Massively Concurrent Red-Black Trees with Hardware Transactional Memory

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

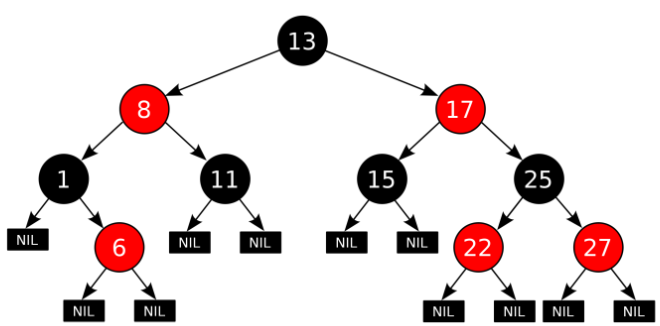
PDP 2016

Motivation

- Hardware Transactional Memory (HTM) has become mainstream
  - IBM Power8, Blue Gene/Q, zEC12
  - Intel Haswell, Broadwell …
- Available on highly multi-threaded machines (10s to 100s of hardware threads available)
- We need to evaluate it on real-life applications

Red-Black trees

- Classic data structure
- Widely used for *dictionary* implementations
- Challenging to devise efficient concurrent implementations using locks or atomic primitives
- Their properties favor the usage of HTM



National Technical University of Athens
CSLab

Our Contributions

- We have implemented concurrent red-black trees with HTM
- We have evaluated then with high number of threads
  - **Intel Haswell-EP**: 56 hardware threads
  - **IBM Power8**: 160 hardware threads
- We have found out that:
  - Programming with HTM can be simple …
    - … but to get performance one needs to put some extra effort
  - Different challenges faced on each system
    - Depending on each HTM’s resources
    - Different optimizations applied

Transactional Memory

- Programming model alternative to locks
  - Aims to simplify programming (complexity is moved to the TM system)
  - Programming effort similar to coarse-grained locking …
  - … with performance similar or even better than fine-grained locking
  - More robust than locks
- Programming with TM
  - Programmer annotates regions of code to be executed atomically (***transactions***)
  - TM system guarantees **atomic** execution of transactions
  - atomic: either all writes become visible (**commit**) or none of them (**abort**)
  - TM system keeps track of read- and write- sets for each transaction and if a **conflict** is detected some transaction is aborted

Lock-based code

```
... other code ...
lock(global_lock);
... critical section ...
unlock(global_lock);
... other code ...
```

→

TM code

```
... other code ...
tx_begin();
... critical section ...
tx_end();
... other code ...
```

Transactional Memory

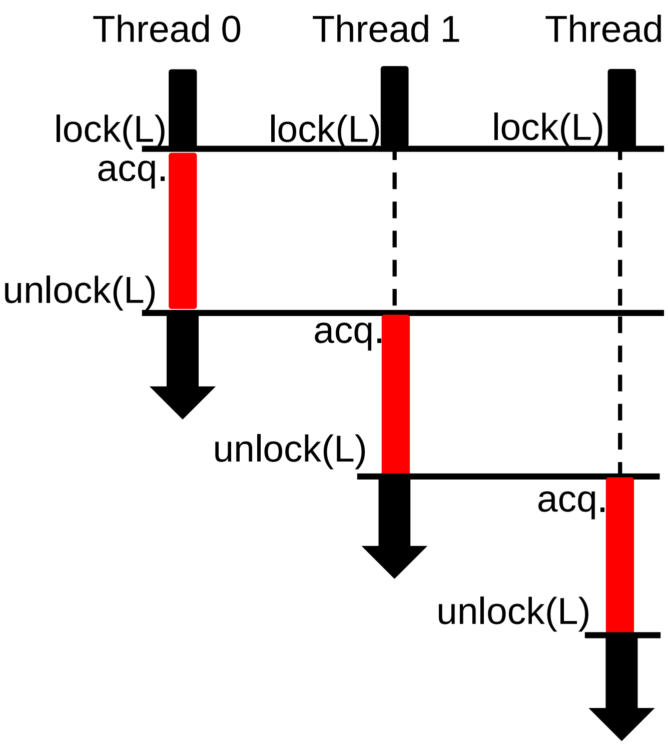
Lock-based code

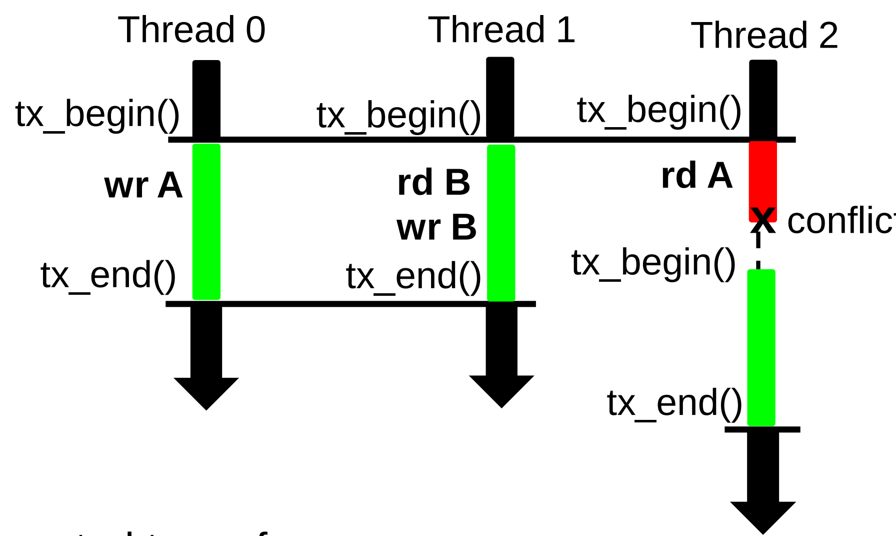
```
... other code ...
lock(global_lock);
... critical section ...
unlock(global_lock);
... other code ...
```

→

TM code

```
... other code ...
tx_begin();
... critical section ...
tx_end();
... other code ...
```





TM is expected to perform

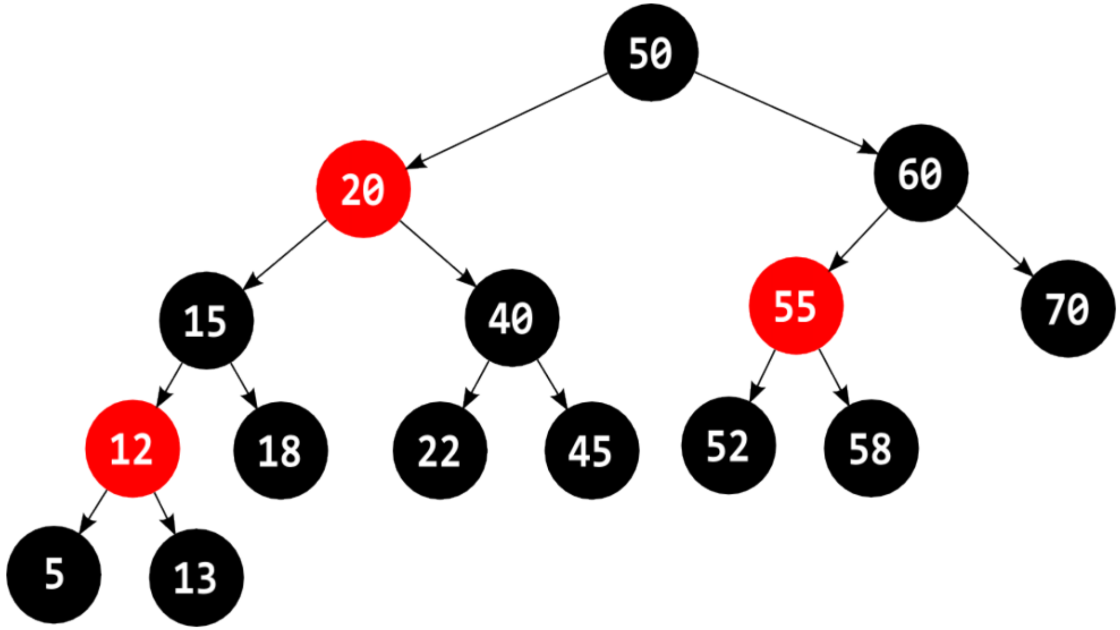
- better than locks when no coflicts are present
- worse than locks when conflicts arise

National Technical University of Athens CSLab

Hardware Transactional Memory

- Hardware implementation of TM (HTM)
  - ISA extensions to support the transactional model
    - `xbegin`: begins a transaction
    - `xend`: ends a transaction
    - `xabort`: explicitly aborts a transaction with some abort code
  - Currently supported on various processors
    - IBM Power8, Blue Gene/Q, zEC12
    - Intel Haswell, Broadwell …
- On the plus side
  - eliminates the overheads of Software TM
- But
  - Hardware limitations
    - read-/write- sets are limited by the hardware buffers
  - Best-effort implementations (a transaction may never commit)
    - aborts due to read-/write- set buffers overflow, interrupts, cache line eviction …
    - programmer needs to specify non-transactional fallback code

Red-Black Trees in a nutshell

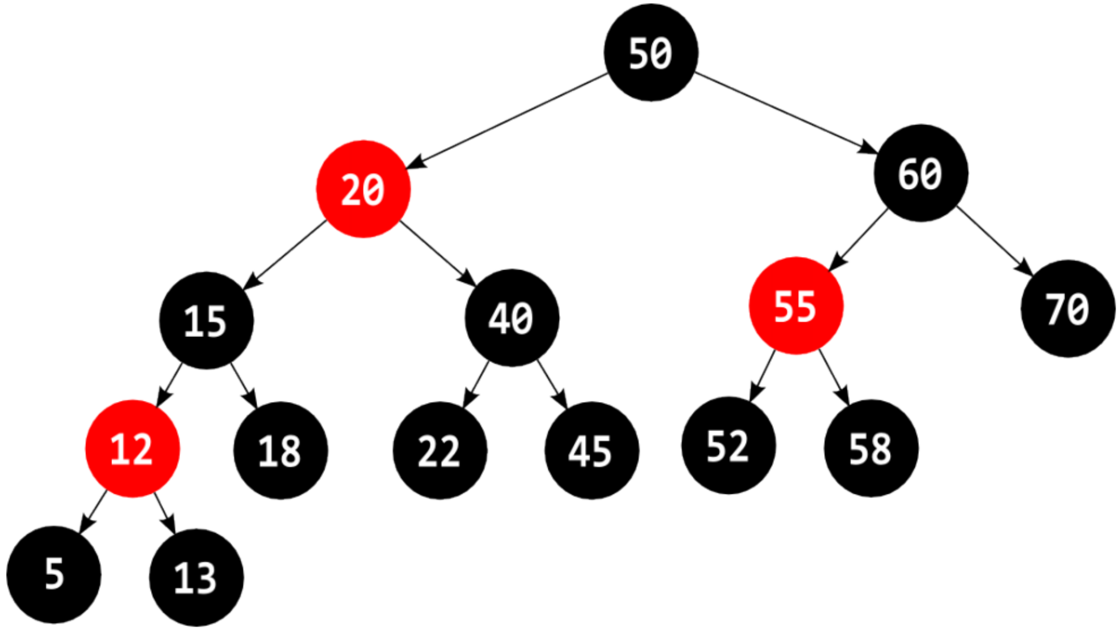


Definition

1. A node is either **red** or **black**
2. The root is always black
3. All leaves are black
4. Every red node must have two black children
5. Every path from a given node to any of its descendant leaves contains the same number of black nodes.

The above properties guarantee that the tree is almost balanced

Red-Black Trees in a nutshell

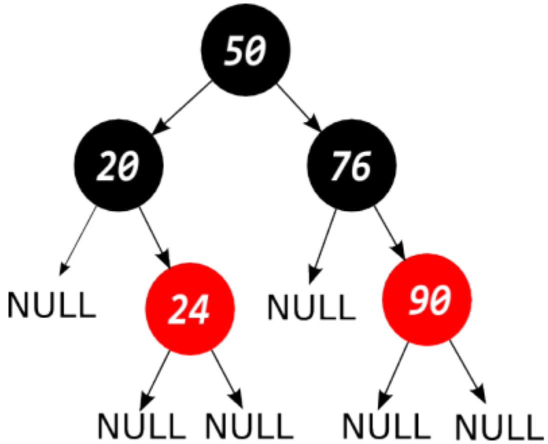


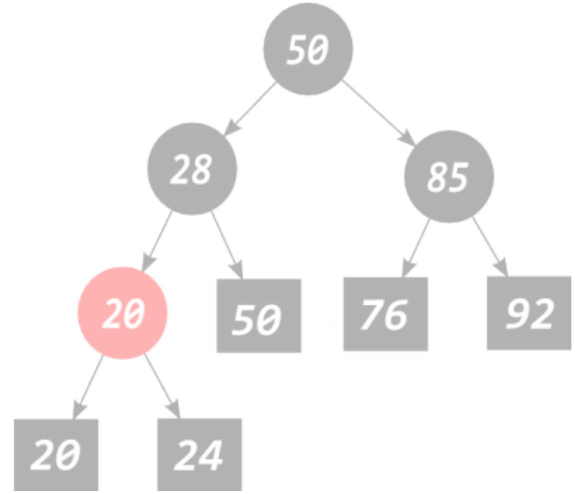
Applications

Dictionary ADT
- Collection of (key, value) pairs

Supports three operations:
- Lookup(key)
- Insert(key, value)
- Delete(key)

Internal vs. External RBTs





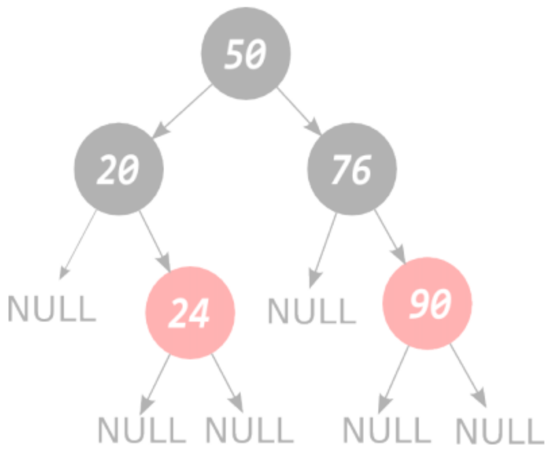
Internal

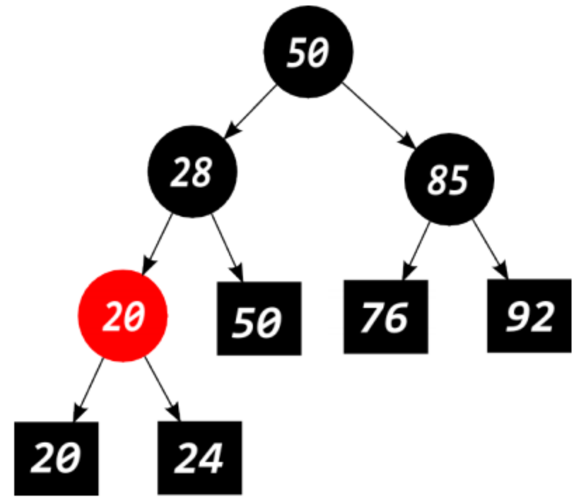
- Both keys and values are stored in every node

External

- Values are only stored in the leaves
- Internal nodes are only used for routing to the appropriate leaf
- \- Occupy more memory
- \+ Simplify delete operation
- All our implementations are external trees

Internal vs. External RBTs





Internal

- Both keys and values are stored in every node

External

- Values are only stored in the leaves
- Internal nodes are only used for routing to the appropriate leaf

\- Occupy more memory

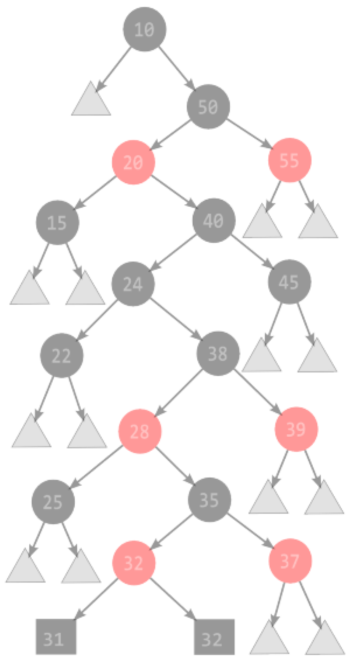
\+ Simplify delete operation

- All our implementations are external trees

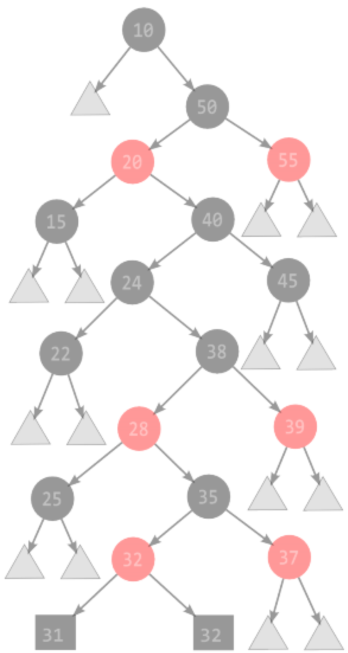
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]

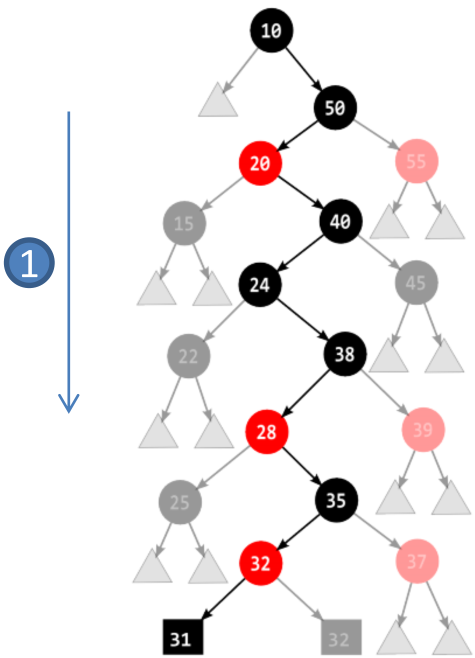


[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

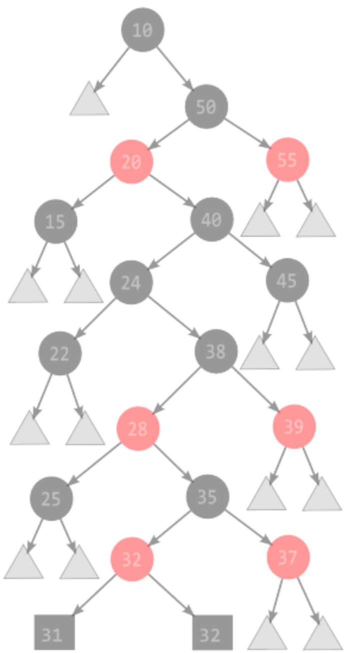
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



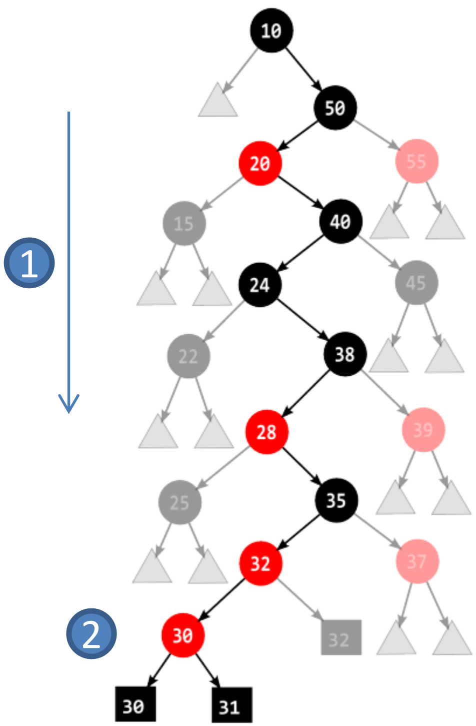
1 Root->leaf (top-down) traversal (read-only phase)

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

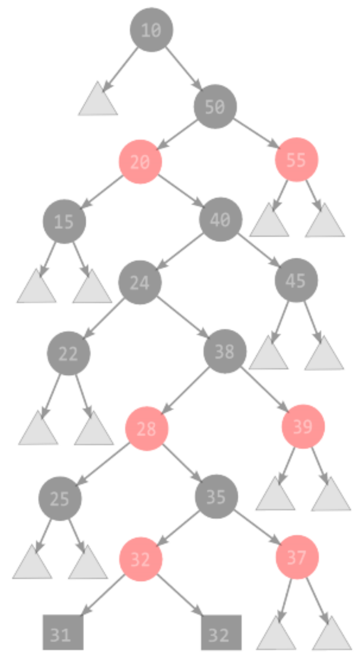
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



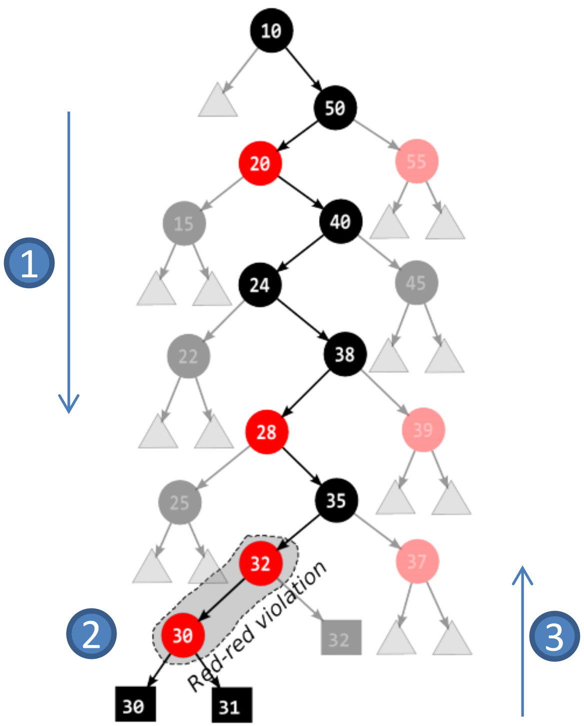
1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

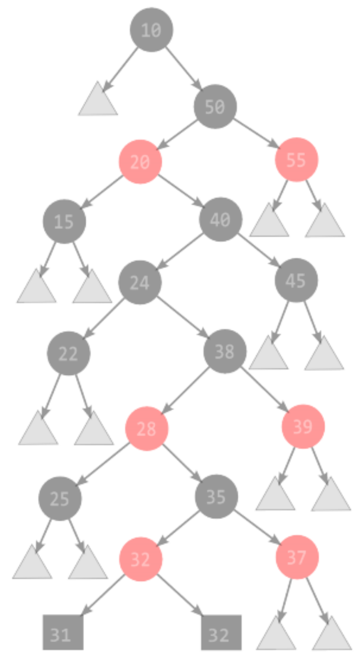
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



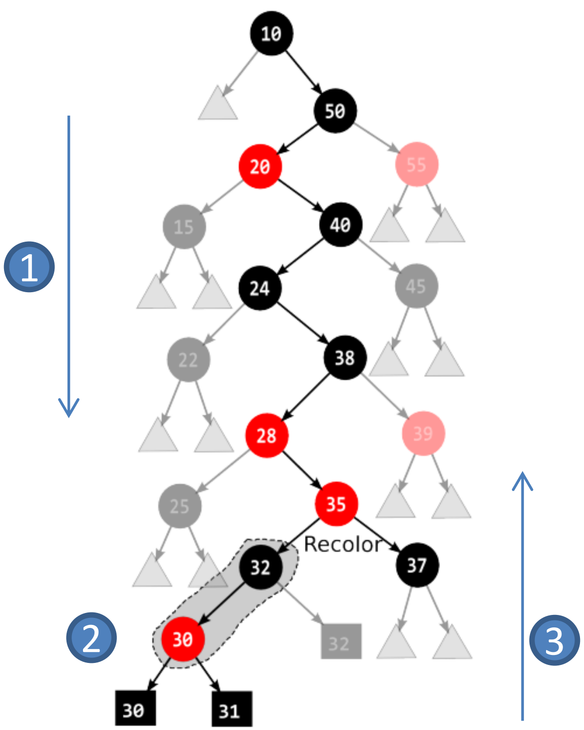
1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

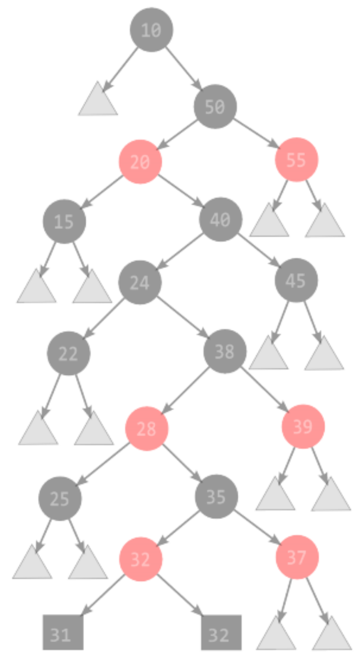
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



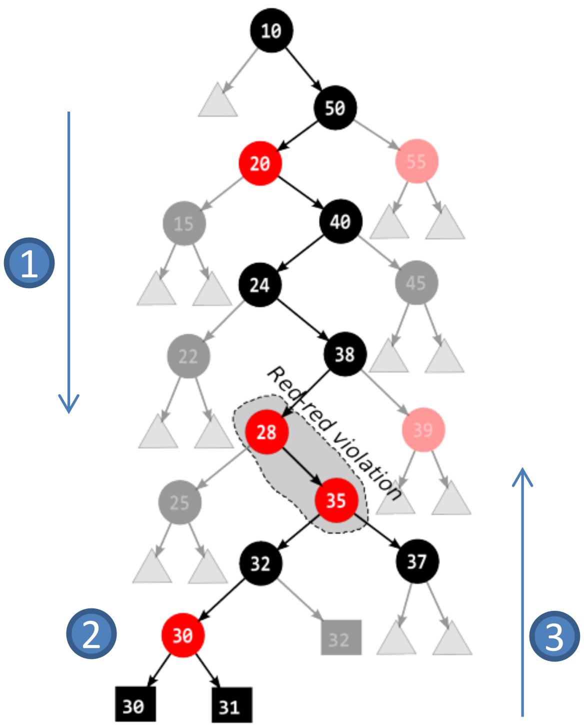
1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

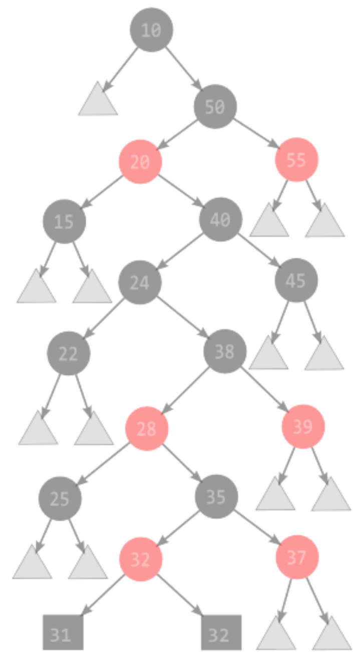
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



