2011

The Ordering Requirements of Relativistic and Reader-Writer Locking Approaches to Shared Data Access

Philip William Howard
Portland State University

Josh Triplett
Portland State University

Jonathan Walpole
Portland State University

Paul E. McKenney

The Ordering Requirements of Relativistic and Reader-Writer Locking Approaches to Shared Data Access

Philip W. Howard
Portland State University
phil.w.howard@gmail.com

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

Josh Triplett
Portland State University
josh@joshtriplett.org

Paul E. McKenney
Linux Technology Center
IBM Beaverton
paulmck@linux.vnet.ibm.com

ABSTRACT

The semantics of reader-writer locks allow read-side concurrency. Unfortunately, the locking primitives serialize access to the lock variable to an extent that little or no concurrency is realized in practice for small critical sections. Relativistic programming is a methodology that also allows read-side concurrency. Relativistic programming uses different ordering constraints than reader-writer locking. The different ordering constraints allow relativistic readers to proceed without synchronization so relativistic readers scale even for very short critical sections. In this paper we explore the differences between the ordering constraints for reader-writer locking and relativistic programs. We show how and why the different ordering constraints allow relativistic programs to have both better performance and better scalability than their reader-writer locking counterparts.

1. INTRODUCTION

Concurrent programs control access to shared data using a variety of synchronization techniques, the most common of which is mutual exclusion. Since mutual exclusion serializes accesses to shared data, it results in programs that have sequential bottlenecks and hence limited scalability.

Reader-writer locks were developed in an attempt to allow read-mostly workloads to scale by allowing read-side concurrency. However, many reader-writer lock implementations use atomic read-modify-write instructions to a common lock variable. Contention for the lock variable and the serialization caused by atomic read-modify-write instructions limit both performance and scalability. In fact, Lev et. al. [12] shows that the production Solaris reader-writer lock does not scale at all for certain read-only workloads. Triplett et. al. [18, 19] shows that the production Linux reader-writer lock does not scale at all for certain read-only workloads.

There have been many attempts to improve performance and scalability of reader-writer locks. Hsieh and Weihl [11] proposed a reader-writer lock that used per-processor locks for readers. This approach avoids contention between readers allowing good scalability for read-only work loads. However, a writer must acquire all the read-locks. As the number of readers increases, the cost to a writer for acquiring all the read-locks becomes exorbitant. Read performance suffers significantly in the presence of writers because readers are excluded for the duration of the write, and writes are slow because writers have to acquire a potentially large number of read-locks.

Lev et. al. [12] present a reader-writer lock based on a Scalable Non-Zero Indicator (SNZI). Their observation is that writers do not need to know the number of readers, only if there are zero or not zero. They construct a tree of SNZI objects where the root of each subtree knows whether the subtree contains zero or non-zero readers. A node only needs to communicate with its parent when that node moves in or out of its zero state. Readers can be distributed throughout the tree so that the number of other readers they contend with is minimal. Writers can check the root of the tree for the existence of readers. Their implementation scales for read-only workloads but suffers significantly with even 1% writers.

The lack of scalability in reader-writer locks presents a need for another synchronization mechanism for read-mostly work loads. RCU (Read-Copy-Update) fills that need within the Linux kernel. RCU is similar to reader-writer locks in that it distinguishes between readers and writers, it allows readers to proceed concurrently, and it was designed for read-mostly work loads. However, RCU provides better performance and scalability than reader-writer locks [9, 18, 19]. Because of its performance characteristics, RCU is gaining wide spread use in the Linux kernel. Figure 1 shows that the use of RCU within the Linux kernel is continuing to grow and has exceeded that of reader-writer locks. RCU is not limited to the Linux kernel. There is a user mode library [5, 4] that is portable and is gaining use in non-kernel applications.

