Relativistic Red Black Trees
Relativistic Programming

• Concurrent reading and writing improves performance and scalability
  – concurrent readers may disagree on the order of concurrent updates
  – orders may be non-linearizable
  – is this OK?

• Relativistic programming provides tools for enforcing the order that is necessary for correctness
  – linearizability is not always necessary!
The Natural World is Relativistic
Implications for Parallel Computing

• Communication over distance takes time
  – recipients at different distances will receive information/results at different times
  – potential to receive information out of order

• Forcing all recipients to agree on the order can only be done by delaying receipt
  – i.e. nobody gets it until the slowest has received it
  – delays slow down computation
  – the approach is inherently non-scalable
The Natural World is also Causal
Implications for Parallel Computing

• Scalability is all about allowing local computation to proceed unhindered
  – but violations of causality are confusing

• Delays can be introduced to preserve causality
  – i.e., if two updates are causally related, we can ensure that all readers see them in their correct order (the order the programmer specified)
  – if they are unrelated, then don’t force all to agree on an order: its unnecessary and expensive
Simple (atomic) Operations

List delete operation

- readers either see it or they don't
- no ordering problems
- no incoherency
Complex (non-atomic) Operations

List move operation

- readers might see before state, after state, or two during states
Dealing with Complex Operations

• What options do we have for list move?
  – do we add new before removing old?
  – do we remove old before adding new?

• What can concurrent readers observe?
  – both old and new?
  – neither old nor new?
  – old but not new?
  – new but not old?

• Which options are OK, and how can we enforce them?
Hiding Incorrect States

• If its OK to see either old, or new, or both, then we must hide the “neither” state
  – any reader that fails to find the old must find the new
  – is it enough to insert the new before removing the old?
  – if the new appears earlier in the list than the old then we need to wait for readers before we delete the old!

• If we must only see the before or after state then we need atomic transactions (later)
Relativistic Programming Primitives

• Generalization of RCU primitives
  – write-lock, write-unlock
    • for synchronization among writers
  – start-read, end-read
    • to delimit read sections
  – wait-for-readers
    • to wait for all pre-existing readers
  – deferred-free
    • to safely reclaim memory
  – publish, read
    • to remove reordering problems
Rules for Relativistic Programming

• Writers must keep data in a continually consistent state
  – a reader must be able to safely traverse a data structure at any time
  – individual updates must take effect at a well-defined point with respect to a concurrent reader
    • this is like linearizability
    • but it does not necessarily imply that all readers must agree on the order of unrelated updates!
Rules for Relativistic Programming

- When updates must be seen in order, writers must insert the appropriate delay
  - wait-for-readers
  - sometimes rp-read is enough

- Readers must delimit their read sections and not hold references between read sections
  - just like conventional locking rules

- To simplify CPU and compiler reordering issues
  - readers must use the rp-read to access shared data
  - writers update shared data using the rp-publish
Rules for Relativistic Programming

• Sometimes writers can reason about read traversal order and avoid using wait-for-readers
  – readers will naturally see updates in order even without it
  – example: moving an element from earlier to later in a singly linked list, by copying then removing, guarantees that all readers will see one or the other or both
But How Generally Useful is This?

- We have some primitives and a few simple rules
- They work well for simple list operations
- They also work well for hash-tables
- But is this enough for more complex data structures?
Red Black Trees

• RB-trees store sorted <key, value> pairs

• They guarantee $O(\log(N))$ performance for inserts, deletes, and lookups
  – they limit the depth of the tree (partially balanced)
  – updates require restructuring of the tree

• They are very difficult to parallelize
  – difficult to avoid deadlock with per-node locking
  – most implementations use a single global lock
  – they are a stiff challenge for Relativistic Programming!
Red Black Tree Properties

- All nodes on the left branch of a subtree have a key less than that of the root of the subtree
- All nodes on the right branch of a subtree have a key greater or equal to that of the root
- Each node has a color (red or black)
- Both children of a red node are black
- The black depth of every leaf is the same
  - this is the balance property
Rebalancing

• Updates potentially require the tree to be restructured to maintain its balance properties
  – changes can affect any node between the update and root
  – locking requires a lock for each affected node
    • acquiring locks on the way up can deadlock with readers that are descending
    • acquiring all possible locks ahead of time degenerates to coarse grain locking
  – conventional approaches use a single lock for the tree
    • no concurrency
Restructuring is also a concern for RP

A thread searching for B could fail to find it
Restructuring is also a concern for RP

A thread searching for B could fail to find it
Lookups

• Readers ignore color and do not access parent pointers

• Readers stop searching when they find a key that matches

• Updates can change colors and place multiple copies in the tree so long as they appear in valid sort positions, without affecting readers

• Relativistic lookups are essentially sequential code!
Inserts

• A new node is always inserted as a leaf

• Concurrent readers will see it if they dereference its pointer before the update publishes the pointer

• If the insert breaks the tree’s color or balance properties it must be recolored or rebalanced
  – that's the! tricky part
Deletes

- Nodes only deleted from the bottom of the tree
  - this may require some restructuring (called a swap)

- Readers will either see the deleted node, or they will not, depending on when they dereference its pointer

- The node's memory must not be reclaimed while readers are still accessing it
  - use rp-free

- If the delete leaves the tree unbalanced it must be restructured
Swap and Deletion of Node B

