Portland State University
PDXScholar

Computer Science Faculty Publications and Presentations | Computer Science

1-2011

Relativistic Red-Black Trees

Philip William Howard
Portland State University

Jonathan Walpole
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/compsci_fac

Part of the Computer and Systems Architecture Commons, and the Databases and Information Systems Commons

Let us know how access to this document benefits you.

Citation Details
Howard, Philip W., and Jonathan Walpole. Relativistic red-black trees. Technical Report 10-06, Portland State University, Computer Science Department, 2010.

This Technical Report is brought to you for free and open access. It has been accepted for inclusion in Computer Science Faculty Publications and Presentations by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

Relativistic Red-Black Trees

Philip W. Howard
Portland State University
pwh@cecs.pdx.edu

Jonathan Walpole
Portland State University
walpole@cs.pdx.edu

Abstract

Operating system performance and scalability on shared-memory many-core systems depends critically on efficient access to shared data structures. Scalability has proven difficult to achieve for many data structures. In this paper we present a novel and highly scalable concurrent red-black tree.

Red-black trees are widely used in operating systems, but typically exhibit poor scalability. Our red-black tree has linear read scalability, uncontended read performance that is at least 25% faster than other known approaches, and deterministic lookup times for a given tree size, making it suitable for realtime applications.

Keywords synchronization, data structures, scalability, concurrent programming, red-black trees

1. Introduction

The advent of many-core hardware introduces the need for highly scalable operating system designs. Many-core hardware poses a special challenge for symmetric shared-memory operating system architectures because it dramatically increases the degree of concurrency and at the same time decreases the locality of accesses to kernel data. The conventional strategy of using mutual exclusion severely limits scalability by serializing accesses to shared data and requiring extensive inter-core communication.

Some researchers have chosen to address this challenge by throwing out symmetric shared-memory multiprocessor operating system architecture, focusing instead on OS architectures that are non-symmetric in their use of shared-memory p1855745, or forgo shared-memory altogether [Baumann 2009]. Our research takes a different approach. We continue to assume a shared-memory operating system architecture can scale [Boyd-Wickizer 2010] and work toward this by weakening the ordering constraints that normally govern concurrent accesses to shared data structures. In this paper we illustrate our approach by considering a particular data structure, namely a red-black tree.

Red-black trees are used to store sorted ⟨key,value⟩ pairs, and are widely used in operating systems. They are used in the Linux kernel for I/O schedulers, the process scheduler, the ext3 file system, and in many other places [Landley 2007]. Linux kernel primitives that manipulate red-black trees do not have concurrency control embedded in them. Instead, higher level uses of the primitives must manage concurrency. This is typically done through mutual exclusion which does not scale [Piggin 2010].

Our approach is similar to RCU-based approaches that have been applied to simpler Linux data structures such as lists and hash tables [Triplett 2010]. We refer to these algorithms as "relativistic" because they weaken the ordering requirements on concurrent reads and updates such that each reader observes the data structure in its own temporal frame of reference. Our research is attempting to generalize the concept of relativistic programming. While this goal has yet to be reached, the development of a relativistic data structure as complex as a red-black tree represents a significant milestone.

Our relativistic red-black tree has the following performance and scalability properties:

1. Linear scalability of read accesses even in the presence of concurrent updates. This property has been tested out to 64 hardware threads.
2. Updaters can proceed concurrently with any number of readers, but not other updaters.
3. Safe, fast, wait-free read access in the presence of updates.

By fast we mean performance approaches that of unsynchronized access[1]. Our read access achieves 93% of the throughput of unsynchronized read access over a wide range of tree sizes and thread counts. Our implementation is also 25% faster than the best lock based implentation for an un-

[1] Unsynchronized access is safe for single threaded or read-only implementations, but not for multi-threaded implementations that include updates.

[Copyright notice will appear here once 'preprint' option is removed.]

contended read. As contention increases, the advantage of our implementation grows significantly.

By wait-free we mean that the read path does not use locks, does not block, and never needs to wait for another thread (neither a reader nor an updater). Furthermore, the read path does not require any atomic instructions and on x86 does not require memory barriers.

The rest of this paper is outlined as follows: Section 2 gives an overview of red-black trees, the operations they support, and the mechanisms that are used to preserve the balanced nature of red-black trees. This section also discusses the state of the art for parallelizing red-black trees. Section 3 discusses the ordering constraints that are, and are not, preserved by relativistic programming. This section also provides a justification for why these weakened ordering constraints are appropriate for concurrent red-black trees. Section 4 presents our implementation for a relativistic red-black tree. Section 5 shows the performance of our implementation compared with red-black trees implemented using other synchronization mechanisms. Section 6 discusses some of the issues and trade-offs involved in performing complete tree traversals (as opposed to single look-ups). Finally, Section 7 presents concluding remarks.

2. Red-Black Trees

Since red-black trees are well known and well documented [Guibas 1978, Plauger 1999, Schneier 1992], we do not give a complete explanation of them. Rather, we give a brief overview to facilitate a discussion of our relativistic implementation. In particular, we discuss the individual steps that make up red-black tree algorithms without discussing the glue that combines these steps. This is because the glue is not impacted by the relativistic implementation.

Red-black trees are partially balanced, sorted, binary trees. The trees store ⟨key,value⟩ pairs. They support the following operations:

insert(key, value) inserts a new ⟨key,value⟩ pair into the tree.

lookup(key) returns the value associated with a key.

delete(key) removes a ⟨key,value⟩ pair from the tree.

first()/last() returns the first (lowest keyed) / last (highest keyed) value in the tree.

next()/prev() returns the next/previous value in key-sorted order from the tree.

