CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2013)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3157

Relativistic red-black trees

Philip W. Howard [1,*,†] and Jonathan Walpole [2]

[1] *University of Puget Sound*
[2] *Portland State University*

SUMMARY

This paper presents algorithms for concurrently reading and modifying a red-black tree (RBTree). The algorithms allow wait-free, linearly scalable lookups in the presence of concurrent inserts and deletes. They have deterministic response times for a given tree size and uncontended read performance that is at least 60% faster than other known approaches. The techniques used to derive these algorithms arise from a concurrent programming methodology called relativistic programming. Relativistic programming introduces write-side delay primitives that allow the writer to pay most of the cost of synchronization between readers and writers. Only minimal synchronization overhead is placed on readers. Relativistic programming avoids unnecessarily strict ordering of read and write operations while still providing the capability to enforce linearizability. This paper shows how relativistic programming can be used to build a concurrent RBTree with synchronization-free readers and both lock-based and transactional memory-based writers. Copyright © 2013 John Wiley & Sons, Ltd.

Received 30 June 2011; Revised 3 February 2013; Accepted 11 September 2013

KEY WORDS: synchronization; data structures; scalability; concurrent programming; red-black trees

1. INTRODUCTION

Red-black trees (RBtrees) are used to store sorted ⟨key,value⟩ pairs. They guarantee O(log(N)) performance for lookup, insert, and delete operations. They also have relatively low cost for maintaining the balance properties of the tree. RBtrees are used extensively in operating systems. In the Linux kernel, they are used for I/O schedulers, the process scheduler, the ext3 file system, and in many other places [1]. RBtrees are also used by many applications and are provided by class libraries such as java.util.TreeMap [2].

Unfortunately, it is difficult to implement concurrent RBtrees with good performance and scalability properties. Most implementations use a single global lock to protect concurrent accesses. Coarse grained locking prevents these implementations from scaling because accesses are serialized by the lock. Because accesses can be easily divided into reads (single lookups and complete traversals) and writes (inserts and deletes), a reader–writer lock can be used, which allows read parallelism. This approach scales for some number of read threads, but eventually contention for the lock dominates and the approach no longer scales (see the performance data in Section 4.1 for evidence of this behavior).

Fine grained locking of RBtrees is problematic. Updates may affect all the nodes from where the update occurred back to the root. If a writer acquires a write lock on all nodes that might change, then the root node becomes a sequential bottleneck. If one attempts to only acquire write locks on the nodes that will actually be changed, it is difficult to avoid deadlock. If write 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

*Correspondence to: Philip W. Howard, University of Puget Sound, OR 97201, USA.
†E-mail: phil.w.howard@gmail.com

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 rebalance the tree.

Transactional memory provides a more automatic approach to disjoint access concurrency. Many researchers use an RBTree as one of the benchmarks for their software transactional memory (STM) systems. Some of these show good scalability [3, 4]; however, they tend to compare their STM implementation against other STM implementations. Doing so masks the fact that overall performance suffers because of the high overhead of all STM systems. Dragojevic *et al.* [5] compared STM's against a sequential implementation and found that for RBtrees it often took four to eight cores to equal the performance of a single core non-TM solution. With some implementations, they found that performance was worse than a sequential implementation over the full range of processors available on a given system.

Bronson *et al.* [6] developed a concurrent AVL (Adelson-Velskii and Landis) tree[‡]. Their approach allows readers to proceed without locks, but the readers have to perform consistency checks at each node to see if the tree has changed or is in the process of changing. If a concurrent change is detected, the reader has to wait or retry. Because readers do not acquire locks, fine grained locking of the writers is less problematic. Their AVL tree is quite complicated, and this complexity degrades performance as more code must execute at each node of the tree. Our approach allows over twice the read performance and up to three times the single threaded update performance (see Section 4 for details). Their approach allows concurrent scalable updates but performance is no better than for a more straightforward STM implementation.

In order to accelerate update operations, a number of researchers have attempted to decouple rebalancing from insert and delete [7, 8]. Decoupling allows updates to proceed more quickly because individual inserts and deletes do not have to rebalance the tree. The rebalancing work can potentially be performed in parallel and some redundant work can be skipped. Decoupling does not improve read access time. Readers still need some form of performance limiting synchronization to protect them against concurrent updates.

Pugh [9, 10] took another approach to concurrency—rather than parallelizing an existing data structure, he created a new one. Skip lists are linked lists that have room in each node for extra pointers. The extra pointers allow searches to skip over nodes, thus finding the target node faster. Skip lists require extra memory for the additional pointers, but they have the advantage of only requiring local operations for insert and delete. Skip lists have probabilistic expected cost of O($log(N)$), and the locality of inserts and deletes mean they also have good scalability properties. Pugh's approach of designing a new data structure to meet the concurrency challenges of RBtrees is effective but is more labor intensive than our approach of applying relativistic programming to existing RBtrees. The fact that designing a new data structure was considered necessary lends further evidence to our argument that RBtrees present a very difficult concurrency challenge. The difficulty of this challenge is a key motivation for our choice of RBtrees as a test case for relativistic programming.

Binder *et al.* [11] developed a multiversion search tree. Their approach maintains a minimum of two copies of the tree: a write-tree used for updates and one or more read-trees used for reads. When a new read-tree is created, only nodes that have been modified since the last read-tree creation are copied. The rest of the read-tree is constructed with references to the already existing read-tree(s). This approach has the advantage that readers only access read-only data, so they can proceed without synchronization. If a reader makes multiple queries to the same read-tree, it is guaranteed that the tree will remain constant across all reads. This approach has several disadvantages: the visibility of writes is delayed until the next read-tree is created. Not all applications can tolerate this delay. This approach also requires the duplication of the entire tree[§]. This copy incurs both a memory cost and a performance cost as CPU time must be expended to periodically create a new read-tree.

As will be seen, our approach bears some similarities to the multiversion search tree of Binder *et al.* In our approach, readers do not need to synchronize with each other nor with writers, and

[‡]AVL trees are similar to RBtrees, but they have a different balance property.
[§]The nodes in the read-tree have fewer fields than the nodes in the write-tree, thus reducing the memory cost of the copy. However, there must be a copy of the entire structure of the tree.

in our approach, some nodes are copied. The two approaches are also markedly different. In our approach, rather than having a persistent copy of the tree for readers, node copies exist only for the duration of the write operation that required the copies. Our approach gives early access to updates, and our approach does not require the overhead of a thread that periodically scans the tree looking for changed nodes.

Our RBTree implementation is based on a new concurrent programming methodology known as relativistic programming. Relativistic programming is a relaxed consistency programming methodology that allows reads to proceed concurrently with writes to the same data. Relativistic programming relaxes the ordering requirements between reads and writes by allowing each reader to observe the data structure in its own temporal frame of reference rather than requiring a total order on all operations. This relaxation allows for very efficient and highly scalable reads. We show in Section 4 that our implementation has low overhead, wait-free, linearly scalable read performance out to at least 64 hardware threads even in the presence of concurrent updates.

Unlike some other concurrent methodologies, relativistic programming yields algorithms that are easy to understand and reason about. Although relativistic programming allows for relaxed ordering constraints, the ordering is not chaotic. Primitives are available to control what ordering is allowed. In fact, many relativistic data structures are linearizable (see Section 6 for details).

Although relativistic programming is not yet a fully general technique, the development of a relativistic data structure as complex as a concurrent RBTree is a significant milestone. Other work has shown how relativistic programming can be used to perform operations on simpler data structures such as hash tables and lists [12–14], but none of these other data structures posed the challenges of an RBTree. An RBTree requires multi-node updates to rebalance the tree. Our work shows that relativistic techniques can handle these multi-node updates without requiring readers to wait.

The rest of this paper is outlined as follows: Section 2 discusses the ordering constraints that are, and are not, preserved by relativistic programming. This section also provides a justification for why these ordering constraints are appropriate for concurrent RBtrees. Section 3 presents our implementation for a relativistic RBtree. Section 4 shows the performance of our implementation compared with RBtrees implemented using other synchronization mechanisms. Section 5 discusses some of the issues and trade-offs involved in performing complete tree traversals (as opposed to single lookups). Section 6 shows how our implementation has the correctness properties provided by linearizability even though it provides the performance benefits of relativistic programming. Finally, Section 7 presents concluding remarks.

2. RELATIVISTIC PROGRAMMING

Relativistic programming is a methodology for developing concurrent programs. The name, and fundamental intuition, is borrowed from Einstein's theory of relativity in which each observer has their own temporal frame of reference. Specifically, the observed order of events may vary among different observers due to the different time delays incurred by the communication of those events over the distance from their source to the observer (Figure 1). This approach to modeling event ordering in a distributed environment is both natural and intuitive, so long as causality is preserved.

The relativistic approach to modeling event order stands in stark contrast to the notion of linearizability [15–17] in which all participants must agree on an order. Linearizability is a useful concept for reasoning about the correctness of a concurrent implementation in terms of its equivalence to some sequential implementation. However, linearizability is not a necessary correctness condition for concurrent data structures, and in fact enforcing it requires unnecessary coordination among participants that limits scalability and hurts performance [18]. Correctness criteria are data structure specific and must be specified and enforced in new implementations of sequential or concurrent data structures. Relativistic programming is concerned with minimizing the coordination required to enforce these correctness criteria in concurrent implementations.

