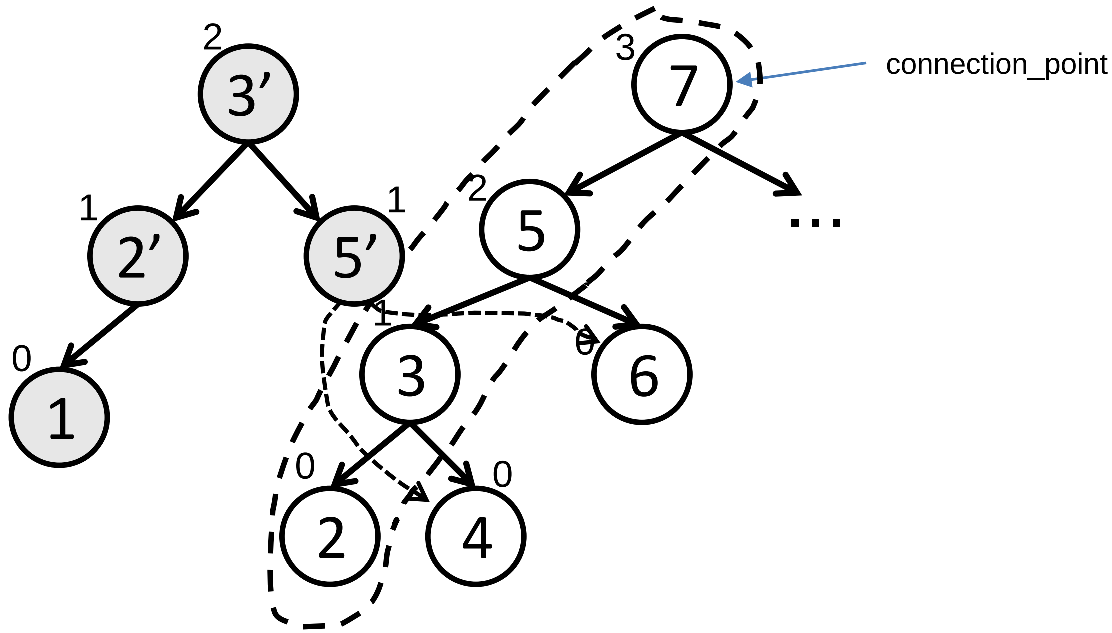
Example: *insert(key = 1)*



RCU-HTM: insert operation

1. Traverse the tree to locate the insertion point
   - During traversal we maintain a stack of pointers to the traversed nodes
2. Perform the insertion and rebalance using copies
   - The reverse traversal uses the saved stack of pointers
   - For each copied node we store the observed children pointers
3. Validate the modified copy
   - For each copied node check that children pointers haven't been modified since we copied the node
   - Also validate the access path followed during traversal
4. Install the copy
   - Change connection_point's child

Example: *insert(key = 1)*

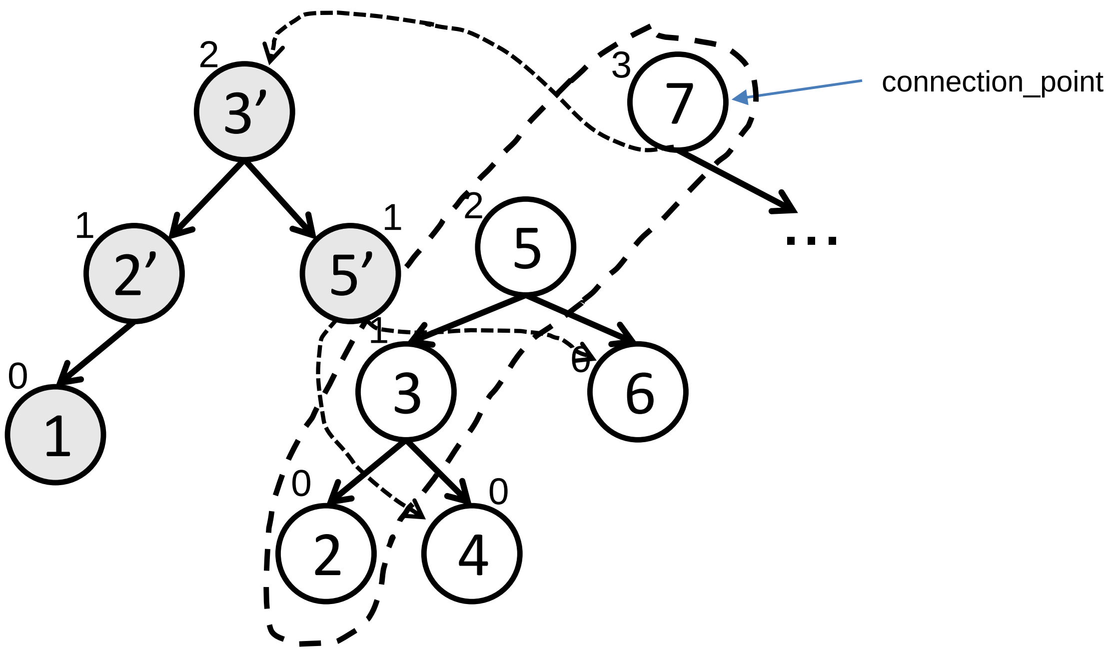


National Technical University of Athens CSLab

RCU-HTM: insert operation

1. Traverse the tree to locate the insertion point
   - During traversal we maintain a stack of pointers to the traversed nodes
2. Perform the insertion and rebalance using copies
   - The reverse traversal uses the saved stack of pointers
   - For each copied node we store the observed children pointers
3. Validate the modified copy
   - For each copied node check that children pointers haven't been modified since we copied the node
   - Also validate the access path followed during traversal
4. Install the copy
   - Change connection_point's child

Example: *insert(key = 1)*



National Technical University of Athens CSLab

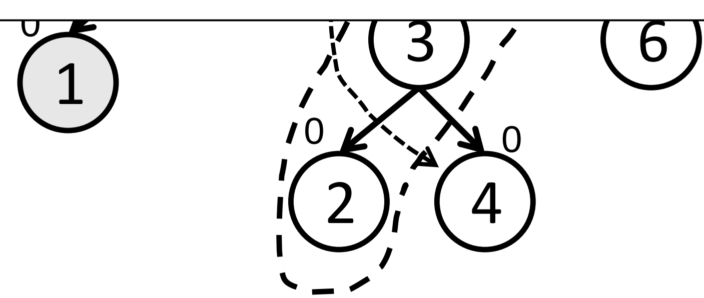
RCU-HTM: insert operation

1. Traverse the tree to locate the insertion point
   - During traversal we maintain a stack of pointers to the traversed nodes
2. Perform the insertion and rebalance using copies
   - The reverse traversal uses the saved stack of pointers
   - For each copied node we store the observed children pointers
3. Validate the modified copy
   - For each copied node check that children pointers haven’t been modified since we copied the node
   - Also validate the access path followed during traversal
4. Install the copy
   - Change connection_point’s child

Steps 3 and 4 performed atomically inside an HTM transaction
If the validation in step 3 fails we abort the transaction and restart the operation

For the non-transactional fallback path we use a lock that allows only a single updater.