Red-black trees are sorted by preserving the following properties:

1. All nodes on the left branch of a subtree have a key $\leq$ the key of the root of the subtree.
2. All nodes on the right branch of a subtree have a key $>$ the key of the root of the subtree.

The tree is balanced by assigning a color to each node (red or black) and preserving the following properties:

1. Both children of a red node are black.
2. The black depth of every leaf is the same. The black depth is the number of black nodes encountered on the path from the root to the leaf.

These invariants are sufficient to guarantee $\mathrm{O}(\log(N))$ lookups because the longest possible path (alternating black and red nodes) is at most twice the shortest possible path (all black nodes). The operations required to rebalance a tree following an insert or delete are limited to the path from the inserted/deleted node back to the root. The rebalancing is, worst case, $\mathrm{O}(\log(N))$ meaning that inserts and deletes can also be done in $\mathrm{O}(\log(N))$.

We will use the following definitions in the explanation of the tree operations:

internal node A node with two non-empty children.

leaf A node with at least one empty child.

Observe that if *next()* is called on any internal node, the result is always a leaf. This is true because *next()* is the left-most node of the right subtree.

The following steps are used to implement red-black trees:

Insertion New nodes are always inserted at the bottom of the tree. This is possible because if *prev(new-node)* is an internal node, then from the observation above, the new node must be a leaf. If *prev(new-node)* is a leaf, the new node will be a child of that node on an empty branch. The insert may leave the tree unbalanced. If so, restructures or recolors (see below) are required to restore the balance properties of the tree.

Delete Nodes are always deleted from the bottom of the tree (possibly following a swap—see below). The delete may leave the tree unbalanced. If so, restructures or recolors (see below) are required to restore the balance properties of the tree.

Swap If an interior node needs to be deleted, it is first swapped with *next(deleted-node)* prior to removal. This makes the node to be deleted a leaf.

Restructures Restructure operations, sometimes called a rotations, are used to rebalance the tree. Restructures always involve three adjacent nodes: child, parent, and grandparent. See Figure 1 for an illustration of the two types of restructure operations.

Recolor Nodes get recolored as part of the rebalancing process. Recoloring doesn't involve changing the structure of the tree, only the colors applied to particular nodes.

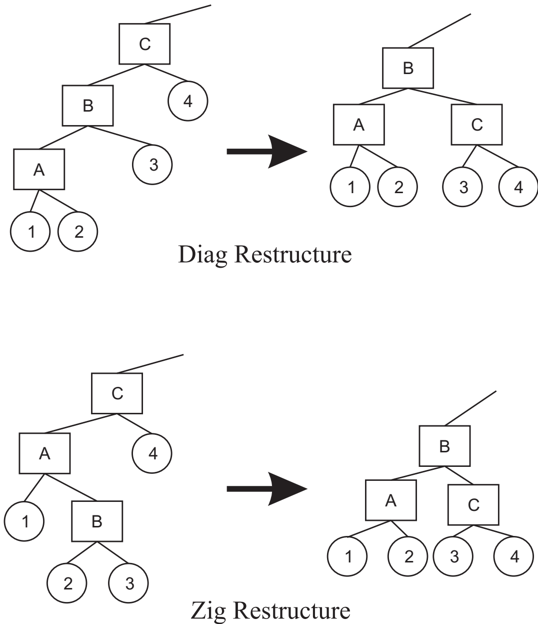

Figure 1. Restructure operations used to rebalance a red-black tree. There are left and right versions of these, but they are symmetric so only the left version is shown here.

2.1 Concurrent red-black trees

In thinking about concurrent red-black trees, it is useful to make a distinction between events and operations. In our terminology, operations are composed of events. Events are steps in an operation which have a visible effect. Events can be thought of as instantaneous. Since operations are typically composed of multiple events, they have a duration.

We will use the following definitions to describe concurrent implementations:

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.

If either of the following two relations holds, operations a and b are said to be concurrent:

$$S_a \Rightarrow S_b \Rightarrow F_a$$

$$S_b \Rightarrow S_a \Rightarrow F_b$$

Graphically, this means that the time-lines of the two operations overlap. There is no implied happens-before relation between the effects of two concurrent operations. The effects of the two operations could occur in any order.

Implementations of objects that allow updates to happen concurrently with reads, require additional properties so that every intermediate representation of the data structure can be mapped to a value of the abstract object [Herlihy 1990]. For a sorted tree, the following properties must be maintained:

1. Lookups will always find a node that exists in the tree.
2. Traversals will always return nodes in the correct order without skipping any nodes that exist in the tree.

Because reads have a duration, and because updates can proceed concurrent with reads, it's possible that the tree will change during a read. As a result, we need to be specific in what we mean by "nodes that exist in the tree". In particular, it means the following: if operation r is a read looking for node N (or a complete traversal of the tree), operation i is the insert of node N, and operation d is the delete of node N, then

1. If $F_i \Rightarrow S_r$ and $F_r \Rightarrow S_d$ then N exists in the tree and must be observed by r in the correct traversal order.
2. If $F_r \Rightarrow B_i$ or if $F_d \Rightarrow S_r$ then N does not exist in the tree and must not be observed by r.
3. if i is concurrent with r then N may or may not be observed by r depending on whether the relative view of r is $E_i \Rightarrow E_r$ or $E_r \Rightarrow E_i$.
4. if d is concurrent with r then N may or may not be observed by r depending on whether the relative view of r is $E_r \Rightarrow E_d$ or $E_d \Rightarrow E_r$.

Another way to state these properties is as follows: Properties one and two state that any update that strictly precedes a read must be observable by the read, and any update that strictly follows a read must not be observable by the read. Properties three and four state that any update that is concurrent with the read may or may not be observable by the read.