The relaxed ordering of relativistic programs can be illustrated by the timeline in Figure 2. Operations $W1$ and $W2$ are non-causally related writes and operations $R1$ and $R2$ are reads. Both reads run concurrent with both writes. It is possible for $R1$ to observe $W1$ but not $W2$ and for $R2$ to observe $W2$ but not $W1$. In other words, the readers would observe the writes as happening in

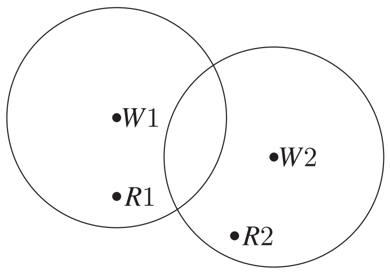


Figure 1. Events $W1$ and $W2$ are flashes of light. The light takes time to travel through space. The circles show the wave front as the light travels from its source. Observer $R1$ has already seen event $W1$ (the wave front has passed $R1$) but not event $W2$. Observer $R2$ has already seen event $W2$ but not $W1$. Thus, the two observers will disagree on the order of events $W1$ and $W2$. If $W2$ was causally dependent on $W1$, then $W2$ could not occur until after the wave front from $W1$ reached the location of $W2$. As a result, the wave front for $W2$ would be contained within the wave front of $W1$, and all observers would see $W1$ prior to $W2$.

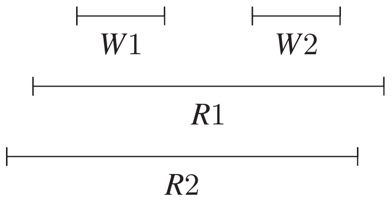


Figure 2. Readers $R1$ and $R2$ can observe writes $W1$ and $W2$ as happening in a different order.

different orders: Reader $R1$ would conclude that $W1$ happened first, and $R2$ would conclude that $W2$ happened first. It is important to note that the order mentioned earlier represents the reference frame of a particular reader. There is no 'global observer' to determine the 'correct' order. Each reader has their own relative view of concurrent operations, which may differ from the view of other concurrent readers. The scope of reordering is bounded. A given reader can see updates concurrent with itself in any order. However, two updates that are not concurrent with the same reader will be observed in the same order by all readers.

Relativistic programming does not impose a total order on operations. Instead, relativistic programming uses barrier-like write-side primitives to preserve causal relationships between operations. Placing the burden for preserving causality on the writers is natural and fits the illustration given in Figure 1. If $W2$ is causally dependent on $W1$, then there is a restriction on when $W2$ can happen, but there is no restriction on when $R1$ and $R2$ can observe state.

If there are no causal dependencies between operations, they are allowed to be reordered. Reads are viewed as observers—the presence of a read does not cause an effect. For this reason, reads can proceed at any time and never have to wait for reads or writes. Writes, on the other hand, do cause effects. Some writes may have to be delayed in order to preserve the causality of their effects.

Referring back to Figure 2, if the two writes were causally related, then any reader that observed $W2$ must also observe $W1$. If no readers were concurrent with both writes, then this ordering would be guaranteed. If there is a delay between the two writes such that all reads that existed at the conclusion of the first write have finished before the second write takes place, then a read could be concurrent with the first write or the second write but not both. Any reader capable of seeing the second write would be ordered strictly later than the first write, thus preserving causality.

The cost of preserving causality is incurred almost exclusively by writers. Writers are able to delay certain operations just long enough to preserve causal ordering. Readers proceed using only light-weight, wait-free primitives, making them high performance and linearly scalable.

Even though relativistic programming does not require a total order on operations, many relativistic algorithms are linearizable. We show in Section 6 that the lookup and update operations on our relativistic RBtree are linearizable. In Section 5, we examine various traversal algorithms, some of which are not linearizable.

2.1. *When is this OK?*

To get the maximum benefit from relativistic programming, the programmer must understand what operations are causally related, and what mechanisms are available to preserve causality. With locking, all of the operations within a critical section are considered atomic with respect to other critical sections. Atomicity makes it easier to reason about correctness; however, atomicity greatly restricts concurrency and thus performance and scalability. With relativistic programming, it is possible to impose a total order on events, but doing so would have a similarly negative impact on write performance. Note that because writers bear the cost of preserving causality, totally ordering writes would not impact the performance and scalability of read-only operations. This is in sharp contrast to locking solutions, which impact both read and write performances.

If two events are independent or commutative, then their order does not matter—there is no causal relationship between them. If there is a causal relationship between two events, then their order does matter and a relativistic program must ensure that this relationship is preserved. To illustrate these concepts, we will consider two examples.

A node can be added to a linked list by creating the node, initializing it, and then linking the node into the list. There is a causal relationship between creating the node (allocating memory for it) and initializing the node. The memory must clearly be allocated before it can be initialized. Note that prior to linking the node into the list, the memory is essentially private to the creating thread so this part of the code can be considered sequential. Compilers guarantee the causal relationships implied in sequential code, so no additional steps need to be taken to preserve this relationship. The individual writes involved in initializing the node are often independent and commutative. Beyond this, because no other thread can see this node, it does not matter what order the fields of the node are initialized. There is also a causal relationship between initializing the node and linking it into the list. Once the node is linked into the list, readers can access the node, so it is important that the node be initialized prior to being accessed. A relativistic implementation must guarantee that this causal relationship is preserved. There is a primitive called *rp-publish* for just this purpose. The *rp-publish* primitive guarantees that all previous writes are visible to readers prior to the value being written by the *rp-publish*. If the *rp-publish* primitive is used to link the node into the list, it will guarantee that all writes that occurred earlier in program order (namely, the initialization of the node) are visible to other threads before the value written by the *rp-publish*. This guarantee is sufficient to preserve the causal relationship between initializing the node and linking it into the list.

For the second example, consider removing a node from a linked list by unlinking the node and then freeing the memory for that node. There is a causal relationship between these two steps: the memory can be freed because the node is no longer part of the list. However, a reader may have obtained a reference to the node prior to it being unlinked. If the memory for the node is reclaimed the reader could observe the reclamation (by crashing because the memory was reused for another purpose) without having observed the unlink. To preserve causality, the writer must guarantee that no readers hold a reference to the unlinked node prior to freeing the memory for that node. Reference counts are a common mechanism used to preserve this form of causality. However, reference counts mean that readers must also write memory when they update the reference count. The relativistic programming mechanism used to preserve this type of causality is to wait until all readers, which existed at the time of the unlink, have finished prior to reclaiming the memory for the node. New readers cannot obtain a reference to the node—it has been unlinked from the list. So, once all readers concurrent with the unlink have finished, it is safe to reclaim the memory. The *wait-for-readers* primitive is used for this purpose.

2.2. *Relativistic Programming Primitives*

This section describes the primitives that are used by relativistic programs. The primitives fit into two categories. The first category is used to bound sections of code much as *lock*, and *unlock* bound critical sections when mutual exclusion is used.

The second category is used to preserve causality. When program order is sufficient to preserve causality, the pair of primitives *rp-publish* and *rp-read* are used to prevent optimizing compilers and relaxed consistency CPUs from reordering execution in such a way that causality is violated[¶]. There are cases where program order alone is not sufficient to preserve causality. In these cases, the *wait-for-readers* primitive can be used to delay subsequent writes such that all readers will see the causally dependent writes in the correct order.

The individual primitives are as follows:

write-lock, write-unlock	These primitives provide synchronization between writers. The use of these primitives does not impact readers—readers proceed oblivious to the presence of a writer. Relativistic programming does not specify what form of synchronization is to be used between writers. Most implementations use mutual exclusion; however, in other work [19, 20], we have shown that STM can be used to synchronize relativistic writers. Writes can happen transactionally, while reads happen relativistically completely outside the transactional system.
start-read, end-read	These primitives bound the code where a reader holds references to the data structure. This is analogous to mutual exclusion where references to the data structure are not allowed outside critical sections. Most relativistic programming implementations have low-latency, constant-time implementations of these primitives so their use has minimum impact on read performance.
wait-for-readers	This primitive waits until 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 the *wait-for-readers* operation to wait for every thread that was inside a read-section at the beginning of the *wait-for-readers* to exit that read-section. Read-sections that begin after the *wait-for-readers* will not delay the *wait-for-readers*.
rp-free	This primitive is called by writers to schedule the future reclamation of memory. It is the equivalent of calling *wait-for-readers* followed by freeing the memory. However, the operation is performed asynchronously so the calling thread does not wait; *rp-free* simply queues the request and then returns.
rp-publish	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. *rp-publish* takes two arguments: a destination and a value. It has the effect of assigning the value to the destination after appropriate memory barriers have been issued.
rp-read	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-read* needs only read the pointer; no barriers are required.

In order to get the read-side benefits of relativistic programming, there must be efficient implementations of *start-read* and *end-read*. In particular, they must be low overhead and wait-free. Efficient implementations are possible because readers do not synchronize with writers in the traditional sense. Readers never have to wait for writers or other readers. Writers on the other hand, do block for readers through the *wait-for-readers* primitive. Readers need to communicate their existence to writers that call *wait-for-readers*, but writers do not need to communicate their existence to readers. This asymetry allows read-side primitives that do not require any expensive synchronization [21]. To demonstrate this claim, we present two possible implementations. A complete description of production worthy implementations is presented elsewhere [14, 22–25].

[¶]Many descriptions of concurrent algorithms ignore the fact that modern CPUs are free to reorder the execution of code. The combination of compilers and CPUs tends to preserve sequential properties of the code but not concurrent properties. Relativistic programming acknowledges this fact and provides primitives to control this behavior so that correct concurrent execution is preserved.