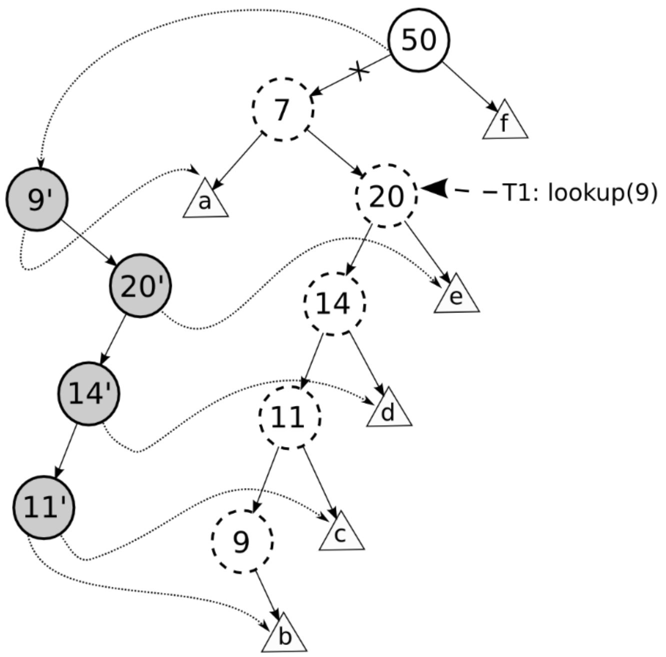
Example: *insert(key = 1)*



National Technical University of Athens CSLab

RCU-HTM: delete operation

- Similar to insert
- One difference:
  - When we delete a node with two children we need to copy the whole path to its successor



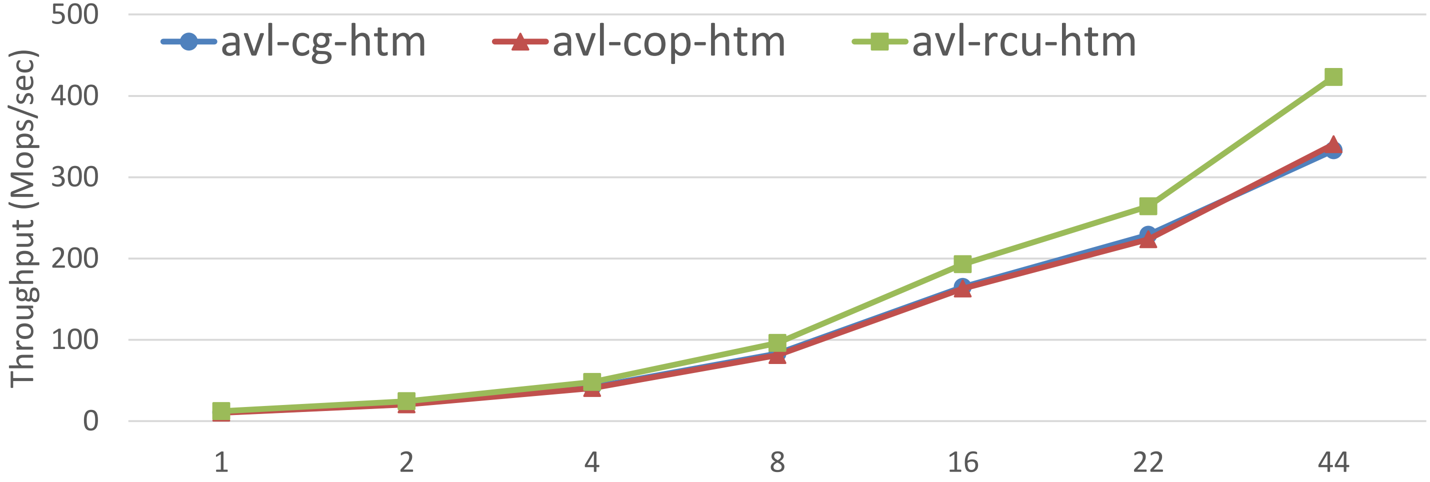
EXPERIMENTAL RESULTS

Experimental Setup

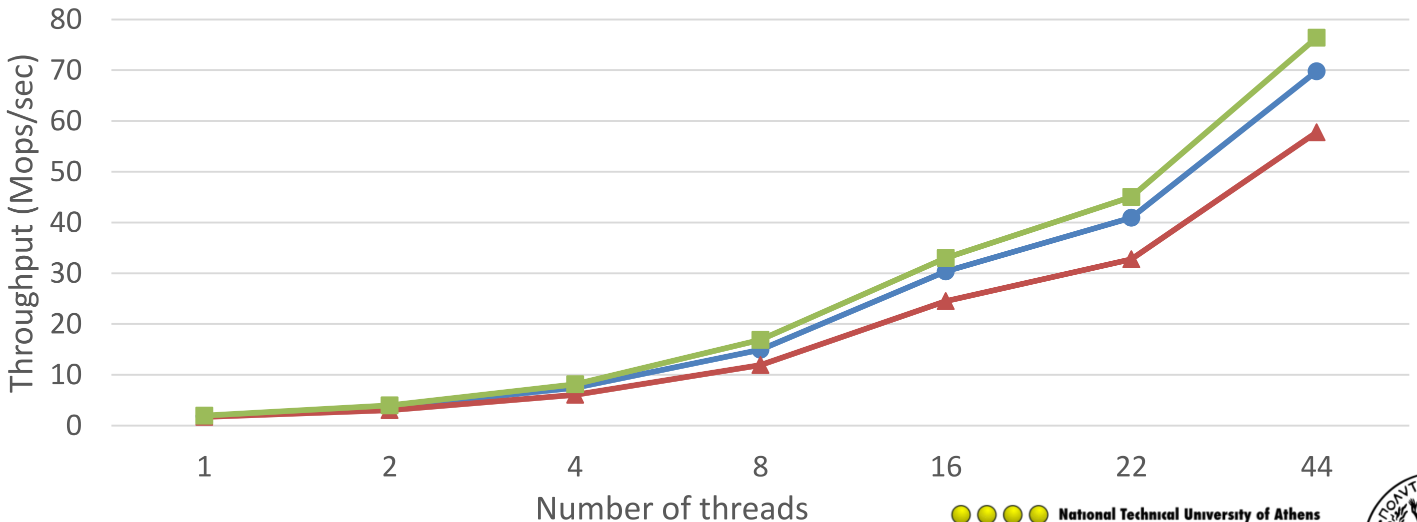
- Intel Broadwell-EP Xeon E5-2699 v4
  - 22 cores / 44 hyperthreads @ 2.2GHz
  - 64 GB of RAM
- GCC 4.9.2, -O3 optimizations enabled
- Scalable memory allocator (jemalloc)
- No memory reclamation
- All threads pinned to hardware threads (hyperthreads enabled only at 44-threaded executions)
- Experiments:
  - Threads run for 2 seconds, executing randomly chosen operations (lookups/inserts/deletes)
  - 3 Workloads: 100%, 80% and 20% lookups, and the rest equally divide between insertions and deletions
  - 3 tree sizes: 2K keys, 20K keys and 2M keys