2.2 The State of the Art

The most common way to synchronize access to a red-black tree is through locking. Unfortunately, this approach doesn't scale because accesses are serialized. Since accesses can be easily divided into reads (lookups) and writes (inserts, deletes), a reader-writer lock can be used which allows read parallelism. This approach scales for some number of read threads, but eventually the contention for the lock dominates and the approach no longer scales (see the performance data in Section 5.1 for evidence of this).

Fine grained locking of red-black trees is problematic. Since updates may affect all the nodes from where the update occurred back to the root, the simplest approach of acquiring a write lock on all nodes that might change degrades to coarse grain locking—all updaters must acquire a write lock on the root. If one attempts to only acquire write locks on the nodes that will actually be changed, it is difficult to avoid deadlock. If the locks are acquired from the bottom up, a reader progressing down the tree, but above the updater, may

acquire a lock that prevents the write from completing. If the locks are acquired from the top down, another updater may change the structure of the tree between the time the initial change was made (e.g. an insert) and the time when the necessary locks are acquired to perform a restructure.

Transactional Memory approaches provide a more automatic approach to disjoint concurrency. However, as the changes required to rebalance a tree progress up the tree, more and more concurrent read transactions would get invalidated. We haven't done any investigation to determine what percentage of concurrent transactions might get invalidated, so we can not predict the performance impact of the invalidations.

Bronson [2010] developed a concurrent AVL tree[2]. Their approach allows readers to proceed without locks, but the readers have to check each step of the way to see if the tree has changed or is in the process of changing. If so, the reader has to wait and retry. Since readers don't acquire locks, this simplifies the fine grained locking of the writers. Their approach is quite complicated and this degrades read performance as more code must execute at each node of the tree. Their approach allows concurrent updates and their performance data show good scalability. We are working to port their implementation from Java to C to perform a fair side-by-side comparison, but we have not yet completed this work. Work done to date indicates that our read approach is much faster.

A number of researchers have attempted to decouple rebalancing from insert and delete [Guibas 1978, Hanke 1998]. This allows updates to proceed more quickly because individual inserts and deletes don't have to rebalance the tree. The rebalancing work can potentially be done in parallel and some redundant work can be skipped. None of this improves read access time, and readers and writers still need some synchronization between them.

3. Relativistic Programming

The name for relativistic programming is borrowed from Einstein's theory of relativity in which each observer is allowed to have their own frame of reference. In relativistic programming, each reader is allowed to have their own frame of reference with respect to the order of updates.

Relativistic programming is characterized by the following two properties:

1. Writes can occur concurrently with reads.
2. Writes are not totally ordered with respect to reads.

Consider the time-line in figure 2. If operations A and B are writes and operation C is a read, then C can observe the writes in either order. In particular, since A is concurrent with C, both $E_A \Rightarrow E_C$ and $E_C \Rightarrow E_A$ are equally valid. The same is true of B and C. Combining all three operations, the ordering $E_B \Rightarrow E_C \Rightarrow E_A$ is valid even though this violates the happens-before relation between the non-concurrent A and B.

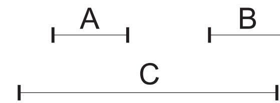

Figure 2. Operation C can see operations A and B in any order.

It is important to note that the order mentioned above represents the reference frame of a particular reader. There is no "global observer" which determines the "correct" order. Each reader has their own relative view of concurrent operations which may differ from the view of other concurrent readers.

3.1 Is this OK?

While it might be disconcerting to have writes appear to happen in different orders, there are two conditions which, if met, make this acceptable. The conditions are as follows:

1. The underlying data structure does not have an inherent time order
2. The updates are independent or commutative

Last In First Out Stacks and First In First Out Queues have an inherent time order (thus the First and Last in their names). As a result, these data structures are not good fits for relativistic implementations[3]. However many other data structures (lists, dictionaries, trees, etc.) have no such inherent time order and thus allow a relativistic implementation.

To illustrate what is meant by independent or commutable updates, consider a phone company that uses a tree to maintain phone book information. If two customers call the phone company to change their service, the two calls can be handled in either order. Neither call affects the other so they are independent. They are also commutable because the results are the same regardless of the order. Any query that saw neither, either, or both updates is equally valid. Even printing a phone book that included neither, either, or both updates is equally valid. If either of these customers called to complain about their inclusion or omission from the phone book, the phone company could legitimately reply that the book was printed either just before or just after their information was entered into the system.

3.2 Memory Management

Relativistic programming requires a mechanism to reclaim memory that has been freed by one thread while still in use by another. Freed memory comes from two sources: nodes that are removed from the tree and the "old" copy of

[2] AVL trees are similar to red-black trees, but they have a different balance property.

[3] Some researchers have proposed weak ordering for LIFO's and FIFO's. This would yield a Later In Earlier Out or Earlier In Earlier Out structure that would be suitable for relativistic techniques.

nodes that were updated using copy-on-update semantics. In either case, it is possible that a concurrent reader may have a reference to the node that needs to be reclaimed.

In order for the reclamation to be safe, reclamation must be delayed long enough to ensure that no readers have a reference to the memory. To facilitate this, readers are required to bound all operations with *begin* and *end* primitives. Readers aren't allowed to hold references to the data outside these read sections. While outside a read section, a reader is considered quiescent (not actively reading). If all readers have been in a quiescent state following the deletion of the last global reference to a node, no reader can hold a reference to the node so the node's memory can be safely reclaimed. Any period of time in which all readers have been in a quiescent state is called a grace period [McKenney 2004].

3.3 Relativistic Programming Primitives

This section describes the primitives that are used by relativistic programs. The description here focuses on the purpose and use of the primitives. There are a variety of implementations readily available which are described elsewhere [Desnoyers 2009, McKenney 2003].