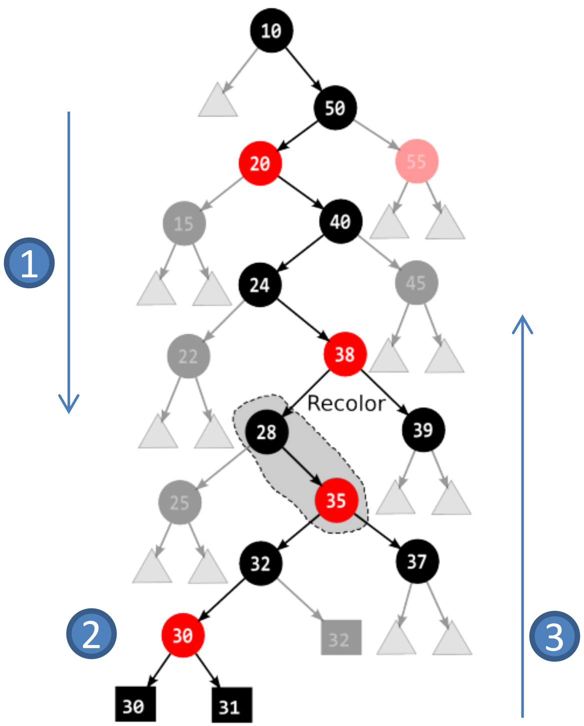
1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

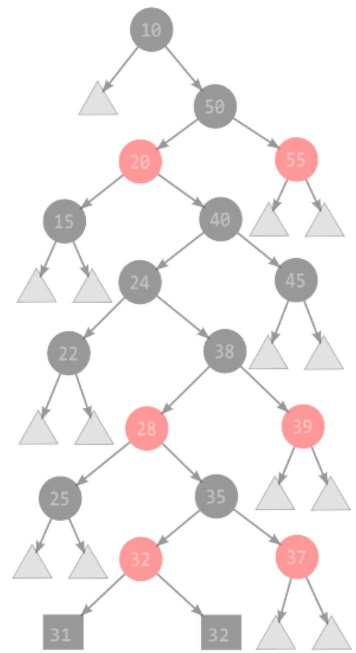
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



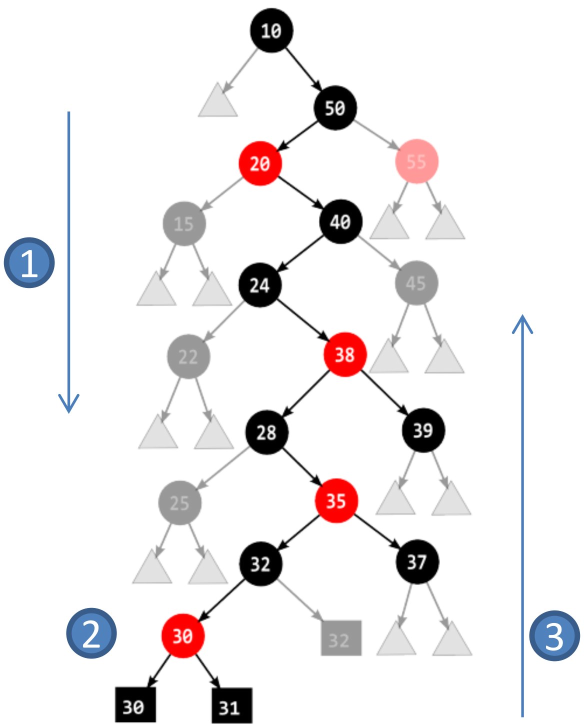
1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

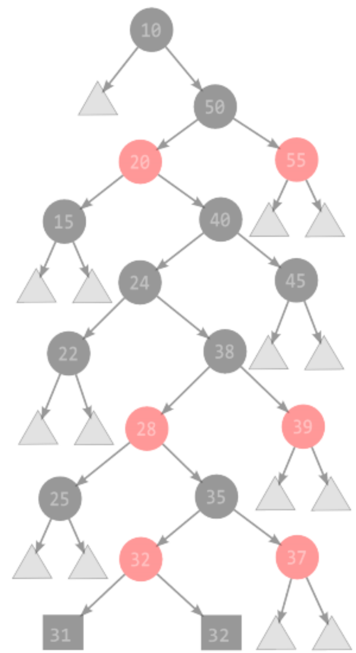
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



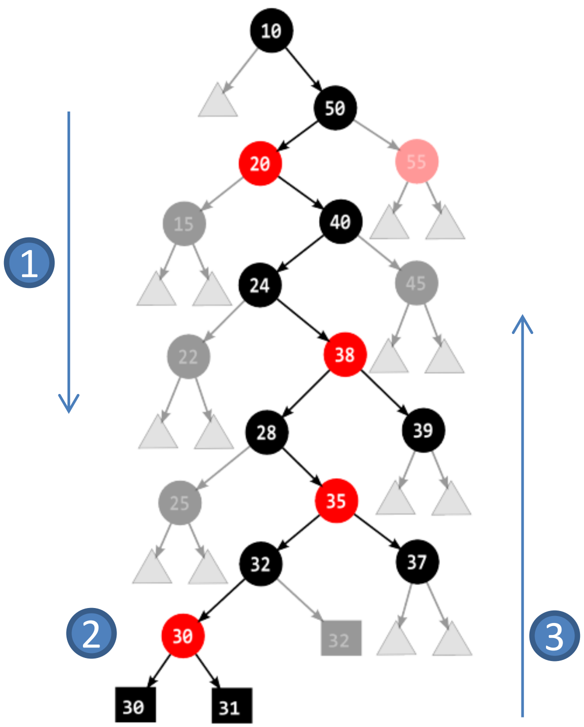
1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

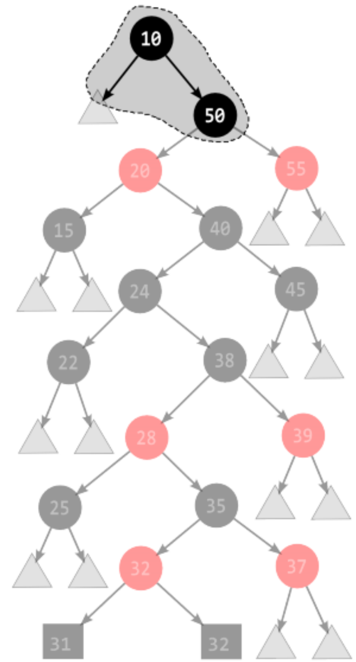
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

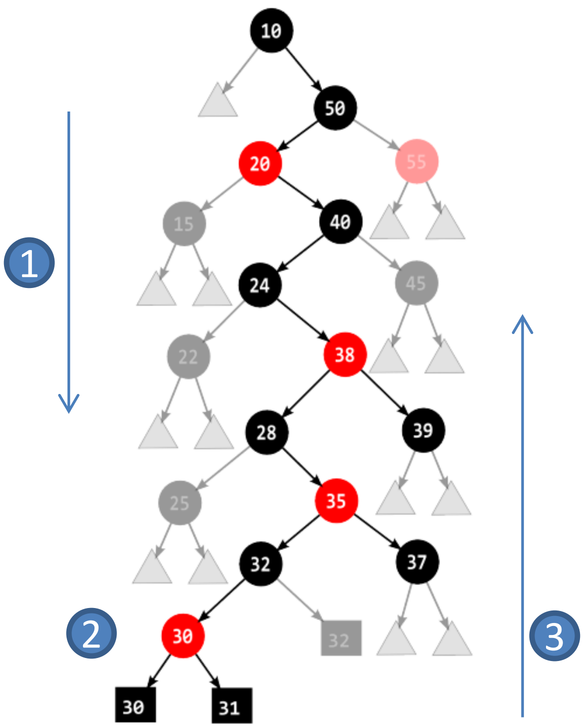
1. Root -> Leaf (top-down) traversal

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

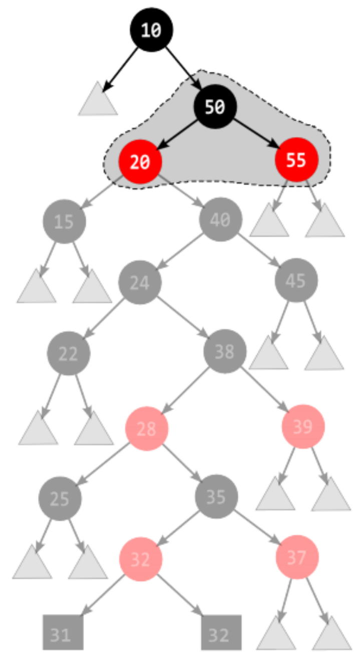
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



Bottom-Up:

1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

Top-Down:

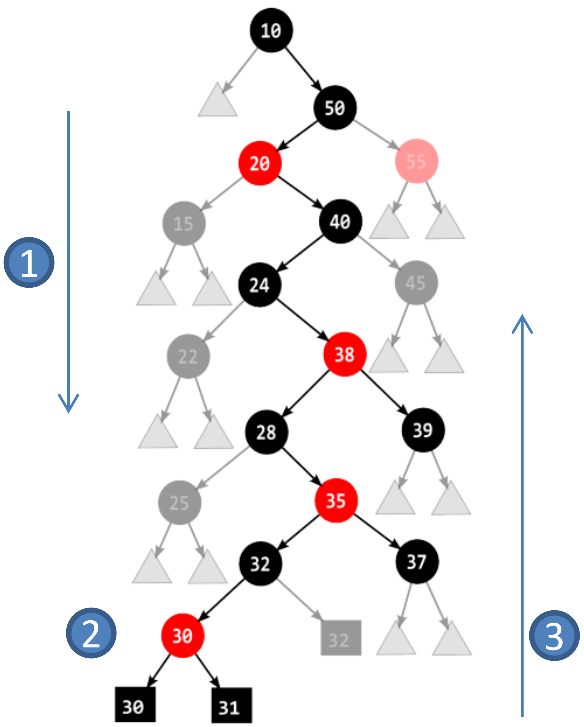
1. Root -> Leaf (top-down) traversal

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

Bottom-Up vs. Top-Down RBTs

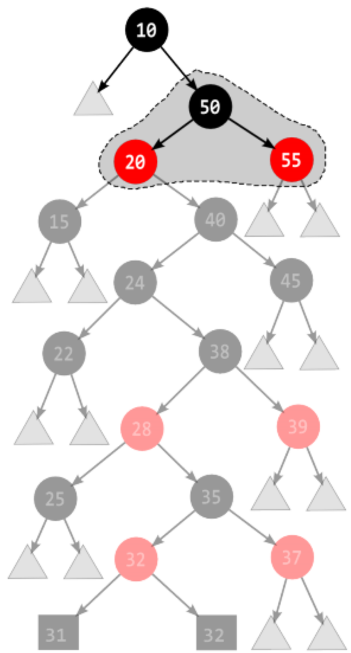
Example: `insert(30)`

Bottom-Up[1]



1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

Top-Down[2]



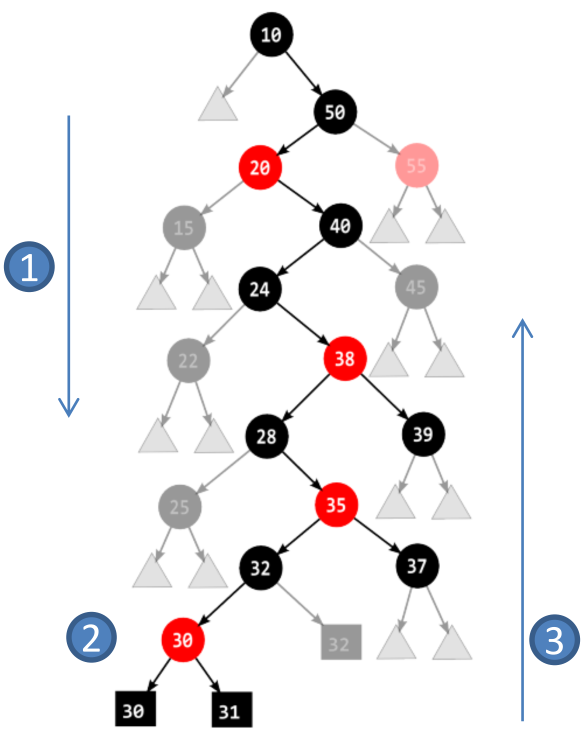
1. Root -> Leaf (top-down) traversal
2. Perform necessary modifications on the way to the leaf

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

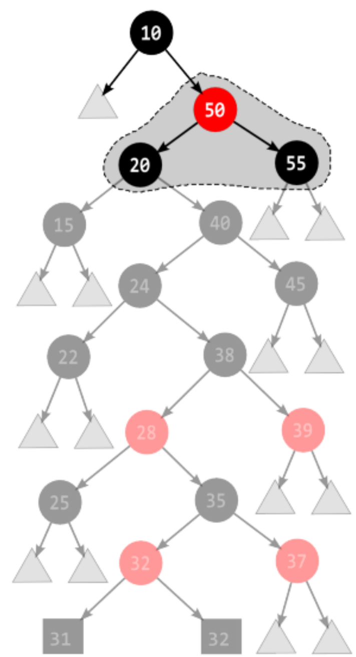
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

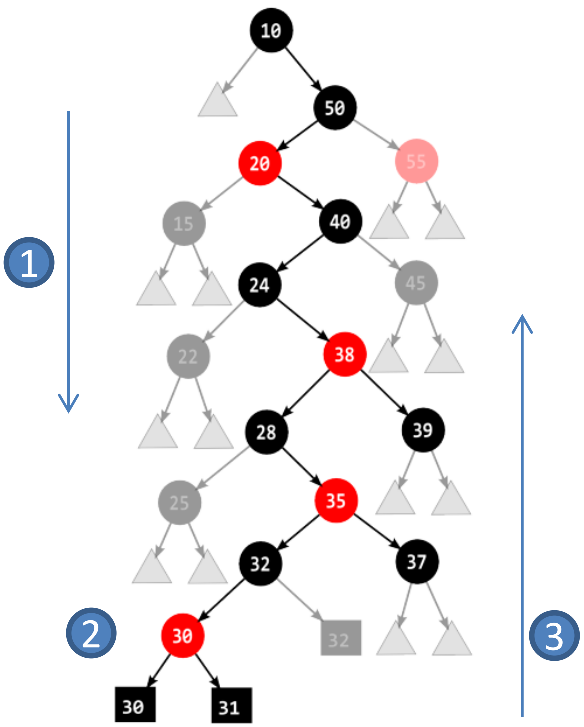
1. Root -> Leaf (top-down) traversal
2. Perform necessary modifications on the way to the leaf

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

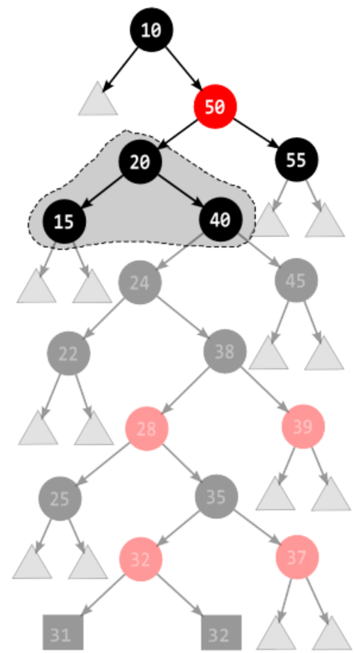
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

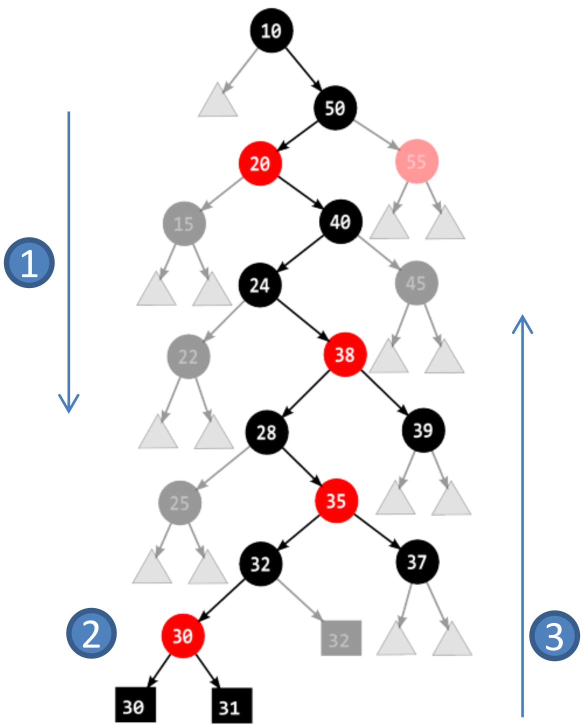
1. Root -> Leaf (top-down) traversal
2. Perform necessary modifications on the way to the leaf

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

Bottom-Up vs. Top-Down RBTs

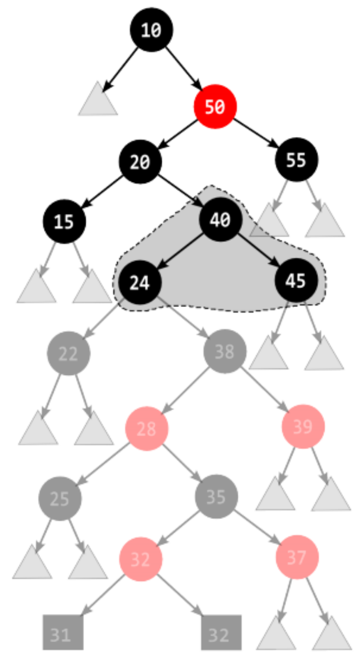
Example: `insert(30)`

Bottom-Up[1]



1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

Top-Down[2]



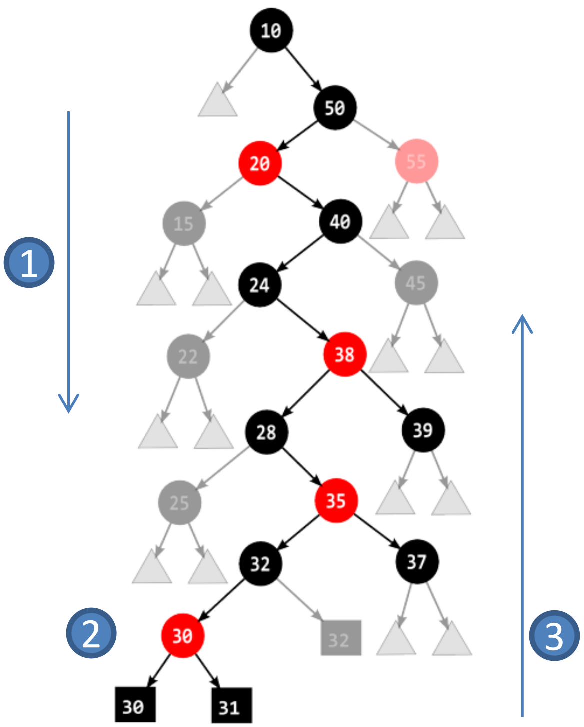
1. Root -> Leaf (top-down) traversal
2. Perform necessary modifications on the way to the leaf

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

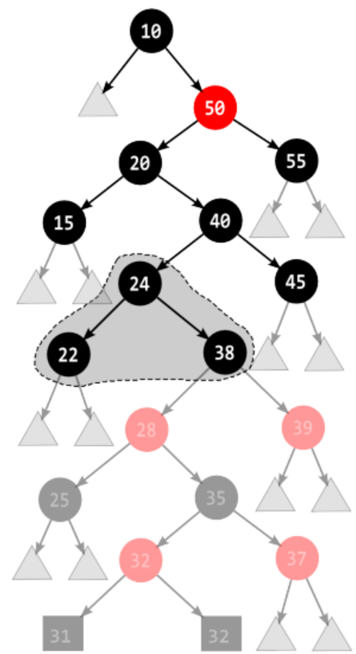
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



Bottom-Up:

1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

Top-Down:

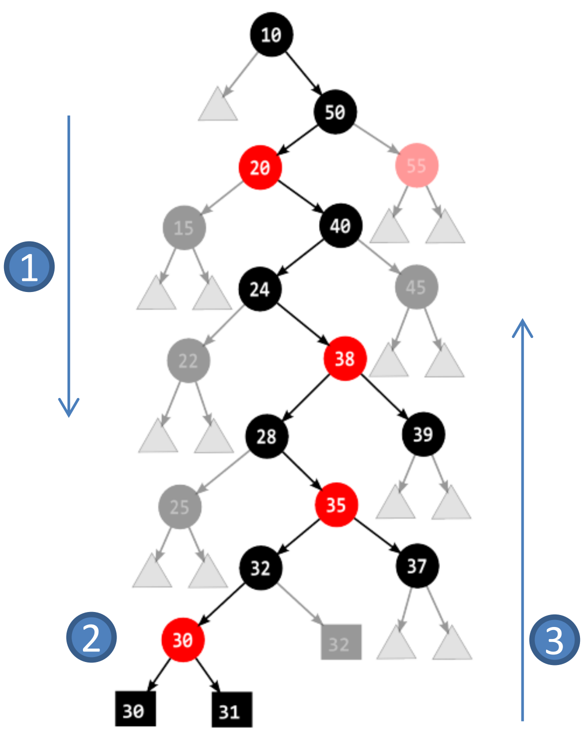
1. Root -> Leaf (top-down) traversal
2. Perform necessary modifications on the way to the leaf

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

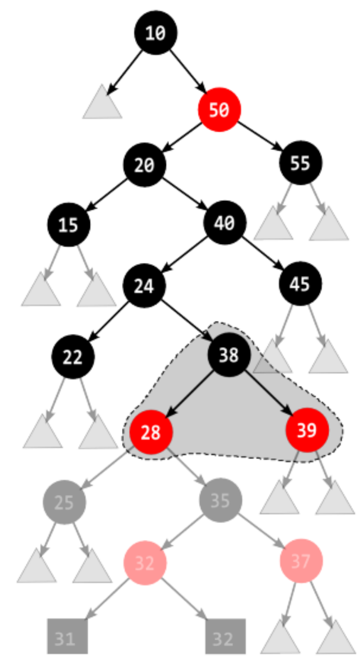
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



Bottom-Up:

1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

Top-Down:

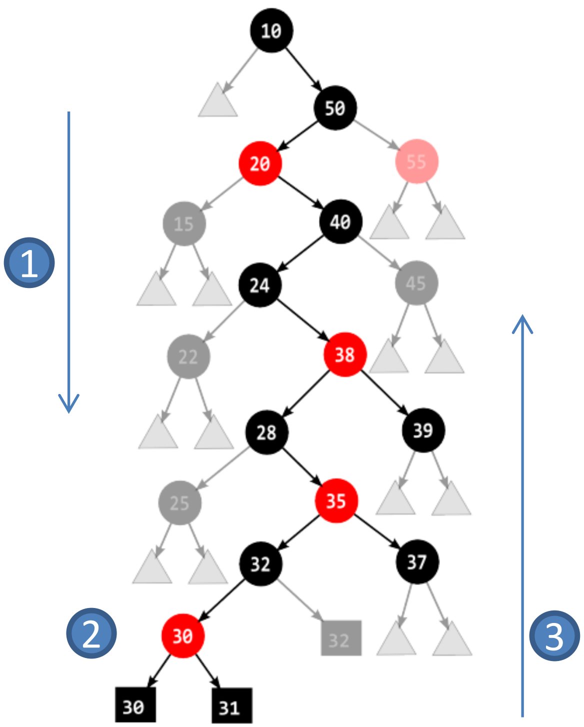
1. Root -> Leaf (top-down) traversal
2. Perform necessary modifications on the way to the leaf

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

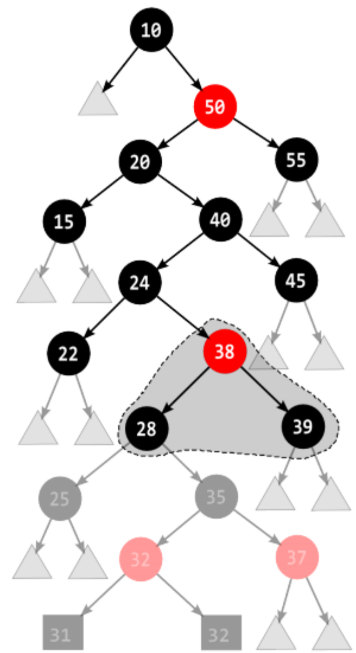
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

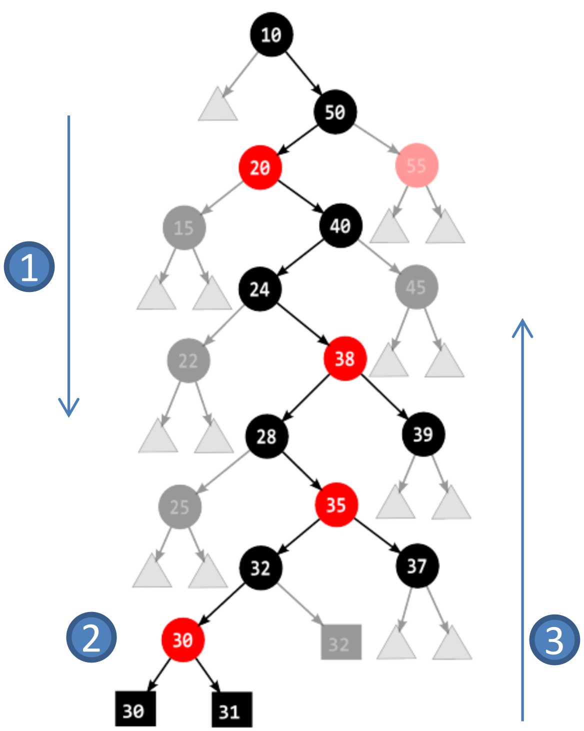
1. Root -> Leaf (top-down) traversal
2. Perform necessary modifications on the way to the leaf

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

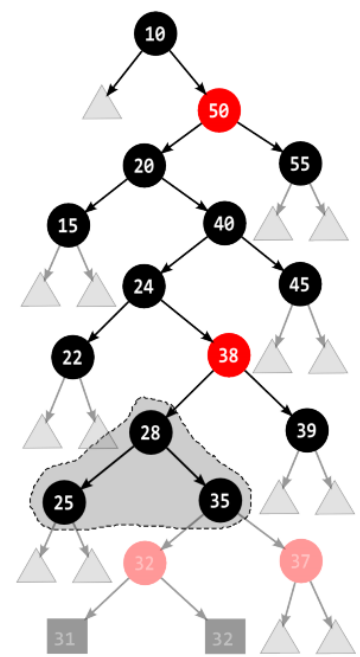
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

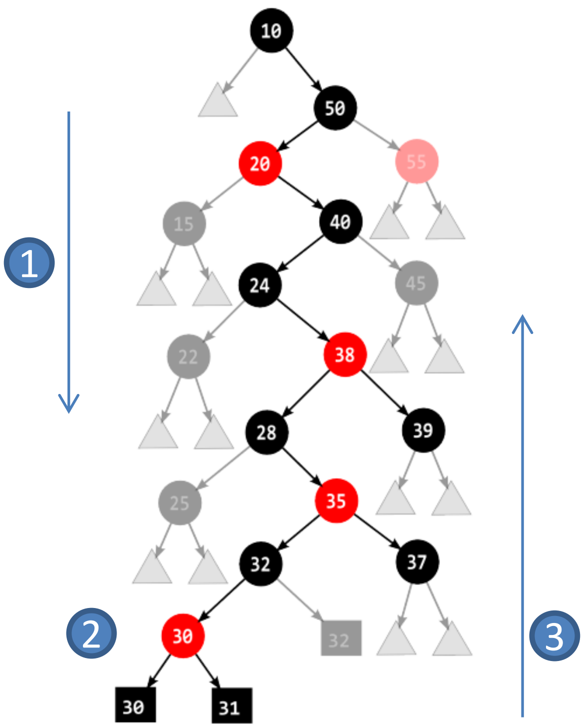
1. Root -> Leaf (top-down) traversal
2. Perform necessary modifications on the way to the leaf