Comparison with HTM-based approaches

2K keys
100% lookups



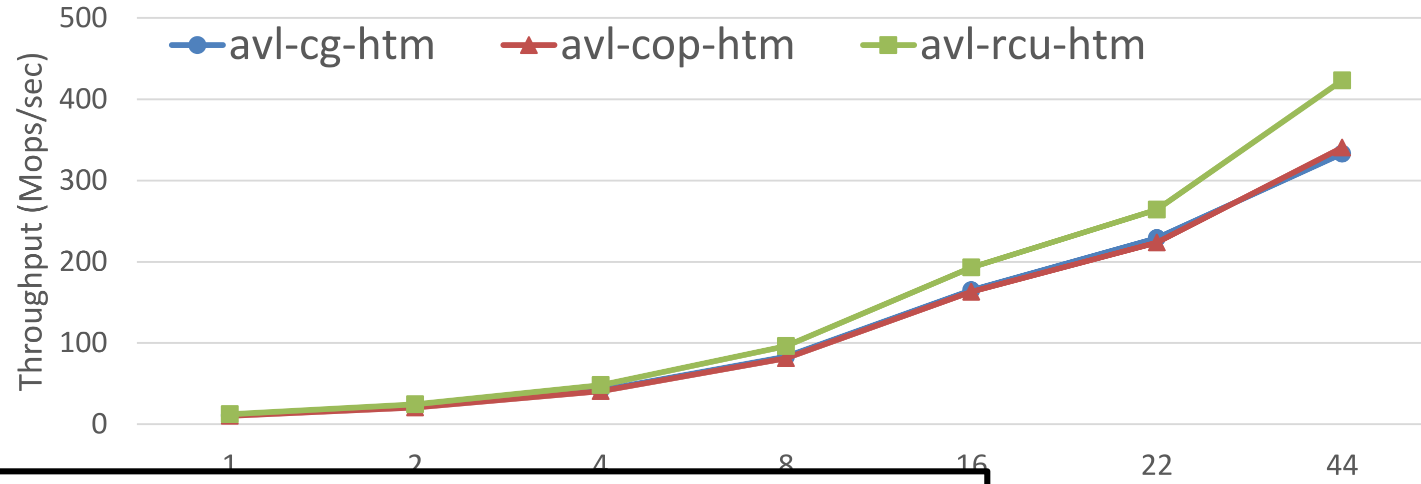
2M keys
100% lookups



National Technical University of Athens
CSLab

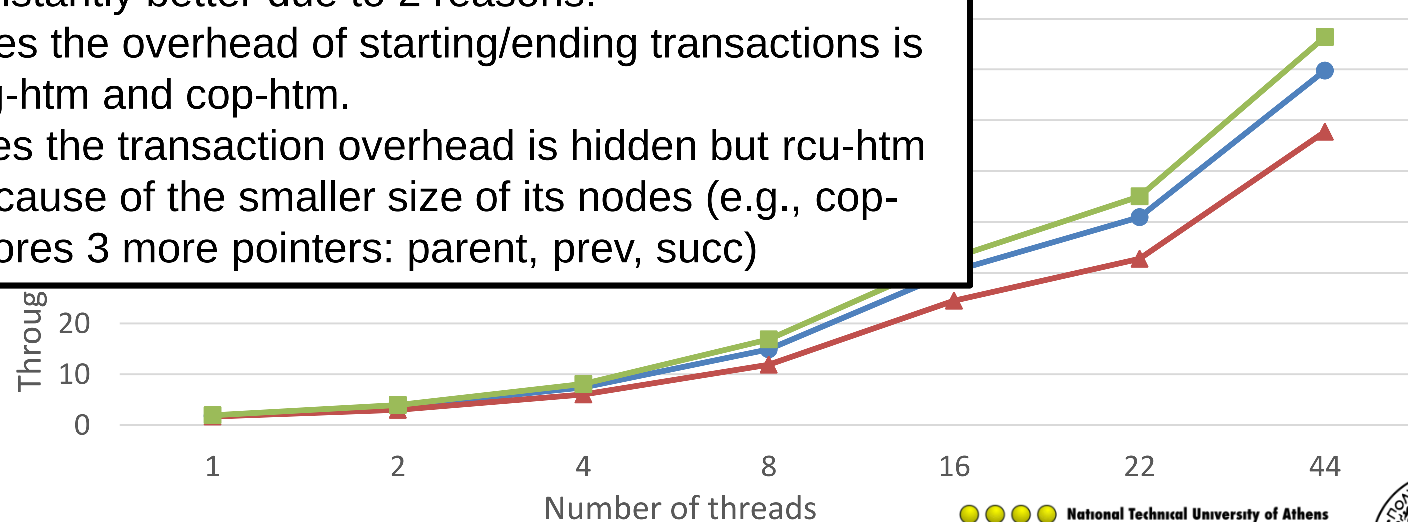
Comparison with HTM-based approaches

2K keys
100% lookups



Read-only workloads

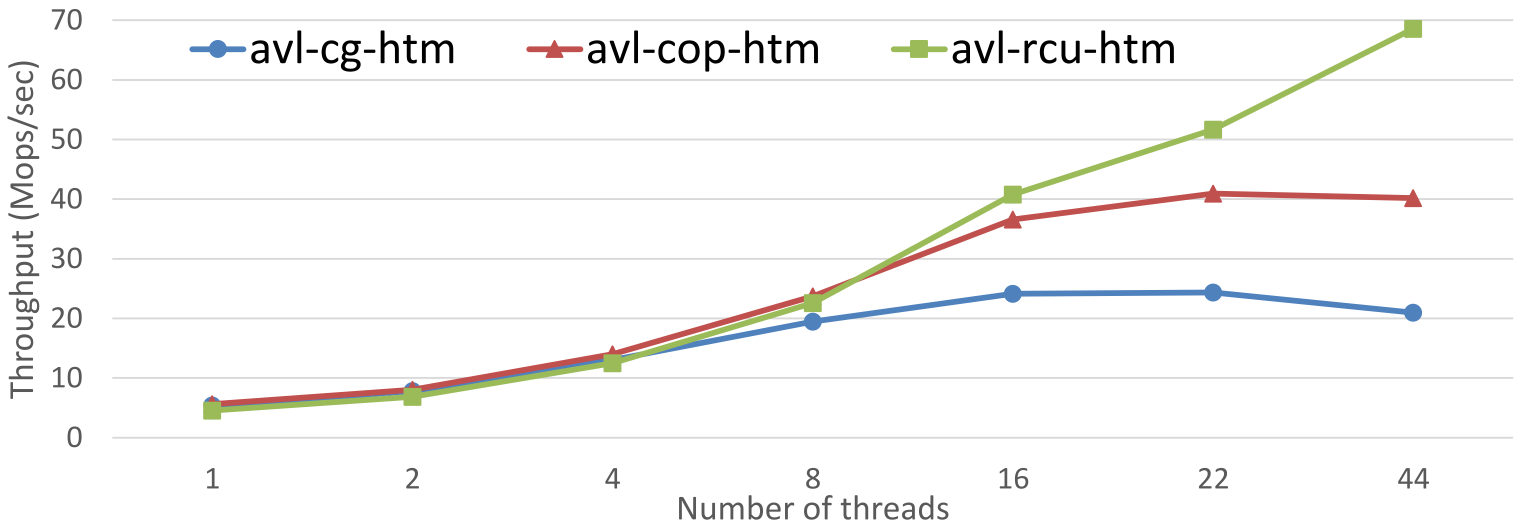
- No conflict/capacity aborts → all HTM-based trees scale