3.3.1 write-lock, write-unlock

These primitives provide mutual exclusion between writers. The use of these primitives does not impact readers—readers proceed oblivious to the presence of a writer. Note that strictly speaking, write-lock and write-unlock are not relativistic programming primitives. Mutual exclusion between writers is not required by relativistic programming, but all known implementations use mutual exclusion between writers.

3.3.2 rp-start-read, rp-end-read

These primitives bound the code where a reader holds references to the data structure. They allow a writer to know when it's safe to reclaim the memory for a no longer used node. Most relativistic programming implementations have non-blocking, wait-free, low-latency, constant-time implementations of these primitives so their use has minimum impact on read performance.

3.3.3 rp-wait-gp

This primitive waits for a grace period. A grace period has expired when all current read-sections have terminated. It is not necessary for all threads to be outside their read-sections at the same time. It is only necessary for any thread inside a read-section at the beginning of rp-wait-gp to exit that read-section. Once a thread exits a read-section, it can begin a new one without interfering with rp-wait-gp. Stated more formally,

$\forall$ readers r and any rp-wait-gp w if $S_r \Rightarrow S_w$ then $F_r \Rightarrow F_w$

3.3.4 rp-free

This primitive is called by writers to schedule the future reclamation of memory. This allows decoupling the freeing of memory from reclaiming that memory. Updaters do not have to block until a grace period has expired, they can schedule the memory for reclamation in the future and then continue. rp-free guarantees that a grace period will expire before the memory is reclaimed.

3.3.5 rp-release

This primitive is used by writers when they want to make a node visible to readers. It includes whatever barriers are required to ensure that the updates to the node are visible before the node itself can be reached.

3.3.6 rp-consume

This primitive is used to dereference the pointer to nodes. It includes whatever barriers are required to enforce dependent read consistency. On most architectures, rp-consume need only read the pointer; no barriers are required.

4. A Relativistic Red-Black Tree Algorithm

Many concurrent methodologies produce very complicated code because both readers and updaters have to check at each step to see if anything has changed. With relativistic techniques, developing a reader is almost as easy as developing a non-concurrent reader. The only restrictions placed on readers are that they use rp-consume when dereferencing pointers and that they not hold any references to the data structure outside read-sections bounded by the rp-start-read and rp-end-read primitives.

In order for readers to be able to proceed without having to check the consistency of the data, the updaters need to keep the data in an always-consistent state. This requires three things: that rp-release be used when updating pointers, that rp-free be used when freeing no-longer-used memory, and that care be taken in the order that updates are made. The final requirement, ordering, takes two forms. In the first case, the updater must not allow a reader to see partial changes to a node. Updaters use copy-on-update to make all the changes to a a private copy of the node, then atomically switch the new node with the old one using rp-release. In the second case, when the structure of the tree is changed, care must be taken to ensure that readers don't get lost. To illustrate this, consider the zig restructure depicted in Figure 1. If a reader is at node A looking for node B at the time the restructure happens, the reader will never find B. This is because after the restructure, B is above A instead of below it.

The remainder of this section explains how we implemented updaters with these principles. Our implementation imposes a single restriction on the trees: we do not allow duplicate keys in the tree. This is a restriction that is imposed on many non-relativistic implementations as well, so we don't find it overly restrictive.

We make the following observations about readers performing a lookup (for traversals, see Section 6):

1. Readers ignore the color of nodes.
2. Readers don't access the parent pointers in nodes.
3. Temporarily having the same item in the tree multiple times won't affect lookups. A positive result will return the first copy encountered. A negative result (item not in tree) will return "not found" even if other keys are duplicated in the tree.

The implications of these observations are that updaters can change the color and parent pointers without affecting readers; updaters can also temporarily allow duplicates provided both duplicates are in valid sort order locations. Given the above observations, the following can be said about the steps in an update:

Insertion New nodes are always inserted at the bottom of the tree. No nodes are moved or freed. A concurrent reader will either see the new node or not depending on whether rp-release $\Rightarrow$ rp-consume. But concurrent readers will never see an inconsistent state.

Delete Nodes are only deleted from the bottom of the tree. Similar to insert, a concurrent reader will either see the deleted node or not depending on the order of operations, but a reader will never see an inconsistent state. The memory for the deleted node must not be reclaimed while concurrent readers have a reference to it. Using the rp-free primitive will ensure that the proper delay occurs before the memory is reclaimed.

Swap Since a concurrent reader searching for the swapped node might be at a point in the tree between the swapped node's new and old positions, special handling is required to ensure that such a reader sees the swapped node (see section 4.1). This is because the swapped node exists in the tree and therefore it must be observable in correct traversal order.

Restructures Much like swap, restructures involve moving nodes. This requires special handling to keep the tree in an always-consistent state (see section 4.2).

Recolor Since readers ignore the color of nodes, recoloring does not affect the read consistency of the tree.

The two operations that require special handling in a relativistic implementation are Swap and Restructure. These are described in greater detail below.

4.1 Swap algorithms

Section 4.1.1 discusses the general swap algorithm. Section 4.1.2 discusses an optimized special case swap.

4.1.1 General Swap

Consider the delete of node B shown in Figure 3. Since B is an internal node, B will be swapped with C $(= next(B))$ prior to deletion. There is a special case where the swap node happens to be the right child of B. This is dealt with in Section 4.1.2

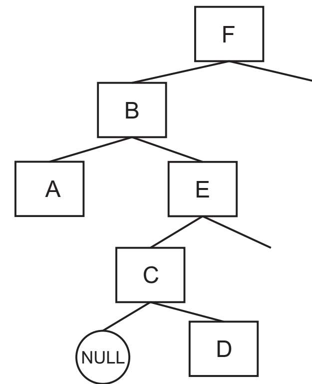

Figure 3. Tree before deletion of node B

Rather than performing separate swap and delete steps, the two are combined as a single step as shown in listing 1. A new node C' is created. C' has the same color as B and the same children as B, but the key and data values of C.