Although RCU bears a superficial similarity to reader-writer locks, it is built on a very different premise and has different ordering properties. This paper explores those differences. Section 2 introduces relativistic programming—the

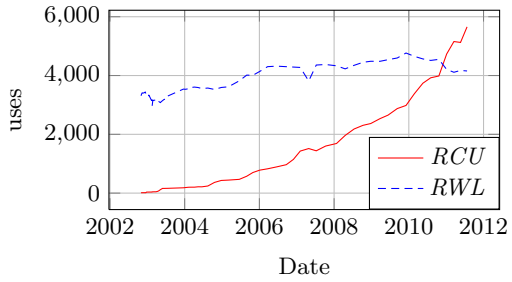


Figure 1: Count of the number of uses of RCU primitives and Reader Writer Locking primitives in the Linux Kernel.

methodology for using RCU. Sections 3–5 discuss the ordering properties of programs built using reader-writer locking and relativistic programming. Section 6 validates the performance claims made within the paper and Section 7 presents concluding remarks.

2. THE BASIS OF RELATIVISTIC PROGRAMMING

Relativistic programming differs from other synchronization mechanisms in the way it embraces concurrency. Mutual exclusion allows local computations to run concurrently, but prevents concurrent access to shared data. Reader-writer locks allow read-read concurrency, but prevent read-write concurrency. Fine grained locking allows concurrent access to disjoint sets of shared data, but prevents concurrent access to the same data. Optimistic techniques such as Transactional Memory allow accesses to run concurrently, but conflicting accesses involving at least one write require either waiting or a rollback. Relativistic programming allows read-read concurrency much as reader-writer locking, but also allows read-write concurrency. Unlike fine grained locking or optimistic techniques, relativistic programming allows read-write concurrency to the same data.

The one area of concurrency that relativistic programming does not yet address is write-write conflicts. Relativistic programming depends on other synchronization mechanisms between writers. Most commonly this involves using a single mutex, but can include fine grained locking [18] or even transactional memory [10].

Many optimistic techniques that allow readers and writers to access the same data (even though one or more will be delayed or rolled back) place a large burden on readers to check the consistency of the data. Readers must be able to detect and handle concurrent changes to the data they are accessing. In contrast, relativistic programming places the burden on writers to maintain the level of consistency that readers require. Doing so allows readers to proceed almost as if the reader was sequential code (Section 4.4 discusses the few restrictions on readers). The burden is shifted almost entirely to writers.

Relativistic writers have two barriers that they can use to control ordering. A light weight barrier is used to guarantee that previous memory writes are visible to readers prior to subsequent memory writes. This is used, for example, to guarantee that any writes used to initialize a node are visible to readers prior to the point at which the node becomes reachable. The heavier weight barrier allows writers to wait for the completion of pre-existing readers. One use of this primitive is for a writer to wait for any readers that might have a reference to a deleted node to finish before reclaiming memory for that node. We show in sections 3–5 that these two primitives are sufficient to handle a wide range of concurrent data structures.

Linearizability is a concurrency property that is often used as a correctness criteria for concurrent objects. Linearizability requires an equivalence between the linearizable computation and a legal sequential computation [8]. We claim that linearizability is too strict a criteria for correctness, and that there are correct solutions that are not linearizable. There is a significant performance cost to linearizability [7]. One of relativistic programming's great strengths is that although it can provide linearizable solutions if needed, it can also reap the performance and scalability benefits of non-linearizable behavior while still retaining ease of use.

We use the following as our correctness criteria for relativistic data structures. If the relativistic data structure is an implementation of an abstract data type (ADT), then relativistic programming provides the following properties:

1. Update operations on the ADT are serialized with respect to each other and leave the ADT in a valid state. If the data structure is partitioned, then updates to the same partition are serialized.
2. Read operations on the ADT see the effects of all previous non-concurrent updates
3. Read operations do not see any of the effects of later non-concurrent updates
4. For concurrent update and read operations, reads see either the state of the ADT prior to the update or after the update, but are not allowed to see any other state.