- RCU-HTM is constantly better due to 2 reasons:
  - In small trees the overhead of starting/ending transactions is visible in cg-htm and cop-htm.
  - In large trees the transaction overhead is hidden but rcu-htm is faster because of the smaller size of its nodes (e.g., cop-htm also stores 3 more pointers: parent, prev, succ)



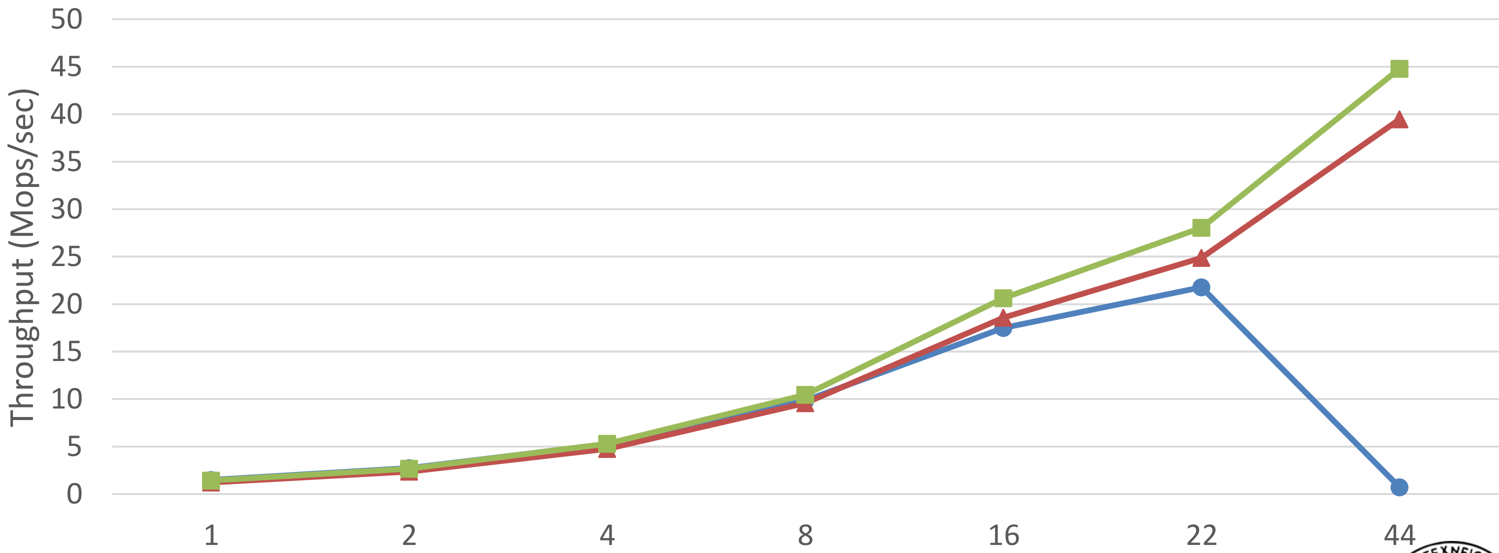
National Technical University of Athens
CSLab