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

   C_prime.color = B.color;
5
   C_prime.left = B.left;
   B.left.parent = C_prime;

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

   C_prime.parent = B.parent;

   F = B.parent;
15 if (F.left == B)
     rp-release(
             F.left, C_prime);
   else
     rp-release(
20           F.right, C_prime);

   rp-free(B);

   rp-wait-gp();
25
   E = C.parent;
   rp-release(
         E.left, C.right);
   E.parent = C.parent;
30
   rp-free(C);
```

Listing 1. Code for swap

The new node C' is linked into the tree in place of B. At this point, the value C is in the tree twice: once at C and once at C'. Any readers looking for C can be divided into two groups: those above C' will find the value at C', those at or below B will find the value at C. In either case, the correct value will be found. However, if the old node C is removed, any readers looking for the value C that were at or below B would miss the value. To avoid this problem, the updater waits for a grace period before removing C from the tree. This ensures that any readers at or below B will complete their read prior to C being removed. The resulting tree is show in figure 4.

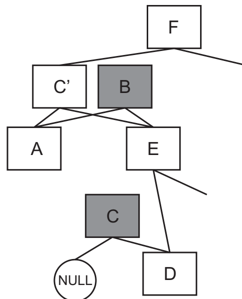

Figure 4. Tree after deletion of node B. The gray nodes are scheduled for reclamation.

This algorithm differs from a non-RP algorithm in the following ways:

1. A copy of node C is placed in B's position rather than node C itself.
2. rp-release is used to make pointer assignments to guarantee that changes to a node are visible before the node itself is reachable.
3. rp-free is used to release memory to ensure that no readers have references to the memory when it is released.
4. A grace period is included so that no readers will miss seeing node C.

4.1.2 Special case: swap node is child of B

In the tree shown in figure 5, C is $next(B)$. It also happens to be the right child of B. This represents a special case and no new nodes need to be created. The changes are made as shown in listing 2. C takes the color of B. The left child of B becomes the left child of C. The node A now appears in the tree twice (once below B and once below C), but no loops are created. Any reader encountering the tree in this state will find A regardless of where they were in their traversal when then changes were made.

B is removed from the tree by linking C into the tree in it's place. B is then freed asynchronously by calling rp-free. The tree is now as shown in figure 6.

This algorithm differs from a non-RP algorithm only in the use of rp-release to make pointer assignments, and rp-free to release memory.

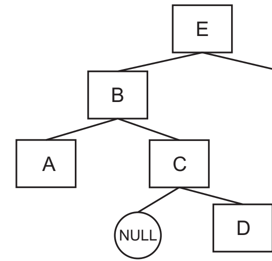

Figure 5. Tree before deletion of node B.

```
1  C = next(B.right);

   C.color = B.color;
   C.left = B.left;
5  B.left.parent = C;

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

   rp-free(B);
```

Listing 2. Code for special case Swap

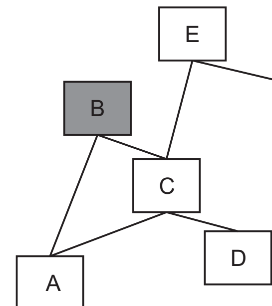

Figure 6. Tree after deletion of node B. The gray node is scheduled for reclamation.

4.2 Restructure

There are two cases for restructures depending on whether the three nodes involved form a diagonal or a “zig”. Each of these can be further classified depending on whether it is left or right, but the left and right cases are symmetric, so only the left case will be described here.

4.2.1 Diag Left

Figure 7 shows a subtree with three nodes labeled A, B, C which need to be rotated so that B is the root of the subtree. The changes are made as shown in listing 3

C' is a copy of node C. The right child of B becomes the left child of C'. C' is then linked into the tree as the

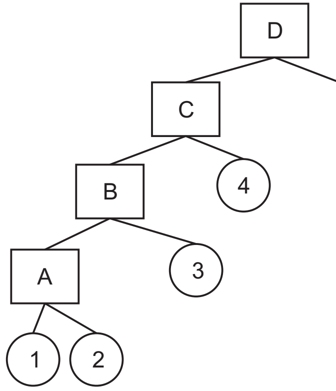

Figure 7. Diagonal arrangement of nodes before restructure.

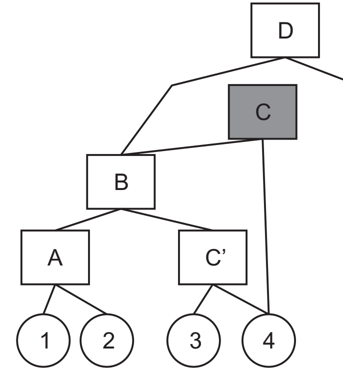

Figure 8. Diagonal arrangement of nodes after restructure. Gray node is scheduled for reclamation.

right child of B. At this point, the value C is in the tree twice. This is similar to swap in Section 4.1.1. However, in this case, the copy is placed lower in the tree rather than higher in the tree. As a result, the original node C can be removed without waiting for a grace period. This is because any readers between C and C' will still see C' even after C is removed from the tree. The resulting tree is shown in figure 8.

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

5  rp_release(
         B.right, C_prime);
   C_prime.parent = B;

   D = C.parent;
10
   if (D.left == C)
     rp_release(D.left, B);
   else
       rp_release(D.right, B);
15
   B.parent = D;

   rp-free(C);
```

Listing 3. Code for diag left restructure

This algorithm differs from a non-RP algorithm as follows: a copy of a node was made rather than changing a node in place, and RP primitives were used for pointer assignment and memory reclaimation.