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

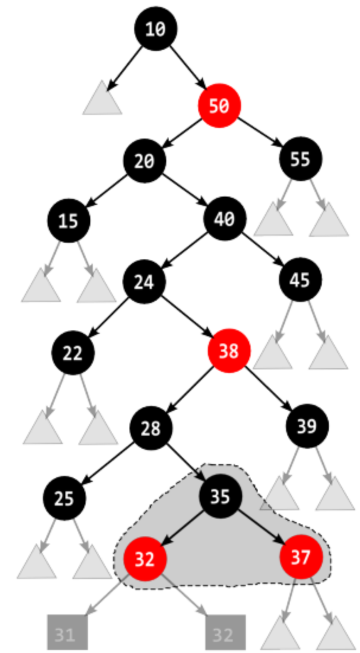
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

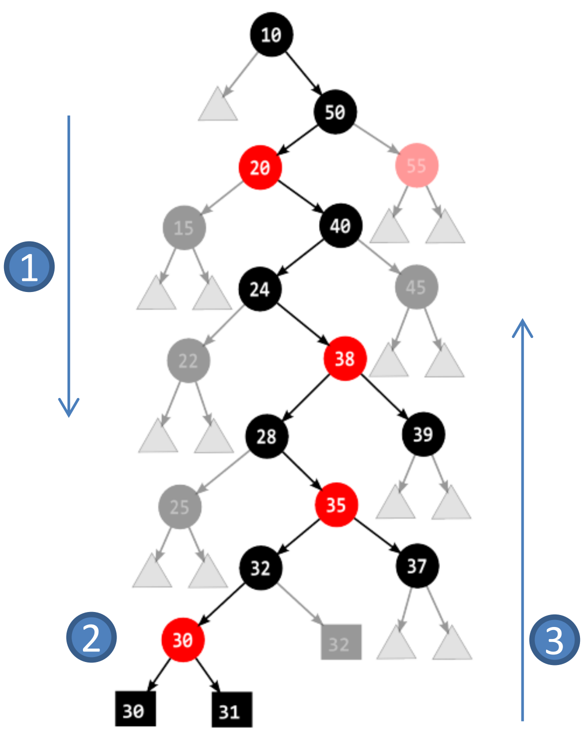
1. Root -> Leaf (top-down) traversal
2. Perform necessary modifications on the way to the leaf

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

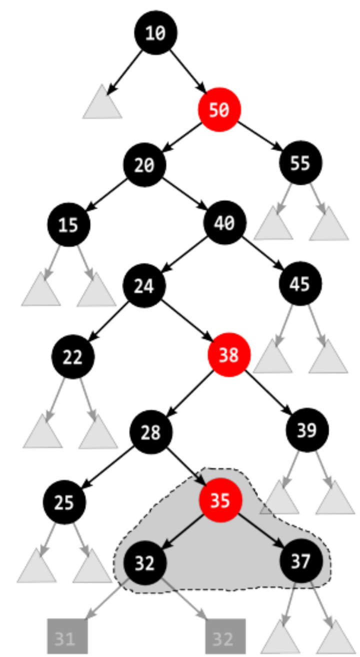
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

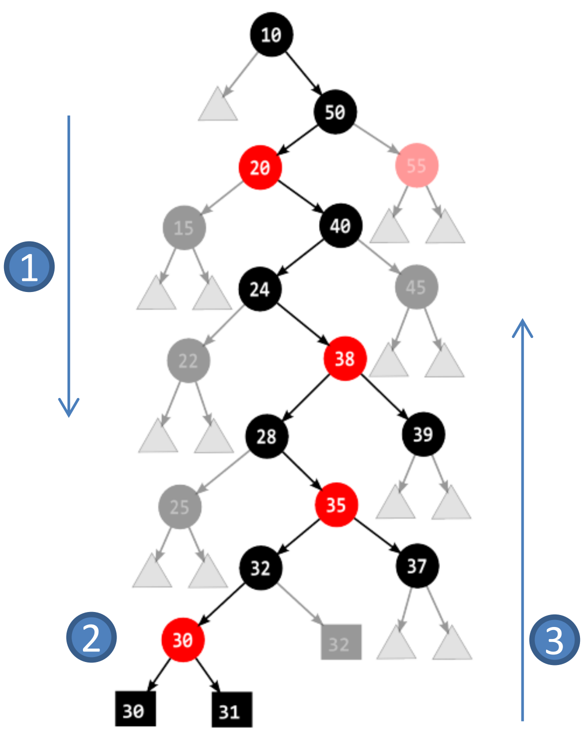
1. Root -> Leaf (top-down) traversal
2. Perform necessary modifications on the way to the leaf

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

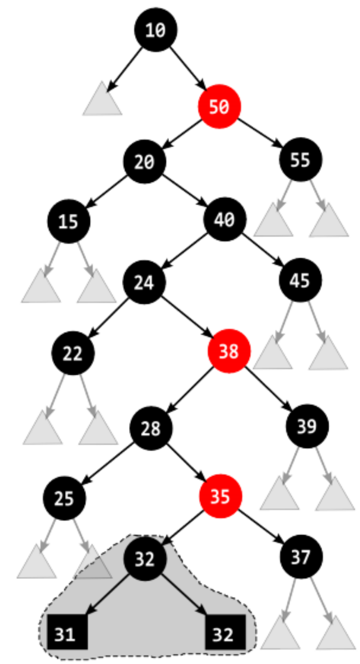
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

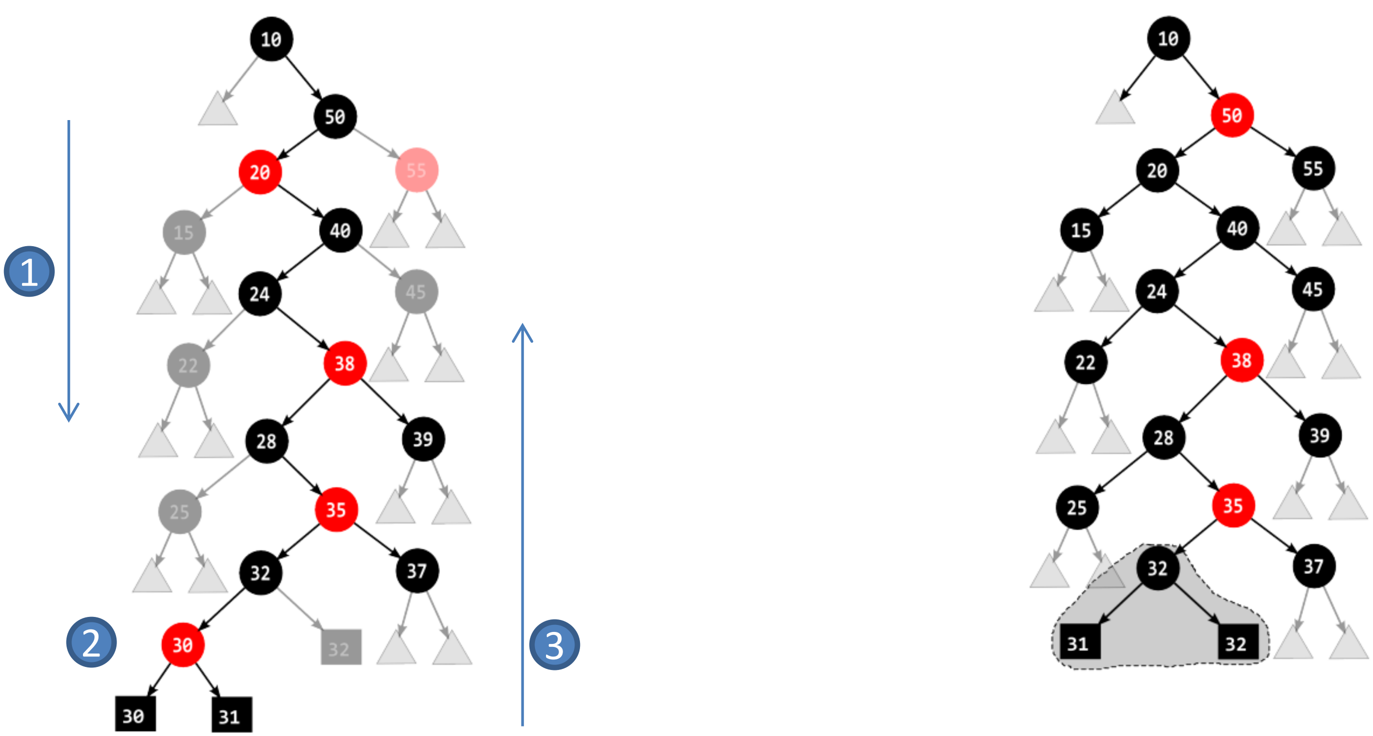
1. Root -> Leaf (top-down) traversal
2. Perform necessary modifications on the way to the leaf

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1] | Top-Down[2]



1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

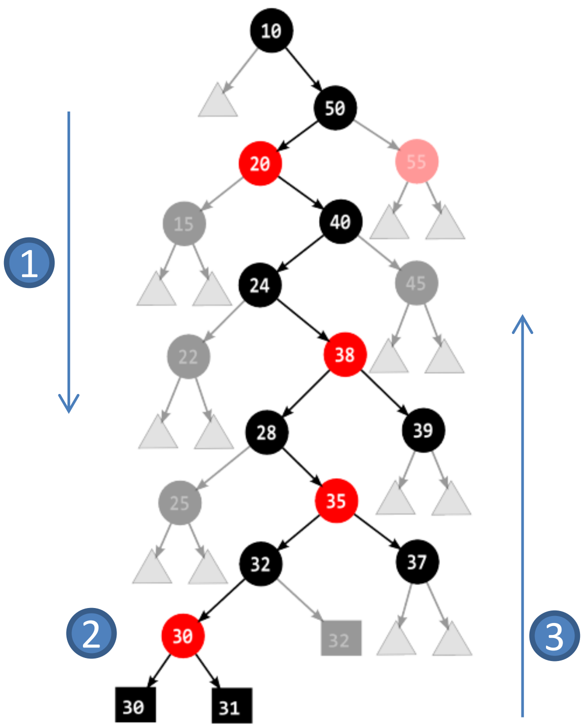
1. Root -> Leaf (top-down) traversal
2. Perform necessary modifications on the way to the leaf
3. When the new key is inserted no leaf->root traversal is necessary

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

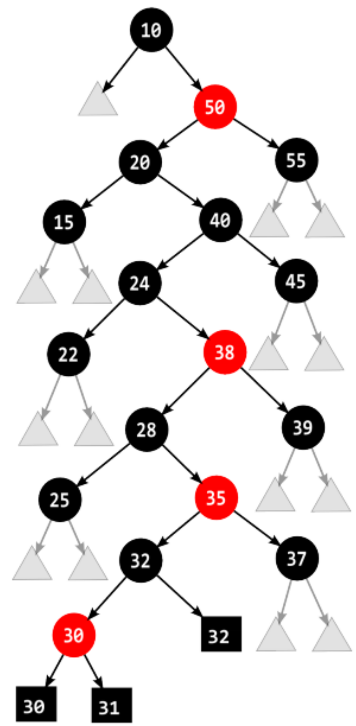
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



1. Root->leaf (top-down) traversal (read-only phase)
2. Insert the new key
3. Leaf -> Root traversal (bottom-up) to fix red-black tree violations

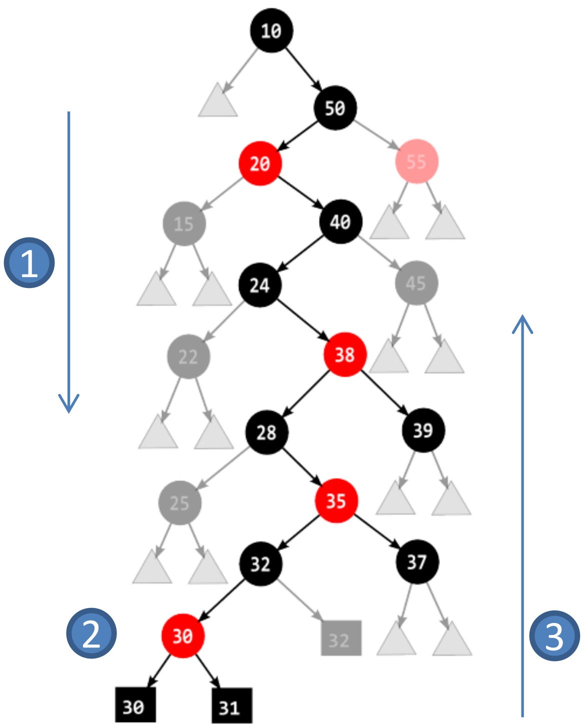
1. Root -> Leaf (top-down) traversal
2. Perform necessary modifications on the way to the leaf
3. When the new key is inserted no leaf->root traversal is necessary

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

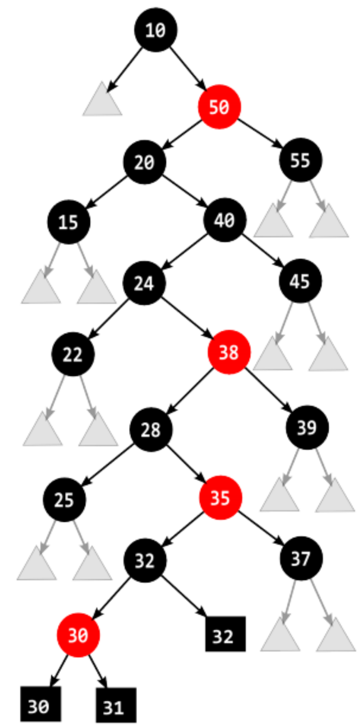
Bottom-Up vs. Top-Down RBTs

Example: `insert(30)`

Bottom-Up[1]



Top-Down[2]



- 2 traversals
- Less tree modifications (only the necessary)
- Not convenient for fine-grained synchronization approaches (threads might traverse the tree in opposite directions)

- 1 traversal
- More tree modifications (some could be avoided)
- Convenient for fine-grained synchronization approaches

[1] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms. The MIT Press, 3rd ed., 2009.
[2] R. A. Tarjan, Efficient top-down updating of red-black trees, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

A naïve HTM red-black tree: *bu-cg-htm*

code: insert operation in *bu-cg-htm*

```
int aborts = MAX_TX_RETRIES;

AGAIN:
while (SGL is taken)
    ;

int status = xbegin();
if (status == TX_BEGIN_SUCCESS) {
    if (SGL is taken)
        xabort(0xFF);

    rbt_insert(...);
    xend();
} else {
    if (--aborts > 0)
        goto AGAIN;

    acquire_lock(SGL);
    rbt_insert(...);
    release_lock(SGL);
}
```

- Bottom-Up external RBT
- Each operation enclosed in a single HTM transaction
- Single global lock (SGL) fallback path.

A naïve HTM red-black tree: *bu-cg-htm*

code: insert operation in *bu-cg-htm*

```
int aborts = MAX_TX_RETRIES;

AGAIN:
while (SGL is taken)
    ;

int status = xbegin();
if (status == TX_BEGIN_SUCCESS) {
    if (SGL is taken)
        xabort(0xFF);

    rbt_insert(...);
    xend();
} else {
    if (--aborts > 0)
        goto AGAIN;

    acquire_lock(SGL);
    rbt_insert(...);
    release_lock(SGL);
}
```

- Bottom-Up external RBT
- Each operation enclosed in a single HTM transaction
- Single global lock (SGL) fallback path.

Begin a transaction.
Return values:

1. *TX_BEGIN_SUCCESS*: a transaction has just began.
2. != *TX_BEGIN_SUCCESS*: a previously initialized transaction has aborted. Return value contains information about abort reason.

A naïve HTM red-black tree: *bu-cg-htm*

code: insert operation in *bu-cg-htm*

```
int aborts = MAX_TX_RETRIES;

AGAIN:
while (SGL is taken)
    ;

int status = xbegin();
if (status == TX_BEGIN_SUCCESS) {
    if (SGL is taken)
        xabort(0xFF);

    rbt_insert(...);
    xend();
} else {
    if (--aborts > 0)
        goto AGAIN;

    acquire_lock(SGL);
    rbt_insert(...);
    release_lock(SGL);
}
```

- Bottom-Up external RBT
- Each operation enclosed in a single HTM transaction
- Single global lock (SGL) fallback path.

Critical section executed in transactional mode

A naïve HTM red-black tree: *bu-cg-htm*

code: insert operation in *bu-cg-htm*

```
int aborts = MAX_TX_RETRIES;

AGAIN:
while (SGL is taken)
    ;

int status = xbegin();
if (status == TX_BEGIN_SUCCESS) {
    if (SGL is taken)
        xabort(0xFF);

    rbt_insert(...);
    xend();
} else {
    if (--aborts > 0)
        goto AGAIN;

    acquire_lock(SGL);
    rbt_insert(...);
    release_lock(SGL);
}
```

- Bottom-Up external RBT
- Each operation enclosed in a single HTM transaction
- Single global lock (SGL) fallback path.

Abort if lock is taken (another thread executes in non-transactional mode)

A naïve HTM red-black tree: *bu-cg-htm*

code: insert operation in *bu-cg-htm*

```
int aborts = MAX_TX_RETRIES;

AGAIN:
while (SGL is taken)
    ;

int status = xbegin();
if (status == TX_BEGIN_SUCCESS) {
    if (SGL is taken)
        xabort(0xFF);

    rbt_insert(...);
    xend();
} else {
    if (--aborts > 0)
        goto AGAIN;

    acquire_lock(SGL);
    rbt_insert(...);
    release_lock(SGL);
}
```

- Bottom-Up external RBT
- Each operation enclosed in a single HTM transaction
- Single global lock (SGL) fallback path.

Execute operation in transactional mode

A naïve HTM red-black tree: *bu-cg-htm*

code: insert operation in *bu-cg-htm*

```
int aborts = MAX_TX_RETRIES;

AGAIN:
while (SGL is taken)
    ;

int status = xbegin();
if (status == TX_BEGIN_SUCCESS) {
    if (SGL is taken)
        xabort(0xFF);

    rbt_insert(...);
    xend();
} else {
    if (--aborts > 0)
        goto AGAIN;

    acquire_lock(SGL);
    rbt_insert(...);
    release_lock(SGL);
}
```

Commit transaction

- Bottom-Up external RBT
- Each operation enclosed in a single HTM transaction
- Single global lock (SGL) fallback path.

National Technical University of Athens CSLab

A naïve HTM red-black tree: *bu-cg-htm*

code: insert operation in *bu-cg-htm*

```
int aborts = MAX_TX_RETRIES;

AGAIN:
while (SGL is taken)
    ;

int status = xbegin();
if (status == TX_BEGIN_SUCCESS) {
    if (SGL is taken)
        xabort(0xFF);

    rbt_insert(...);
    xend();
} else {
    if (--aborts > 0)
        goto AGAIN;

    acquire_lock(SGL);
    rbt_insert(...);
    release_lock(SGL);
}
```

- Bottom-Up external RBT
- Each operation enclosed in a single HTM transaction
- Single global lock (SGL) fallback path.

Code executed on abort

A naïve HTM red-black tree: *bu-cg-htm*

code: insert operation in *bu-cg-htm*

```
int aborts = MAX_TX_RETRIES;

AGAIN:
while (SGL is taken)
    ;

int status = xbegin();
if (status == TX_BEGIN_SUCCESS) {
    if (SGL is taken)
        xabort(0xFF);

    rbt_insert(...);
    xend();
} else {
    if (--aborts > 0)
        goto AGAIN;

    acquire_lock(SGL);
    rbt_insert(...);
    release_lock(SGL);
}
```

- Bottom-Up external RBT
- Each operation enclosed in a single HTM transaction
- Single global lock (SGL) fallback path.

Retry in transactional mode

A naïve HTM red-black tree: *bu-cg-htm*

code: insert operation in *bu-cg-htm*

```
int aborts = MAX_TX_RETRIES;

AGAIN:
while (SGL is taken)
    ;

int status = xbegin();
if (status == TX_BEGIN_SUCCESS) {
    if (SGL is taken)
        xabort(0xFF);

    rbt_insert(...);
    xend();
} else {
    if (--aborts > 0)
        goto AGAIN;

    acquire_lock(SGL);
    rbt_insert(...);
    release_lock(SGL);
}
```

- Bottom-Up external RBT
- Each operation enclosed in a single HTM transaction
- Single global lock (SGL) fallback path.

Execute operation in non-transactional mode (SGL fallback path)

A naïve HTM red-black tree: *bu-cg-htm*

code: insert operation in *bu-cg-htm*

```
int aborts = MAX_TX_RETRIES;

AGAIN:
while (SGL is taken)
    ;

int status = xbegin();
if (status == TX_BEGIN_SUCCESS) {
    if (SGL is taken)
        xabort(0xFF);

    rbt_insert(...);
    xend();
} else {
    if (--aborts > 0)
        goto AGAIN;

    acquire_lock(SGL);
    rbt_insert(...);
    release_lock(SGL);
}
```

- Bottom-Up external RBT
- Each operation enclosed in a single HTM transaction
- Single global lock (SGL) fallback path.

Spin until the fallback lock is released

National Technical University of Athens CSLab

Experimental Platforms

	Intel Haswell-EP	IBM Power8
Processors	2 x Intel Xeon E5-2697 v3	2 x IBM Power8
# Cores	2 x 14	2 x 10
# Threads	56 (2-way SMT)	160 (8-way SMT)
Core clock	2.6 GHz	3.7 GHz
L1 (Data)	32KB, 8-way, 64B block size	64KB, 8-way, 128B block size
L2	256KB, 8-way, 64B block size	512KB, 8-way, 128B block size
L3	35MB, 20-way, 64B block size (shared per die)	80MB, 8-way, 128B block size (shared per die)
Memory	64 GB (4 NUMA regions)	256 GB (4 NUMA regions)

Experimental Platforms: HTM characteristics

	Intel Haswell-EP	IBM Power8
Versioning	Lazy	
Progress guarantees	Best effort	
Conflict detection	Eager	
Conflict granularity	Cache line	
Cache line size	64B	128B
TX read-set (total / per HW thread)	4MB / 2MB	8KB / 1KB
TX write-set (total / per HW thread)	22KB / 11KB	8KB / 1KB

Abort reasons	Conflict	Transactional conflict
		Non-transactional conflict
	Capacity	
	Explicit	

Experimental Platforms: HTM characteristics

	Intel Haswell-EP	IBM Power8
Versioning	Lazy	
Progress guarantees	Best effort	
Conflict detection	Eager	
Conflict granularity	Cache line	
Cache line size	[illegible]	[illegible]
TX read-set (total / per HW thread)	4MB / 2MB	8KB / 1KB
TX write-set (total / per HW thread)	22KB / 11KB	8KB / 1KB

Haswell-EP can support larger transactions!

Abort reasons	Conflict	Transactional conflict
		Non-transactional conflict
	Capacity	
	Explicit	

Experimental Setup

Code

- Written in C, using GCC intrinsics and inline assembly for HTM instructions
- GCC 4.9.2/4.9.1, -O3 optimization
- Nodes are padded and aligned to occupy exactly one cache line

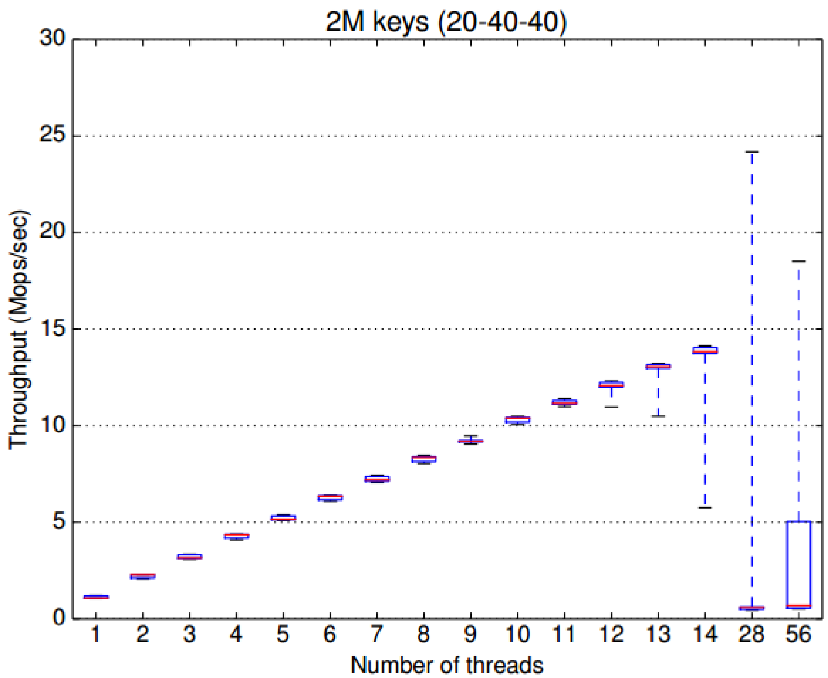
HTM Evaluation

- Tree sizes (3 configurations)
  - Medium/Large trees: 2M/20M/100M keys
  - Tree initialized to contain half keys in specified range (e.g. 1M keys in the 2M tree)
- Operations Workload (%lookups-%insertions-%deletions)
  - Read-intensive: 80-10-10
  - Read-write: 50-25-25
  - Write-intensive: 20-40-40