Comparison with HTM-based approaches

2K keys
20% lookups



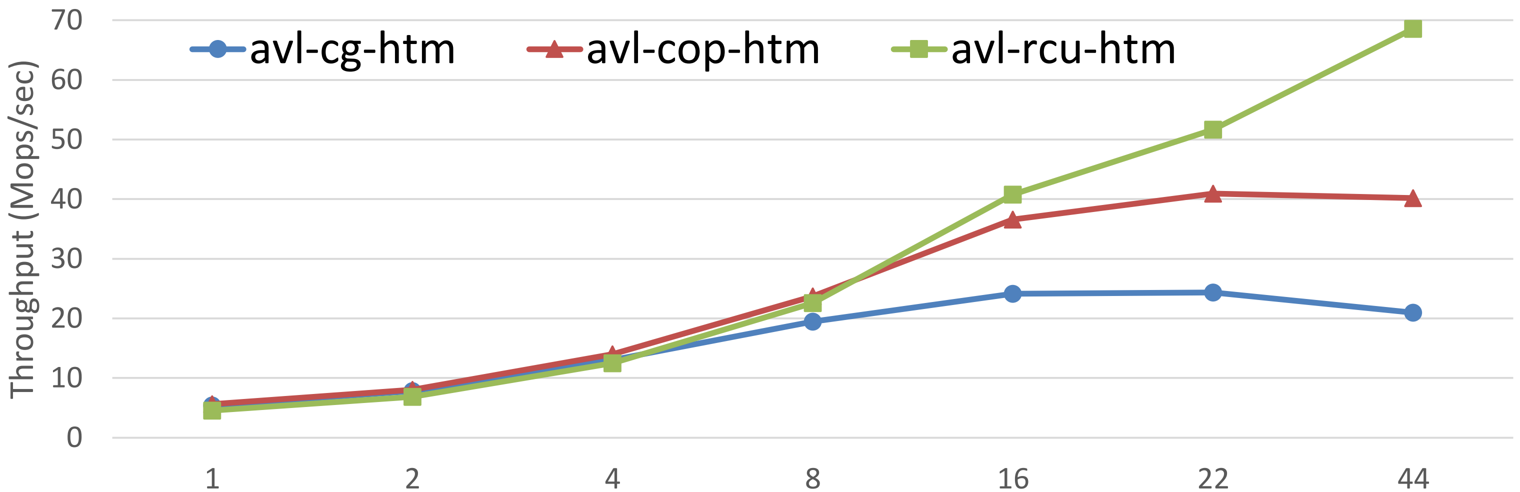
2M keys
20% lookups



National Technical University of Athens CSLab

Comparison with HTM-based approaches

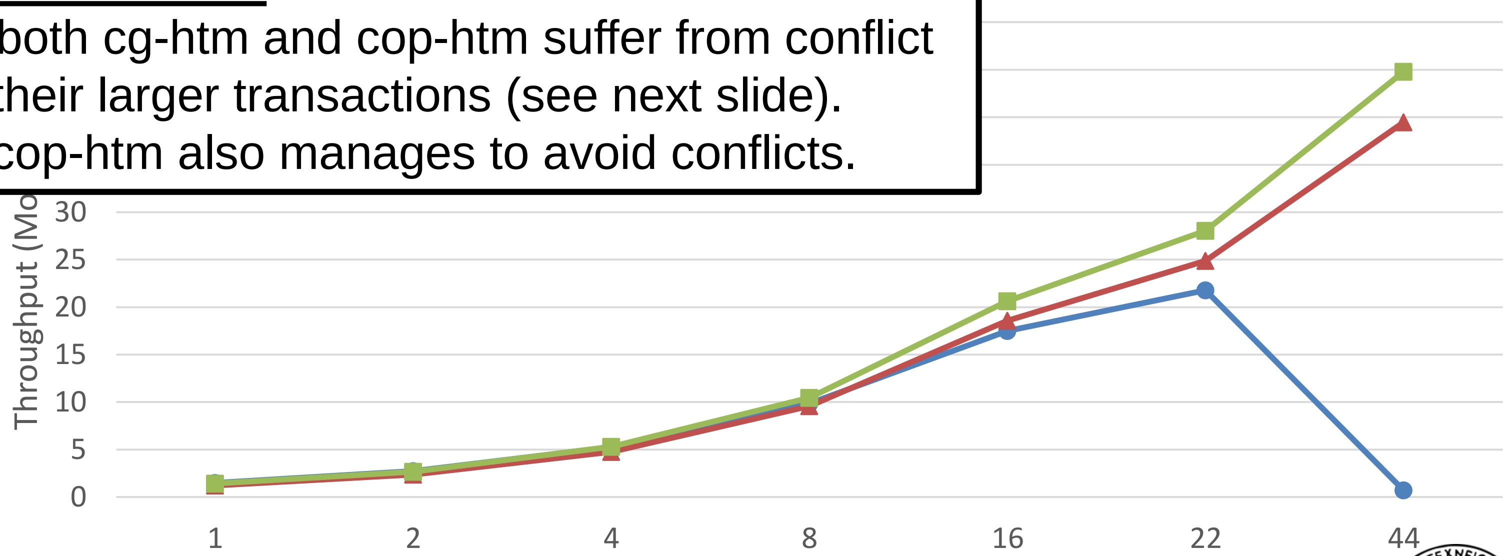
2K keys
20% lookups



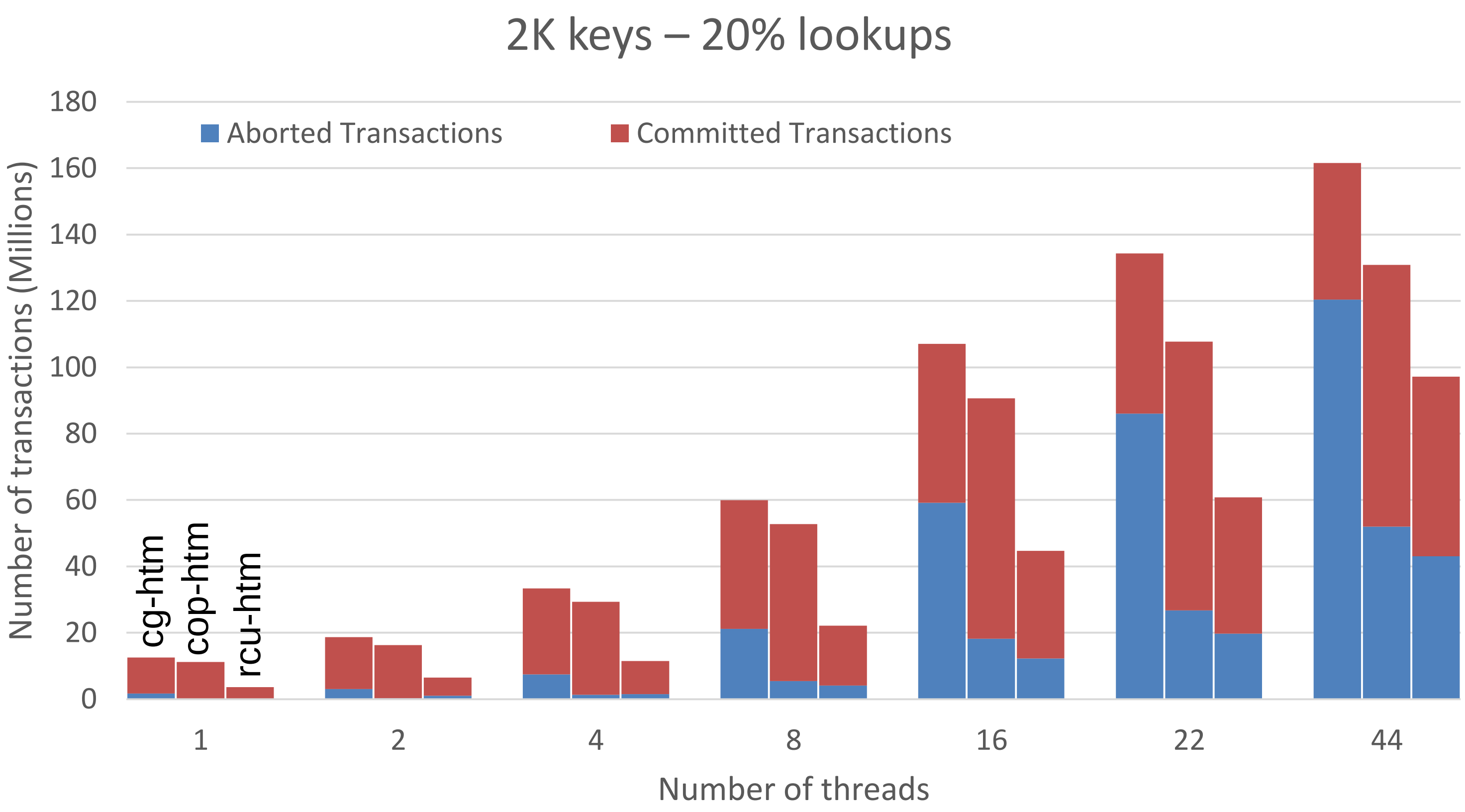
Write-dominated workloads

- In small trees both cg-htm and cop-htm suffer from conflict aborts due to their larger transactions (see next slide).
- In large trees cop-htm also manages to avoid conflicts.