4.2.2 Zig Left

Figure 9 shows a subtree with three nodes labeled A, B, C which need to be rotated so that B is the root of the subtree (this is known as a double rotation [Guibas 1978]). There are two ways to accomplish this: either a copy of B can be placed above A and C, or copies of A and C can be placed below B. Since the first method involves moving the copy up in the tree, it requires a grace period. In our implementation, freed nodes are cached so creating copies is fairly fast. Even though the second method requires two copies, performance data showed that the second method is faster, so that method is described here (see listing 4).

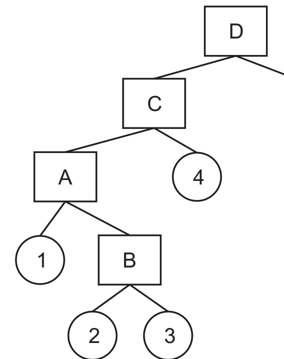

Figure 9. Zig arrangement of nodes before restructure.

A' is a copy of A. The left child of B becomes the right child of A'. A' is linked into the tree as B's left child. At this point, the value A appears in the tree twice. Since the new copy is placed below the original, there is no need for a grace period before removing the original from the tree.

C' is a copy of C. The right child of B becomes the left child of C'. C' is linked into the tree as B's right child. The original nodes A and C are removed from the tree by making B a child of D.

This algorithm differs from a non-RP algorithm as follows: copies of a nodes were made rather than changing nodes in place, and RP primitives were used for pointer assignment and memory reclamation.

5. Performance

Performance data was collected using the following synchronization techniques:

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

rp_release(
    B.left, A_prime);
A_prime.parent = B;

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

rp_release(
    B.right, C_prime);
C_prime.parent = B;

D = C.parent;
if (D.left == C)
  rp_release(D.left, B);
else
  rp_release(D.right, B);

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

Listing 4. Code for zig left restructure

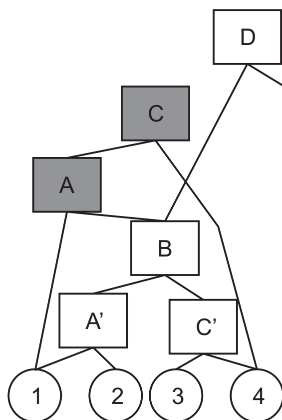

Figure 10. Zig arrangement of nodes after restructure. Gray nodes are scheduled for reclamation.

nolock No synchronization was used. This is **not** a valid implementation, but it was tested as a theoretical upper bound (highest performance, but not a data safe implementation) to compare the other algorithms against.

lock A pthread mutex was used and shared between readers and writers. As a result, there was no parallelism while accessing the tree.

rwlr A reader/writer lock that favors readers. The implementation was derived from Mellor-Crummey and Scott [Mellor-Crummey 1991].

rwlw A reader/writer lock that favors writers. The implementation was derived from Mellor-Crummey and Scott [Mellor-Crummey 1991].

rp This is the relativistic implementation described in this paper.

Note: all the algorithms except rp used a "standard" red-black tree implementation that did not perform copy-on-update.

The test created a tree and preloaded it to a given size with a random set of values. Threads were created to perform operations on the tree (lookups, inserts, and deletes). The threads were allowed to run for a fixed period of time and the total number of operations performed was reported.

Threads were of two types: readers and updaters. Readers performed lookups for values in the tree. Updaters removed a value from the tree and then inserted a different value. By pairing deletes and inserts, the size of the tree remained fixed.

Tests were performed on trees of size 64 and 64K nodes. The graphs for both sizes were very similar, so only the graphs for trees of size 64K are presented here. Comments indicate where there were differences in the size 64 graphs.

Tests were run where all the threads were readers and where there was one updater and multiple readers. Since all the synchronization algorithms only allow a single updater at a time, no data was collected with updaters contending with other updaters.

Performance data was collected on a Sun UltraSPARC T2 running SunOS 5.10. The UltraSPARC T2 has eight cores each supporting eight hardware threads for a total of 64 hardware threads. Performance data was also collected on a four quad-core Intel Xeon machine (total of 16 hardware threads). The machine was running Linux 2.6.28. The results for both the Sun and Intel processors were very similar. As a result, the performance data for the Xeon processor is not presented in this paper, but is available at `http://rp.avscorp.com/rbTreePerformanceSupplementalData.pdf`.

5.1 Read Performance

Figure 11 shows the read performance of the red-black tree. The performance of rwlw and lock were strictly worse than rwlr, so they were left off the figure for clarity. The reads

here are individual lookups, not complete traversals. For complete traversals, see section 6.3. The following observations can be made from the figure:

1. rp read performance scales linearly to at least 64 threads.
2. rp read performance approaches unsynchronized performance. rp values were approximately 93% of the nolock values.

When we plotted read performance with and without a concurrent updater, the two lines were indistinguishable. The average difference in read performance with and without a concurrent updater was less than 1%. This shows that a concurrent updater does not impact read performance.

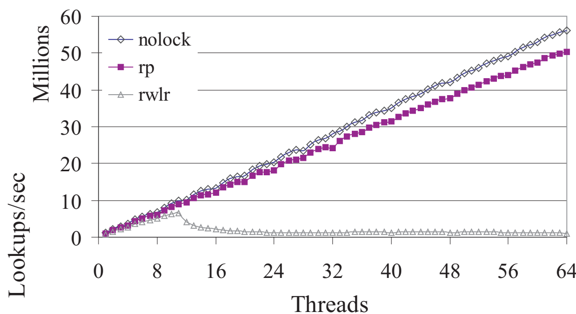

Figure 11. Read performance of 64K node red-black trees using a variety of synchronization techniques.