The pseudo-code in Listing 1 shows a potential implementation of *start-read*, *end-read*, and *wait-for-readers*. There is a global Epoch counter that is incremented on each call to *wait-for-readers*. There is an array of epochs, one per reader. When a reader starts, it stores the global epoch into its per-reader epoch. When the reader finishes, it clears its per-reader epoch. Because only one reader writes to each location in the per-reader epoch list, no synchronization is required for these writes. The *wait-for-readers* primitive waits for each reader to either not be reading or to be reading in an epoch newer than the one being waited on. No atomic read-modify-write instructions are required in the read-path[‖]. The pseudo-code for *wait-for-readers* includes an atomic increment followed by a fetch of the incremented value. Because the fetch is not atomic with the increment, it is possible that the fetch will retrieve a value larger than the result of the *atomic-inc*. However, the algorithm still behaves correctly because *wait-for-readers* will wait for any readers that were in the epoch that existed prior to the call to *wait-for-readers*. If an atomic *add-and-fetch* primitive is available, it could be used instead of the *atomic-inc* and non-atomic fetch.

```
start-read() {
  Reading[reader] = Epoch;
}

end-read() {
  Reading[reader] = 0;
}

wait-for-readers() {
  int my_epoch;

  atomic_inc(Epoch);
  my_epoch = Epoch; // guaranteed larger than Epoch prior to call

  for (readers) {
    while (Reading[reader] != 0 &&
        Reading[reader] < my_epoch)
    {}
  }
}
```

Listing 1. Possible implementation of relativistic programming primitives

A second implementation, used by Classic RCU [22, 25] within the Linux Kernel, makes use of the fact that RCU read-sections are not allowed to yield the CPU (e.g., through blocking calls). For preemtible kernels, *start-read* and *end-read* disable preemption for the duration of the read-section. For non-preemptible kernels, these primitives are completely optimized away. Under these conditions, read-sections always run to completion—they are never interrupted and the scheduler cannot take their time-slice from them. As a result, a context switch on a given processor implies that no read-section was present at the time of the switch. The *wait-for-readers* (called *synchronize_rcu* within the Linux kernel) waits for (or forces) a context switch on each processor. Because each processor goes through a context switch, each processor has gone through a state where there are no read-sections. Therefore, all read-sections that existed prior to the *wait-for-readers* are guaranteed to have completed.

2.3. *Always-consistent data*

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

[‖]Some architectures require a memory barrier, which was not shown in the pseudo-code. Memory barriers do not require the round-trip communication of atomic read-modify-write instructions, and they are parallelizable so they affect neither performance nor scalability as much as atomic read-modify-write instructions.

a reader is almost as easy as developing a non-concurrent reader. The only restrictions placed on readers are as follows:

1. Read-sections must be bounded by the *start-read* and *end-read* primitives.
2. Readers must not hold references to the data structure outside read-sections. This second limitation is exaclty the same as with mutual exclusion where a reader is not allowed to hold references to the data outside a critical section protected by locks.
3. The *rp-read* primitive must be used when reading data written by *rp-publish*.

For readers to be able to proceed without having to check the consistency of the data, updaters need to keep the data in an always-consistent state. Correct relativistic writers must ensure that the program order of changes leaves the data structure in an always-consistent state and that relativistic readers will see causally dependent writes in program order. The following three requirements are used to meet these objectives:

1. *rp-publish* must be used when making nodes visible to readers.
2. *rp-free* must be used when freeing no-longer-used memory.
3. Care must be taken in the order that updates are made so readers never see an invalid state.

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 private copy of the node, then atomically switch the new node with the old one using *rp-publish*. In the second case, when structural changes are made, care must be taken to ensure that readers can only see valid states. To illustrate this, consider the zig restructure depicted in Figure 3 (the zig restructure is part of the rebalancing process for RBtrees). If a reader is at node C looking for node B at the time the restructure happens, the reader may not find B. This is because after the restructure, B is above C instead of below it. For this reason, the restructure operation must be performed such that readers can never miss the node represented by B (see Section 3.2.2 for our solution to this problem).

2.4. Justifying relativistic ordering

As stated earlier, relativistic programming does not require all readers to agree on the order of updates. Any time multiple readers are concurrent with multiple non-causally related writers, the readers can disagree on the order of updates. Consider the timeline shown in Figure 4. It is

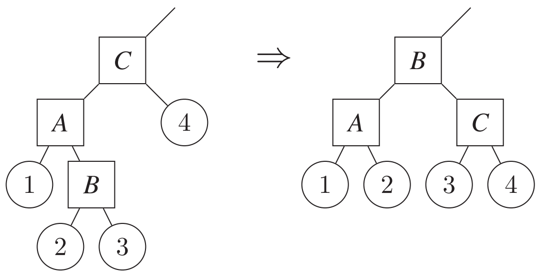


Figure 3. A reader at node C looking for B at the time of the restructure will fail to see B unless the updater is careful in the order in which changes are made to the tree.

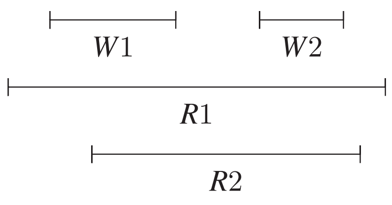


Figure 4. Readers $R1$ and $R2$ can observe writes $W1$ and $W2$ as happening in a different order.

possible for $R1$ to see $W1$ but not $W2$ and for $R2$ to see $W2$ but not $W1$. With a locking solution, all readers would agree on the order of the writes. For example, if the start of the operation in Figure 4 is the point in time when a reader–writer lock was requested, then if the reader–writer lock favored readers both $R1$ and $R2$ would occur before either writer. If the reader–writer lock favored writers or was fair, then $R1$ would occur before $W1$, and both readers would occur before $W2$. But note that regardless of the reader–writer implementation, the total time for all four operations would be longer because reads and writes could not run concurrently.

What are the conditions under which this out-of-order behavior is acceptable? Suppose the directory information of a phone company was stored in a RBTree. Reads would be individual lookups of a phone number. Writes would be updates to the directory such as a new customer signing up for service (an insert) or a customer canceling service (a delete). A complete traversal of the tree would be required to print a phone book.

If a lookup was concurrent with multiple independent updates, then the lookup would be interested in, at most, one of those updates. If the lookup was for some customer A, it does not matter whether that lookup occurred before or after an update for some other customer B. Under these conditions, the order of the updates is irrelevant so the relativistic ordering is acceptable.

Traversals are more challenging because they observe all nodes in the tree. Traversals are of long duration with respect to updates (traversals are $O(N)$, while updates are $O(\log(N))$), so it is reasonable to expect that many updates might be concurrent with a single traversal. The traversal will see some collection of the concurrent updates but not the others. And more specifically, two concurrent traversals may see different collections of updates, and the state seen by any single traversal may not reflect a state that existed in any instant of time as defined by the order of the writes. With our phone company example, it is easy to argue that this does not matter. Everyone understands that phone books are out-of-date by the time they are received. In addition, customers do not have access to an ordered list of updates so they are not in a position to argue that their phone book is inconsistent with the order of updates. The phone book is understood to be an approximate representation of the state contained in the tree. If an update occurred at about the same time as the phone book was printed, there is no guarantee that that particular update will be included.

For another traversal example, suppose an operating system scheduler kept tasks in a RBtree sorted by the order in which they will be scheduled. If someone displayed a list of tasks in scheduled order, this would require a complete traversal of the tree. By the time the list was examined, it would likely be out-of-date. Some new tasks would have been scheduled, and other tasks would have already been run. An exception to this already out-of-date assumption would be if the scheduler wanted to make a decision based on the global state instead of just scheduling the highest priority task. Under these conditions, the scheduler should prevent concurrent updates while it is making its global decision. Without concurrent updates, there will be no disagreement on order.

We have shown through several examples that the reordering allowed by relativistic algorithms is often acceptable. We do not claim that it is always acceptable, only that it is often the case that when updates are independent and commutable that the reordering allowed by relativistic algorithms can be tolerated.

3. A RELATIVISTIC RED-BLACK TREE ALGORITHM

Because RBtrees are well-known and well-documented [7, 26–28], 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 RBtree algorithms without discussing the glue that combines these steps, because the glue is not impacted by the relativistic implementation.

RBtrees 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. (Some implementations of these primitives pass arguments specifying what the last accessed node was.)

RBtrees are sorted by preserving the following properties:

1. All nodes on the left branch of a subtree have a key less than the key of the root of the subtree.
2. All nodes on the right branch of a subtree have a key greater than or equal to 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 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 most often O(1) but in the worst case is O(log(N)). This means that inserts and deletes can also be performed in O(log(N)).

An insert or delete may violate the balance properties. If so, the tree is rebalanced by recoloring nodes or performing restructure operations (sometimes called rotations). Restructures always involve three adjacent nodes: child, parent, and grandparent. Figure 5 illustrates the two types of restructure operations. Following an insert, at most one restructure is required to restore balance. Following a delete, at most two restructures are required [7].

Our implementation imposes a single restriction on the trees: we do not allow the same key to have different values at the same time. This means that our implementation is suitable for a *map*,

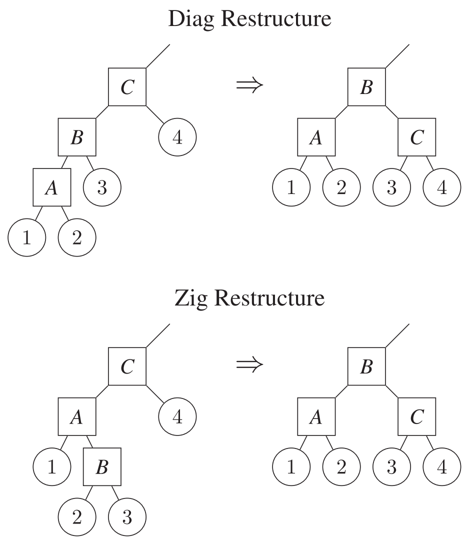


Figure 5. Restructure operations used to rebalance an RBtree. There are left and right versions of these, but they are symmetric so only the left version is shown here.

which only allows a single value for each unique key, but is not suitable for a *multi-map* that allows multiple values for each key.

We make the following observations about readers performing a lookup (for traversals, see Section 5):

1. Readers ignore the color of nodes.
2. Readers do not access the parent pointers in nodes.
3. Temporarily having the same item in the tree multiple times will not affect lookups provided all copies are in correct sort order locations within the tree. A positive result will return the first copy encountered (and because they are duplicates, it does not matter which copy is returned). A negative result for a given key (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.