Items 1–3 are the same as for reader-writer locking. Item 1 states that the data structure invariants must be restored before the write completes. Items 2 and 3 state that non-concurrent operations form a total order that matches the serial order. Item 4 deals with read-write concurrency, and it states that a write can not affect an unrelated read. For example: if a set includes the values 1..5 then a write inserting the value 6 should not cause a concurrent read looking for 5 to return *not found*. Item 4 is intended to allow non-linearizable behavior in that not all concurrent readers have to observe the same states. This aspect of relativistic programming is covered in Section 5.

2.1 Complexities of Concurrent Programming

Modern hardware does not present a sequentially consistent interface to bare metal programmers. Multiple levels of cache, instruction pipelines, and out of order execution units allow modern CPUs to be fast, but these features mean concurrent programmers can not assume sequentially consistent behavior of their code [1, 14, 15, 2].

Optimizing compilers transform source code into efficient machine code whose execution on single-threaded hardware is equivalent to sequential execution of the source code. But the key here is that they assume a *sequential* execution. Both the compiler and the underlying hardware can rearrange code in a way that is safe for a sequential execution but may not be safe for a concurrent execution [3]. Listing 1 shows a function for initializing a shared structure. If another thread was spinning on `while (! data->init-complete)` prior to accessing the other fields in the structure, then line 4 of Listing 1 must execute after lines 2 and 3 otherwise readers may access uninitialized data. Since there are no data dependencies between the three assignment statements, both the compiler's optimizer and the CPU's out of order execution unit can reorder the execution of `init()`. Even if the compiler and execution units leave the order intact, the memory systems in some hardware architectures can re-order memory writes unless they are explicitly told not to.[1]

```
1 void init(struct some_struct *data) {
2   data->field1 = FIELD1_INIT;
3   data->field2 = FIELD2_INIT;
4   data->init_complete = TRUE;
5 }
```

Listing 1: Line 4 must execute after all the others. The compiler, out of order execution unit, or memory subsystem can each re-order this execution.

The impact of reordering can be localized in lock based code. Use of mutual exclusion differentiates between three categories of code: that accessing only thread-private data, that appearing in critical sections, and that comprising the locking primitives themselves. Local code can be treated as sequential because there is no shared data. The critical section can also be treated as sequential because mutual exclusion prevents conflicts involving writes. So only the locking primitives are truly concurrent.

A programmer writing new locking primitives must be concerned about all the complexities of concurrent programming. They must ensure that the primitives enforce the semantics of the lock on the target hardware. They must account for whatever reorderings the CPU and memory system allow. They must also ensure that the locking primitives contain whatever compiler barriers are necessary to prevent the compiler from moving code across a critical section boundary.

If locking primitives work correctly and are used correctly, a programmer using those primitives to develop a multithreaded system can view both their local code and critical section code as if it was sequential code.[2] They do not have to worry about the complexities discussed above. They still have many other complex issues to worry about, but not compiler, execution unit, nor memory system reorderings.

In contrast with reader-writer locking programs, relativistic programs allow reads to run concurrent with writes. They are truly concurrent in the sense that they contain data races—multiple threads accessing the same data where one of the threads performs a write. Because relativistic programs are truly concurrent, programmers need to be aware of and manage the reorderings discussed earlier. Relativistic programming identifies specific operations that need to be ordered and provides a small number of ordering primitives with simple rules for their placement. If these primitives work correctly and used correctly, they will only allow correct orderings of execution. We demonstrate the validity of these claims in the next several sections.

3. INTRODUCTION TO ORDERING CONTROL

In this section, we introduce some of the ordering relationships that occur in concurrent programming. We use those relationships to show how various primitives are used to control ordering so that only orderings that yield correct results are allowed. The primitives handle the complexities discussed in Section 2.1 so the programmer can view their code almost as if it were sequential.