5.2 Update Performance

Figure 12 shows update performance. The X axis shows the total number of threads. The first thread was an updater, the remainder of the threads were readers. The uncontended update performance is indicated by the left most data point on each line. The remainder of the data points show the contended update performance with a varying number of readers.

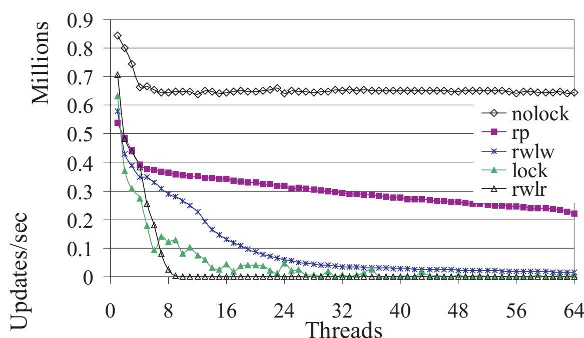

Figure 12. Update performance of 64K node red-black trees with a single updater and multiple readers.

Of all the synchronization methods, rp had the worst uncontended update performance. rp performance was 93% of rwlw performance (the best of the other valid methods). With even a single concurrent reader, rp update performance is better than any of the other synchronization methods (with the exception of nolock which leads to data corruption). As the number of concurrent readers increases, the advantage of rp is more pronounced. With a smaller tree, it takes more concurrent readers to give a clear advantage to rp. With a tree size of 64 nodes, rp had better write-side performance if there were 6 or more concurrent readers.

6. Traversals

The performance data presented in Section 5 was for readers performing lookups. Another read access pattern is a traversal where all the nodes in the tree are accessed in order. There are a number of traversal algorithms available. Some make use of a stack to keep track of what branches still need to be visited. Others make use of parent pointers in each node and whether the just-visited node is the left or right child of the parent. For each of these algorithms, the particular shape of the tree is important. A concurrent update may restructure the tree such that the path represented by the stack no longer exists. Alternately, a node that was below a given node may be restructured above it causing it to be either visited twice or skipped in the traversal. These re-orderings present challenges to a relativistic implementation of a tree because they have the potential to result in an invalid traversal.

Two approaches were taken to solve these problems. Both approaches use two primitives: first() and next(). The primitive first() returns the item in the tree that is first in the sort order. The primitive next() is passed information about a node in the tree and returns the node that follows the specified node in sort order. The two approaches differ in their implementation of next().

6.1 The standard approach

The standard approach to a tree traversal is to treat the entire traversal as a single operation. A lock is acquired at the beginning of the traversal and held until the end of the traversal. The lock prevents any updates from happening during the period the lock is held. To allow this type of traversal using the relativistic read and update algorithms described earlier, the mutex used for the write lock is replaced with a reader-writer lock. This yields three sets of critical section bounding primitives:

1. read start/end bounds a relativistic read and allows relativistic updaters to proceed in parallel. These primitives are used for lookups.
2. write lock/unlock bounds an update critical section. Relativistic readers are not inhibited by this lock.
3. rw lock/unlock bounds a traversal. Multiple rw locks can be acquired at the same time. An rw lock excludes a write

lock, and a write lock excludes an rw lock. An rw lock does not inhibit relativistic readers (read start/end).

6.2 A relativistic approach

Consider the following observations about tree traversals and the relativistic algorithms given in Section 4:

1. Traversals are $O(N)$; Updates are $O(\log(N))$ therefore traversals are expected to take much longer than updates.
2. Some updates require a grace period to expire in the middle of the update. If the grace period is $O(N)$, this will significantly delay updates.
3. The algorithms given in Section 4 assume that readers don't access the parent pointers so parent pointers should not be used by relativistic traversals. It should be possible to perform relativistic updates that keep the parent pointers always consistent, but this would likely require more node copies, and may require additional grace periods in the midst of an update.

Given the above considerations, a relativistic traversal can be constructed using relativistic lookups. The next() primitive is passed the key of the previous node and returns the key and value of the node with the first key greater than the previous key. This allows the same relativistic read and update algorithms described in Section 4 to be used. The consequences are that a traversal will take $O(N \log(N))$ time, and the tree that is traversed may not represent any state present in a globally ordered time. In particular, it's possible that a great number of updates occurred during the time of the traversal. No guarantees can be made as to which of these updates were seen and which weren't. The only guarantee is that next() will return the next node that was in the tree during the relativistic snapshot of time in which next() was called.

6.3 Traversal Performance

Figure 13 shows the read-side performance of the standard approach. With the exception of lock, all synchronization mechanisms scale linearly and have similar performance (only rp and rwlw are shown in the figure for clarity). This is because all the synchronization mechanisms except lock allow read concurrency and because the time is dominated by the traversal, not by the synchronization.

Figures 14 and 15 show write-side performance of the standard approach. Like the performance data in Section 5, there is a single updater and $N-1$ readers. The write-side performance falls off significantly in the presence of readers, so figure 15 shows the same data, but leaves off the uncontended update data points so the details of the others can be seen. Figure 15 shows that the rp and rwlw algorithms have very similar performance. This is because the rwlw synchronization algorithm was used for the rw-lock and write-lock primitives in the rp implementation.

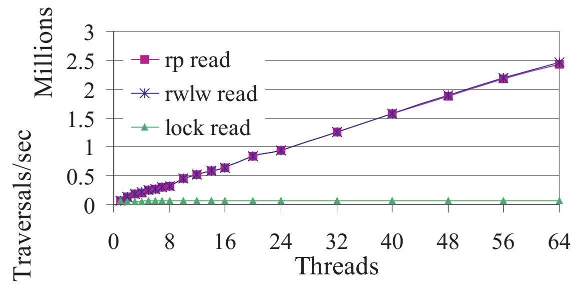

Figure 13. Read performance of standard traversal algorithm. Note that the rp and rwlw lines are on top of each other.