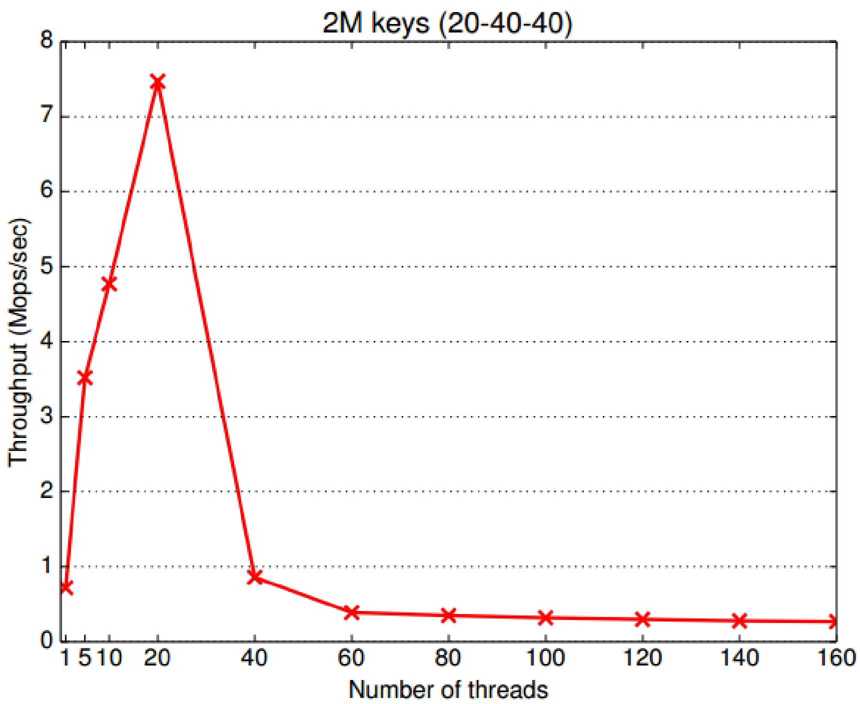
Benchmarks Execution

- Benchmarks duration is 15 seconds
- Employ empty physical cores before SMT contexts
- 10 transactional retries before acquiring SGL
- Results are the average of 20 independent executions (box plots shown when necessary)

bu-cg-htm: Performance



Haswell-EP

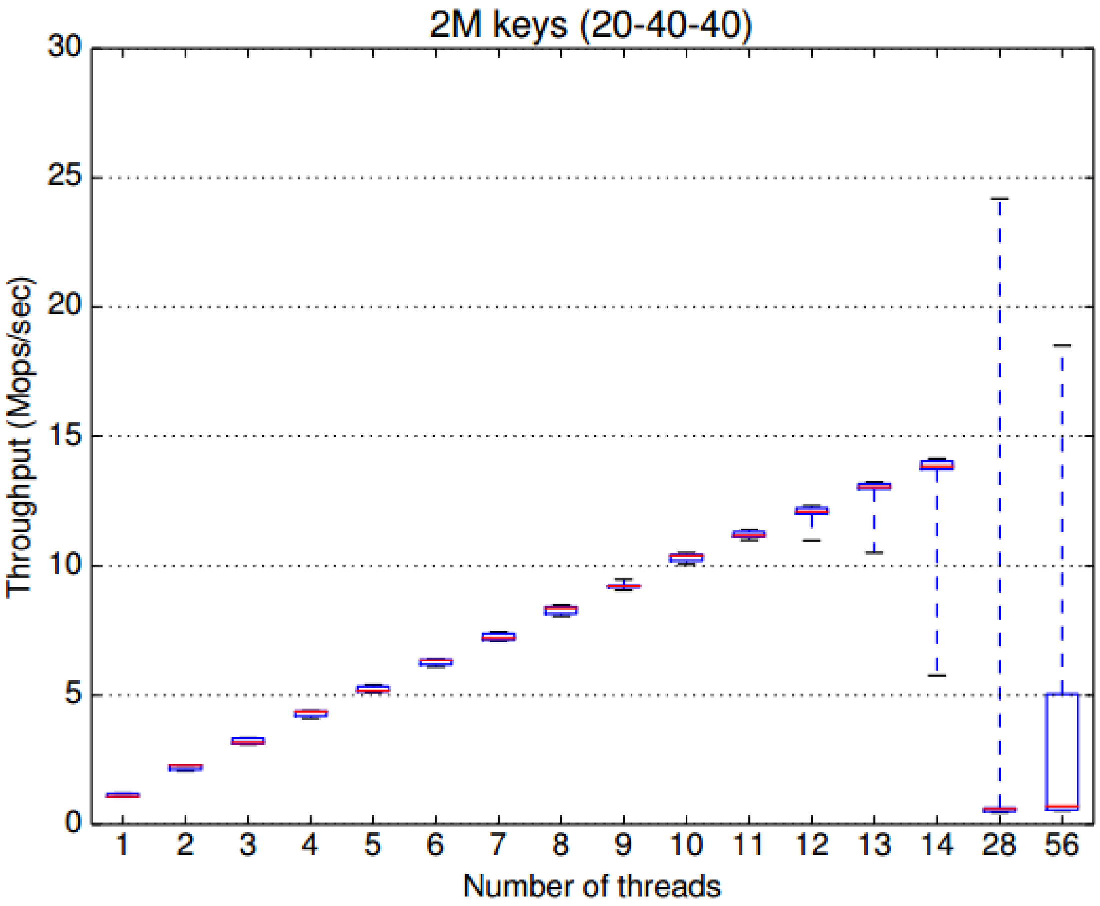


Power8

National Technical University of Athens
CSLab

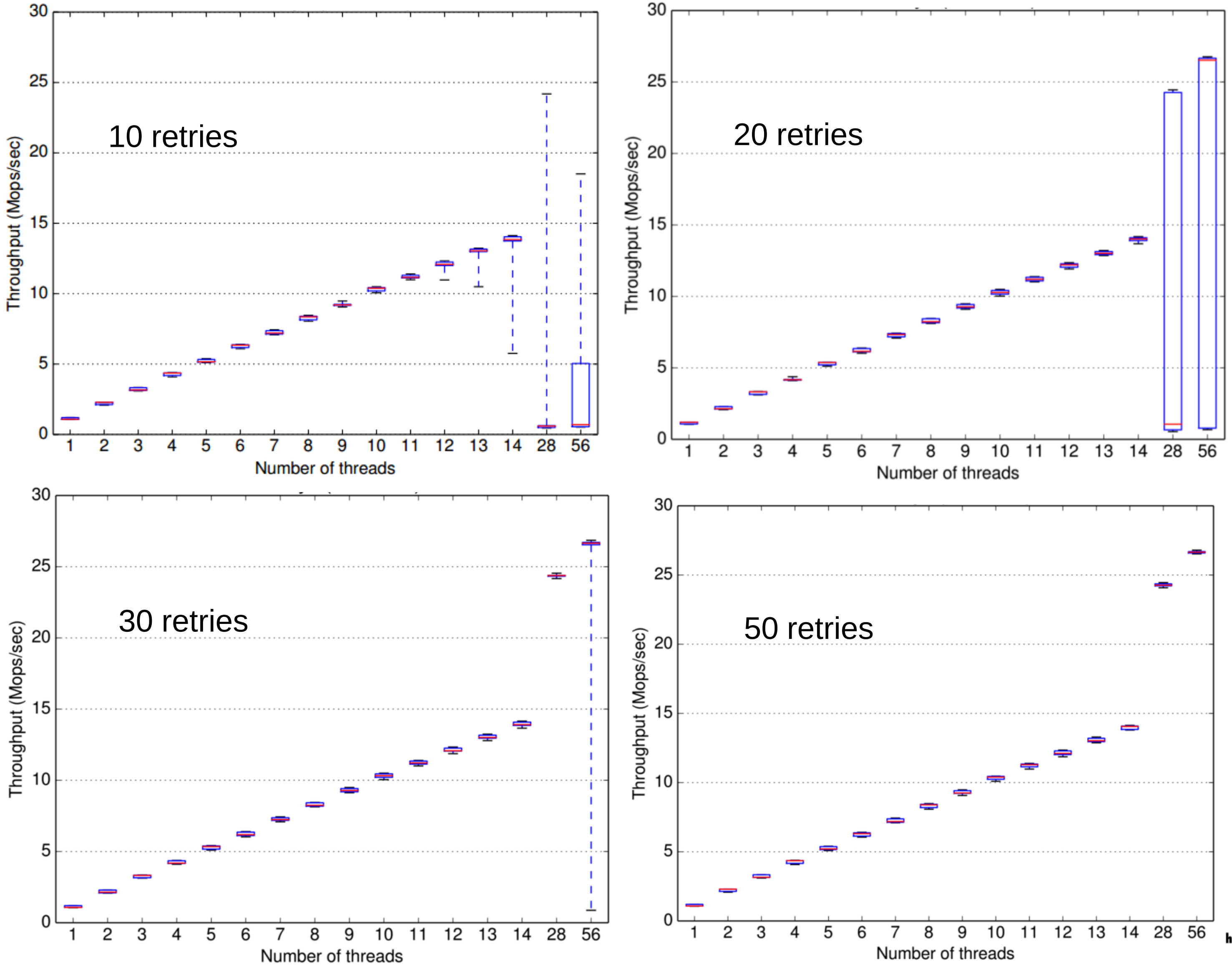
OPTIMIZING FOR HASWELL-EP

Haswell-EP: bu-cg-htm throughput



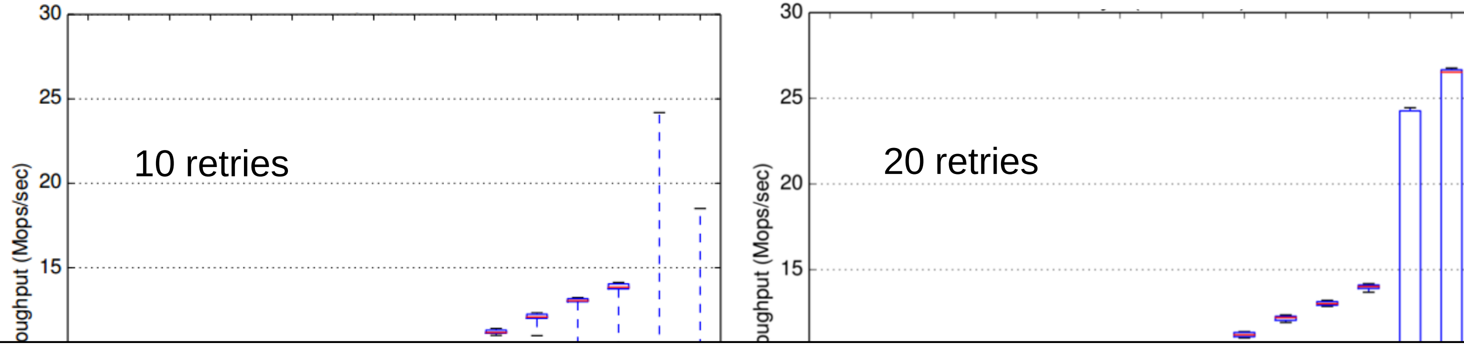
Haswell-EP: tuning number of retries

2M keys (20-40-40)

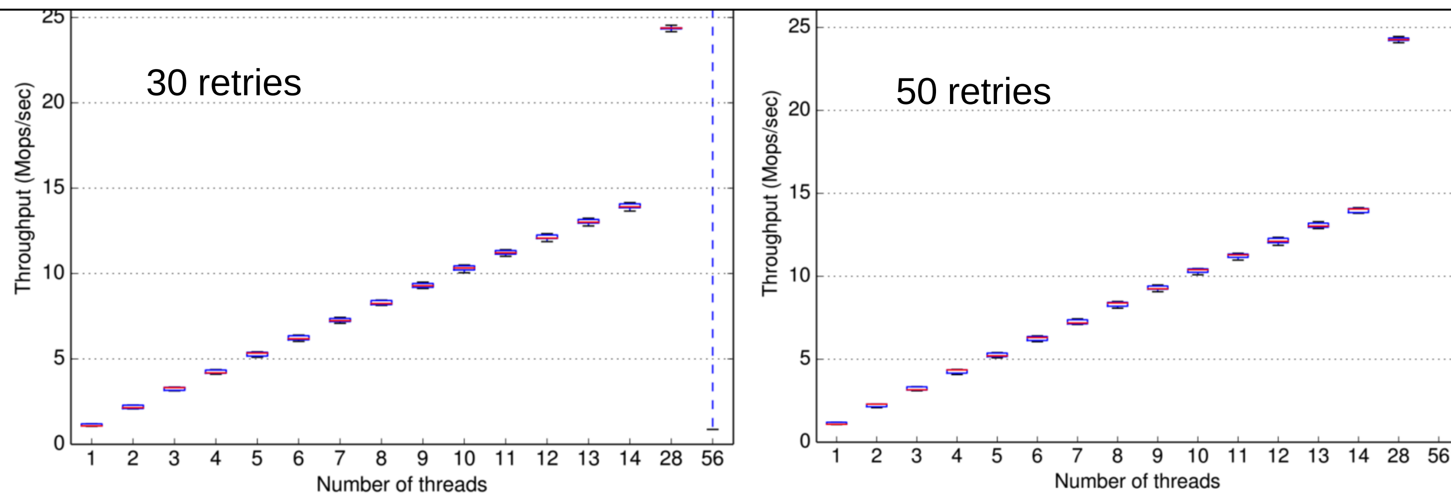


Haswell-EP: tuning number of retries

2M keys (20-40-40)

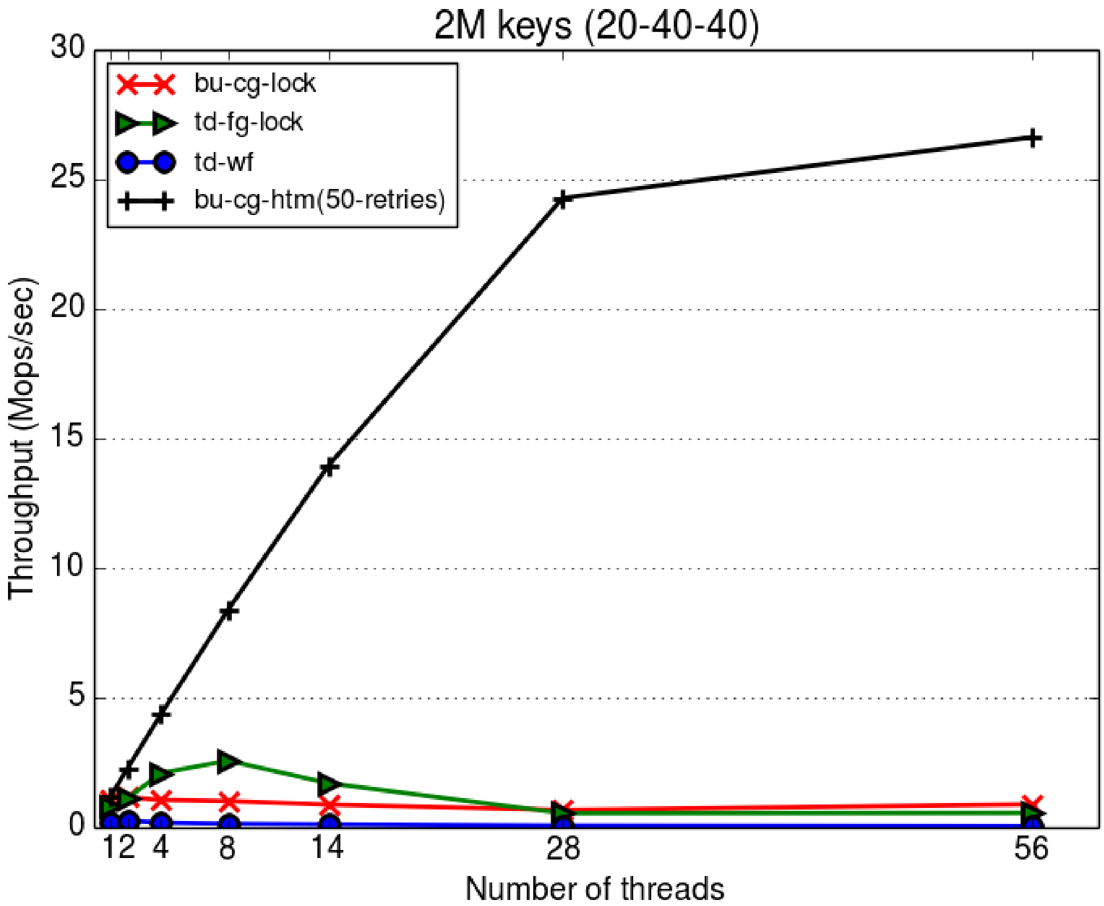


- Tuning the number of transactional retries is vital to get stable and high performance
- As more cores are added in future processors this effect will probably become more intense



CSLab

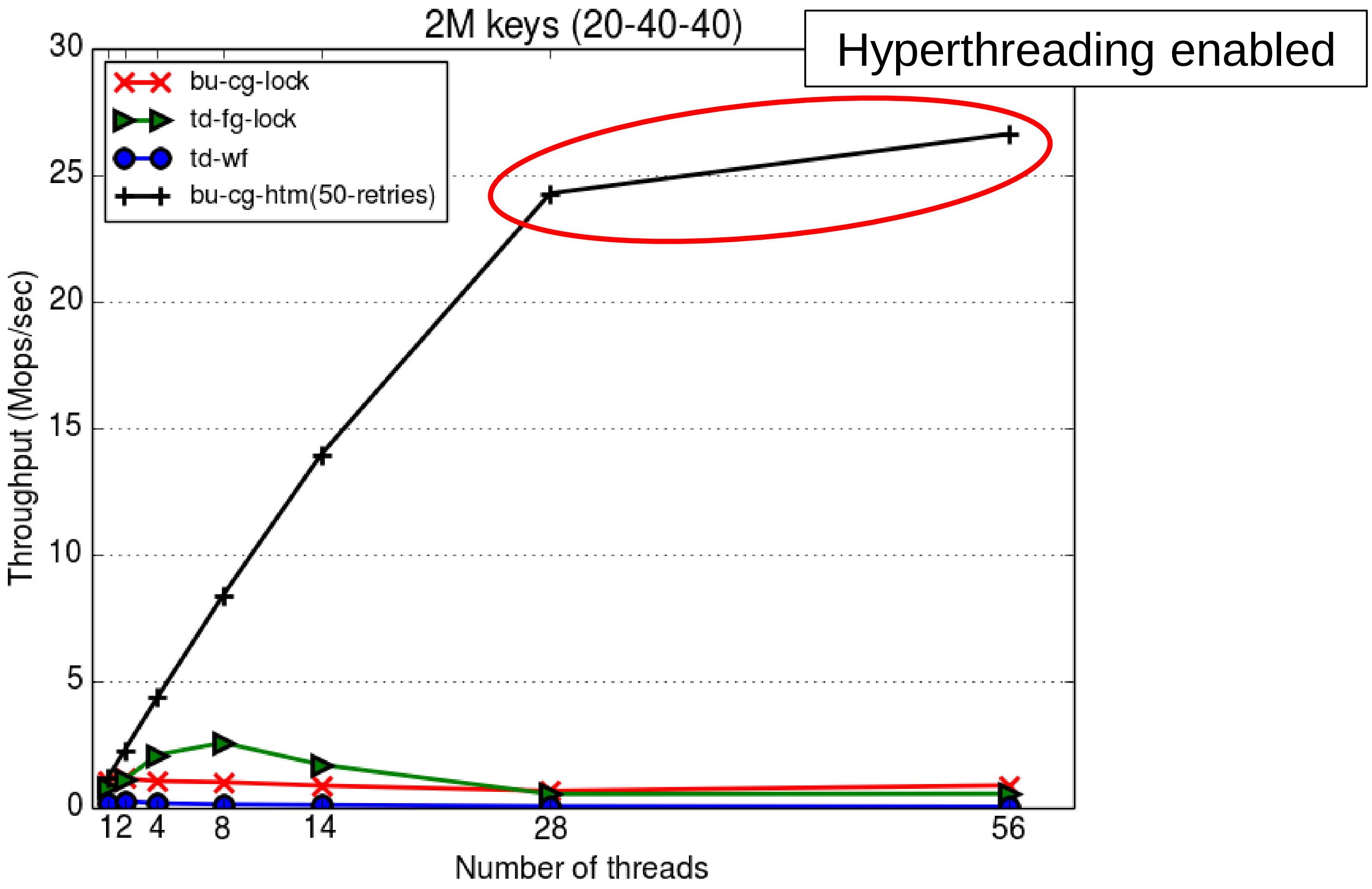
Haswell-EP: Performance



- *bu-cg-lock*: bottom-up coarse-grained locking
- *td-fg-lock*: top-down fine-grained locking
- *td-wf*[1]: top-down wait-free (atomic operations, CAS)

[1] A. Natarajan, L. H. Savoie, N. Mittal, Concurrent Wait-Free Red Black Trees, SSS 2013

Haswell-EP: Performance

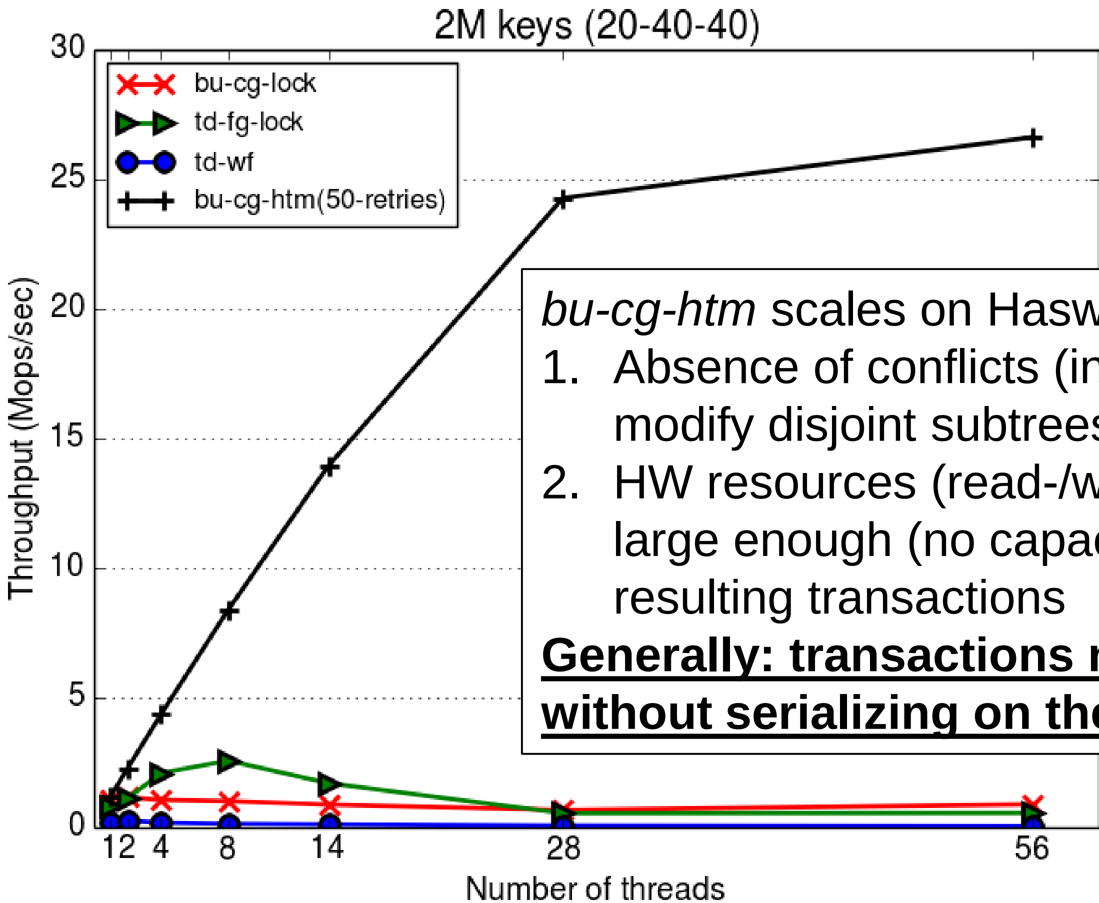


- *bu-cg-lock*: bottom-up coarse-grained locking
- *td-fg-lock*: top-down fine-grained locking
- *td-wf*[1]: top-down wait-free (atomic operations, CAS)

[1] A. Natarajan, L. H. Savoie, N. Mittal, Concurrent Wait-Free Red Black Trees, SSS 2013

National Technical University of Athens CSLab

Haswell-EP: Performance



bu-cg-htm scales on Haswell-EP for 2 reasons:

1. Absence of conflicts (in most cases threads modify disjoint subtrees)
2. HW resources (read-/write- set buffers) are large enough (no capacity aborts) for the resulting transactions

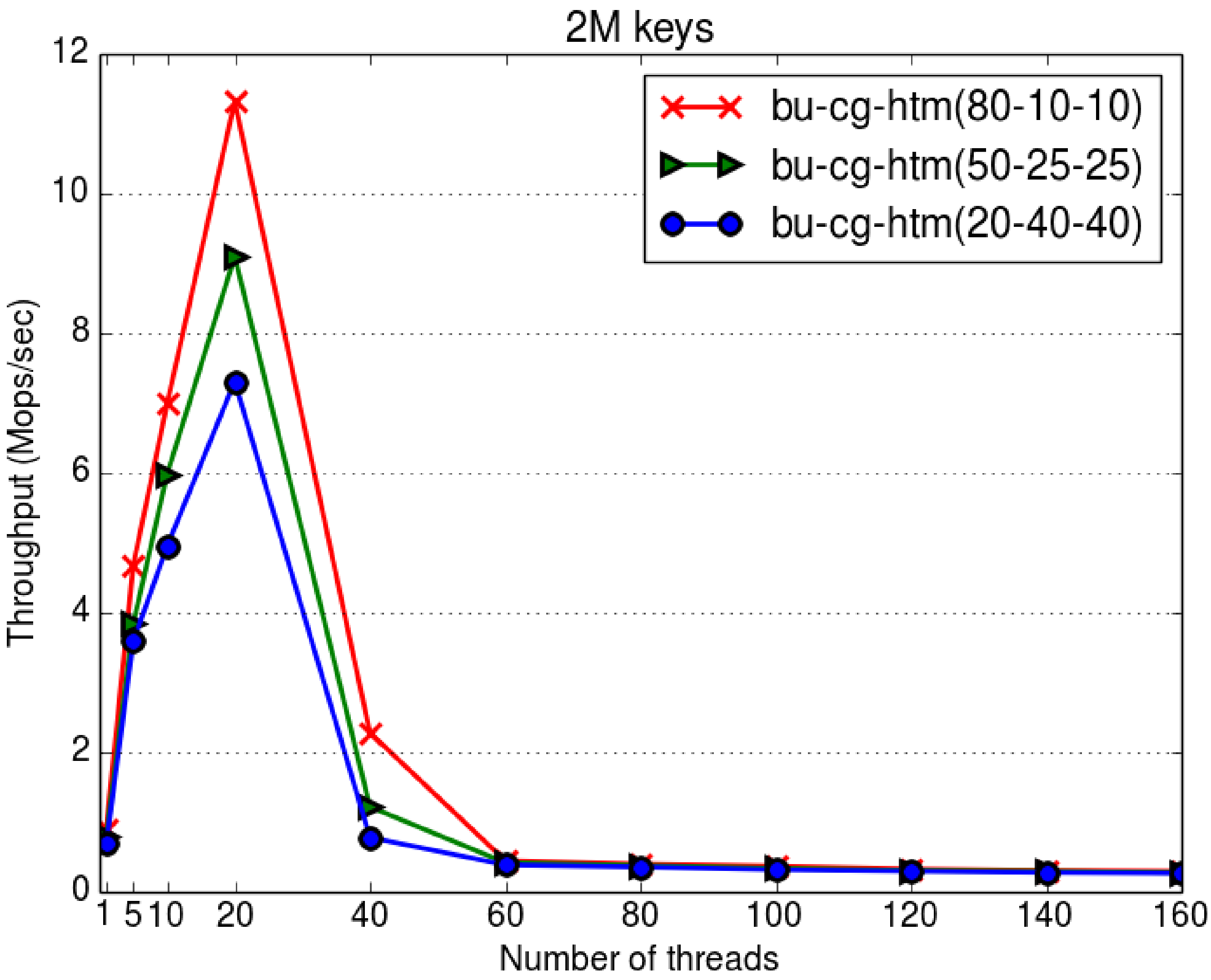
Generally: transactions manage to commit without serializing on the global lock

- *bu-cg-lock*: bottom-up coarse-grained locking
- *td-fg-lock*: top-down fine-grained locking
- *td-wf*[1]: top-down wait-free (atomic operations, CAS)