In this section, we use a linked list delete operation to illustrate ordering relationships. The linked list in Figure 2 begins with three nodes labeled A, B, and C. A write thread removes node B leaving nodes A and C and then reclaims the memory used by B.

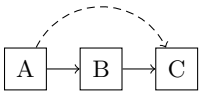


Figure 2: Linked list showing the removal of node B. The dashed line shows the link out of A after the removal.

We use the following names for specific events whose order is critical:

obtain ref Obtain a reference to the node following A. This could either be a reference to B or C depending on whether the reference is obtained before or after B is removed from the list.

drop ref The thread no longer has a reference to the node following A. In most cases, code does not explicitly drop references, so *drop ref* is implied just after the last use of the reference in question.

unlink The writer removes B by changing A so that it points to C instead of B.

reclaim The writer reclaims the memory used by B allowing this memory to be used for another purpose.

[1]Implementations of the Java and C++ memory models give programmers more explicit control over this kind of re-ordering.

[2]Boehm [3] points out that in the absence of a memory model, a compiler may transform code so that data races are created that did not exist in the source. When this happens, code which the programmer thought was data race free (and therefore viewable as sequential) will in fact be concurrent.

Figure 3 shows which orderings must be preserved. Three types of arrows are used to portray three types of happens-before relationships. Normal arrows indicate orderings that are implied by program order. Memory or compiler barriers may be required to preserve these orderings. Dashed arrows show happens-before relationships that are not explicitly controlled; they are simply a notation that one event occurred before another. We refer to these as occurred-before relationships. Bold arrows indicate orderings that must be preserved to preserve correctness. We refer to these as required-before relationships.

Notation: Within text, we use $\rightarrow$ to mean occurred-before so that when we say $A \rightarrow B$ we mean A occurred-before B. We use $\Rightarrow$ to mean required-before so that when we say $A \Rightarrow B$ we mean A is required-before B. If we want to specify an operation for a particular thread, we use subscripts so that $A_{T1} \rightarrow B_{T2}$ means operation A of thread T1 occurs-before operation B of thread T2.

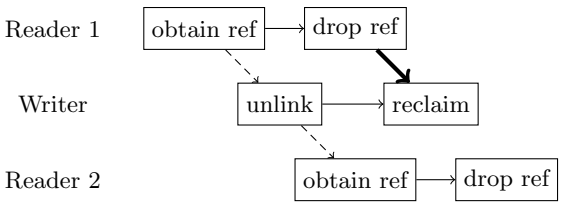


Figure 3: Orderings that have to be enforced by synchronization. Since Reader 1 obtained a reference to a node before it was unlinked, Reader 1 must drop that reference before the node is reclaimed. Reader 2 obtained the reference after the node was unlinked, so there is no dependency between Reader 2 dropping the reference and the writer reclaiming the memory.

The occurred-before relationships between threads can only be inferred when one thread observes a change, or the lack of a change, made the other thread. If $A \rightarrow B$ then either A made a change that was observed by B or B made a change that was not observed by A (because A occurred-before the change was made).

Required-before relationships are often the consequence of a previous occurred-before relationship. Figure 3 shows *obtain* $ref_{Reader\ 1} \rightarrow$ *unlink*. This occurs-before relationship imposes the relationship *drop* $ref_{Reader\ 1} \Rightarrow$ *reclaim*. Since *unlink* $\rightarrow$ *obtain* $ref_{Reader\ 2}$, there is no ordering constraint between Reader 2's *drop ref* and the *reclaim*.