Binary trees can be implemented as *external trees* in which all keys reside at leaves of the tree. Internal nodes are routing nodes used to find a particular leaf. For external trees, internal nodes always have two children, and leaves have no children. External trees make some operations simpler; however, an external tree requires almost double the number of nodes because a tree with N keys requires N leaves and $N - 1$ routing nodes.

Our implementation is an *internal tree*, which stores ⟨key,value⟩ pairs at each node. Internal trees have nodes that are clearly internal nodes because they have two children. They also have nodes that are clearly leaves because they have no children. But internal trees also have nodes with a single child. Single child nodes have attributes of both leaves and internal nodes. We use the following terminology to refer to different types of nodes:

left-leaf	A node with a null left branch.
right-leaf	A node with a null right branch.
leaf	A node that has at least one null child. That is, a node that is either a left-leaf or a right-leaf.
left-internal	A node with a non-null left child.
right-internal	A node with a non-null right child.
internal	A node with two non-null children. That is, a node that is both left-internal and right-internal.

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

Given the above observations, the following can be said about the steps in an update:

Insert New nodes are always inserted at the bottom of the tree. This is possible because if *prev(new node)* is a right-internal node, then from the observation above, the new node must be a left-leaf. If *prev(new node)* was not a right-internal node, then it is a right-leaf, and the new node will be the right child of that node. A concurrent reader will either see the new node or not depending on whether *rp-publish* of the insert happens before the *rp-read* of the pointer changed by *rp-publish*. But concurrent readers will never see an inconsistent state.
The insert may leave the tree unbalanced. If so, restructures or recolors (see the succeeding text) are required to restore the balance properties of the tree.

Leaf Delete A left-leaf can be deleted by having the left-leaf's parent point to the left-leaf's right child. A right-leaf can be deleted by having the right-leaf's parent point to the right-leaf's left child. In both cases, the delete is effected with a single pointer change. Similar to an 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.

The delete may leave the tree unbalanced. If so, restructures or recolors (see the succeeding text) are required to restore the balance properties of the tree.

Interior Delete 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 left-leaf. Because 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 (section 3.1). This is because the swapped node exists in the tree, and therefore, it must be observable in correct traversal order.

The delete may leave the tree unbalanced. If so, restructures or recolors (see the succeeding text) are required to restore the balance properties of the tree.

Restructures Restructure operations, sometimes called rotations, are used to rebalance the tree. Restructures always involve three adjacent nodes: child, parent, and grandparent. See Figure 5 for an illustration of the two types of restructure operations. Much like interior delete, restructures involve moving nodes. This requires special handling to keep the tree in an always-consistent state (Section 3.2).

Recolor Nodes get recolored as part of the rebalancing process. Recoloring does not involve changing the structure of the tree, only the colors applied to particular nodes. Because 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 interior delete and restructure. These are described in greater detail in the succeeding text.

3.1. Interior delete algorithms

Section 3.1.1 discusses the general interior delete algorithm; Section 3.1.2 discusses an optimized special case.

3.1.1. General internal delete Consider the delete of node B shown in Figure 6. Because 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 3.1.2

Rather than performing separate swap and delete steps, the two are combined in a single step as shown in Listing 2. C' is a copy of C. C' is changed so that it has the same color as B and the same children as B but the key and data values of C.

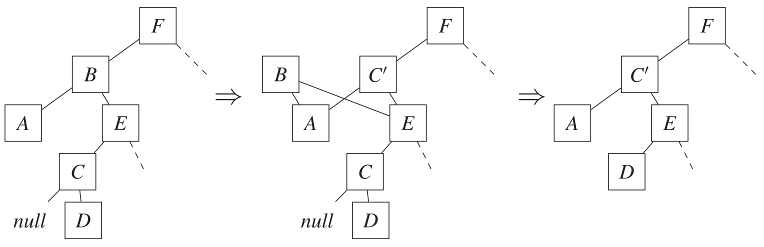


Figure 6. Tree before and after deletion of node B including one intermediate step.

```
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 2. Code for interior delete

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 calls *wait-for-readers* before removing C from the tree, thus ensuring that any readers at or below B will complete their read prior to C being removed. Any new readers will see C' and thus will not need to find C.

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-publish* is used to make reader visible 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 *wait-for-readers* is included so that no readers will miss seeing node C.

3.1.2. Special case: swap node is child of B In the tree shown in Figure 7, 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 3. 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). 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 its place. B is then freed asynchronously by calling *rp-free*.

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

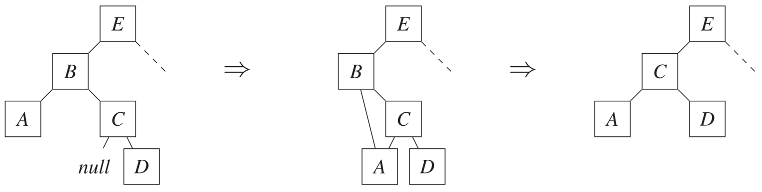


Figure 7. Tree before and after deletion of node B including one intermediate step.

```
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 3. Code for special case interior delete

3.2. 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.

3.2.1. Diag Left Figure 8 shows a subtree with three nodes labeled A, B, and C that need to be rotated so that B is the root of the subtree. The changes are made as shown in Listing 4.

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 right child of B. At this point, the value C is in the tree twice. This is similar to the interior delete in Section 3.1.1. 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 readers. This is because any readers between C and C' will still see C' even after C is removed from the tree.

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.

3.2.2. Zig Left Figure 9 shows a subtree with three nodes labeled A, B, and C that needs to be rotated so that B is the root of the subtree (this is sometimes referred to as a double rotation [7]).

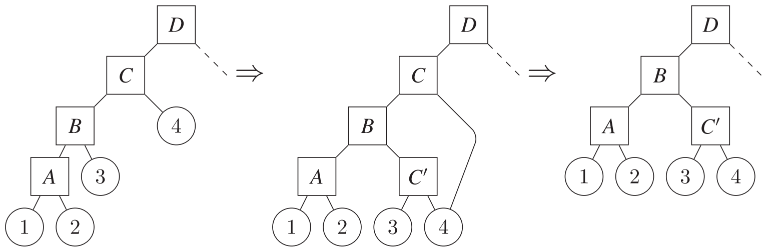


Figure 8. Arrangement of nodes before and after a diag restructure including one intermediate step.

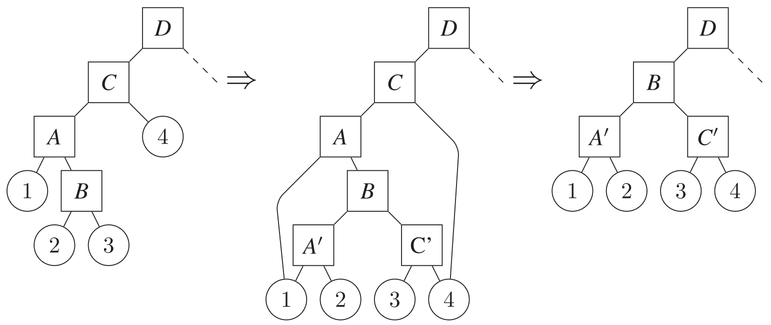


Figure 9. Arrangement of nodes before and after a zig restructure including one intermediate step.

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. Because the first method involves moving the copy up in the tree, it requires a *wait-for-readers*. Even though the second method requires two copies, performance data showed that the second method is faster, so that method is described here (Listing 5).

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. Because the new copy is placed below the original, there is no need for a *wait-for-readers* 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 nodes were made rather than changing nodes in place, and RP primitives were used for pointer assignment and memory reclamation.

4. PERFORMANCE

Our performance claim is not that our implementation is the best possible implementation of a data structure suitable for a Map. Instead, our claim is that relativistic programming is a synchronization mechanism that allows for low overhead, scalable reads even in the presence of concurrent updates. We compare our implementation to RBTree implementations that use other synchronization

```
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);

B.parent = D;

rp-free(C);
```