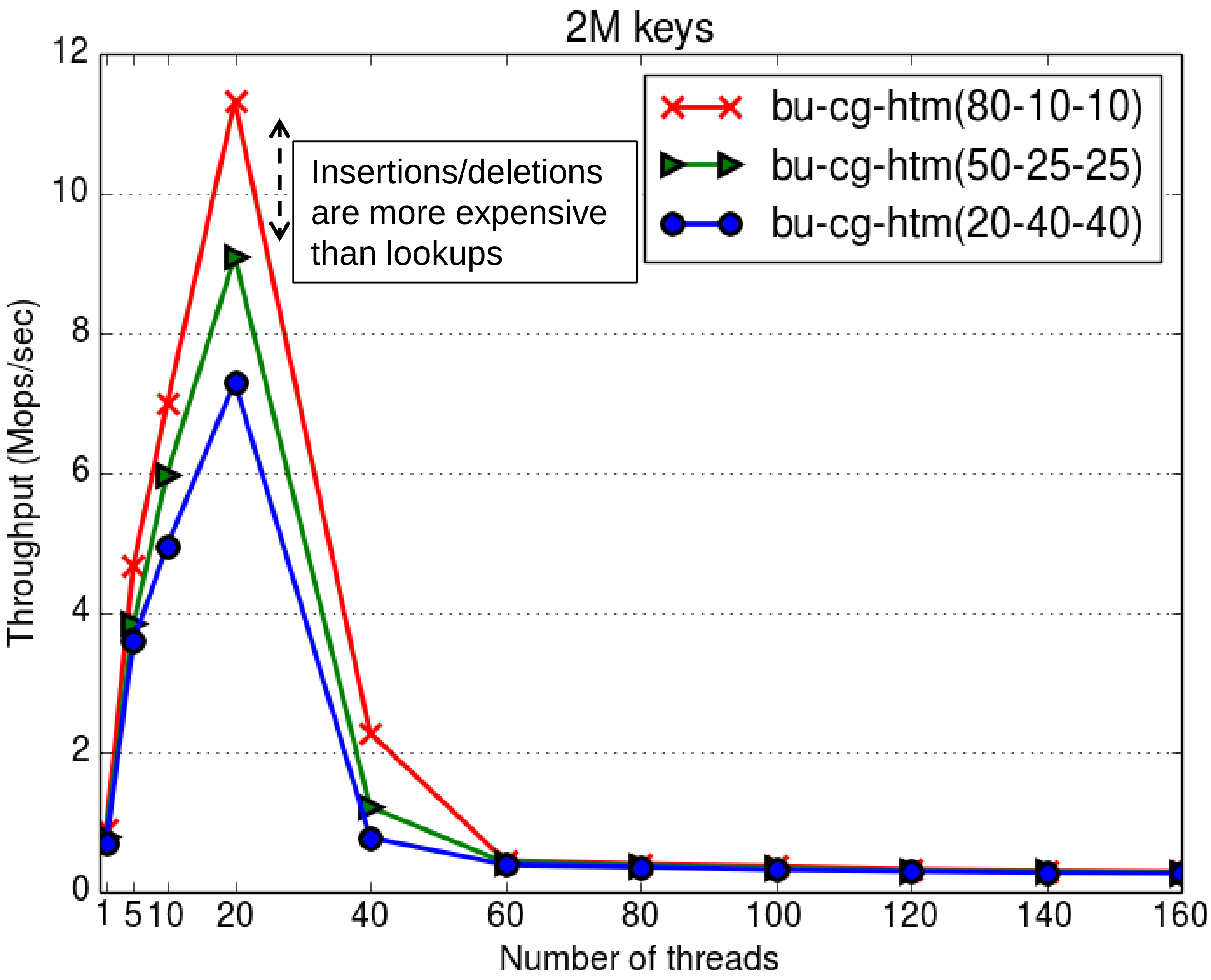
[1] A. Natarajan, L. H. Savoie, N. Mittal, Concurrent Wait-Free Red Black Trees, SSS 2013

OPTIMIZING FOR POWER8

Power8: bu-cg-htm throughput

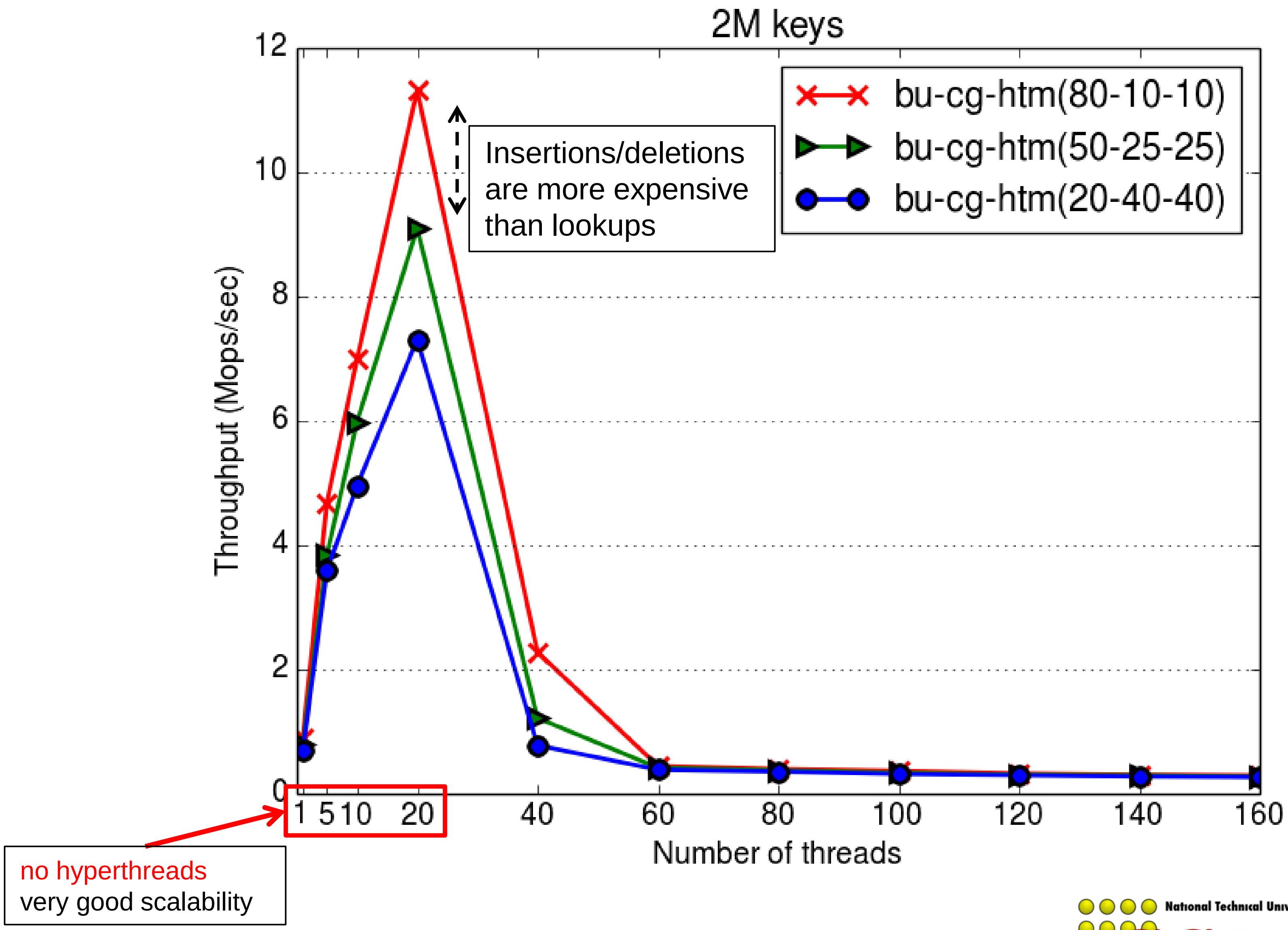


Power8: bu-cg-htm throughput



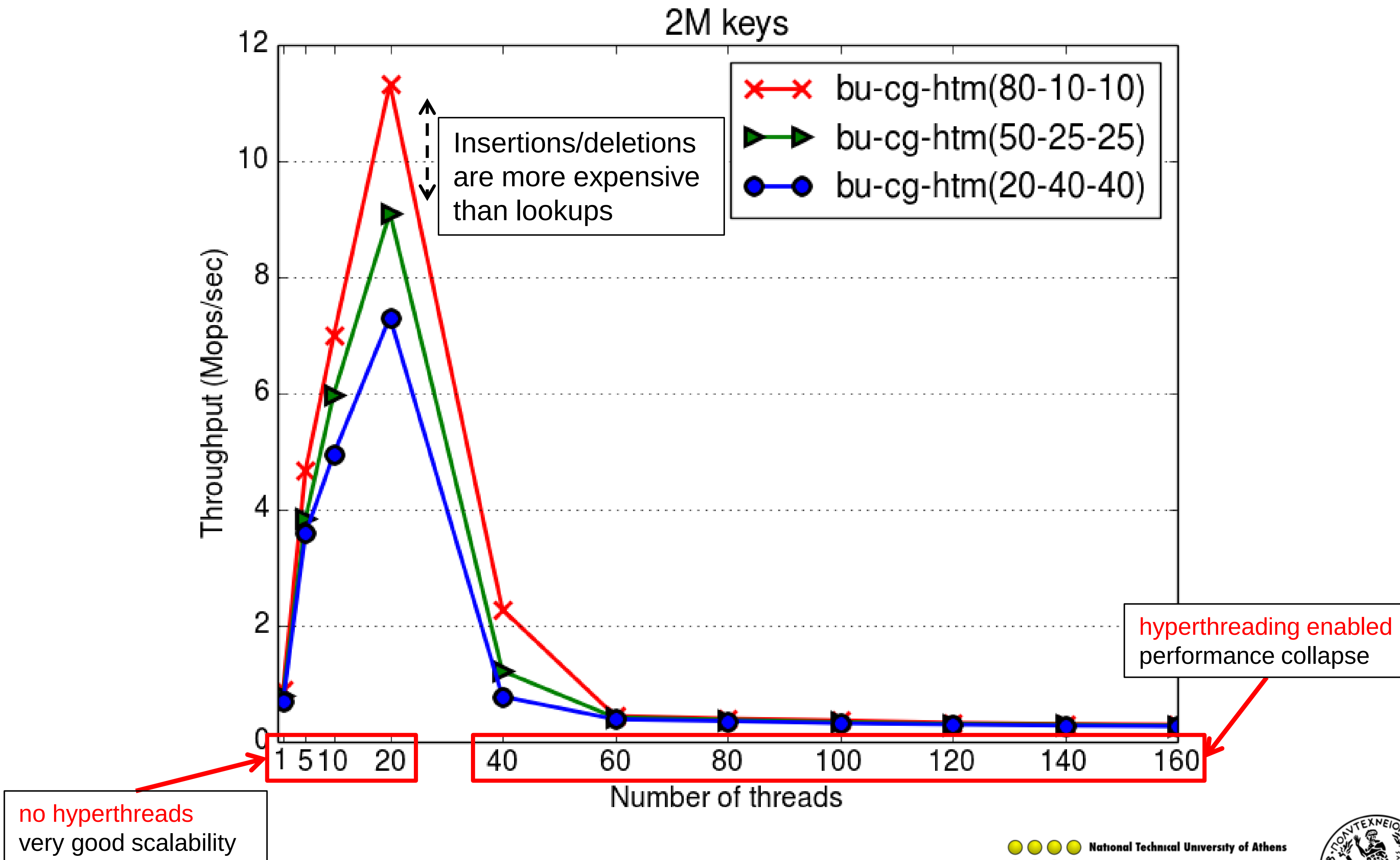
National Technical University of Athens CSLab

Power8: bu-cg-htm throughput



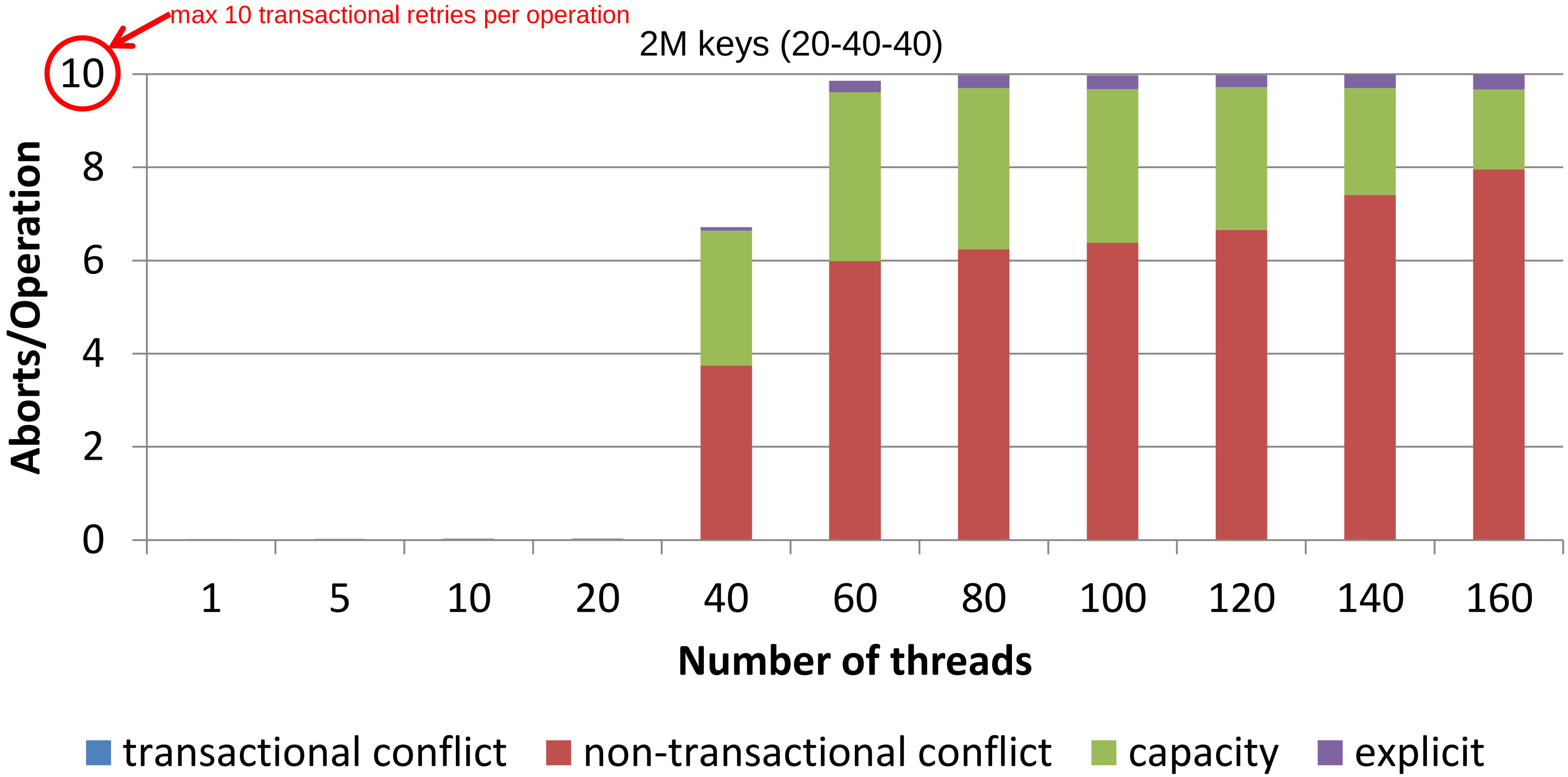
National Technical University of Athens CSLab

Power8: bu-cg-htm throughput



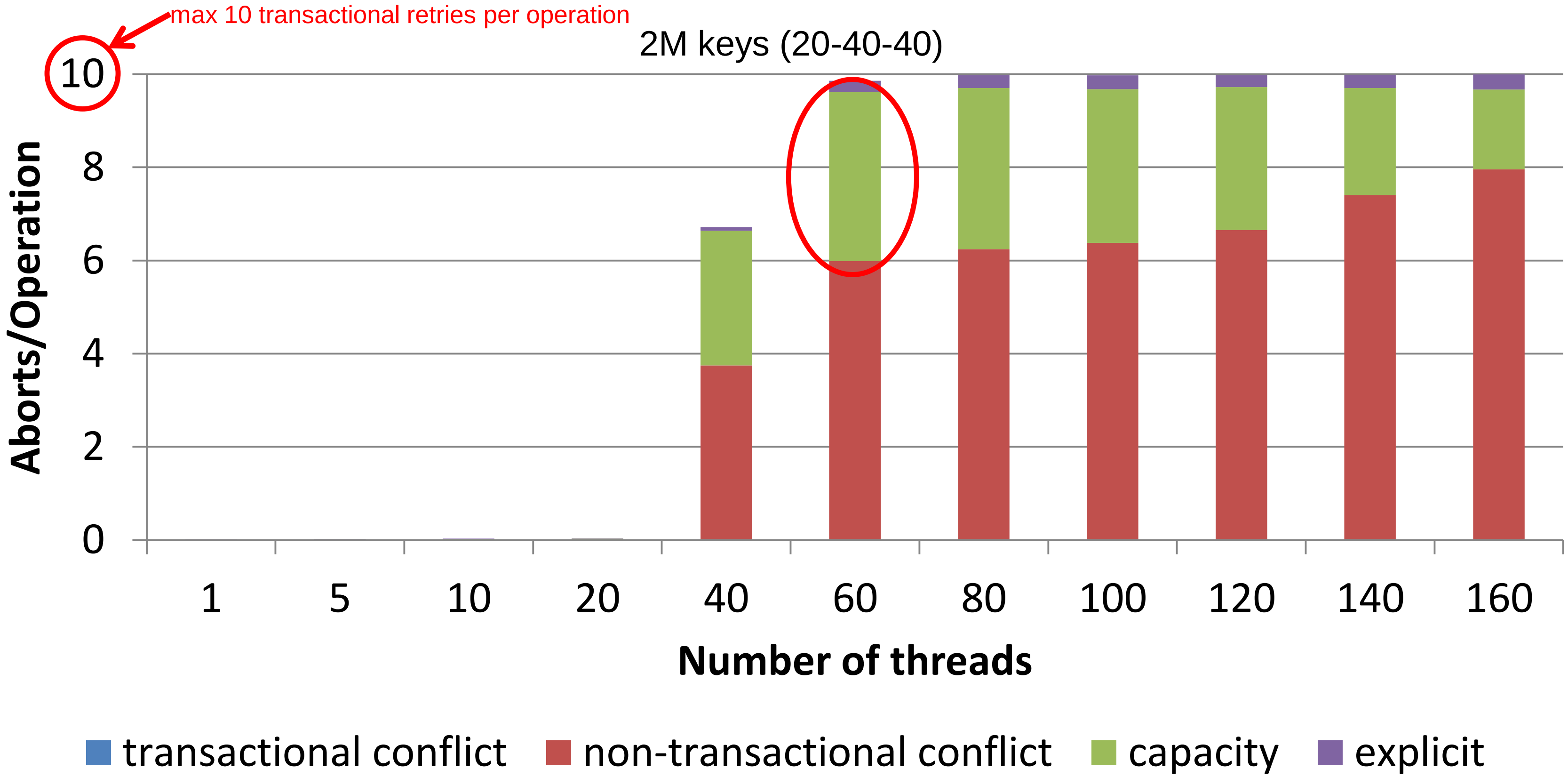
National Technical University of Athens CSLab

Power8: bu-cg-htm aborts' breakdown



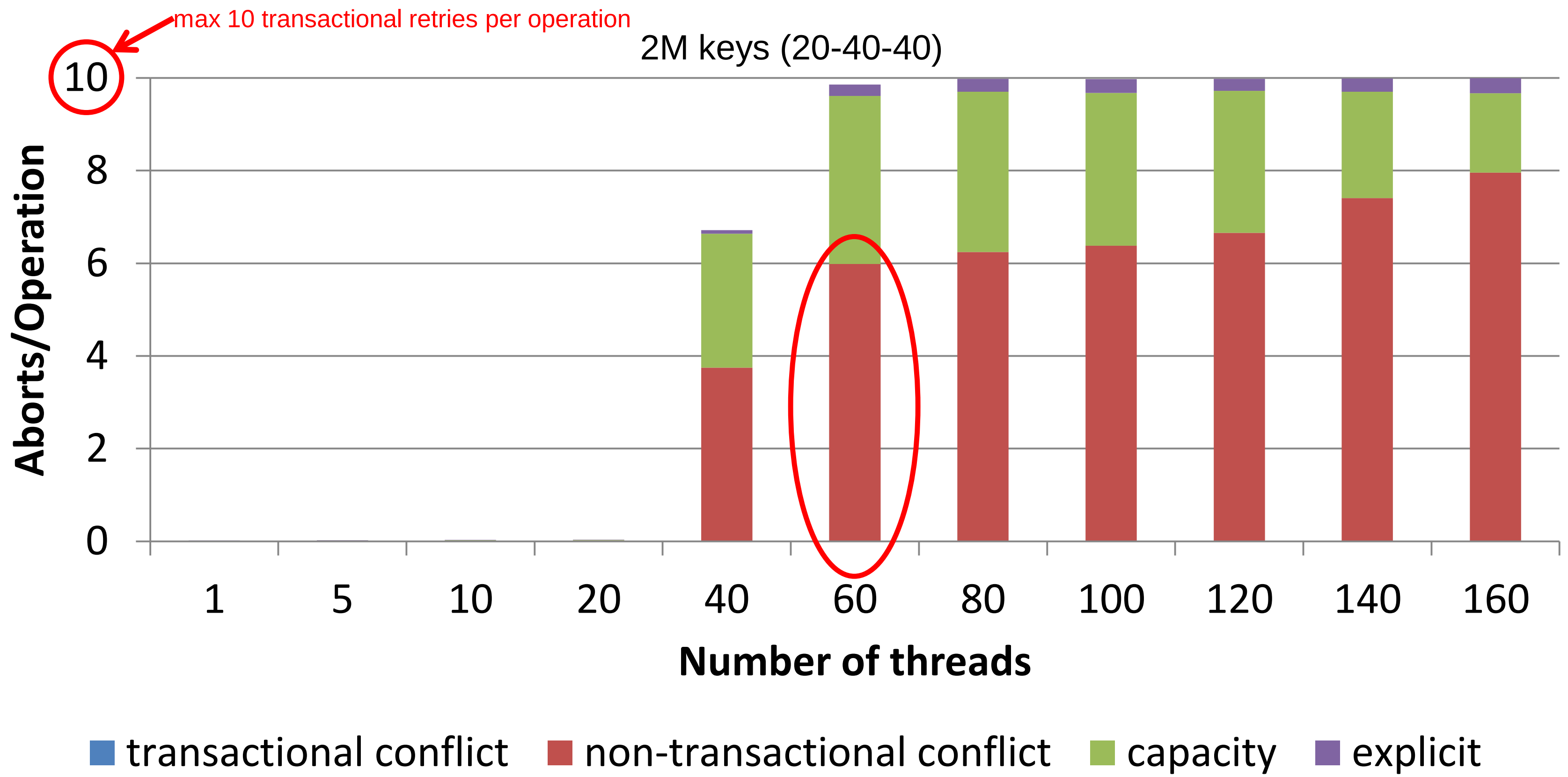
- ***transactional conflicts***: close to zero → non-conflicting operations on red-black tree
- ***non-transactional conflicts***: due to global lock acquisition from other threads
- ***capacity***: read-/write- set overflows
- ***explicit***: due to global lock found taken

Power8: bu-cg-htm aborts' breakdown



> 20 threads → capacity aborts due to sharing of TM buffers between multiple SMT contexts

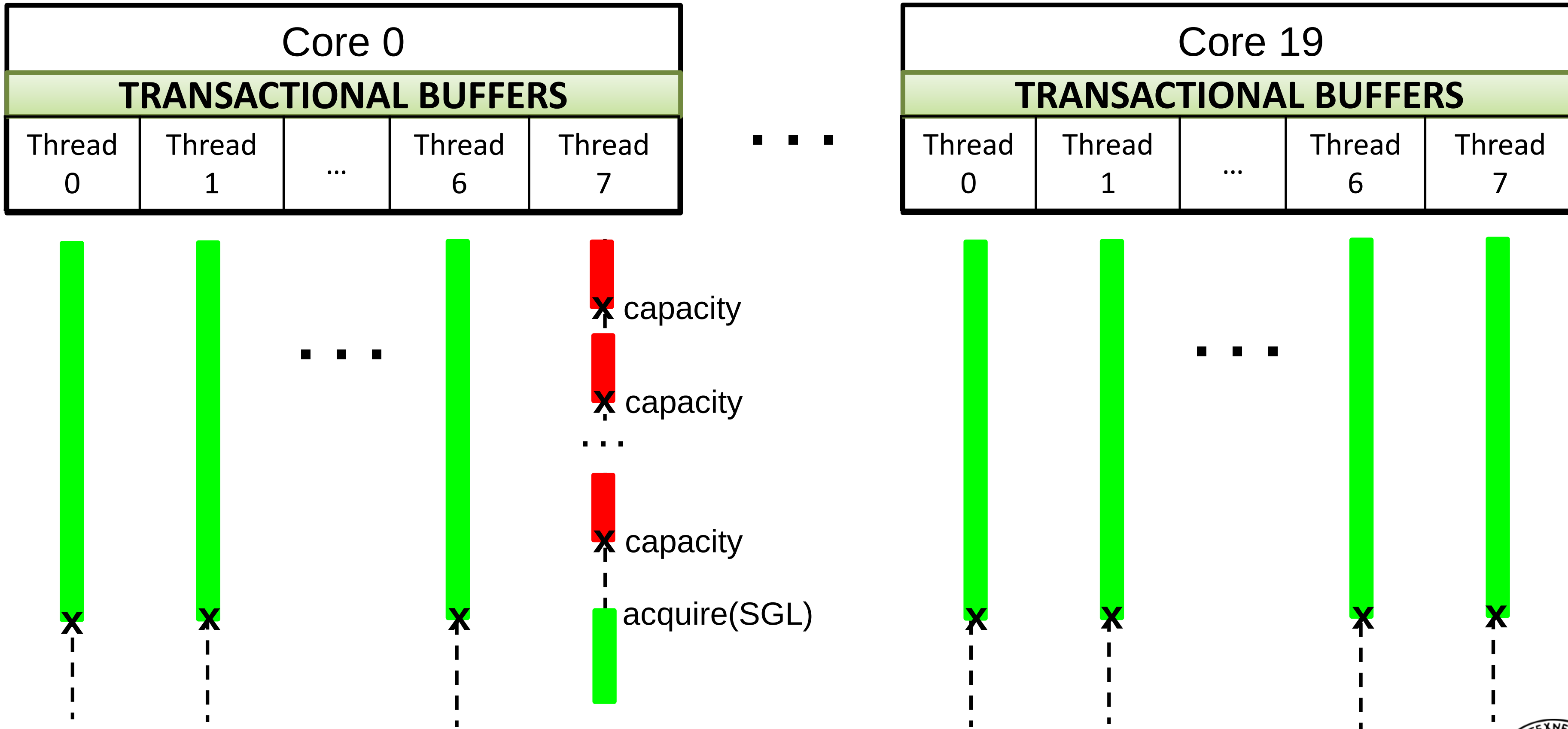
Power8: bu-cg-htm aborts' breakdown



> 20 threads → capacity aborts due to sharing of TM buffers between multiple SMT contexts
Capacity aborts → SGL acquisitions → non-transactional conflict aborts

1st optimization: PCL Fallback

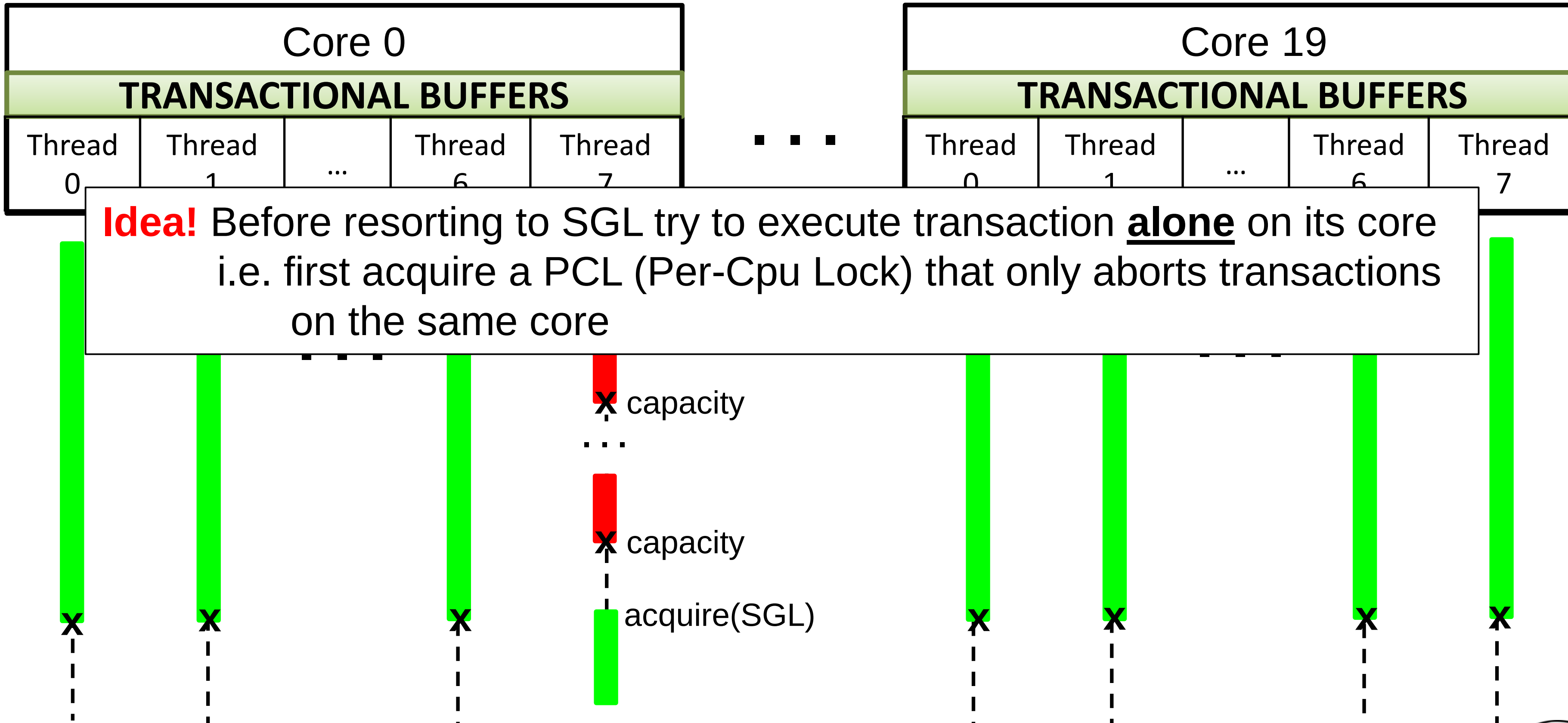
SGL Fallback observation:
When some thread acquires the global lock **all** concurrent transactions are aborted. Even those executed on other cores than the one acquiring the lock (which do not share hardware transactional buffers).



