Relativistic Red-Black Trees

Philip Howard
10/31/2011
phil.w.howard@gmail.com
Red-Black Trees

• **External Property**: Every external node is black

• **Internal Property**: The children of red nodes are black

• **Depth Property**: All external nodes have the same black depth
Reasons for RBTrees

- $O(\log(n))$ performance
- $O(1)$ restructures after insert/delete
- Possibly $O(\log(n))$ recolors (but recolors are cheap)
Insertion

• Search until we get to external node
• Insert at this location adding two empty, black, external nodes underneath
• If root, color black, else color red

• Preserves external, and depth properties; may violate internal property
Delete

• Find node
• If node doesn’t have an external child, swap with next node in in-order traversal order (left most node of the right branch)
• Delete the node from the bottom
Swap example: Delete 5
Diag Restructure

Before:

- A
  - B
    - C
      - 4
      - 3
      - 2
      - 1

After:

- B
  - A
    - 1
    - 2
  - C
    - 3
    - 4
Zig Restructure

Before:

```
A
  /  
B    4
  /
1
 /
2
 /
3
```

After:

```
B
  /  
A    C
  /    /
1    2
    /  /
3    4
```
Performance

• Find, Insert, Remove
  $O(\log(n))$
• At most 1 restructure on insert
• At most 2 restructures on delete
• Possibly $\log(n)$ recolors
Terminology

- synchronize_rcu
- rcu_assign_pointer
- rcu_dereference
- call_rcu
- call_rcu(free)
- wait-for-readers
- rp-publish
- rp-read
- defer-for-readers
- rp-free
Terminology

\[ S_n \] the Start of operation \( n \).

\[ F_n \] the Finish of operation \( n \).

\[ E_n \] the Effect of operation \( n \). For example, if operation \( n \) is an insert, the effect of that operation is that the tree has a new node.

\( a \rightarrow b \) defines a happens-before relation such that \( a \) happens before \( b \).
Terminology

if

\[ S_a \rightarrow S_b \rightarrow F_a \]

or

\[ S_b \rightarrow S_a \rightarrow F_b \]

then \( a \) and \( b \) are concurrent and either

\[ E_a \rightarrow E_b \quad \text{or} \quad E_b \rightarrow E_a \]

is possible
Relativistic Requirements

- Readers don’t block, and are not inhibited by writers
- Lookups always find a node that exists in the tree
- Traversals always return nodes in the correct order without skipping any nodes that exist in the tree
nodes that “exist in the tree”

\[ i \] is insert of \( N \)
\[ d \] is delete of \( N \)
\[ r \] is read (lookup or complete traversal)

- If \( F_i \rightarrow S_r \) and \( F_r \rightarrow S_d \) then \( N \) exists in the tree
- If \( F_r \rightarrow S_i \) or \( F_d \rightarrow S_r \) then \( N \) does not exist in the tree
nodes that “exist in the tree”

\(i\) is insert of \(N\)
\(d\) is delete of \(N\)
\(r\) is read (lookup or complete traversal)

- If \(i\) is concurrent with \(r\) then either \(E_i \rightarrow E_r\) or \(E_r \rightarrow E_i\)
- If \(d\) is concurrent with \(r\) then either \(E_d \rightarrow E_r\) or \(E_r \rightarrow E_d\)
nodes that “exist in the tree”

- Any update that strictly precedes a read is observable by that read
- Any update that strictly follows a read is not observable by that read
- Any update that is concurrent with a read may or may not be observable by that read
Relativistic Implementation
Assumptions

• Readers ignore the color of nodes

• Readers don’t need to access the parent pointers (note: some traversal algorithms require access to parent pointers)

• If ADT is a map (not a multimap), then temporarily having a duplicate node in the tree won’t affect lookups

• Mutual exclusion used between writers
Operations

Insertion  No nodes are moved or freed so this operation is safe.

Recolor   Recoloring does not affect the read consistency of the tree.

Delete    Need to defer reclamation of memory until no thread has a reference to the deleted node.

Swap      Need to guarantee that the node which moves up isn’t missed by readers between the new and old position.