Listing 4. Code for diag left restructure

```
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 5. Code for zig left restructure

mechanisms but do not compare our implementation against other data structures suitable for implementing Maps. One of the implementations we compare against is a concurrent AVL tree. We chose this implementation not because AVL trees differ from RBtrees but because of the synchronization mechanism used by the concurrent AVL tree.

Performance data was collected using the following synchronization techniques:

nolock No synchronization was used. This is not a valid concurrent implementation because it leads to data corruption. Because our benchmarks are comparing the performance of various synchronization mechanisms, *nolock* was included as a zero-cost mechanism.

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 [29].

rwlw A reader/writer lock that favors writers. The implementation was derived from Mellor-Crummey and Scott [29].

rp This is the relativistic implementation described in this paper.

ccavl The concurrent AVL implementation by Bronson *et al*. [6]

Note 1: all the RBTree algorithms except *rp* used a ‘standard’ RBtree implementation that did not perform copy-on-update. Copy-on-update is slower, and we did not want to bias the results against the non-RP implementations.

Note 2: Bronson’s *ccavl* algorithm was originally developed in Java, and it relied on Java’s garbage collection mechanism. We ported their algorithm to C to match the implementation of our other algorithms. However, we did not provide garbage collection for their algorithm. As a result, the performance numbers reported here for *ccavl* are better than what a complete implementation would provide because our numbers do not include the cost of garbage collection.

Note 3: The RCU implementation for the performance data in this section follows the pseudo-code presented in Section 2.2. The code is available at https://github.com/philip-w-howard/RP-Red-Black-Tree.

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.

Each run was repeated 16 times. The operations and values were randomized so that each thread of each run was performing different operations. The results of all 16 runs were averaged, and the average is reported in the figures. The standard deviation was also computed, but the variance from run to run was small enough, in most cases, not to be visible on the graphs. For this reason, error bars were not included in the graphs.

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. Unless otherwise noted, tests were performed on trees of size 64K nodes[**].

Three sets of benchmarks were run. In the first set, all threads were readers. In the second, one thread was an updater, and the rest were readers. This arrangement shows how concurrent reads affect update performance. In the third set, all threads were updaters.

Performance data was collected on a Sun UltraSPARC T2 running SunOS 5.10 and on an Intel Xeon E7310 system running Linux 2.6.32. The UltraSPARC T2 has eight cores each supporting eight hardware threads for a total of 64 hardware threads. The Intel Xeon has four processors each with four cores for a total of 16 hardware threads.

4.1. Read performance

Figure 10 shows the read performance of the RBtree. The reads here are individual lookups not complete traversals. For complete traversals, see Section 5.4. The following observations can be made from the figure:

1. On the Sun, *rp* read performance scales linearly to at least 64 threads.
2. On the Intel, *rp* scales well, but it is less than linear. Because *nolock* is also less than linear, the non-linearity is not due to synchronization cost.
3. *rp* read performance approaches unsynchronized performance. On the Intel, *rp* values were approximately 93% of the *nolock* values.
4. The best competitor to *rp* (other than *nolock*, which is not safe for updates) was *ccavl* on the Intel. *rp* was at least 60% faster than *ccavl* for all thread counts.

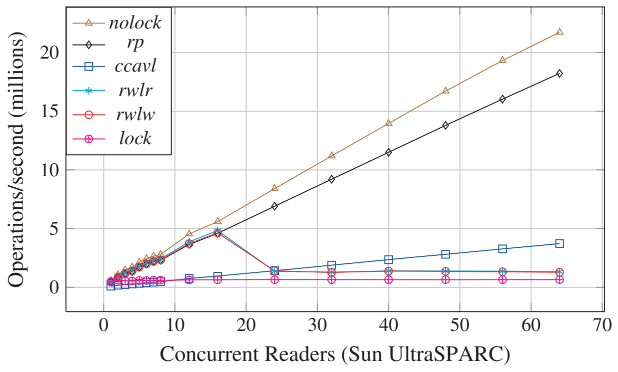
Figure 11 shows the read performance both with and without a concurrent updater. Because this benchmark shows the effects of contention, trees of size 64 were used. The smaller trees generated more contention and thus magnify the effects of contention. On the Sun, the *rp* contended and uncontended read performance lines are indistinguishable showing that the writer did not impact the performance of the readers. On the Xeon, the concurrent writer affected read performance for all algorithms including *nolock*. Because *nolock* was affected, the contention was likely due to memory contention independent of the synchronization mechanism used.

4.2. Update performance

Figure 12 shows contended update performance. The *x*-axis shows the number of read threads. The leftmost data point on each line (zero read threads) shows the uncontended update performance. The remainder of the data points show the contended update performance with a varying number of readers.

Of all the synchronization methods, *ccavl* had the worst uncontended performance. The uncontended performance of *rp* was poor, but with concurrent readers *rp* update performance is better

[**]Our algorithm is intended to scale with the number of processors. We do not comment on its ability to scale with tree size. For this reason, we did not test a wide range of tree sizes.



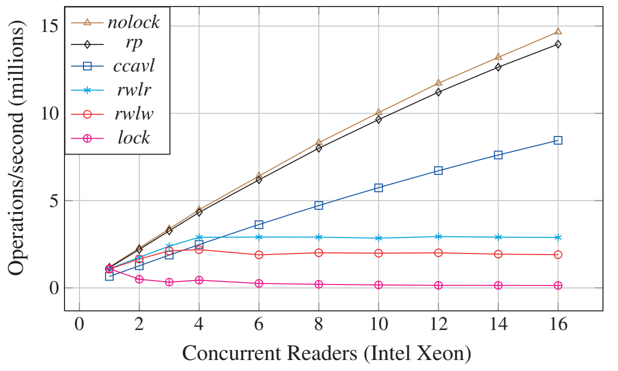


Figure 10. Read performance of 64K node RBtrees. In the UltraSparc data, the *rwlr* and *rwlw* lines are on top of each other because there is little difference between these for read-only workloads.

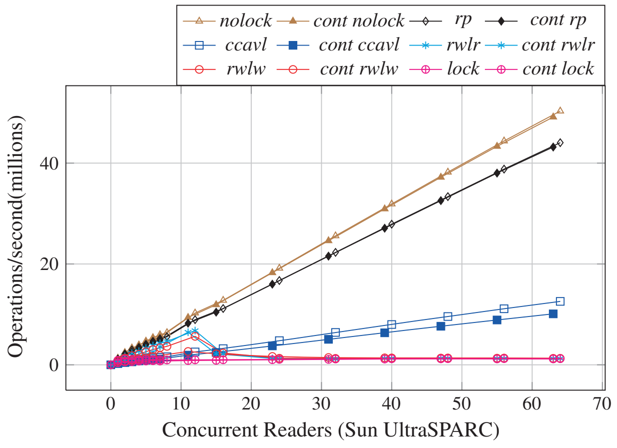
than any of the other synchronization methods (with the exception of *nolock*, which leads to data corruption). As the number of concurrent readers increases, the advantage of *rp* is more pronounced. With a smaller tree, it takes more concurrent readers to give a clear advantage to *rp*. For example, on the Sun, with a tree size of 64 nodes, *rp* had better write-side performance if there were six or more concurrent readers.

4.3. Concurrent updates

The previous section showed the performance of a single writing thread in the presence of a variable number of read threads. The only synchronization method from the previous sections that allowed concurrent writes was *ccavl*. In the previous sections, *ccavl* had poor performance, but those sections did not show *ccavl* in its best light: scalability on the write side.

Although relativistic programming does not directly address concurrent writes, we have shown in other work [19,20] that relativistic programming can be combined with software transactional memory in such a way that the read performance is preserved by relativistic programming, and scalable concurrent write performance is provided by the software transactional memory system. Combining relativistic readers with STM writers was made possible by modifying a software transactional memory system so that it supported the *wait-for-readers*, *rp-free*, and *rp-publish* primitives. These primitives take effect at commit time so the important ordering aspects of the update algorithms are preserved. Because the STM preserves the important ordering properties of the relativistic update, reads can proceed relativistically, completely outside the software transactional memory system. The modified STM, called RP-STM, is described in detail elsewhere [19,20]. RP-STM was built by modifying SwissTM [4].

The STM system used for this work does not build on the Ultrasparc T2 system used for the other benchmarks. So in this section, we present benchmarks only on the Intel Xeon system. We show the performance for the following algorithms:



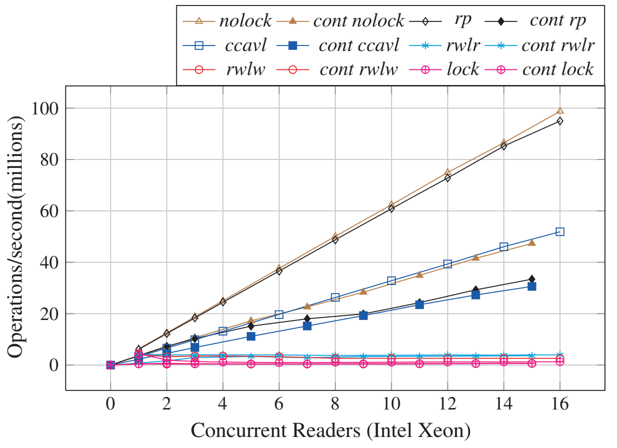


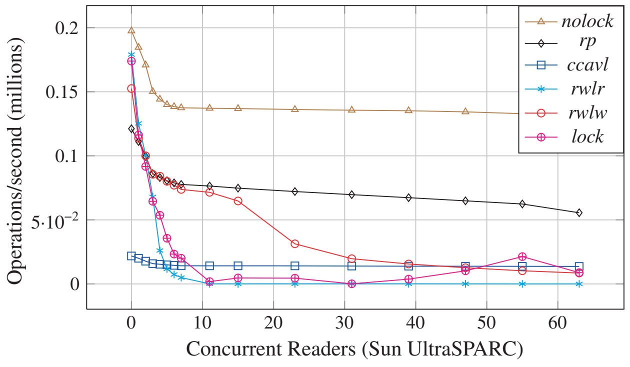
Figure 11. Contended and uncontended read performance of 64 node RBtrees.

rp	This is the relativistic implementation described in this paper. Writers are synchronized with a pthread mutex (but note that the mutex does not affect readers).
ccavl	The concurrent AVL implementation by Bronson *et al*. [6]
RP-STM	The relativistic read, STM update combination described in our other work [19, 20]
SwissTM	The SwissTM system described by Dragojevic *et al*. [4]

Figure 13 shows update performance where all threads are performing updates on a 64K node tree. For low thread counts, *ccavl* is competitive with the STM versions. However, at higher thread counts, the STM versions show much better performance. Because the *ccavl* algorithm is quite complex and difficult to code correctly, there is no advantage to the *ccavl* approach over STM approaches. The *SwissTM* approach shows better write performance than *RP-STM* approach, because the *RP-STM* approach includes the relativistic update, which requires copying some nodes. However, Figure 14, which displays read performance when all threads are performing reads, shows that *RP-STM* has much better read performance, thus justifying the additional cost of the write algorithm.

5. TRAVERSALS

The performance data presented in Section 4 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



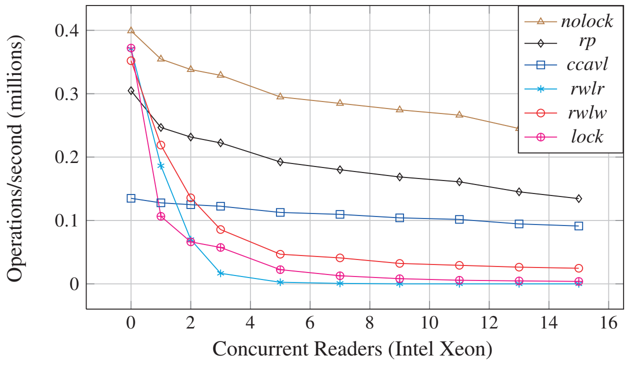


Figure 12. Update performance of 64K node RBtrees with a single updater and multiple readers. The left-most data point shows uncontended write performance. The remainder of the data points show the update performance with a variable number of concurrent readers.