3.1 RWL Ordering Control

Reader-writer locking uses locking primitives to control ordering. There are separate read and write primitives. The semantics of reader-writer locks enforce the following constraints:

1. if *read-lock*$_1 \rightarrow$ *write-lock* then *read-unlock*$_1 \Rightarrow$ *write-lock*
2. if *write-lock* $\rightarrow$ *read-lock* then *write-unlock* $\Rightarrow$ *read-lock*
3. if *write-lock*$_1 \rightarrow$ *write-lock*$_2$ then *write-unlock*$_1 \Rightarrow$ *write-lock*$_2$

The first constraint prevents a writer from starting if any readers hold the lock. The second constraint prevents a reader from starting if a write holds the lock. The third constraint prevents a second writer from starting if another writer holds the lock. Note that there is not a constraint that prevents a second reader from starting if another reader holds the lock. The absence of this constraint is what allows read-read concurrency.

In order for the lock primitives to guarantee mutual exclusion, the primitives must be capable of communicating with other threads. There is two-way communication: *read-lock* must know the results of *write-lock* and *write-lock* must know the results of *read-lock*. This communication is usually performed by atomic read-modify-write instructions on lock variables. The atomic read-modify-write instructions guarantee that there is a total order on the lock operations.

Figure 4a shows two readers and a writer deleting a node from a linked list. For both readers, *read-lock* $\rightarrow$ *write-lock*. This imposes the relationship *read-unlock* $\Rightarrow$ *write-lock* for both readers. However, since reader-writer locks allow read-read concurrency, the relationship *read-lock*$_1 \rightarrow$ *read-lock*$_2$ does not imply any other ordering between the other elements of the two read-sections.

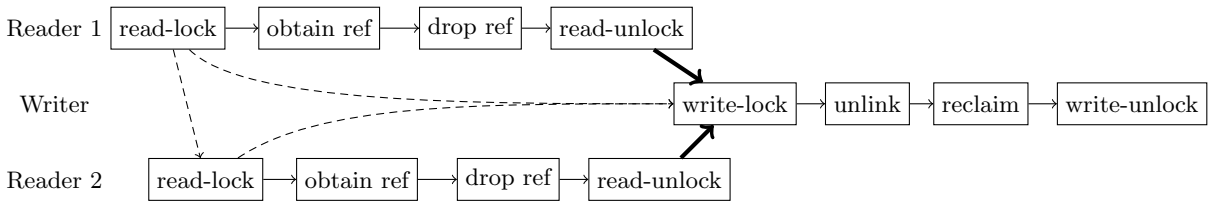
Figure 4b another reader and the same writer as Figure 4a. The relationship *write-lock* $\rightarrow$ *read-lock* implies *write-unlock* $\Rightarrow$ *read-lock*.

The correctness of the reader-writer lock implementation depends on program order dependencies being preserved. In particular, if the *drop ref* in Figure 4a is delayed after the *read-unlock*, then the required-before relationship between *drop ref* and *reclaim* (see Figure 3) may be violated. Also, if *obtain ref* in Figure 4b happens before the *read-lock*, then the *obtain ref* may also happen before *unlink* meaning that the entire reader should have happened before the writer, not after. The locking primitives must not only enforce the locking semantics, they must also contain whatever compiler and memory barriers are required to guarantee that the contents of the critical section do not leak outside. If the locking primitives preserve these properties, then only correct orderings will be allowed.

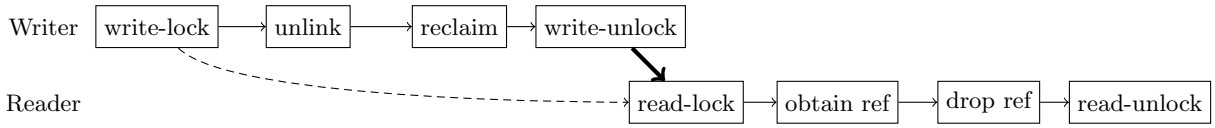
3.2 Relativistic Programming Ordering Control

Relativistic programming does not use mutual exclusion to control ordering. Relativistic readers can begin at any time and never wait for writers or other readers (not even for the duration of an atomic read-modify-write instruction). Instead of mutual exclusion, relativistic programming uses a write-side primitive called *wait-for-readers* to delay certain operations to preserve required-before relationships. *wait-for-readers* waits for all readers that existed at the beginning of the wait to finish. Stated formally:

$\forall$ readers r and any *wait-for-readers* w if (start of r) $\rightarrow$ (start w) then (end r) $\Rightarrow$ (end w)



a) Relationships between readers and a later writer



b) Relationships between a writer and a later reader

Figure 4: Reader-writer locks enforce the ordering through mutual exclusion. The *lock* and *unlock* primitives enforce this ordering.