Restructures Requires moving nodes. Need to ensure that no readers get lost in the transition.
Diag Restructure

Before:

- H
  - F
    - D
      - B
        - A
        - C
      - G
  - E

After:

- H
  - D
    - B
      - A
      - C
    - F
      - E
      - G
Diag Restructure
Diag Restructure

Reader at F looking for D?
Diag Restructure

1. init F’
2. link F’
3. unlink F
RBTree Delete with Swap
Swap algorithm

- find node to delete
- find swap candidate
- create swap-prime
- link swap-prime into tree
- wait a grace period
- remove swap from tree
Diag Restructure

Before:

```
  H
 / \    
F   G    
/ \  /   /
D  E D   H
/ \ / \ / \ / 
B  C B  C  B  F
A   C   A  C  A  E
```

After:

```
  H
 /    
D  G    
/ \  /   /
B  E D   H
/ \ / \ / 
A  C A  C  B  F
```

The restructuring involves moving node G from the second level to the first level, and adjusting the connections accordingly.
RBTree Delete with Swap
Zig Restructure
## Performance

<table>
<thead>
<tr>
<th>lock</th>
<th>No synchronization – NOT A VALID IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>nolock</td>
<td>pthread mutex lock</td>
</tr>
<tr>
<td>rwlr</td>
<td>Reader-Writer lock that favors readers</td>
</tr>
<tr>
<td>rwlw</td>
<td>Reader-Writer lock that favors writers</td>
</tr>
<tr>
<td>rp</td>
<td>Relativistic Programming implementation</td>
</tr>
<tr>
<td>ccavl</td>
<td>Concurrent AVL implementation</td>
</tr>
</tbody>
</table>
Read Performance

Operations/sec

Millions

Threads

NOLOCK reads
RP reads
ccavl reads
RWL_write reads
RWL_read reads
LOCK reads
Contended Write Performance

![Graph showing contended write performance with lines for different write types: NOLOCK writes, RP writes, ccavl writes, RWL_write writes, RWL_read writes. The x-axis represents Threads ranging from 0 to 70, and the y-axis represents Operation/sec with a scale from 0 to 0.25 in millions.]
Linearizability?
Linearizability

• Definition
  Equivalent to a some valid serial execution

• Proof
  Each operation appears to take effect instantaneously
RP RBTree Properties

- **Immutable nodes** – key & value don’t change
- **Sort** – Nodes are always in valid sort order
- **Structural Integrity** – lookups will always succeed
Lookup

Three options at each node

1. correct node was found
2. proceed down left path
3. proceed down right path
Lookup

Last $rp$-$read$ is the linearization point

- **Immutable nodes** means each time we examine a node, we’ll make the same decision.
- **Sort** means we’ll make the right decision at each node.
- **Structural integrity** means concurrent updates won’t break our lookup.

Must show that updates maintain these properties.
Insert

• `rp-publish` is the linearization point

• Must show that restructure does not invalidate *sort* or *structural integrity* properties
Delete without Swap

• \texttt{rp-publish} is the linearization point (assigning \texttt{NULL} or the single child to the parent of deleted node)

• Must show that restructure does not invalidate \texttt{sort} or \texttt{structural integrity} properties
Swap

\[ \text{rp-publish}(C') \text{ is linearization point} \]

\[ \text{Consider linearization point of lookups for B} \]

\[ \text{Preservation of properties?} \]
Diag Restructure

- Linearization point?
- sort?
- structural integrity?
Linearizability?

Write 1 – Write 2

Reader
Traversals
Traversal Options

• Use a stack to represent current location in traversal
• Use parent pointers to track location in traversal
• Other options?
Lookup Assumptions
Valid for Traversals?

• Readers ignore the color of nodes
• Readers don’t need to access the parent pointers (note: some traversal algorithms require access to parent pointers)
• If keys are unique, temporarily having a duplicate node in the tree won’t affect reads
• Mutual exclusion used between writers
Traversal Options

- Reader-Writer lock O(N) traversal – with delayed writes
- Relativistic O(N log(N)) traversal – with undelayed writes
- Relativistic O(N) traversal – with more complicated writes
Contended Update Performance
Read Performance 64K node tree
Scalability (4 readers)

- $O(N)$
- $O(N \log(N))$
Conclusions

What we gained

- Linear scalability for readers
- Writer doesn’t interfere with read performance
- Read performance approaches that of `nolock`
- Contended write performance exceeds that of other valid synchronization methods
Conclusions

What we gave up

- Uncontended write performance is worse than with other synchronization methods
- Traversal performance
Conclusions

• RP is applicable to complex multi-write update data structures

• `wait-for-readers` can be used to control the visibility of writes within an update
(an aside)

When is it OK for readers to see updates in different orders?

When updates are independent or commutative

Example: phone book

deleting a customer and adding a customer are independent UNLESS the new customer gets the old customer’s phone number