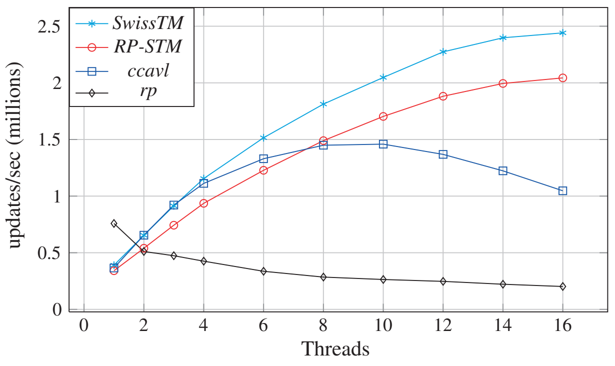


Figure 13. Concurrent update performance of 64K node trees. All threads are performing updates.

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 reorderings present challenges to a relativistic implementation of a tree because they have the potential to result in an invalid traversal.

Three approaches were taken to solve these problems. The approaches differ in their implementation of *next()*. The approaches also differ in their ordering guarantees. All three solutions presented

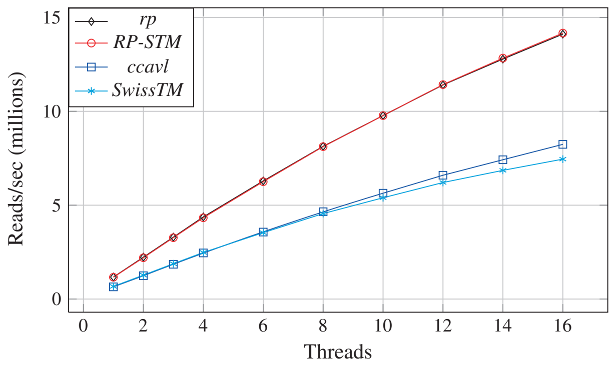


Figure 14. Concurrent read performance of 64K node trees. All threads are performing reads. Note that the *rp* and *RP-STM* lines are on top of each other.

here maintain the properties listed in the succeeding text. The term 'visited' means that a node was returned by the traversal so that it can be processed by user code.

1. All keys that were inserted prior to the beginning of the traversal and that are not deleted until after the end of the traversal will be visited exactly once.
2. Any keys that are inserted or deleted during the traversal will be visited at most once.
3. All keys that are visited will be visited in correct sort order.

One of the solutions we present is linearizable. Linearazability is preserved by preventing updates during a traversal. In other words, for the linearizeable solution, Property 2 does not describe any keys because all inserts and deletes happen either before or after the traversal not during.

Developers sometimes make the assumption that algorithms built on locks, even fine grained locks, provide adequate consistency guarantees. An algorithm that admits to being non-linearizable might be considered suspect. Before presenting our traversal algorithms, let us argue that a solution built on fine grained locks will have the same non-linearizable properties as our non-linearizable solutions. Assume you have a solution built on fine grained locks such that a traverser in one part of the tree does not inhibit updates in other parts of the tree. Consider two updates and two traversers, all concurrent with each other. Update A occurs early in traversal order. Update B occurs late in traversal order. Reader 1 traverses the early part of the tree prior to A but does not traverse the latter part of the tree until after B. Reader 1 observed B but not A. Reader 2 traverses the early part of the tree before A but completes the traversal prior to B. Reader 2 observed A but not B. These operations are not linearizable.

This example does not argue for the correctness of a non-linearizable solution in any particular instance. However, it does show that, for traversals, non-linearizable behavior is possible even when heavier weight synchronization mechanisms are used. In Section 2.4, we described situations where this behavior should be acceptable.

5.1. A reader–writer locking 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 traversal thus the structure of the tree remains unchanged for the duration of the traversal. 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. *start-read/end-read* bounds a relativistic read. These primitives are used for lookups. Relativistic reads do not exclude updates nor traversals.
2. *rw-write-lock/rw-write-unlock* bounds an update critical section. Updates do not exclude relativistic readers but do exclude traversals.

3. *rw-read-lock*/*rw-read-unlock* bounds a traversal. Multiple *rw-read-locks* can be acquired at the same time. A traversal does not exclude other traversals nor relativistic readers but does exclude updates.

Using these primitives, lookups can proceed at any time. Traversals and updates are mutually exclusive, which will cause significant delays to updates while traversals are happening.

5.2. An $O(N\log(N))$ relativistic approach

Consider the following observations about tree traversals and the relativistic algorithms given in Section 3:

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 *wait-for-readers* in the middle of the update. If the *wait-for-readers* must wait for traversals, the wait will be $O(N)$ and will significantly delay updates.
3. The algorithms given in Section 3 assume that readers do not access the parent pointers so parent pointers should not be used by relativistic traversals.

Given the aforementioned 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 approach allows the same relativistic read and update algorithms described in Section 3 to be used. The consequence is that a traversal will take $O(N\log(N))$ time because each call to *next()* does a lookup starting at the root of the tree. However, *wait-for-readers* only delays for single calls to *next* rather than for the full traversal. The tree that is traversed may not represent any state present in a globally ordered time. In particular, it is 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 were not. 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.

Using this approach, traversals will take longer; however, updates will not be significantly impeded by traversals.

5.3. An $O(N)$ relativistic approach

The previous section described an $O(N\log(N))$ approach to traversals that is compatible with concurrent updates. This slower approach was used for two reasons: the update algorithms did not preserve the read consistency of the *parent* pointers in the nodes, and it allowed *wait-for-readers* to only delay for single calls to *next()* rather than for the full traversal.

In Section 3.2.2, we mentioned that two approaches were possible for the zig restructure: one required copying a single node and a *wait-for-readers*, whereas the other required copying two nodes but did not require a *wait-for-readers*. Copying additional nodes can solve both of the reasons for the $O(N\log(N))$ traversals. Copying additional nodes can preserve the read consistency of the parent pointers and can eliminate the need for *wait-for-readers* during an update. Preserving the read consistency of parent pointers means that a relativistic traversal can use the parent pointers so *next()* does not have to perform a lookup starting at the root. Eliminating all *wait-for-readers* calls from updates means that the read-section can span the entire traversal without delaying updates.

The user mode RCU library (available at http://lttng.org/urcu) has recently released code for an RBTree that copies additional nodes thereby preserving the read consistency of the parent pointers and removing all *wait-for-readers* calls from the update path. We chose to present the simpler relativistic algorithm in this paper because it is easier to understand and thus provides a better illustration of how relativistic programming can be applied to a data structure. However, we present benchmarks using the more complex algorithm here, to show how it performs compared with the simpler one.

Using the more complex approach, updates will take longer than with the approach described in this paper; however, traversals will only take $O(N)$. As with the relativistic approach in the previous section, traversals will not necessarily reflect the state of the tree as it existed in any globally ordered time.

5.4. *Traversal performance*

The $O(N)$ relativistic traversal depends on the urcu library, which does not run on SunOS 5.10 so the traversal benchmarks were run on the same four-way quadcore Xeon machine as the concurrent update benchmarks.

Figure 15 shows traversal performance while contending with a single update thread. The tree size was $10,000$ nodes. All three methods scale, however, the performance of the *rp* $O(N \log(N))$ method is noticeably worse than the others because it is $O(N \log(N))$, whereas the other two are $O(N)$. The read performance of the *rwlw* approach was poor for low thread counts because the reader–writer lock favored writers. Once there were enough read threads, multiple readers were ready to claim the read-lock as soon as the writer released the write-lock.

Figure 16 shows the performance of a single update thread in the presence of concurrent traversals on a $10,000$ node tree. The *rwlw* approach has the best uncontended performance, but it degrades rapidly in the presence of traversals because once a traversal acquires the lock, the update must wait for the entire traversal to complete. The *rp* $O(N \log(N))$ algorithm has best overall performance, but this is at the sacrifice of traversal performance as shown in Figure 15. Finally, the *rp* $O(N)$ update algorithm degrades gracefully in the presence of concurrent traversals and still maintains $O(N)$ traversals.

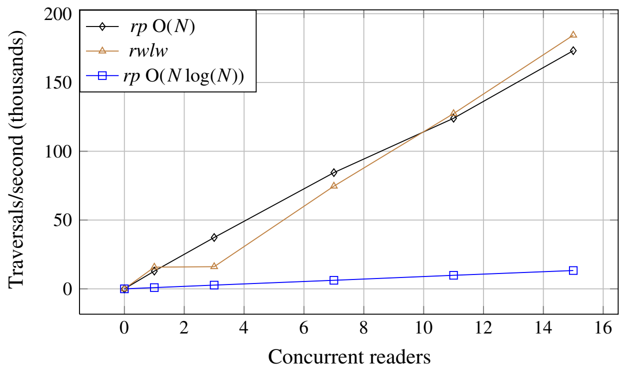


Figure 15. Performance of traversals in the presence of a single update thread on a tree of $10,000$ nodes.

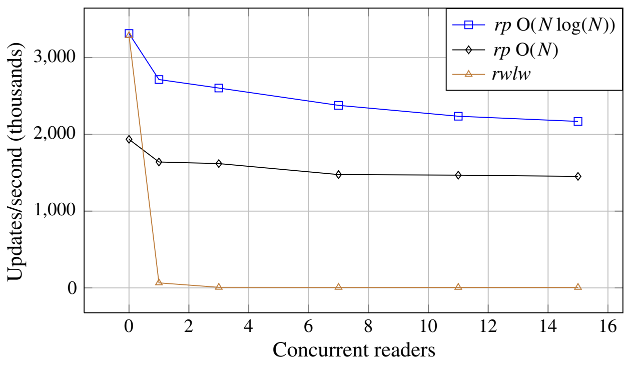


Figure 16. Performance of a single update thread in the presence of concurrent readers performing traversals on a $10,000$ node tree.

The update performance data can be explained as follows: linearizability has a cost, and this cost is reflected in the low contended update performance of the *rwlw* approach. Allowing $O(N)$ traversals has a cost and this is reflected in the difference between the *rp* $O(N\log(N))$ and *rp* $O(N)$ approaches. By assigning a cost to different properties (linearizability and $O(N)$ traversals), a developer can select which traversal method is appropriate based on the importance and cost of the properties.