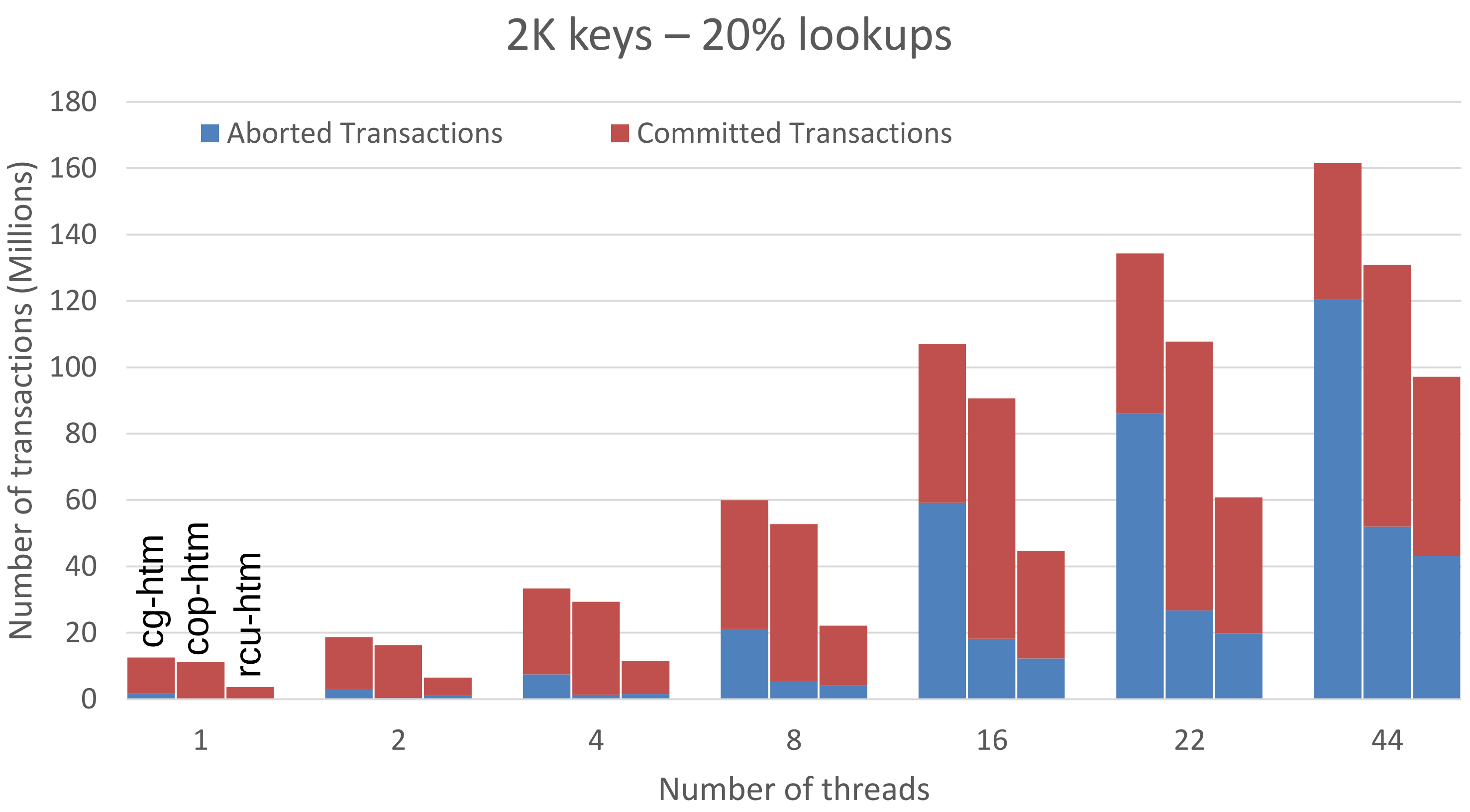
2M keys
20% lookups



Comparison with HTM-based approaches



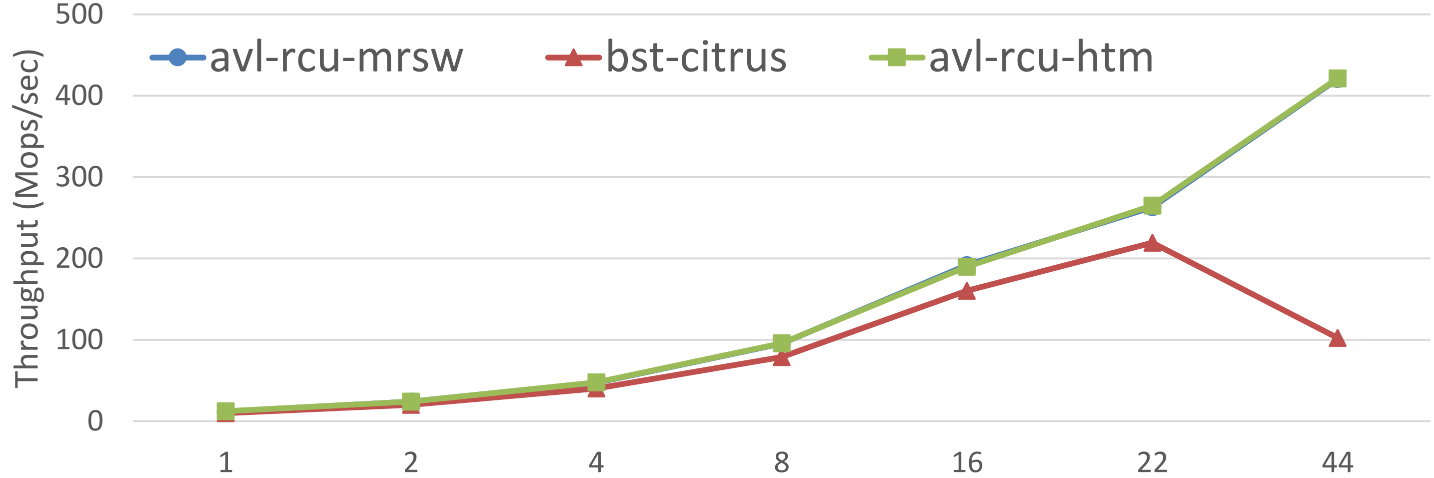
Comparison with HTM-based approaches



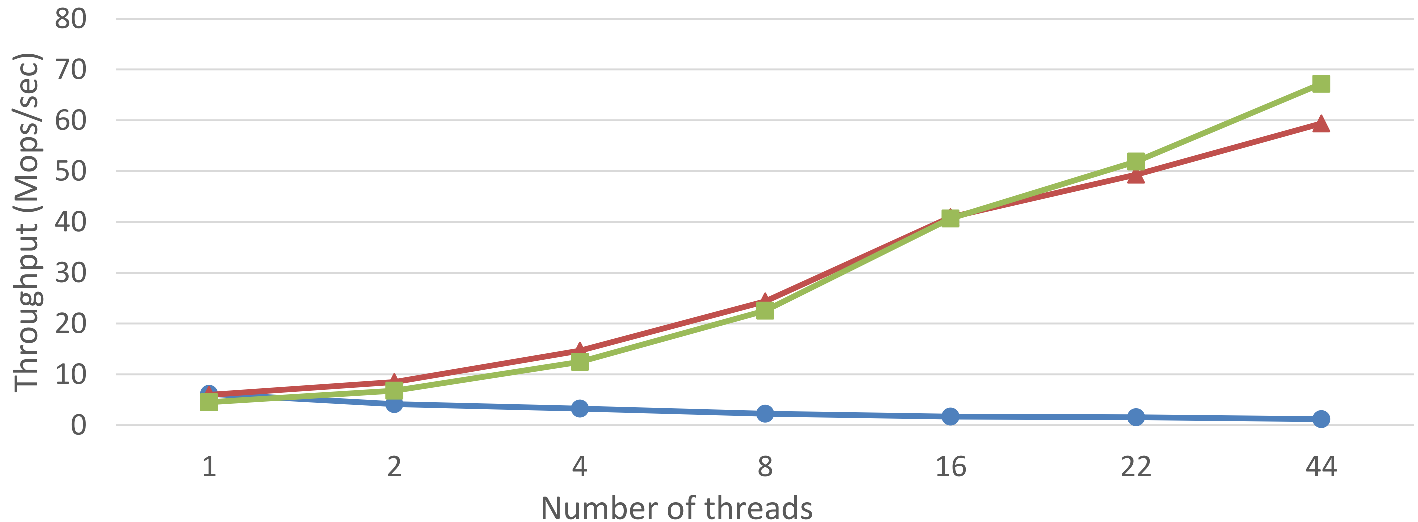
RCU-HTM executes much less transactions and suffers less aborts.