National Technical University of Athens CSLab

1st optimization: PCL Fallback

SGL Fallback observation:
When some thread acquires the global lock **all** concurrent transactions are aborted. Even those executed on other cores than the one acquiring the lock (which do not share hardware transactional buffers).

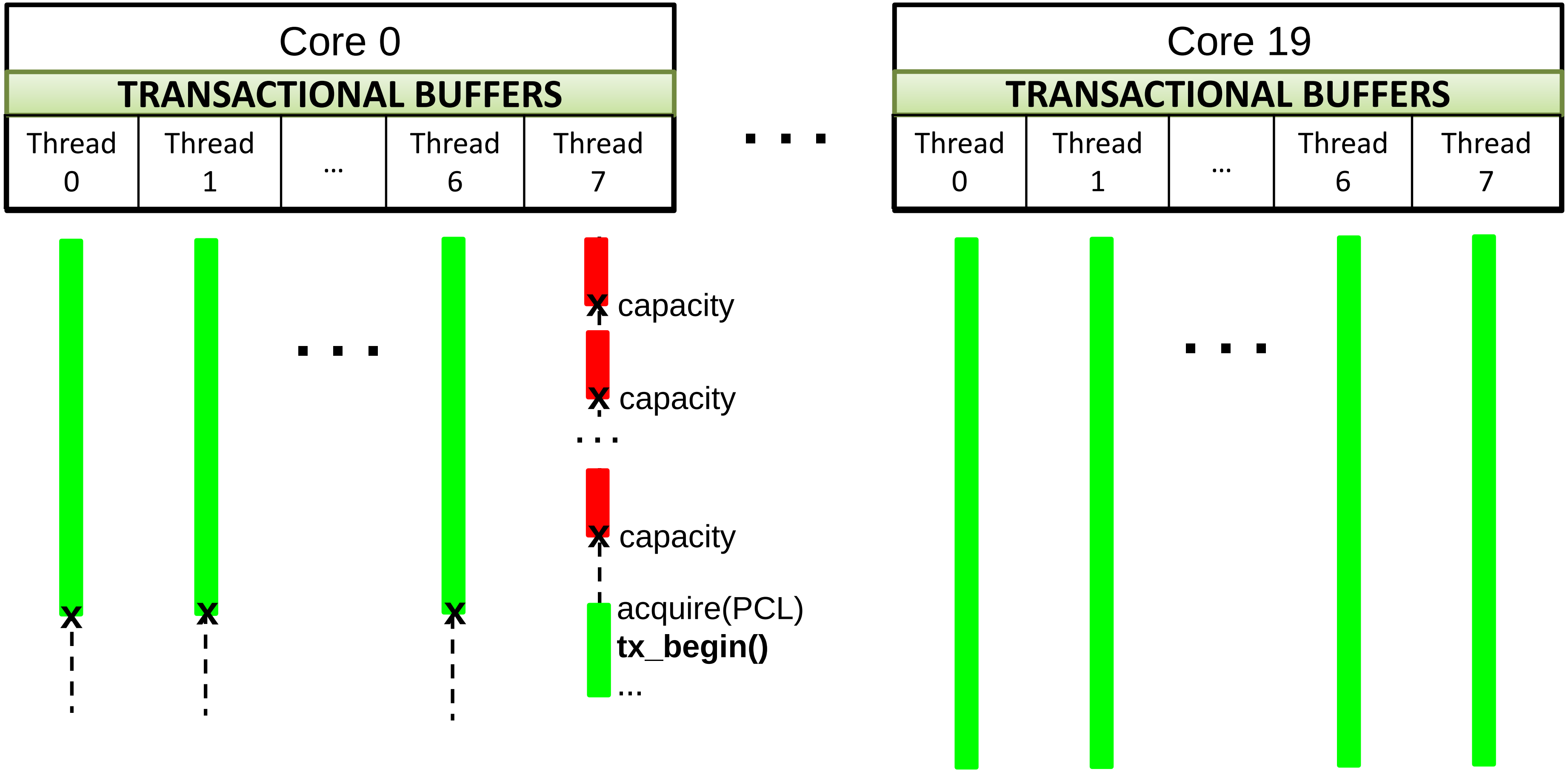


Idea! Before resorting to SGL try to execute transaction **alone** on its core i.e. first acquire a PCL (Per-Cpu Lock) that only aborts transactions on the same core

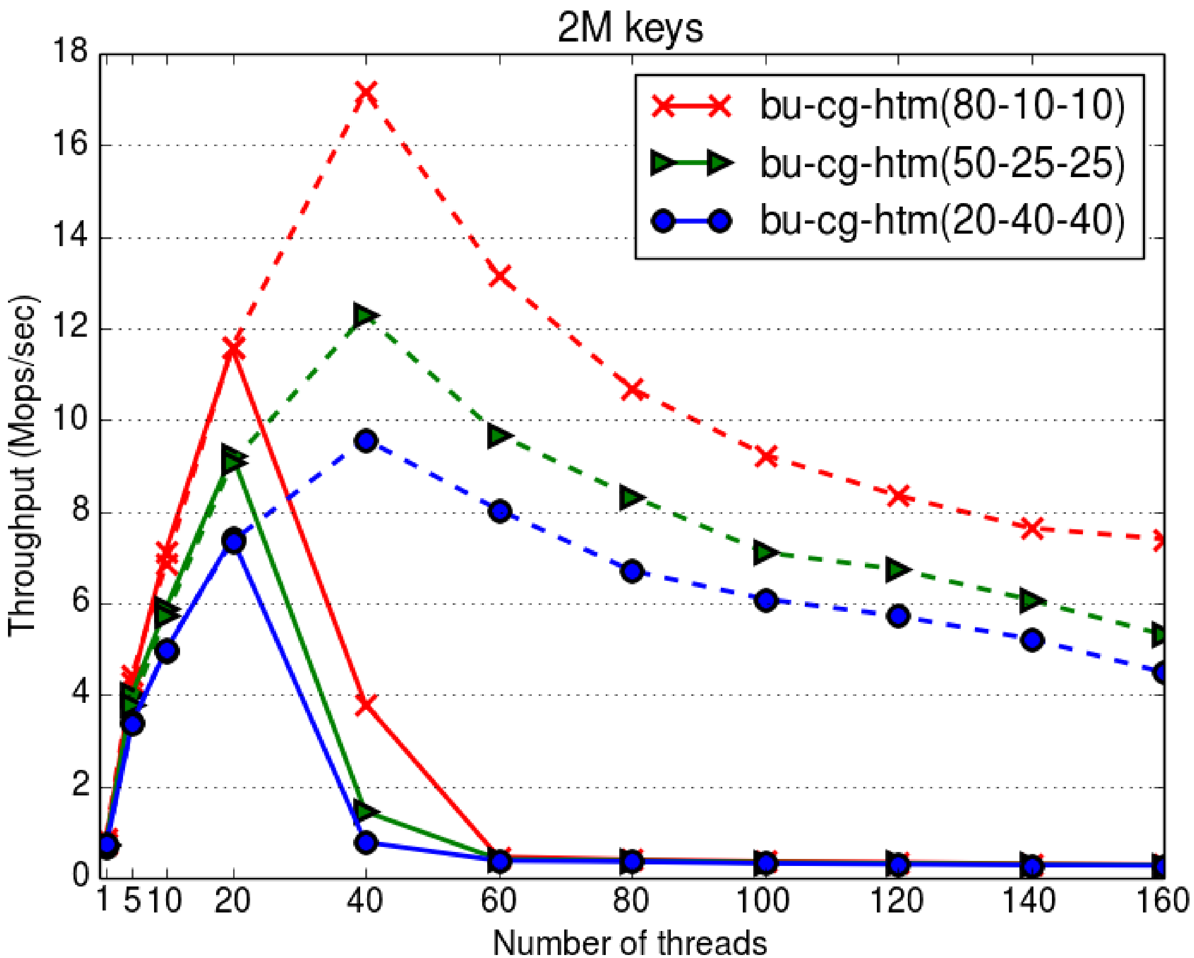
National Technical University of Athens CSLab

1st optimization: PCL Fallback

Idea! Before resorting to SGL try to execute transaction **alone** on its core
i.e. first acquire a PCL (Per-Cpu Lock) that only aborts siblings

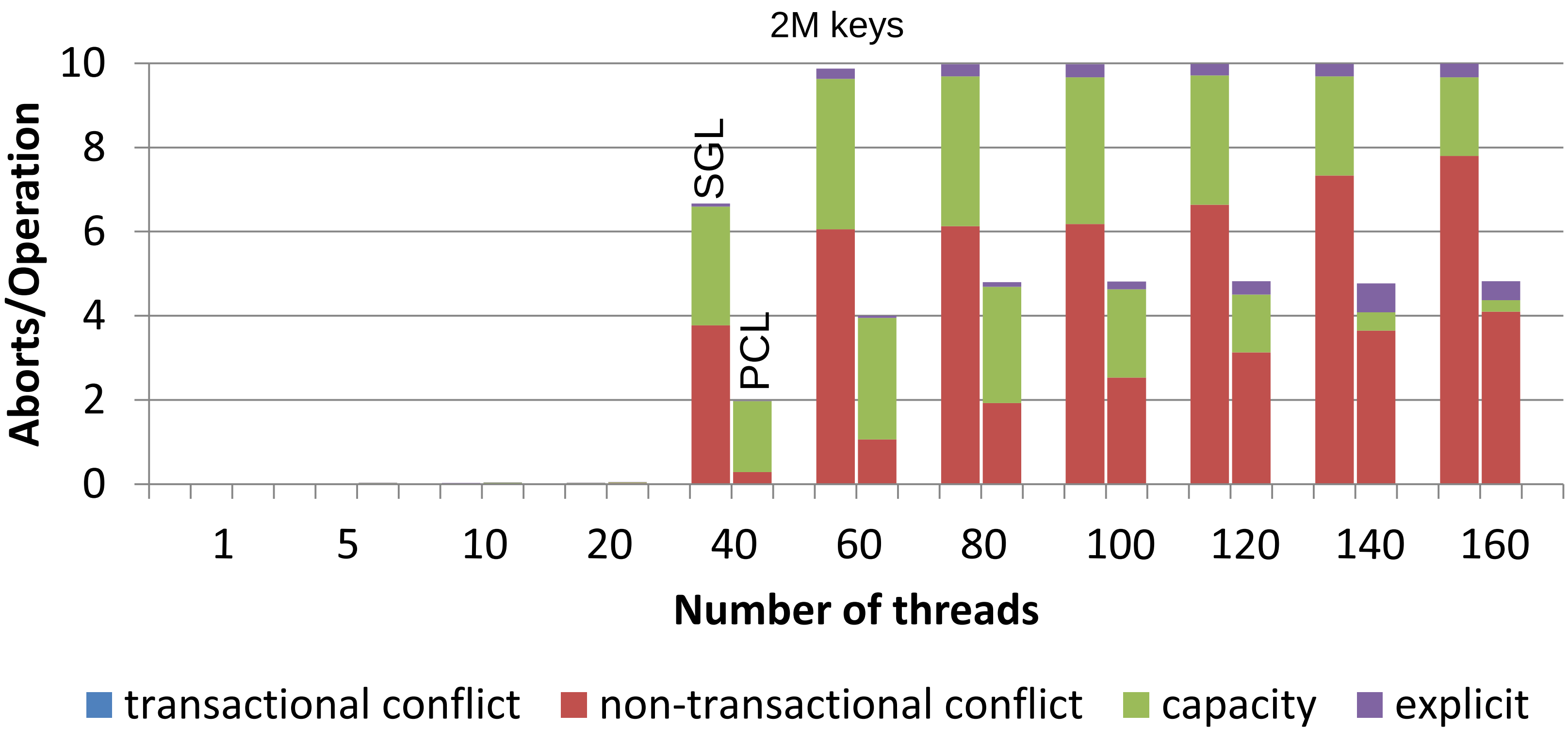


PCL Fallback: throughput

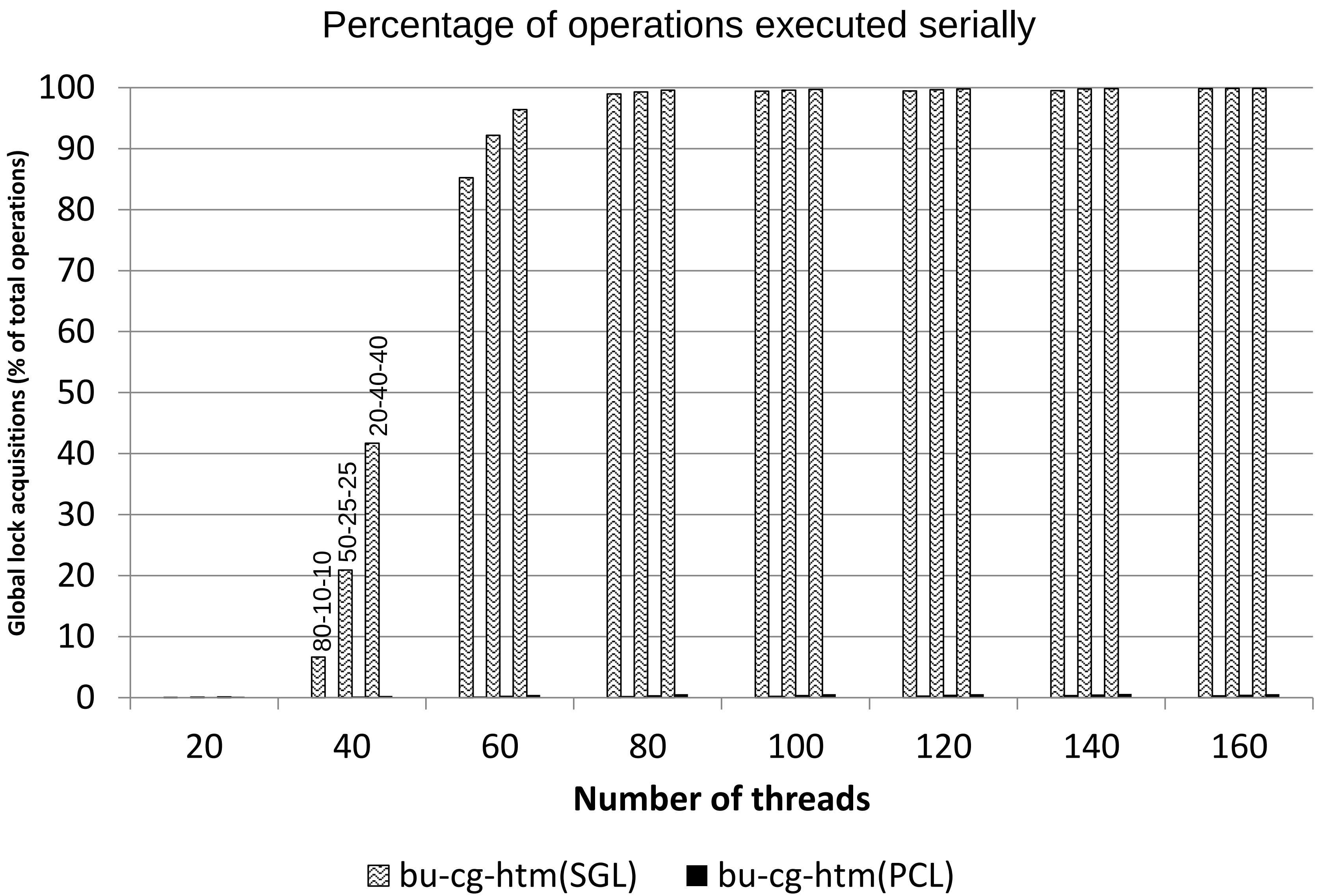


solid lines: SGL fallback
dotted lines: PCL fallback

PCL Fallback: aborts' breakdown

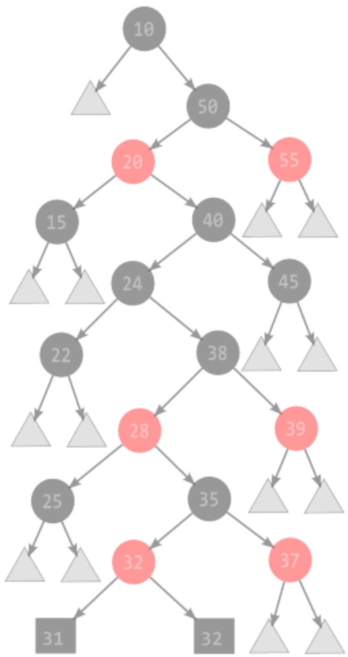


PCL Fallback: lock acquisitions



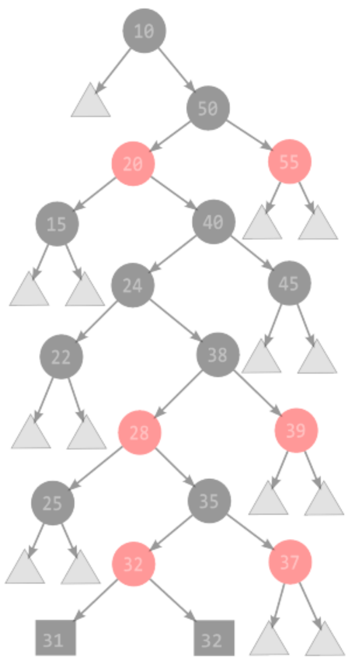
2nd optimization: Fine-grained transactions

- PCL leads to underutilization (1 thread per core executes)
- PCL does not solve the problem of capacity aborts
- Naively enclosing each RBT operation in an HTM transaction results in large transactions
- Review example: insert(30)



2nd optimization: Fine-grained transactions

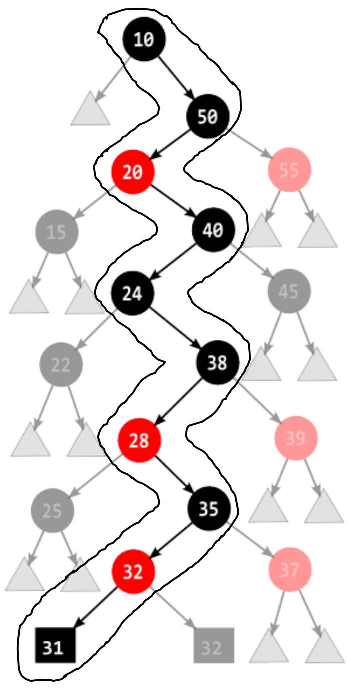
- PCL leads to underutilization (1 thread per core executes)
- PCL does not solve the problem of capacity aborts
- Naively enclosing each RBT operation in an HTM transaction results in large transactions
- Review example: insert(30)



Each transaction in bu-cg-htm encloses 2 phases:

2nd optimization: Fine-grained transactions

- PCL leads to underutilization (1 thread per core executes)
- PCL does not solve the problem of capacity aborts
- Naively enclosing each RBT operation in an HTM transaction results in large transactions
- Review example: insert(30)

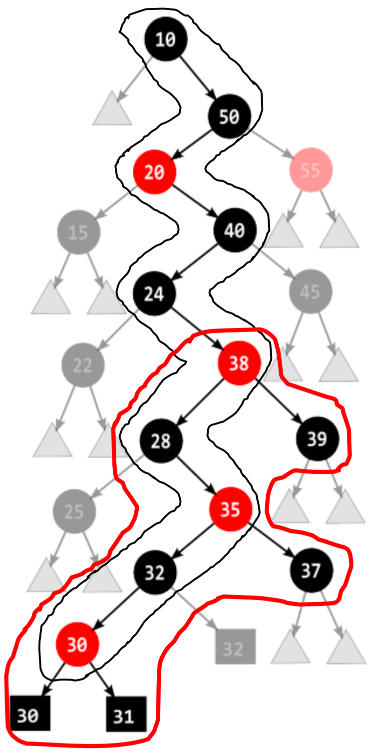


Each transaction in bu-cg-htm encloses 2 phases:

1. Lookup phase

2nd optimization: Fine-grained transactions

- PCL leads to underutilization (1 thread per core executes)
- PCL does not solve the problem of capacity aborts
- Naively enclosing each RBT operation in an HTM transaction results in large transactions
- Review example: insert(30)

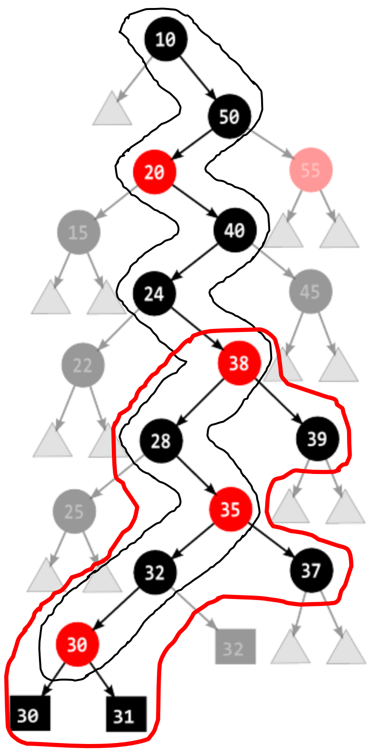


Each transaction in bu-cg-htm encloses 2 phases:

1. Lookup phase
2. Rebalance phase

2nd optimization: Fine-grained transactions

- PCL leads to underutilization (1 thread per core executes)
- PCL does not solve the problem of capacity aborts
- Naively enclosing each RBT operation in an HTM transaction results in large transactions
- Review example: insert(30)

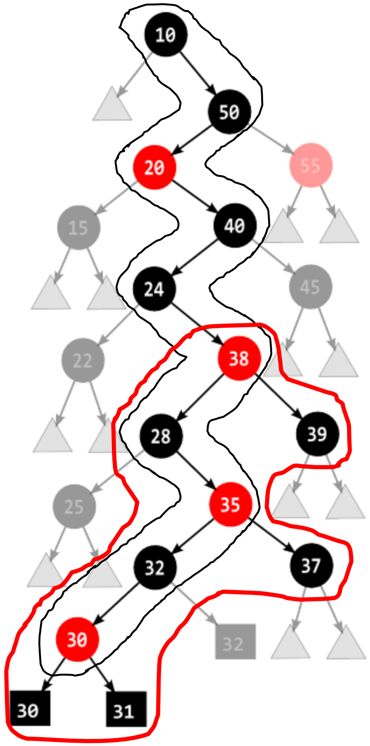


Each transaction in bu-cg-htm encloses 2 phases:
1. Lookup phase
2. Rebalance phase

Transactional size of each phase:
1. Lookup: proportional to the average depth of the tree (typically 20-30 levels)
2. Rebalance: ~97% times < 3 levels, ~75% 1 level

2^{nd} optimization: Fine-grained transactions

- PCL leads to underutilization (1 thread per core executes)
- PCL does not solve the problem of capacity aborts
- Naively enclosing each RBT operation in an HTM transaction results in large transactions
- Review example: insert(30)



Each transaction in bu-cg-htm encloses 2 phases:

1. Lookup phase
2. Rebalance phase

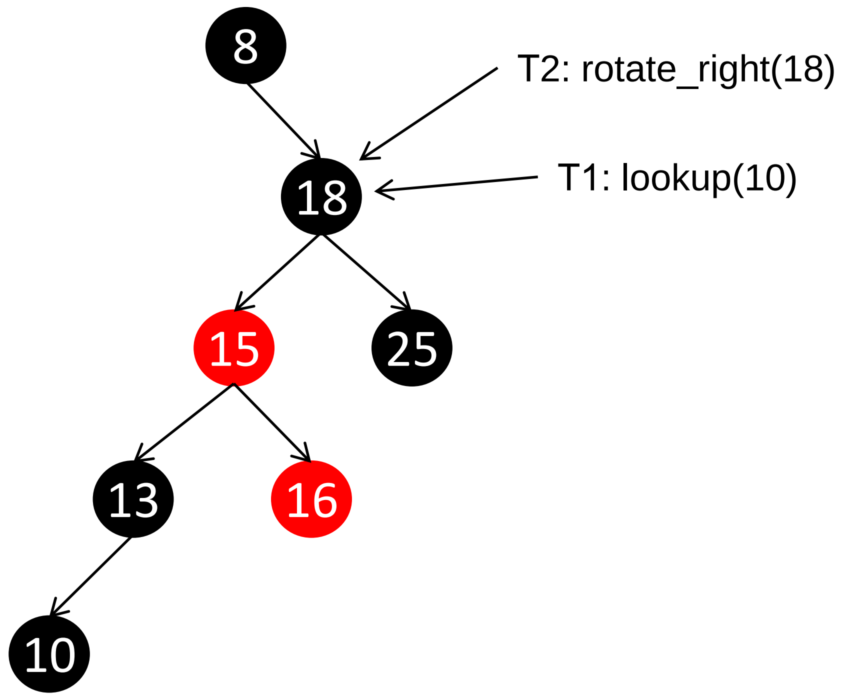
Transactional size of each phase:

1. Lookup: proportional to the average depth of the tree (typically 20-30 levels)
2. Rebalance: ~97% times < 3 levels, ~75% 1 level

Idea! Split lookup phase in multiple shorter transactions

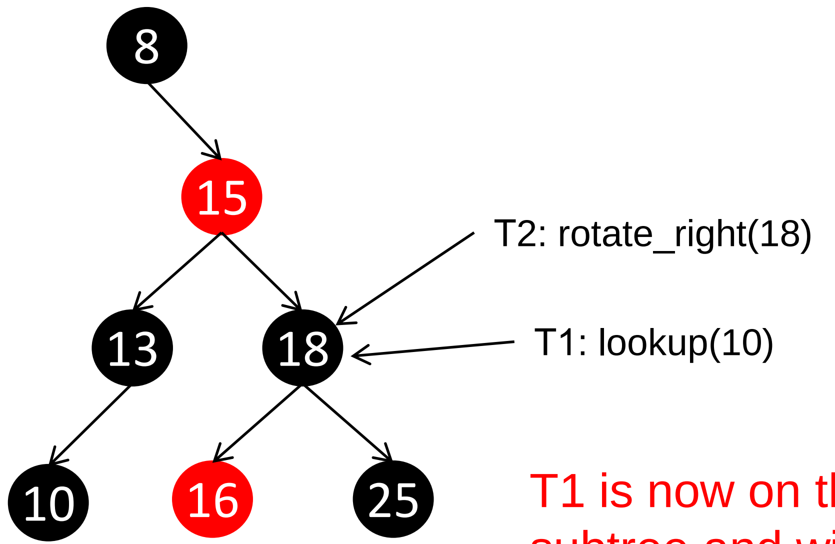
2nd optimization: Fine-grained transactions

- Splitting lookup phase in multiple transactions is not straightforward
- What can go wrong?