6. LINEARIZABILITY

Linearizability [15–17] is a widely accepted correctness criteria for concurrent programs. Linearizability means that sequential reasoning can be used in verifying the correctness of an algorithm. An algorithm is linearizable if each operation takes effect through a single atomic operation. In this section, we show that the lookup, insert, and delete operations meet this criteria and thus are linearizable.

Our arguments for the linearizability of these operations will be based on the following properties of our relativistic implementation:

Immutable nodes	The key and value within a node never change once the node is created.
Sort	The update methods preserve the sort property of trees such that all nodes on the left branch are less than the current node, and all nodes on the right branch are greater than or equal to the current node.
Structural integrity	The update methods never move a node in such a way that a lookup may fail to find that node.

In the following subsections, we examine each of the steps of the relativistic tree algorithms and explain why they are linearizable.

6.1. Lookup

A lookup proceeds by examining the key in a node of the tree (initially the root of the tree) and taking one of the three courses of action based on the key in the node:

1. The correct node has been found and will be returned.
2. The lookup will proceed down the left branch.
3. The lookup will proceed down the right branch.

We argue that the *rp-read* used to transition to the last node examined on a lookup is the linearization point. The following reasons justify the choice of this *rp-read* as the linearization point:

The *rp-read* involves reading a pointer, and this operation is atomic on all modern hardware. The *immutable nodes* property means that even if the key is examined several times (e.g., once to determine if the lookup has found the node in question and again to decide which branch to look down) the answer will be the same each time. The *sort* property means that the correct branch will always be searched. The *structural integrity* property means that a node below a reader will never be moved above the reader in such a way that the reader will not see that node. As a result, the last *rp-read* in a lookup determines the result of the lookup provided the updates preserve the *immutable nodes*, *sort* and *structural integrity* properties. We show in the next sections that these properties are preserved.

6.2. Insert

Nodes are always inserted at the bottom of the tree. The *rp-publish* that makes the new node visible to readers serves as the linearization point for an insert. The *rp-publish* involves writing a pointer, which is atomic on all modern hardware. Prior to the *rp-publish*, the node does not exist in the tree, and following the *rp-publish*, it does. Provided the restructure operation (described later) preserves the *immutable nodes*, *sort* and *structural integrity* properties, the restructure will not invalidate the *rp-publish* as the linearization point.

6.3. Leaf delete

Leaf nodes are deleted with a single assignment. The assignment has the parent point to the non-null branch of the leaf node (if it exists) instead of the leaf node. This assignment serves as the linearization point of the leaf delete. The assignment is atomic on all modern hardware. Prior to the assignment, the node does exist in the tree, and following the assignment, it does not. Provided the restructure operation (described in the succeeding text) preserves the *immutable nodes*, *sort* and *structural integrity* properties, the restructure will not invalidate the assignment as the linearization point.

6.4. Interior delete

When an interior node is deleted, the next node in sort order (the leftmost node on the right branch) is moved up into the location of the deleted node. Figure 17 enumerates the reader visible states in this process. The first reader visible change is when node C' gets linked into the tree in place of node B using *rp-publish*. This *rp-publish* is the linearization point for the removal of B as can be seen by considering lookups for node B. The linearization point for the lookup is the *rp-read* of `F.left`. If the *rp-read* happens before the *rp-publish*, the lookup will find B; otherwise, it will report that B does not exist in the tree. The *rp-publish* preserves the *immutable nodes*, *sort* and *structural integrity* properties.

The second reader visible change is when C is removed from the tree by linking D in its place. This is performed using *rp-publish* to update `E.left`. The combination of the two changes has the effect of moving C to a new location within the tree in possible violation of the *structural integrity* property. However, a *wait-for-readers* occurs between the changes so that any reader at B or E looking for C will find C before it is removed. Any reader looking for C that began after the *rp-publish*, which inserted C', will find C' so it will terminate its search before encountering node E.

After C is removed from the tree, the node still exists in memory because concurrent readers may still have a reference to it. The final change takes place when C is reclaimed. This happens after all current readers have finished so that no readers have a reference to C when it is reclaimed.

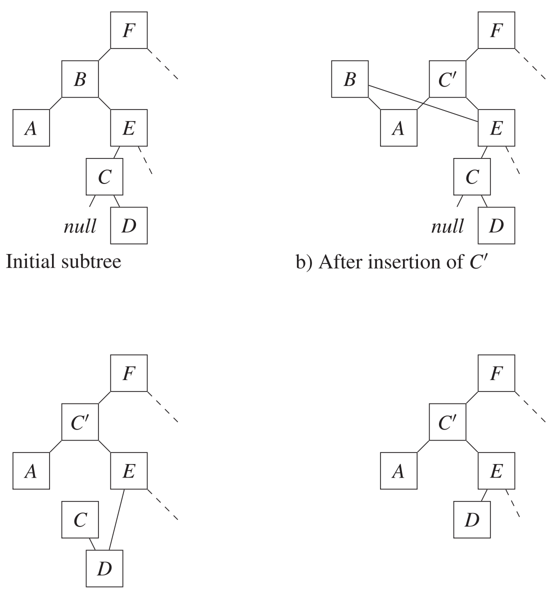


Figure 17. All reader visible states in the deletion of node B through an interior delete.

6.5. Restructure

Restructures come in both the diag and zig forms. Restructures happen only after inserts or deletes. The linearization point for the update occurred in the insert or delete, so we only need to show that the restructure operations do not violate the *immutable nodes*, *sort* or *structural integrity* properties. We consider the diag restructure first.

Figure 18 shows the reader visible states of a diag restructure operation. The first change places C' below B. This temporarily violates the *sort* property because node 4 appears on both the left and right branches of C. However, lookups for all nodes on the subtree rooted at C will terminate correctly. In particular, a reader at B will find node 4 through C'. Any readers at or above C will find 4 as the right branch of C. Any readers looking for C will stop at C, so the fact that there is a duplicate on the left branch of C will not cause any problems. So even though the *sort* property is violated, the violation will not cause any lookups to fail. The second change corrects the temporary violation of the *sort* property.

The second change removes C from the tree leaving C'. The effect of the first two changes is that C is moved to a new location within the tree. However, after each step, any lookups will complete correctly regardless of where in the tree the reader was when the change happened. Thus, the *structural integrity* property is preserved.

After C is removed from the tree, the node still exists in memory because concurrent readers may still have a reference to it. The final change takes place when C is reclaimed. This happens after all current readers have finished so that no readers have a reference to C when it is reclaimed.

Figure 19 shows the reader visible states of a zig restructure. The first change is inserting A' below B. This change does not violate either the *sort* or *structural integrity* properties. The second change inserts C' below B. This change temporarily violates the *sort* property because node 4 is on both the left and right branches of C. As with the diag restructure, all lookups within the subtree will complete correctly. The third change corrects the temporary violation of the *sort* property by removing both A and C. The *structural integrity* property is not violated by this change because A and C continue to be visible to all readers that could potentially be looking for these nodes.

After A and C are removed from the tree, the nodes still exist in memory because concurrent readers may still have a references to them. The final change takes place when A and C are reclaimed. This happens after all current readers have finished so that no readers have a reference to these nodes when they are reclaimed.

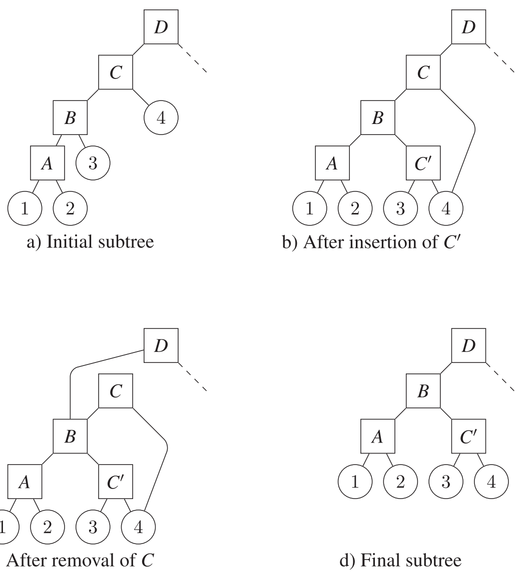


Figure 18. All of the reader visible states in a diag restructure.

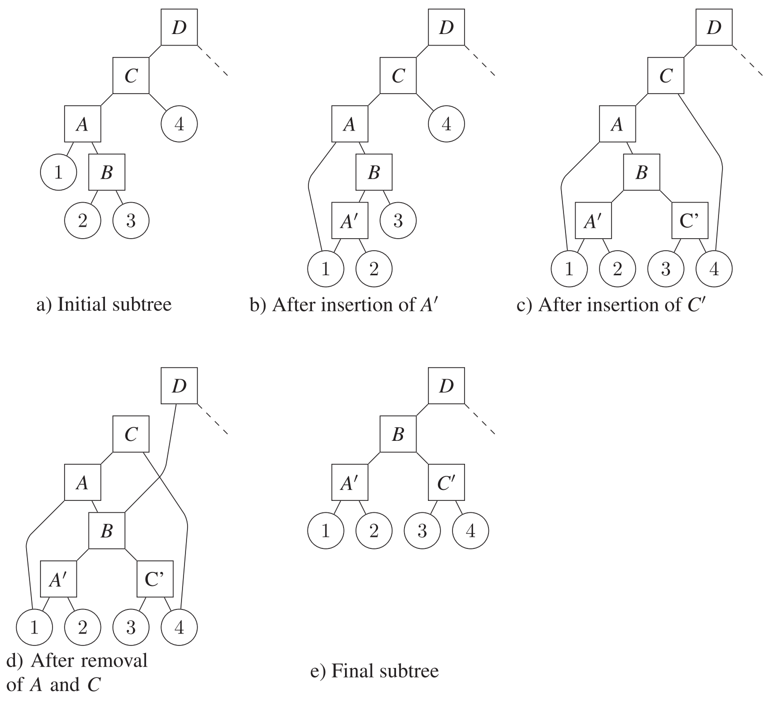


Figure 19. Reader visible states for a zig restructure.

6.6. *Summary of linearizability arguments*

Relativistic programming does not guarantee a total order on all events. The relaxed ordering is what allows the higher performance of relativistic algorithms. However, this does not mean that ordering is totally chaotic. Readers are allowed to proceed almost unsynchronized because the writers guarantee the necessary ordering properties through the use of *rp-publish* and *wait-for-readers*. For example, in the case of the interior delete algorithm, *wait-for-readers* was used to preserve the *structural integrity* property while moving node C higher in the tree. Not only are RP algorithms not chaotic, but we showed that the lookup, insert, and delete operations are linearizable with each other.