Comparison with RCU-based approaches

2K keys
100% lookups



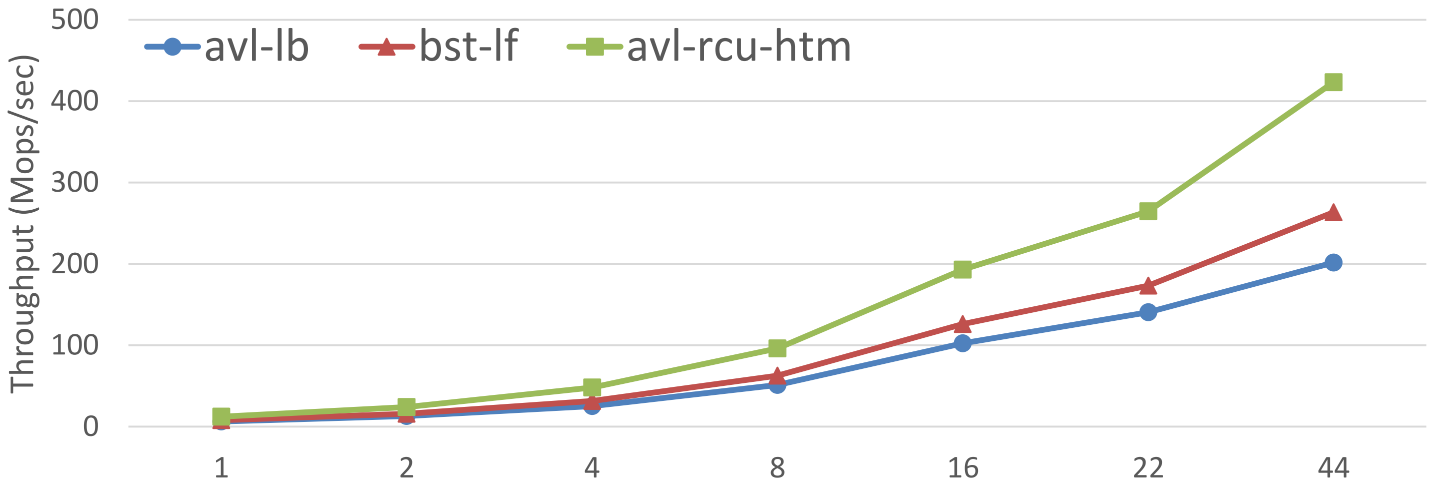
2K keys
20% lookups



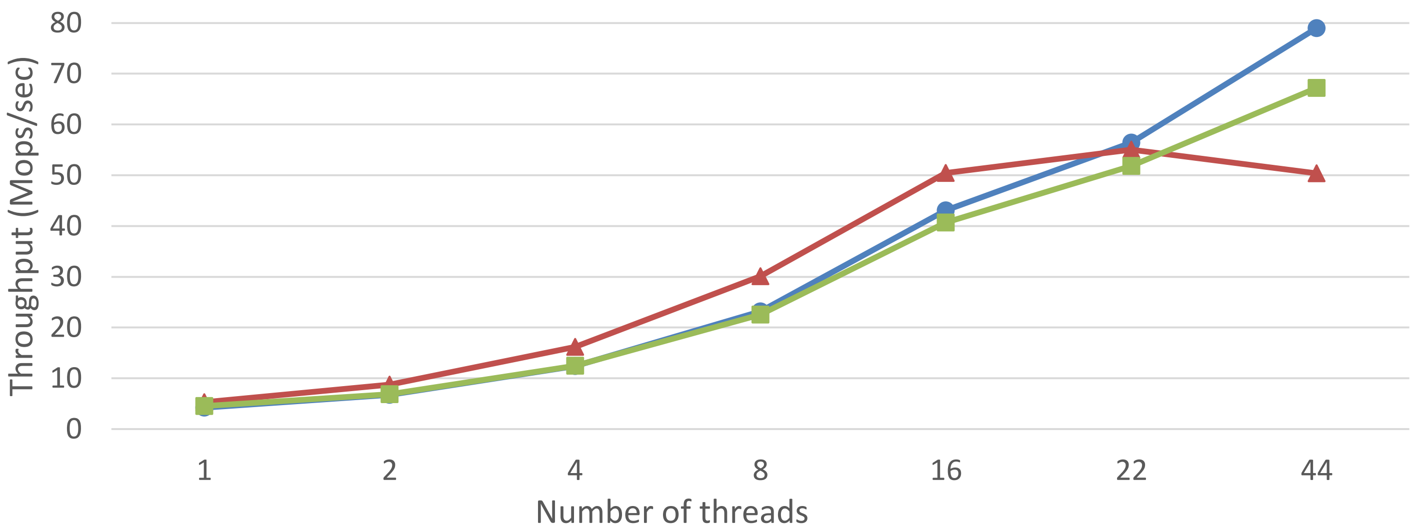
avl-rcu-mrsw: writers synchronized using a single lock
bst-citrus: unbalanced BST, RCU for readers, fine-grain locks for writers [Arbel PODC'14]