2^{nd} optimization: Fine-grained transactions

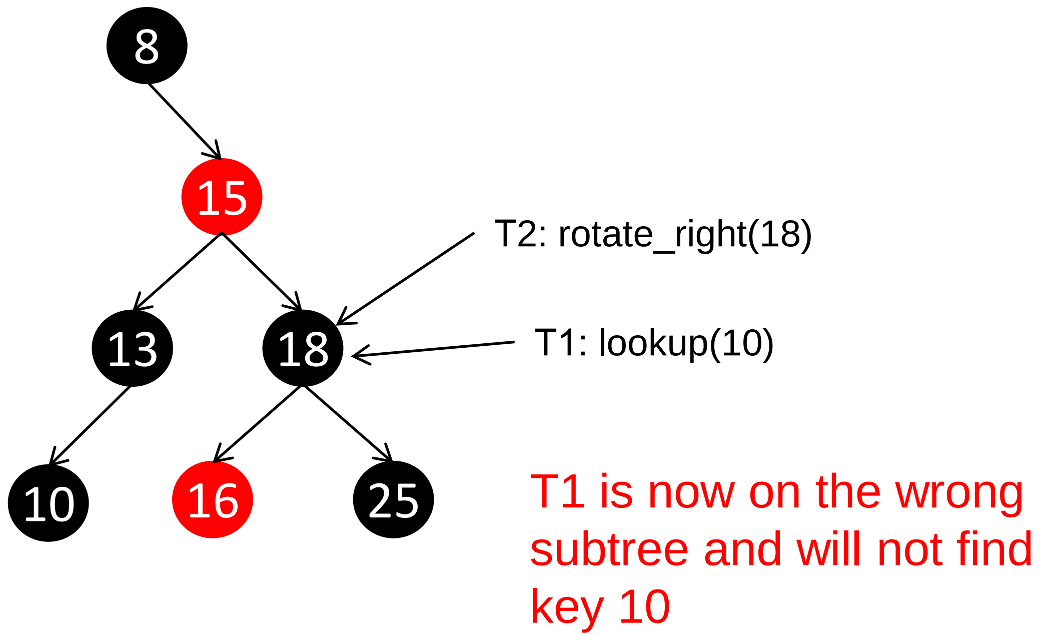
- Splitting lookup phase in multiple transactions is not straightforward
- What can go wrong?



T1 is now on the wrong subtree and will not find key 10

2nd optimization: Fine-grained transactions

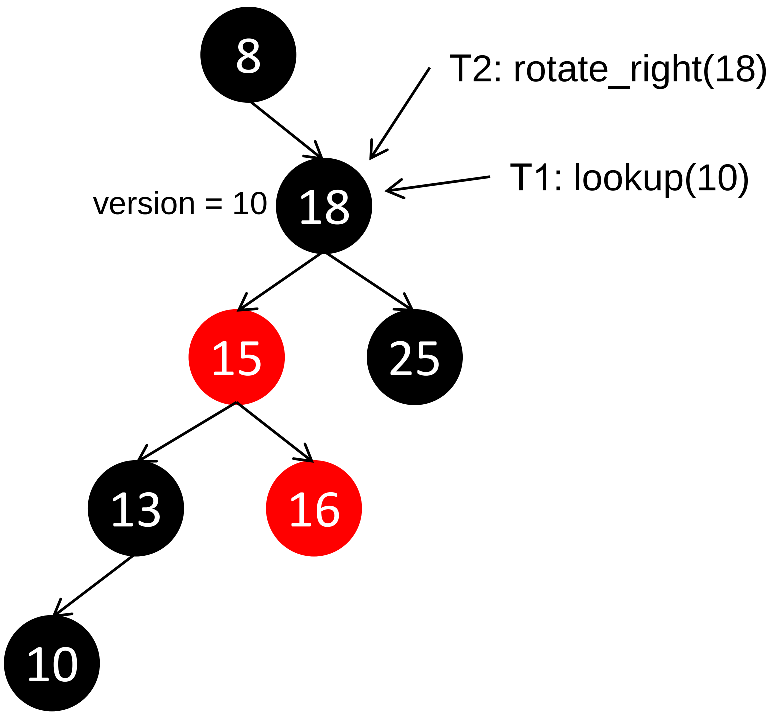
- Splitting lookup phase in multiple transactions is not straightforward
- What can go wrong?



- We need a way to inform threads about concurrent modifications.
  - we add a *version number* on each node

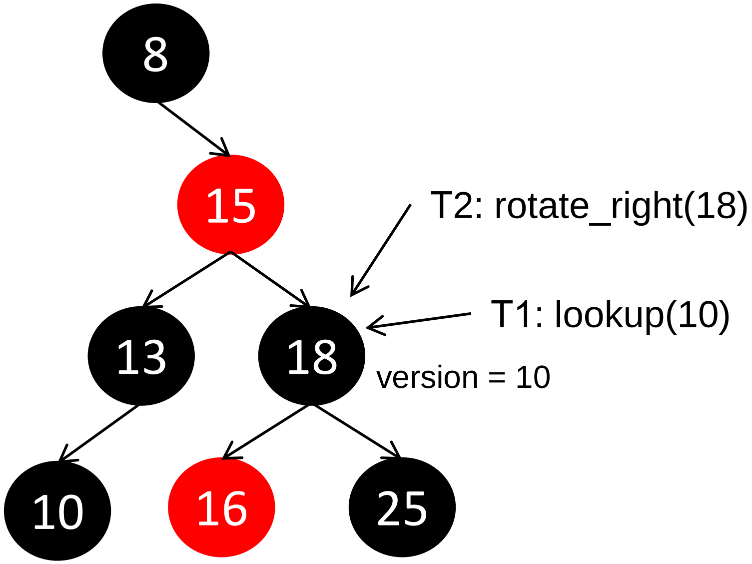
2nd optimization: Fine-grained transactions

- A *version number* is added on each node
- Version number is increased when node is modified
- Fine-grained transactions:
  1. Validate the version of current node
  2. If current node has changed, abort and restart operation from root
  3. Otherwise, move to next node, read its version and commit



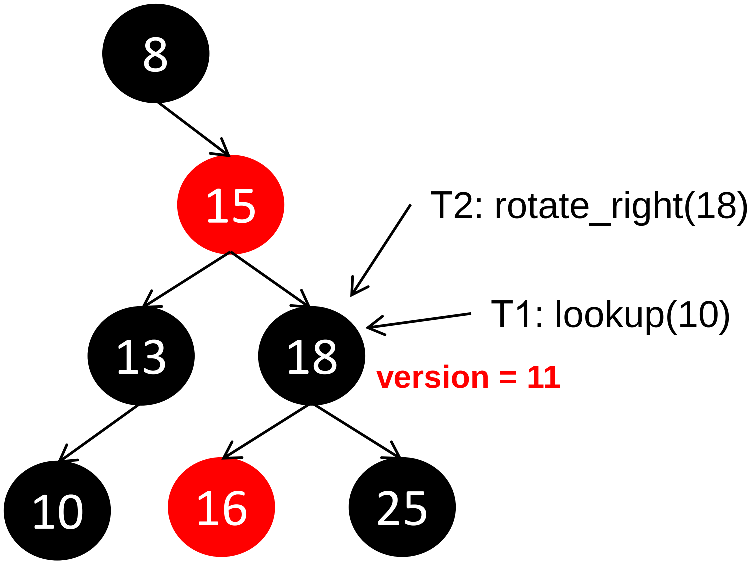
2^{nd} optimization: Fine-grained transactions

- A *version number* is added on each node
- Version number is increased when node is modified
- Fine-grained transactions:
  1. Validate the version of current node
  2. If current node has changed, abort and restart operation from root
  3. Otherwise, move to next node, read its version and commit



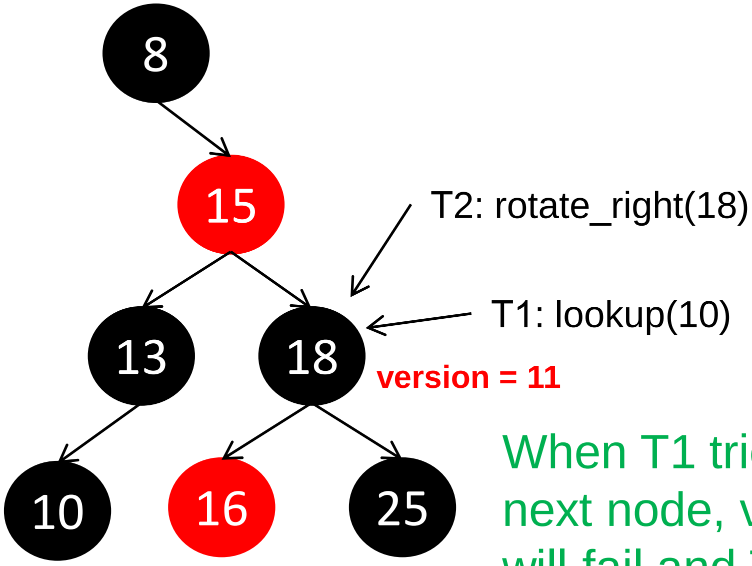
2nd optimization: Fine-grained transactions

- A *version number* is added on each node
- Version number is increased when node is modified
- Fine-grained transactions:
  1. Validate the version of current node
  2. If current node has changed, abort and restart operation from root
  3. Otherwise, move to next node, read its version and commit



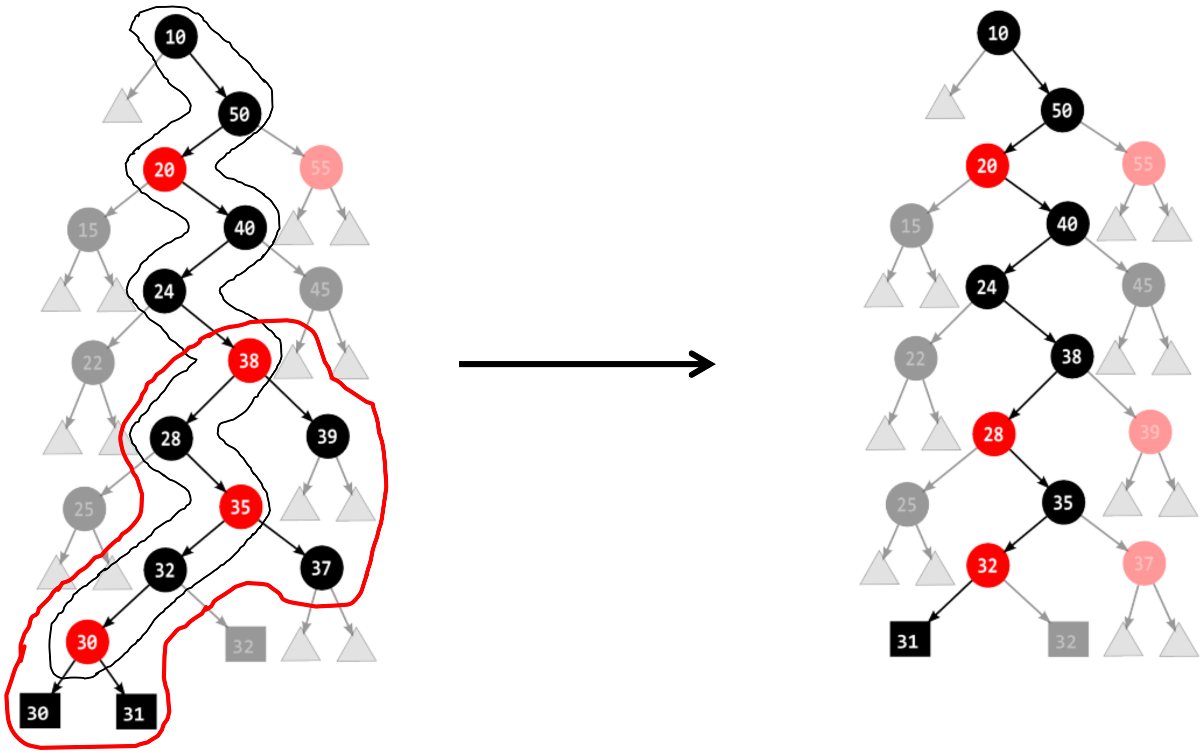
2^{nd} optimization: Fine-grained transactions

- A *version number* is added on each node
- Version number is increased when node is modified
- Fine-grained transactions:
    1. Validate the version of current node
    2. If current node has changed, abort and restart operation from root
    3. Otherwise, move to next node, read its version and commit



When T1 tries to move to the next node, validation of node 18 will fail and T1 will restart from root

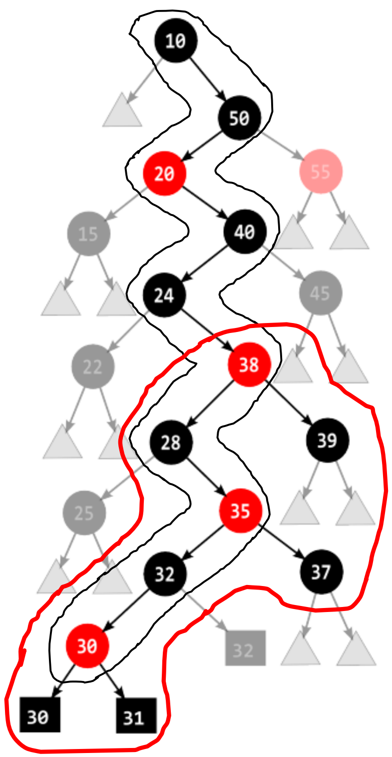
2nd optimization: Fine-grained transactions

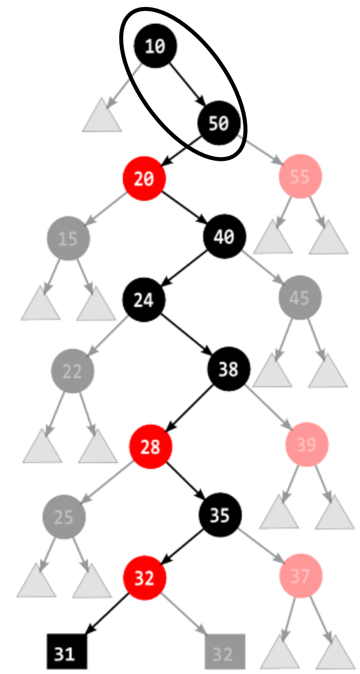


Coarse-grained transactions
<u>1 large transaction</u>

Fine-grained transactions
<u>Multiple short transactions</u>

2nd optimization: Fine-grained transactions



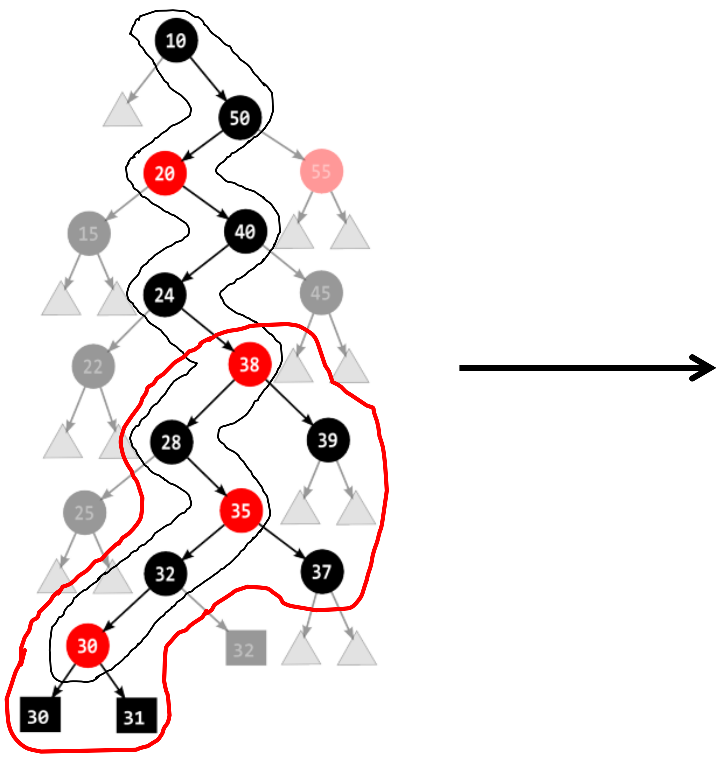


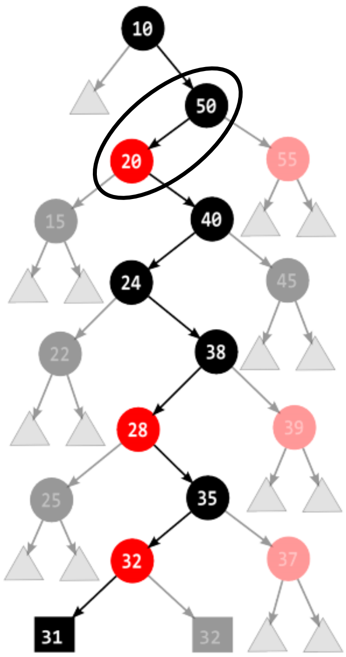
Coarse-grained transactions
1 large transaction

Fine-grained transactions
Multiple short transactions

National Technical University of Athens CSLab

2nd optimization: Fine-grained transactions

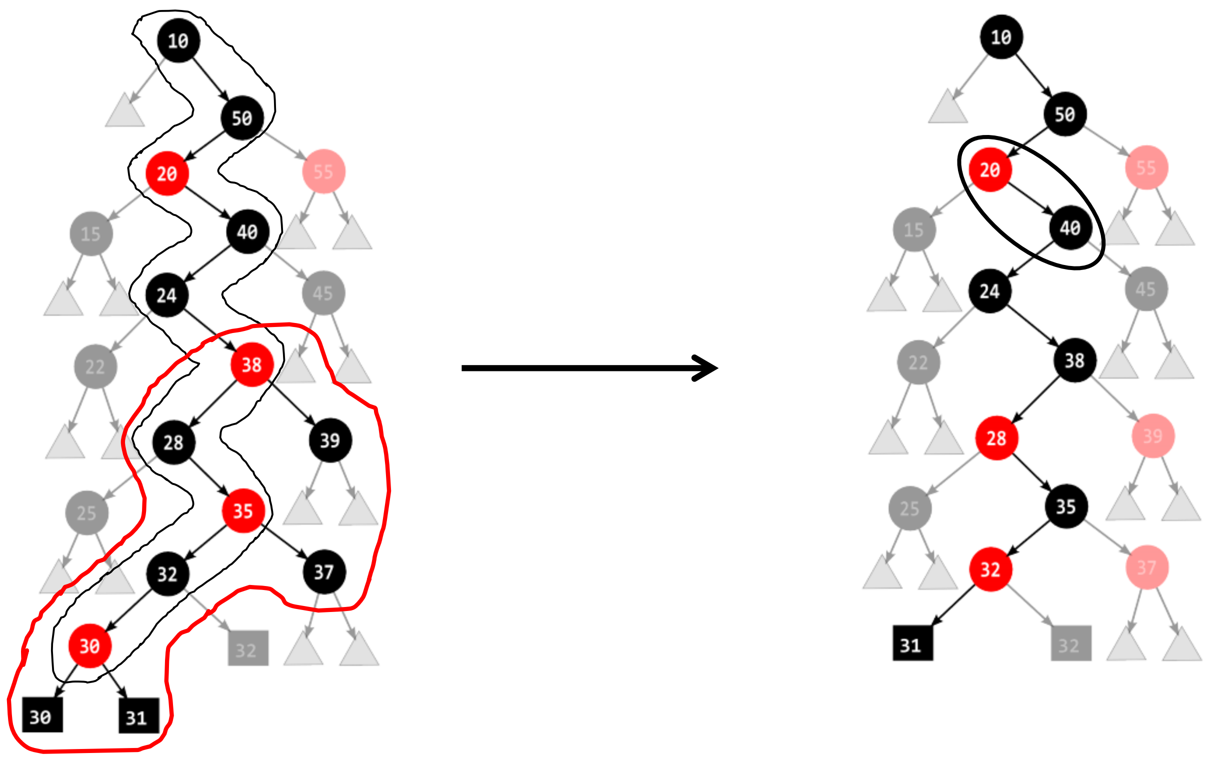




Coarse-grained transactions
1 large transaction

Fine-grained transactions
Multiple short transactions

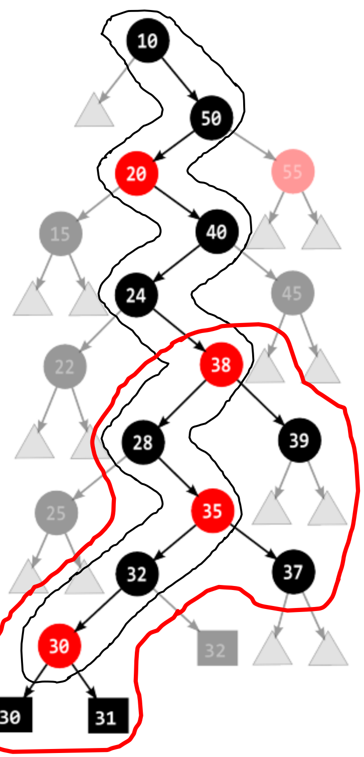
2nd optimization: Fine-grained transactions

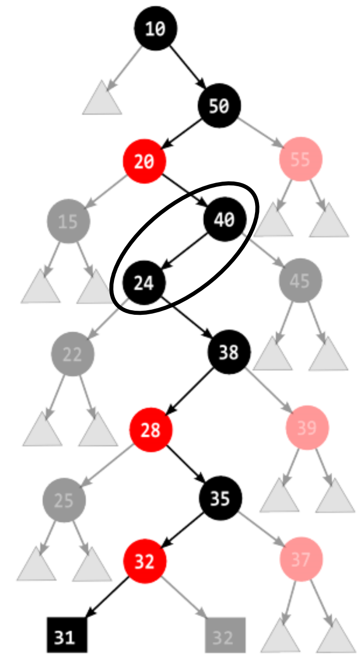


Coarse-grained transactions
1 large transaction

Fine-grained transactions
Multiple short transactions

2nd optimization: Fine-grained transactions

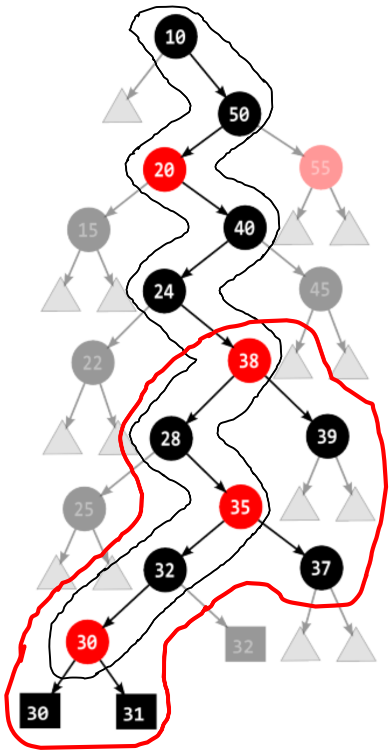


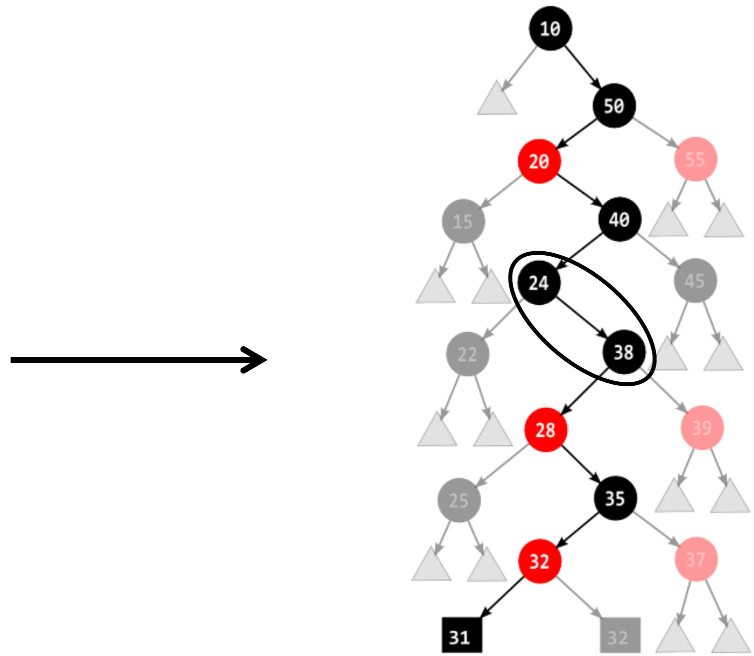


Coarse-grained transactions
1 large transaction

Fine-grained transactions
Multiple short transactions

2nd optimization: Fine-grained transactions



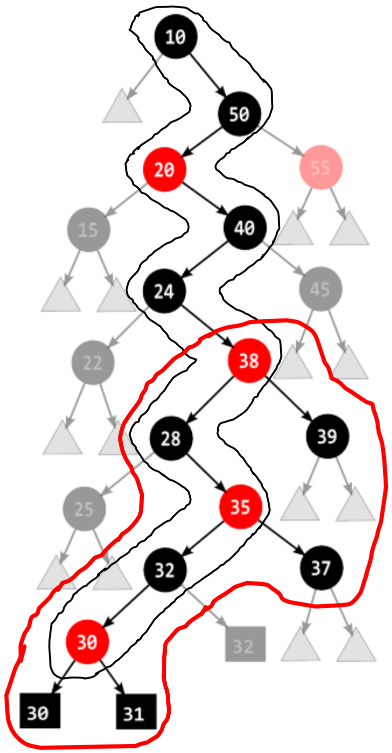


Coarse-grained transactions
1 large transaction

Fine-grained transactions
Multiple short transactions

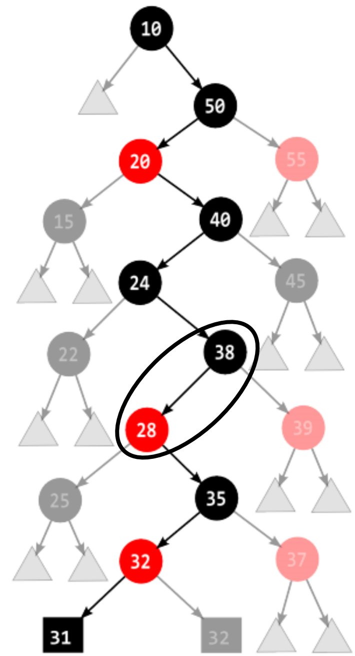
National Technical University of Athens CSLab