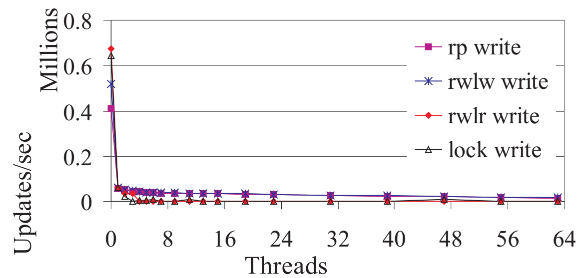

Figure 14. Update performance of standard traversal algorithm

Figure 16 shows the read-side performance of the relativistic approach. For sufficiently high thread counts, rp outperforms the other synchronization methods. At lower thread counts, rp is similar to the other approaches. Although rp scales well, it does not scale linearly. We believe that this is because, for traversals, readers wind up evicting parts of their own L1 caches. This places additional bandwidth requirements on the memory bus.

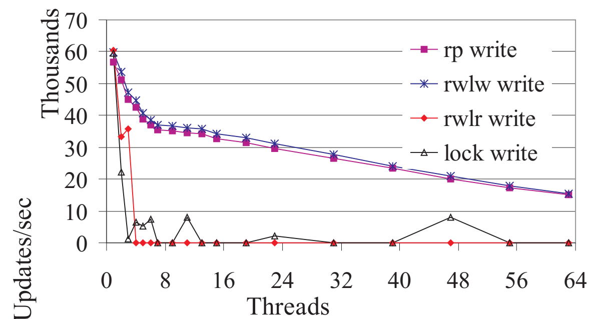

Figure 15. Update performance of standard traversal algorithm showing only contended updates

The update performance of the relativistic approach was essentially the same as the update performance when the

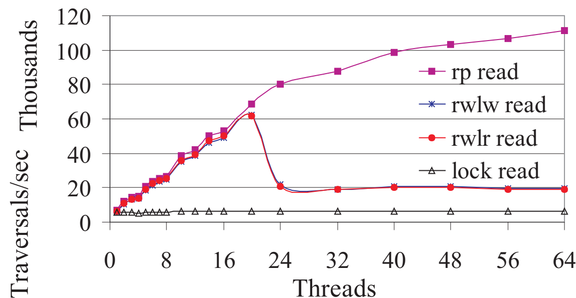

Figure 16. Read performance of RP traversals. Note that rwlr and rwlw lines are on top of each other.

readers were doing lookups instead of traversals (see Figure 12). This is because the relativistic traversals were using relativistic lookups to perform the traversal.

7. Conclusions

We have shown that relaxing the ordering constraints on updates can allow for much more scalable concurrent data structures. This was demonstrated by implementing a relativistic red-black tree. Our implementation has lookup performance that rivals an implementation without any form of synchronization. Further, our implementation scales linearly out to at least 64 hardware threads.

We are continuing to investigate relativistic implementations of other data structures and use patterns. We hope to derive a generalized method for developing relativistic implementations. We are also working to solve the multi-update problem so that relativistic implementation can have concurrent updaters as well as concurrent readers with a single updater.

References

[Baumann 2009] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3.

[Boyd-Wickizer 2010] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *9th USENIX Symposium on Operating System Design and Implementation*, pages 1–16, Vancouver, BC, Canada, October 2010. USENIX.

[Bronson 2010] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 257–268, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-877-3.

[Desnoyers 2009] Mathieu Desnoyers. *Low-Impact Operating System Tracing*. PhD thesis, École Polytechnique de Montréal, December 2009. [Online]. Available: `http://www.lttng.org/pub/thesis/desnoyers-dissertation-2009-12.pdf`.

[Guibas 1978] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *SFCS '78: Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, Washington, DC, USA, 1978. IEEE Computer Society.

[Hanke 1998] Sabine Hanke. The performance of concurrent red-black tree algorithms. Technical report, Albert-Ludwigs University at Freiburg, 1998.

[Herlihy 1990] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Language Systems*, 12(3):463–492, 1990. ISSN 0164-0925.

[Landley 2007] Rob Landley. Red-black trees (rbtree) in linux. *kernel.org documentation*, January 2007. [Online]. Available: `http://www.kernel.org/doc/Documentation/rbtree.txt`.

[McKenney 2003] Paul E. McKenney. Kernel korner: using RCU in the Linux 2.5 kernel. *Linux J.*, 2003(114):11, 2003. ISSN 1075-3583.

[McKenney 2004] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: `http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf` [Viewed October 15, 2004].

[Mellor-Crummey 1991] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *PPOPP '91: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 106–113, New York, NY, USA, 1991. ACM. ISBN 0-89791-390-6.

[Piggin 2010] Nick Piggin. The memory map semaphore (mmap-sem). *lwn.net*, August 2010. [Online]. Available: `http://lwn.net/Articles/399148/`.

[Plauger 1999] P. J. Plauger. A better red-black tree. *C/C++ Users J.*, 17:10–19, July 1999. ISSN 1075-2838. URL `http://portal.acm.org/citation.cfm?id=330304.330305`.

[Schneier 1992] Bruce Schneier. Red-black trees. *Dr. Dobb's J.*, 17(4):42–46, 1992. ISSN 1044-789X.

[Triplett 2010] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Scalable concurrent hash tables via relativistic programming. *SIGOPS Oper. Syst. Rev.*, 44:102–109, August 2010. ISSN 0163-5980. URL `http://doi.acm.org/10.1145/1842733.1842750`.