Comparison with state-of-the-art

2K keys
100% lookups



2K keys
20% lookups

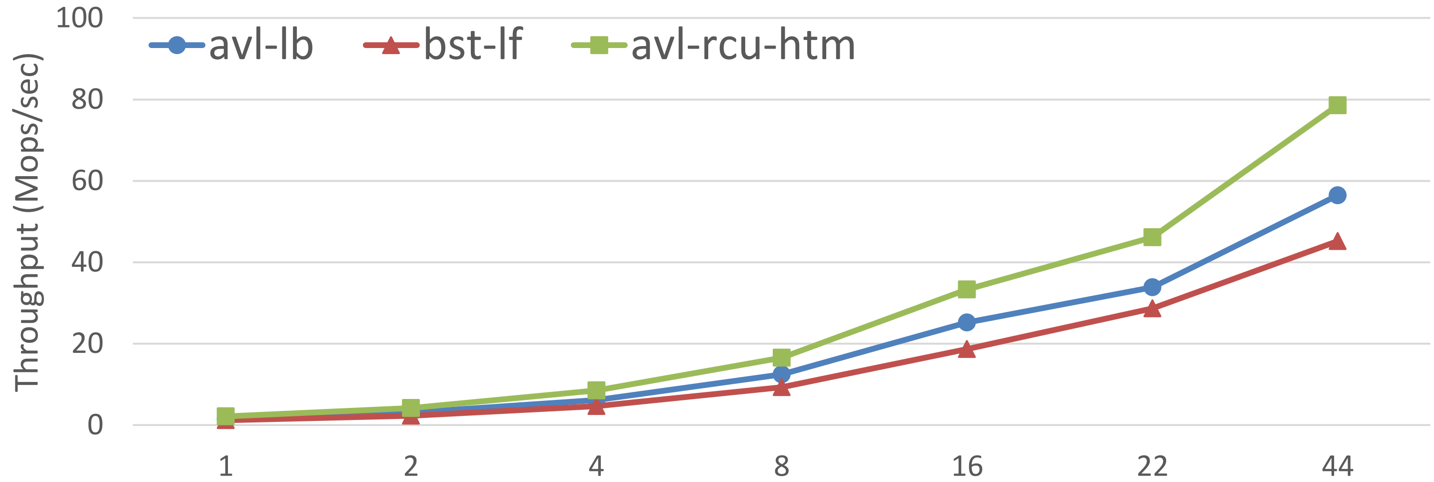


avl-lb: relaxed balance lock-based AVL tree [Bronson PPOPP'10]
bst-lf: unbalanced lock-free (CAS-based) tree [Natarajan PPoPP'14]

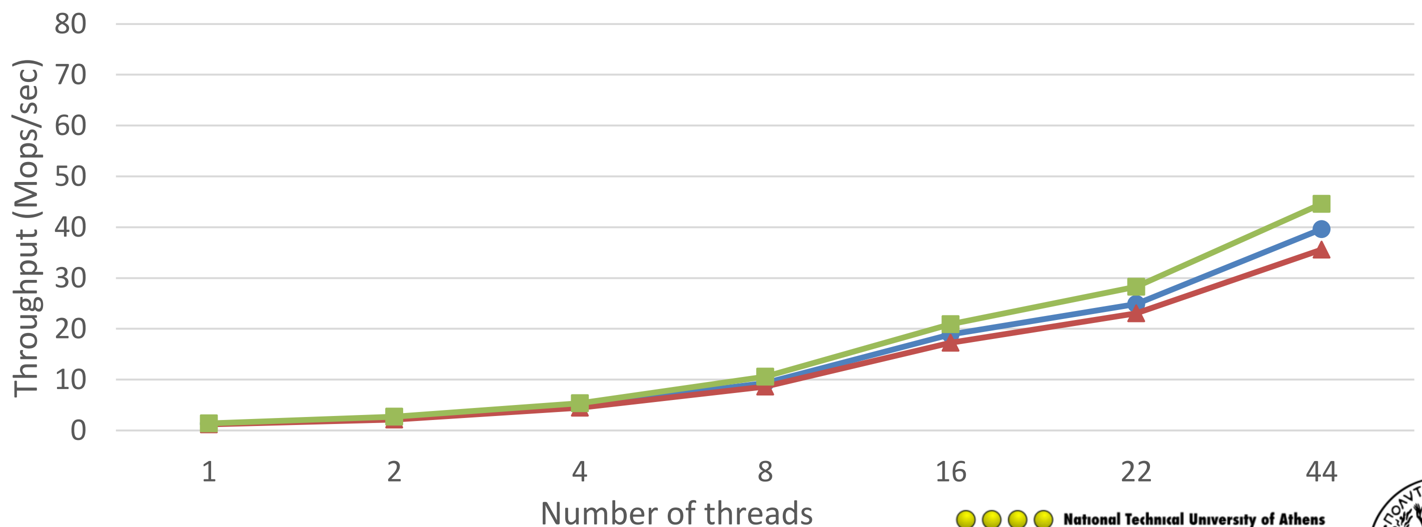
National Technical University of Athens
CSLab

Comparison with state-of-the-art

2M keys
100% lookups



2M keys
20% lookups



National Technical University of Athens CSLab

CONCLUSIONS & FUTURE WORK

Conclusions & Future Work

- RCU-HTM combines RCU with HTM and provides concurrent BSTs that are:
  - Internal
  - Strictly balanced
  - Efficient both for readers and updaters
- Future work
  - Memory reclamation
  - Formal proof of correctness (linearizability)
  - More BSTs (e.g., B+-trees, Splay trees, etc.)

THANK YOU!

QUESTIONS?

ACKNOWLEDGMENT

Intel Corporation for kindly providing the Broadwell-EP server on which we executed our experiments.