Most synchronization mechanisms enforce total ordering even where total ordering is not required. Imposing a total order limits performance and scalability. Relativistic programming allows the developer to decide how much ordering is required and only pay for the required ordering. In Section 5, we presented several traversal algorithms. The reader–writer lock approach is totally ordered with respect to updates and is linearizable using the acquisition of the lock as the linearization point. This approach greatly degrades update performance because updates must wait for traversals to finish. The relativistic approaches are not linearizable because a relativistic traversal is not totally ordered with the set of writes that are concurrent with the traversal. The relaxation in ordering means that updates do not have to wait for traversals. There were two options for relativistic traversals: one required $O(N \log(N))$ traversals, whereas the other allowed $O(N)$ traversals but at the expense of slower updates. These different approaches to traversals provide a good illustration of the types of ordering versus performance trade-offs that are available with relativistic programming.

7. CONCLUSIONS

This paper presented new algorithms for a concurrent red-black tree. The algorithms allow wait-free, linearly scalable concurrent lookups in the presence of inserts and deletes. They have deterministic response times and uncontended read performance that is at least 60% faster than other known approaches.

Our approach is unique in that readers do not need to explicitly synchronize with writers. Readers need to advertise their existence, but this can be carried out without the use of costly synchronization instructions. Writers can use the knowledge of the existence of readers to delay operations as required to preserve causality. This synchronization mechanism means that readers have similar performance characteristics of completely unsynchronized accesses—even in the presence of concurrent updates—but the data safety characteristics of more conventional synchronization mechanisms.

We showed that our algorithms for lookup, insert, and delete are linearizable meaning that our algorithms, although built on a relaxed consistency synchronization mechanism, provide the same consistency guarantees as algorithms built with heavier weight mechanisms. We showed both linearizable and non-linearizable algorithms for complete tree traversals and showed the performance trade-offs associated with the different algorithms. We also argued that a traversal algorithm based on fine grained locks would not be linearizable and that such an algorithm would have the same consistency guarantees as our non-linearizable algorithms.

Finally, our algorithms were inspired by, and implemented as a test case for, relativistic programming. Relativistic programming is a new concurrent programming methodology that does not, by default, require all participants to agree on the order of events. Instead, primitives are provided that can be used to enforce causality without incurring significant cost on the read-side. Work is ongoing to formalize and generalize the relativistic programming methodology so that it is available and accessible to the general concurrent programming community.

ACKNOWLEDGEMENTS

This material is based upon work supported in part by the National Science Foundation under Grant No. CNS-0719851.

REFERENCES

1. Landley R. Red-black trees (rbtree) in Linux. *kernel.org documentation*; January 2007. [Online]. Available from: http://www.kernel.org/doc/Documentation/rbtree.txt [Accessed on 26 September 2013].
2. Java platform standard ed. 6. [Online]. Available from: http://download.oracle.com/javase/6/docs/api/java/util/TreeMap.html [Accessed on 26 September 2013].
3. Fernandes SM, Cachopo J. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11. ACM: New York, NY, USA, 2011; 179–188. Available from: http://doi.acm.org/10.1145/1941553.1941579 [Accessed on 26 September 2013].
4. Dragojević A, Guerraoui R, Kapalka M. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09. ACM: New York, NY, USA, 2009; 155–165. Available from: http://doi.acm.org/10.1145/1542476.1542494 [Accessed on 26 September 2013].
5. Dragojevic A, Felber P, Gramoli V, Guerraoui R. Why STM can be more than a research toy. *Communications of the ACM* 2011; **54**(4):70–77.
6. Bronson NG, Casper J, Chafi H, Olukotun K. A practical concurrent binary search tree. In *Ppopp '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM: New York, NY, USA, 2010; 257–268.
7. Guibas LJ, Sedgewick R. A dichromatic framework for balanced trees. In *SFCS '78: Proceedings of the 19th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society: Washington, DC, USA, 1978; 8–21.
8. Hanke S. The performance of concurrent red-black tree algorithms. *Technical Report*, Albert-Ludwigs University at Freiburg, 1998.
9. Pugh W. Concurrent maintenance of skip lists. *Technical Report*, University of Maryland at College Park, College Park, MD, USA, 1990. Available from: http://drum.lib.umd.edu/bitstream/1903/542/2/CS-TR-2222.pdf [Accessed on 26 September 2013].
10. Pugh W. Skip lists: a probabilistic alternative to balanced trees. *Communications in ACM* June 1990; **33**:668–676. Available from: http://doi.acm.org/10.1145/78973.78977 [Accessed on 26 September 2013].
11. Binder W, Mosincat A, Spycher S, Constantinescu I, Faltings B. Multiversion concurrency control for the generalized search tree. *Concurrency and Computation: Practice and Experience* 2009; **21**(12):1547–1571.
12. Triplett J, McKenney PE, Walpole J. Scalable concurrent hash tables via relativistic programming. *SIGOPS Operating Systems Review* August 2010; **44**:102–109. Available from: http://doi.acm.org/10.1145/1842733.1842750 [Accessed on 26 September 2013].

13. Triplett J, McKenney PE, Walpole J. Resizable, scalable, concurrent hash tables. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11. USENIX Association: Berkeley, CA, USA, 2011; 11.
14. McKenney PE. Kernel korner: using RCU in the Linux 2.5 kernel. *Linux Journal* 2003; **2003**(114):11.
15. Herlihy MP, Wing JM. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Language Systems* 1990; **12**(3):463–492.
16. Vafeiadis V, Herlihy M, Hoare T, Shapiro M. Proving correctness of highly-concurrent linearisable objects. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06. ACM: New York, NY, USA, 2006; 129–136. Available from: http://doi.acm.org/10.1145/1122971.1122992 [Accessed on 26 September 2013].
17. Herlihy M. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems* November 1993; **15**:745–770. Available from: http://doi.acm.org/10.1145/161468.161469 [Accessed on 26 September 2013].
18. Herlihy M, Shavit N, Waarts O. Linearizable counting networks. *Distributed Computing* February 1996; **9**:193–203. Available from: http://portal.acm.org/citation.cfm?id=1063369.1063372 [Accessed on 26 September 2013].
19. Howard PW, Walpole J. A relativistic enhancement to software transactional memory. In *HotPar'11: Proceedings of the 3rd USENIX Workshop on Hot Topics in Parallelism*, HotPar'11. USENIX Association: Berkeley, CA, USA, 2011; 15.
20. Howard PW. Extending relativistic programming to multiple writers. *Ph.D. Thesis*, Portland State University University, 2012. Available from: http://pdxscholar.library.pdx.edu/open_access_etds/114/ [Accessed on 26 September 2013].
21. Attiya H, Guerraoui R, Hendler D, Kuznetsov P, Michael MM, Vechev M. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11. ACM: New York, NY, USA, 2011; 487–498. Available from: http://doi.acm.org/10.1145/1926385.1926442 [Accessed on 26 September 2013].
22. McKenney PE. Exploiting deferred destruction: an analysis of read-copy-update techniques in operating system kernels. *Ph.D. Thesis*, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available from: http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf [Accessed on 26 September 2013].
23. Desnoyers M, McKenney PE, Stern AS, Dagenais MR, Walpole J. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems* 2012; **23**(2):375–382. Available from: http://dl.acm.org/citation.cfm?id=2122267.2122521 [Accessed on 26 September 2013].
24. Desnoyers M. Low-impact operating system tracing. *Ph.D. Thesis*, École Polytechnique de Montréal, December 2009. [Online]. Available from: http://www.lttng.org/pub/thesis/desnoyers-dissertation-2009-12.pdf [Accessed on 26 September 2013].
25. Hart TE, McKenney PE, Brown AD, Walpole J. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing* December 2007; **67**:1270–1285. Available from: http://portal.acm.org/citation.cfm?id=1316099.1316427 [Accessed on 26 September 2013].
26. Bayer R. Symmetric binary b-trees: data structure and maintenance algorithms. *Acta Informatica* 1972; **1**:290–306. Available from: http://dx.doi.org/10.1007/BF00289509 [Accessed on 26 September 2013].
27. Plauger PJ. A better red-black tree. *C/C++ Users Joural* July 1999; **17**:10–19. Available from: http://dl.acm.org/citation.cfm?id=330304.330305.
28. Schneier B. Red-black trees. *Dr. Dobb's Journal* 1992; **17**(4):42–46.
29. Mellor-Crummey JM, Scott ML. 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*. ACM: New York, NY, USA, 1991; 106–113. Available from: http://doi.acm.org/10.1145/109625.109637 [Accessed on 26 September 2013].