2nd optimization: Fine-grained transactions



Coarse-grained transactions

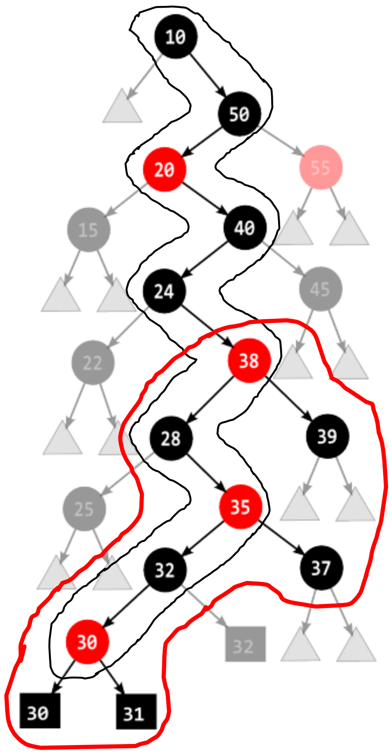
<u>1 large transaction</u>



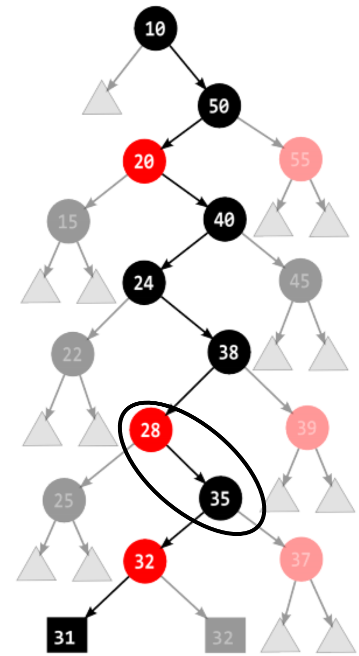
Fine-grained transactions

<u>Multiple short transactions</u>

2nd optimization: Fine-grained transactions

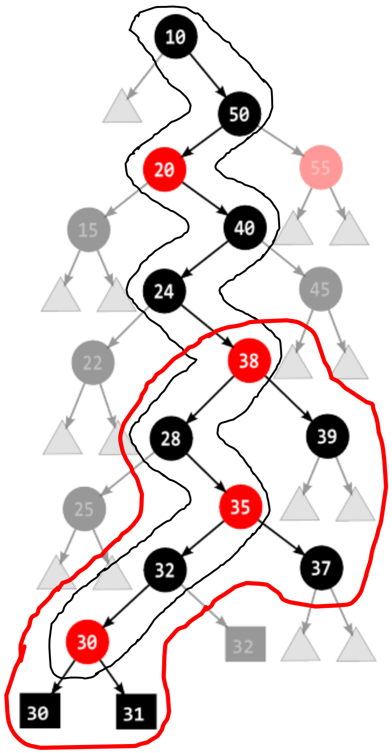


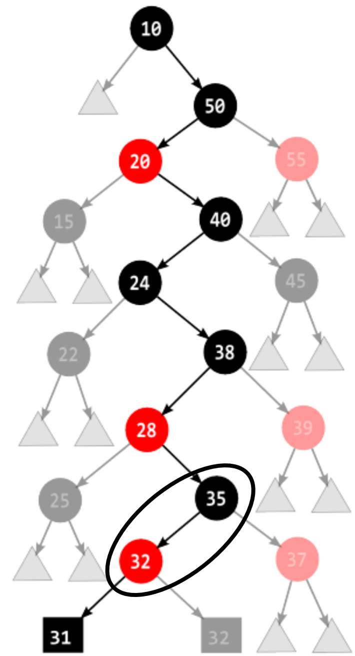
Coarse-grained transactions
<u>1 large transaction</u>



Fine-grained transactions
<u>Multiple short transactions</u>

2nd optimization: Fine-grained transactions



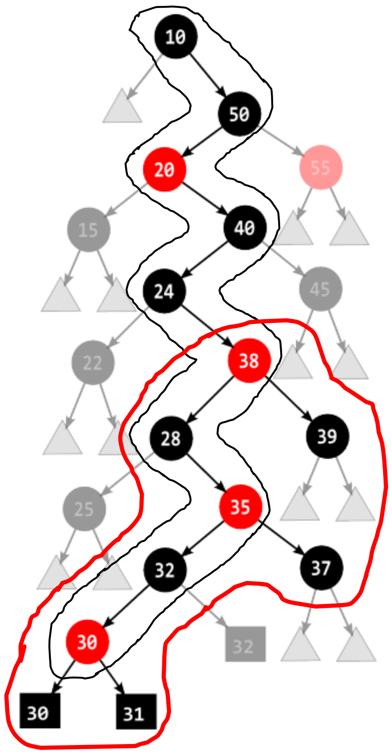


Coarse-grained transactions
<u>1 large transaction</u>

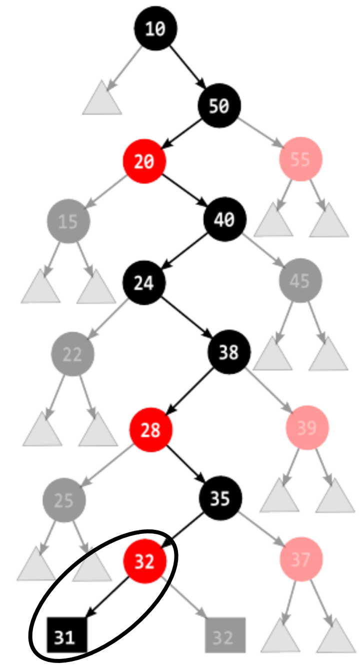
Fine-grained transactions
<u>Multiple short transactions</u>

National Technical University of Athens CSLab

2nd optimization: Fine-grained transactions

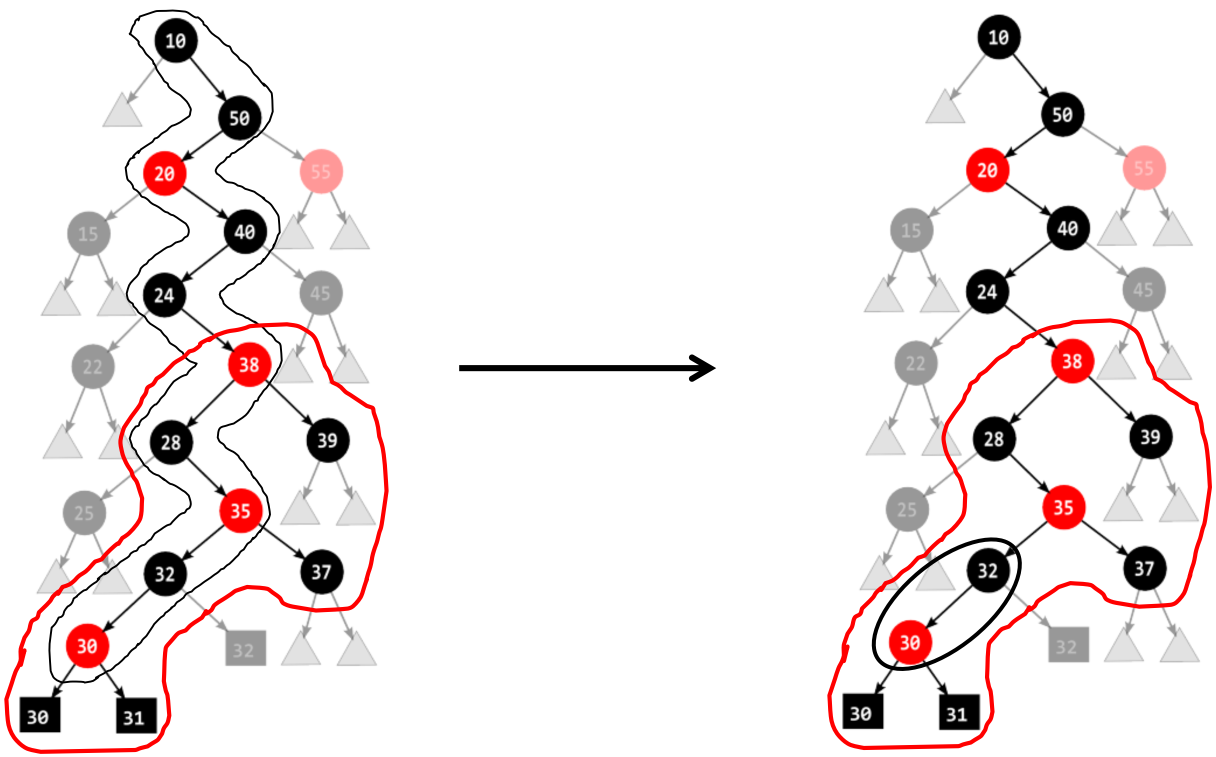


Coarse-grained transactions
1 large transaction



Fine-grained transactions
Multiple short transactions

2nd optimization: Fine-grained transactions

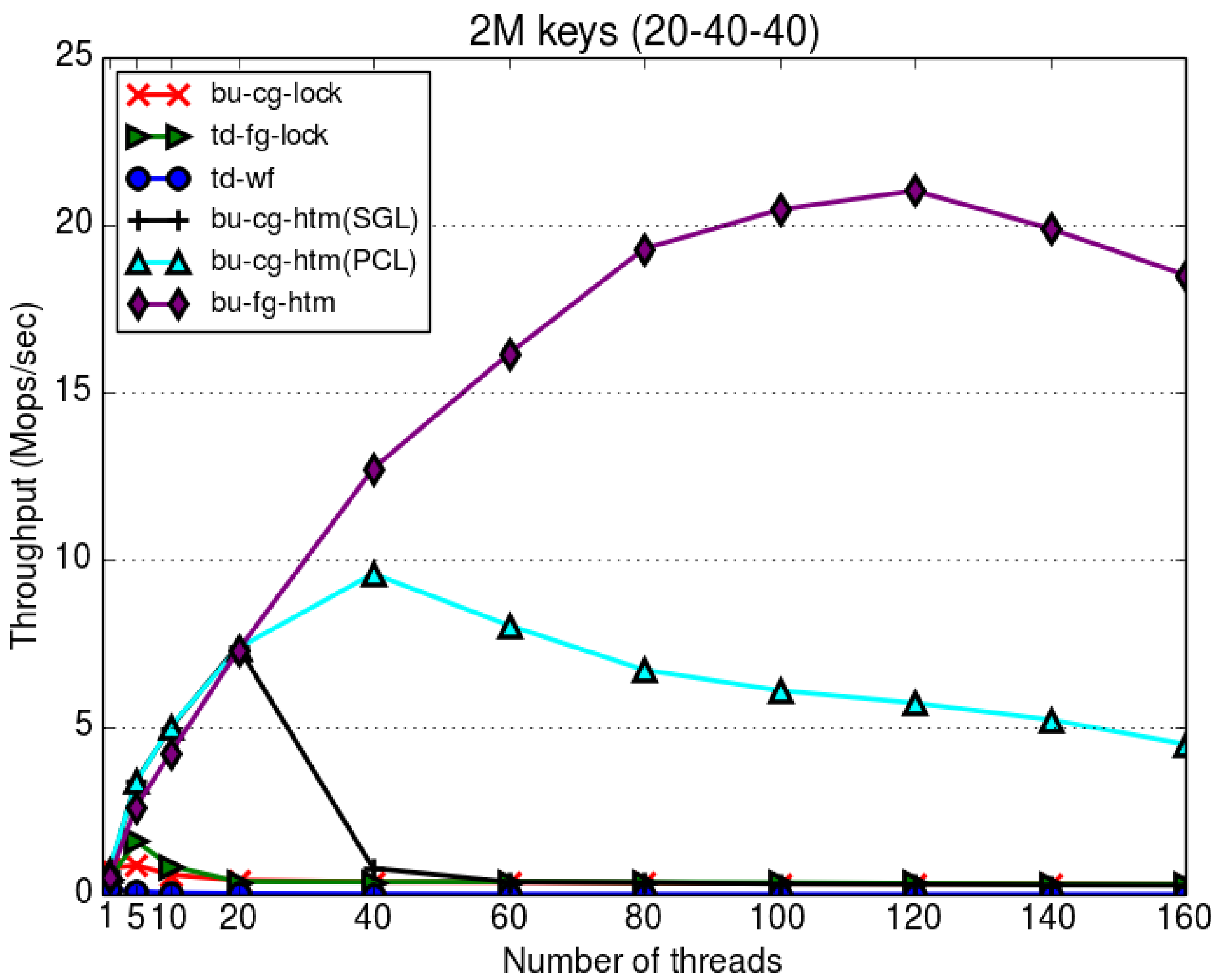


Coarse-grained transactions
1 large transaction

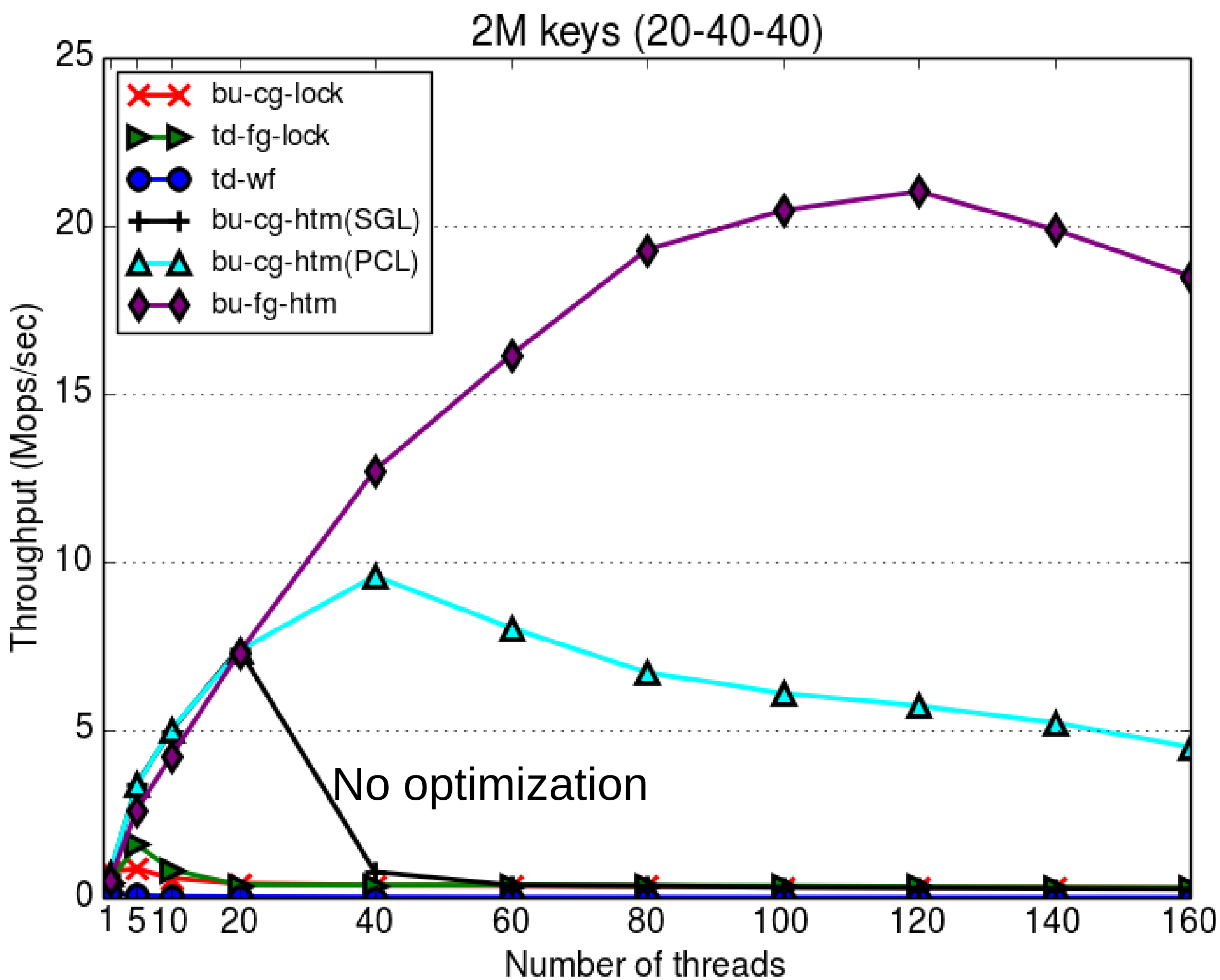
Fine-grained transactions
Multiple short transactions

National Technical University of Athens
CSLab

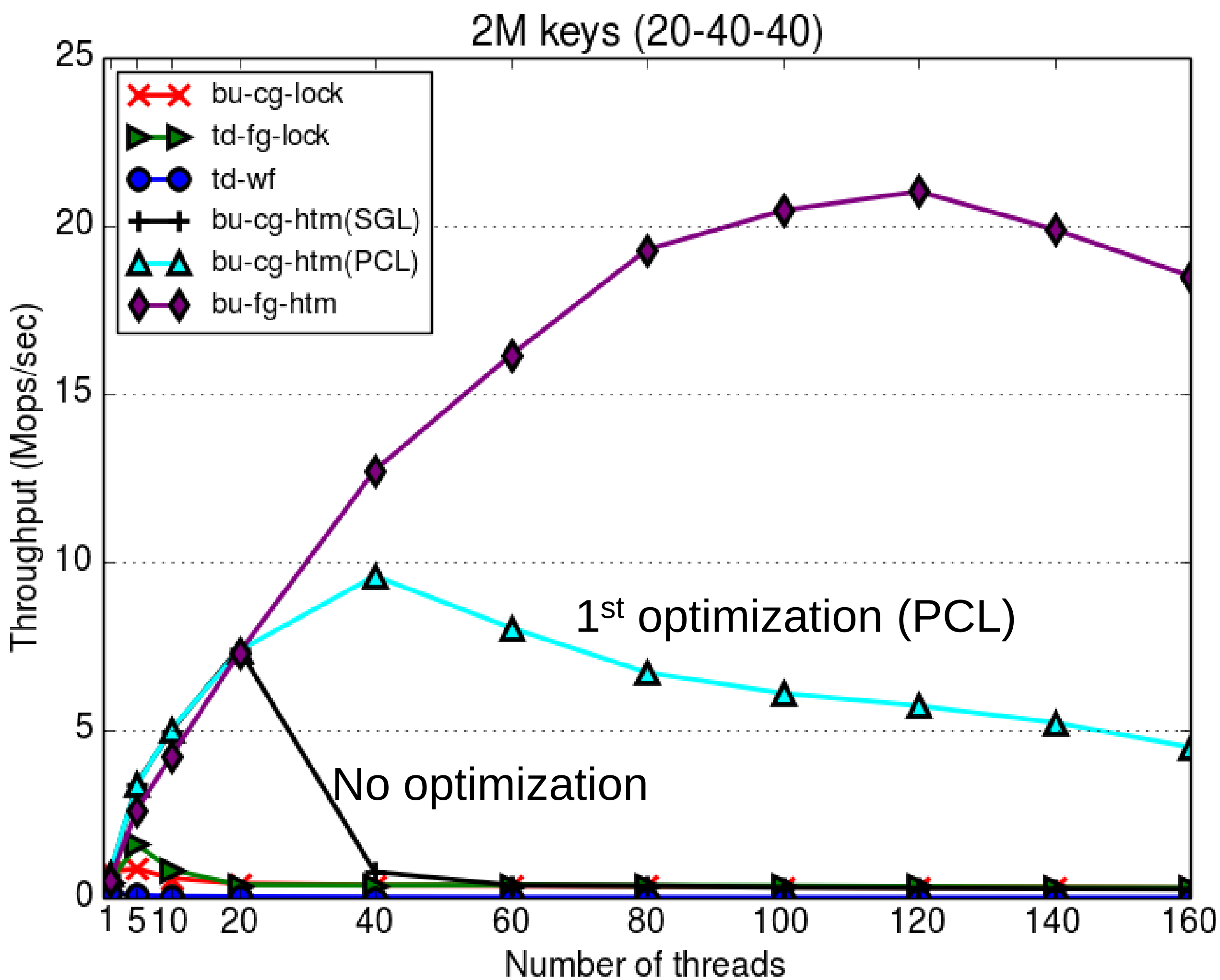
Fine-grained transactions: throughput



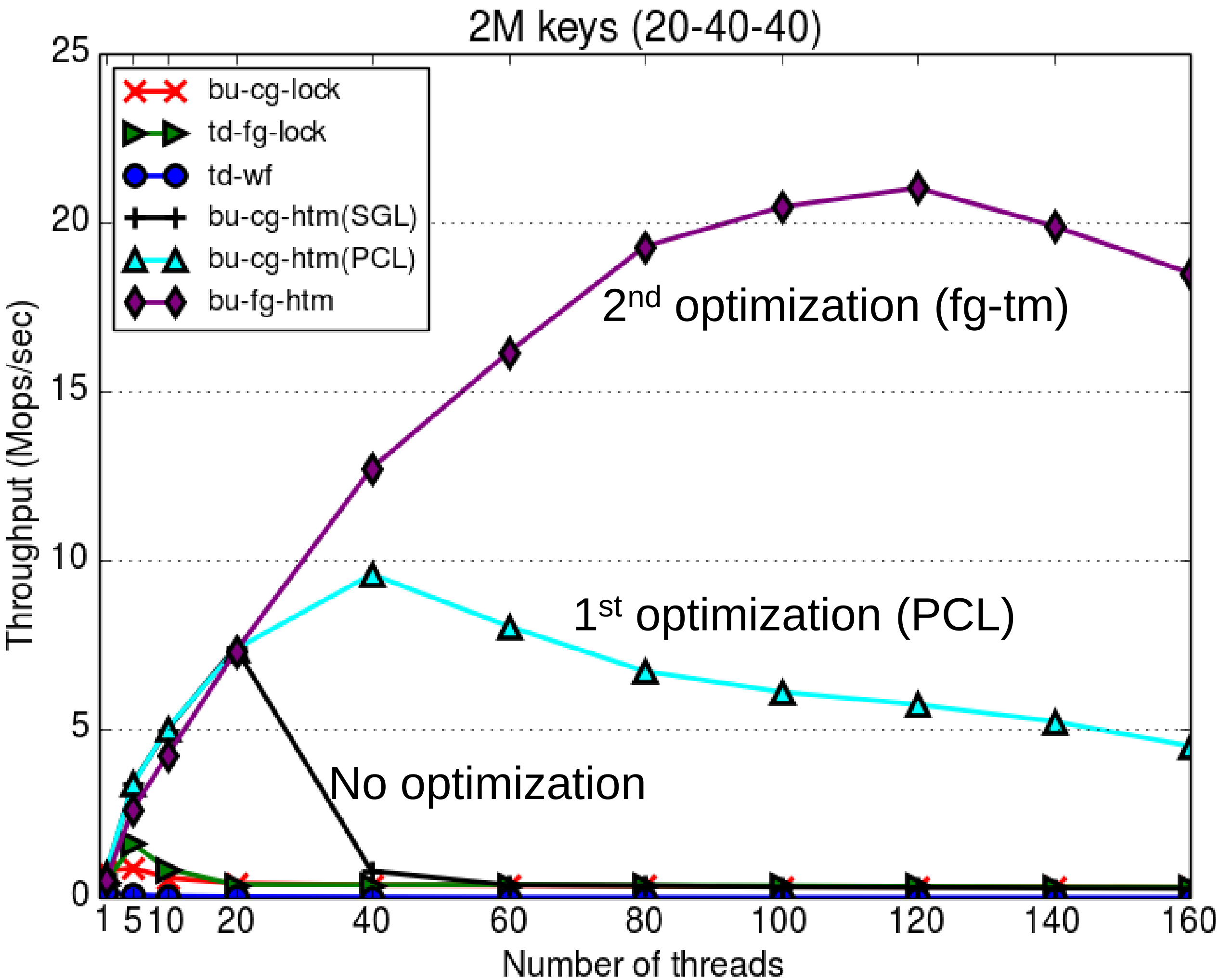
Fine-grained transactions: throughput



Fine-grained transactions: throughput



Fine-grained transactions: throughput



National Technical University of Athens CSLab

Fine-grained transactions: aborts/operation

#Threads	bu-cg-htm			bu-fg-htm		
	2M	**20M**	**100M**	**2M**	**20M**	**100M**
20	0.02	0.23	0.27	0.001	0.002	0.007
40	6.2	9.7	9.9	0.001	0.003	0.007
60	9.8	9.9	9.9	0.002	0.004	0.01
80	9.9	9.9	9.9	0.009	0.008	0.012
120	9.9	9.9	9.9	0.24	0.17	0.11
160	9.9	9.9	9.9	1.15	1.07	0.85

- *bu-fg-htm* manages to keep very low abort rate, independently of tree size

Conclusions & Future work

- Programming with HTM might be simple
  - achieving high performance is not
  - hardware limitations need to considered
  - different HTM systems need different software optimizations
- Future Work
  - Extend evaluation to more data structures / algorithms (e.g. graph algorithms)
  - Evaluation on machines with more physical cores / hardware threads

THANK YOU!

QUESTIONS?

ACKNOWLEDGMENT

Intel Corporation and **IBM Hellas** for kindly providing the two servers.
I-PARTS project of Action ARISTEIA, co-financed by European Union (European Social Fund) and Hellenic national funds through the Operational Program Education and Lifelong Learning (NSRF 2007-2013).