Move B's next node (C) to B's position
- create new copy of C (C')
- assign B's children to C'
- give C' B's parent (B is now unreachable)
- defer free of B's memory

Defer deletion of C
- wait for readers, then reassign C's children to C's parent (E)

Defer free of C's memory
Code for Swap

```
C = next(B);
C_prime = C.copy();

C_prime.color = B.color;
C_prime.left = B.left;
C_prime.left.parent = C_prime;

C_prime.right = B.right;
C_prime.right.parent = C_prime;

F = B.parent;
C_prime.parent = F;

if (F.left == B)
    rp-publish(F.left, C_prime);
else
    rp-publish(F.right, C_prime);

rp-free(B);
wait-for-readers();

E = C.parent;
rp-publish(E.left, C.right);
C.right.parent = E;

rp-free(C);
```

Listing 1: Code for swap
No copy is necessary
C adopts B's left child (A)
A appears twice in the tree (its temporarily a DAG)
  - this does not affect readers
Defer free of B

Figure 5. Tree before and after deletion of node B including one intermediate step
Code for Special Case Swap

C = next(B);
C.color = B.color;
C.left = B.left;
C.left.parent = C;

E = B.parent;
if (E.left == B)
    rp-publish(E.left, C);
else
    rp-publish(E.right, C);

rp-free(B);

Listing 2: Code for special case Swap
Diagonal Restructure

Figure 6. Arrangement of nodes before and after a diag restructure including one intermediate step.
Code for Diagonal Restructure

```
1  C_prime = C.copy();
   C_prime.left = B.right;
   C_prime.left.parent = C_prime;

5  rp-publish(B.right, C_prime);
   C_prime.parent = B;

    D = C.parent;

10 if (D.left == C)
    rp-publish(D.left, B);
else
    rp-publish(D.right, B);

15 B.parent = D;

   rp-free(C);
```

Listing 3: Code for diag left restructure
Zig Restructure

Figure 7. Arrangement of nodes before and after a zig restructure including one intermediate step.
Code for Zig Restructure

```c
A_prime = A.copy();
A_prime.right = B.left;
A_prime.right.parent = A_prime;

rp-publish(B.left, A_prime);
A_prime.parent = B;

C_prime = C.copy();
C_prime.left = B.right;
C_prime.left.parent = C_prime;

rp-publish(B.right, C_prime);
C_prime.parent = B;

D = C.parent;
if (D.left == C)
    rp-publish(D.left, B);
else
    rp-publish(D.right, B);

rp-free(A);
rp-free(C);
```

Listing 4: Code for zig left restructure
Read (lookup) Performance

Figure 8. Read performance of 64K node red-black trees using a variety of synchronization techniques.
Sequential Write (insert/delete) Performance

Figure 9. Update performance of 64K node red-black trees with a single updater and multiple readers. The left-most data point shows uncontended write performance. The remainder of the data points show the update performance with a variable number of concurrent readers.
Write-side Synchronization

• Global locking
  – no concurrent writes
• Fine-grain locking
  – deadlock
• CCAVL
  – concurrent, but different data structure
• STM - transactional memory
  – disjoint access parallelism (to be discussed later)
Concurrent Write (insert/delete) Performance

Figure 10. Concurrent update performance of 64K node trees.
Concurrent Read (lookup)
Performance

Concurrent read performance of 64K node trees. Note that the \textit{rp} and \textit{RP-STM} lines are on top of each other.
Linearizability

• Lookup, insert and delete operations are linearizable
  – there is a well defined point at which they take effect
  – they are primitive operations with only one update (can’t be seen out of order!)
Linearizability

- **Lookups**
  - last rp-read is the linearization point
- **Inserts**
  - rp-publish is the linearization point
- **Deletes**
  - store is the linearization point
- **Traversals are not necessarily linearizable**
Traversals

• Traversals visit the nodes in order
  – they make use of the next operation
• Traversals are challenging for RP because they read more than one location
  – restructures might cause nodes to be missed or duplicated
Traversals

• Three approaches explored:
  – treat a traversal as a single indivisible operation
    • acquire a lock and hold it for the duration
    • replace the write lock with a reader-writer lock
  – $O(N \log(N))$ relativistic traversals
    • can't use parent pointers
    • use relativistic lookups to construct the traversal
      (traversal take $O(N \log(N))$)
    • updates only wait for lookups
    • traversal is non-linearizable
Traversals

- Third approach: $O(N)$ relativistic traversal
  - requires modification of update operations to allow readers to use parent pointers
  - requires additional node copies to preserve the parent pointers
Traversals

Figure 12. Update performance in the presence of concurrent readers performing traversals
Traversals

Figure 13. Read performance
Conclusions

• Relativistic programming can be applied to a data structure as complex as a Red-Black Tree with excellent performance and scalability results
Spare Slides
Reader-visible States in Swap

a) Initial subtree

b) After insertion of $C'$
Reader-visible States in Swap

c) After removal of $C$

d) Final subtree
Reader-visible States in Diagonal Restructure

a) Initial subtree
b) After insertion of C'}
Reader-visible States in Diagonal Restructure

- c) After removal of C
- d) Final subtree
Reader-visible States in Zig Restructure

a) Initial subtree

b) After insertion of $A'$

c) After insertion of $C'$
Reader-visible States in Zig Restructure

d) After removal of A and C

e) Final subtree