In Figure 5 the *wait-for-readers* is depicted in two parts—the beginning of the wait and the end of the wait. Splitting the wait makes it easier to identify the occurred-before and required-before relationships in relativistic algorithms.

The *start-read* and *end-read* primitives require compiler and memory barriers to prevent *obtain ref* from occurring before *start-read* and to prevent *drop ref* from occurring after *end-read* (these are the same barrier requirements for reader-writer locking primitives). Similarly, the *wait-for-readers* primitive must have compiler and memory barriers to prevent *unlink* from occurring after *start wait* and to prevent *reclaim* from occurring before *end wait*. With these barriers in place, it can be seen from Figure 5 that any reader for which *obtain ref* $\rightarrow$ *unlink* must also have *start read* $\rightarrow$ *start wait*. This means that *end-read* $\Rightarrow$ *end wait* is sufficient to guarantee that *drop ref* $\Rightarrow$ *reclaim*.

Note that in Figure 5 there is no happens-before relationship between Reader 1's *obtain ref* and the Writer's *unlink*. Figure 5 could have been drawn with the *unlink* to the left of Reader 1's *obtain ref* without affecting any other ordering in the diagram. Using the *start-read* $\rightarrow$ *wait-for-readers* is more conservative than using *obtain ref* $\rightarrow$ *unlink* because *wait-for-readers* may wait longer than necessary, but it is guaranteed to wait at least long enough. That is, *wait-for-readers* might wait for readers that obtained their reference after the *unlink*, but *end-read* $\Rightarrow$ *end wait* is sufficient to guarantee that any reader for which *obtain ref* $\rightarrow$ *unlink* will *drop ref* prior to the *reclaim*. If *start wait* $\rightarrow$ *start-read* as shown in Reader 2 of Figure 5 then it must also be true that *unlink* $\rightarrow$ *obtain ref* so there is no required-before relationship between *drop ref* and *unlink*.

With reader-writer locking, there is two way communication between the locking primitives. Readers need to know about the state of the write lock and writers need to know about the state of the read lock. With relativistic programming, the communication is only one way: writers performing *wait-for-readers* need to know about the existence of readers, but readers do not need to know about the existence of writers (whether they are doing a *wait-for-readers* or not). The asymmetry of the communication means that it is possible to implement *start-read* and *end-read* with no more than a memory barrier. Atomic read-modify-write instructions are not required.

4. EXTENDING TO OTHER OPERATIONS

The previous section analyzed the ordering requirements for a Delete operation. In this section, we will look at three other operations. The first one is an Insert. The other two are called Move Down and Move Up. The names are taken from moving a node in a tree to a lower or higher location within a tree but the principles apply any time a node is moved within a directed acyclic graph. Down or Up is relative to the traversal order of readers.

There is no reason to repeat the analysis of reader-writer locking implementations because the analysis is the same regardless of the operation: correctness is preserved using mutual exclusion. So in this section, we only look at the relativistic programming implementations.

4.1 Insert

The *wait-for-readers* primitive is a heavy weight barrier. We illustrated its use in memory reclamation in Section 3.2. Relativistic programming has a lighter weight barrier called *rp-publish*. In this section we illustrate the use of *rp-publish* while inserting a node in a linked list.

Figure 6 shows a node D being added to a linked list. The update takes place in two steps. In the first step, not visible to readers, the new node D is created and initialized (including the pointer to E). In the second step, the node

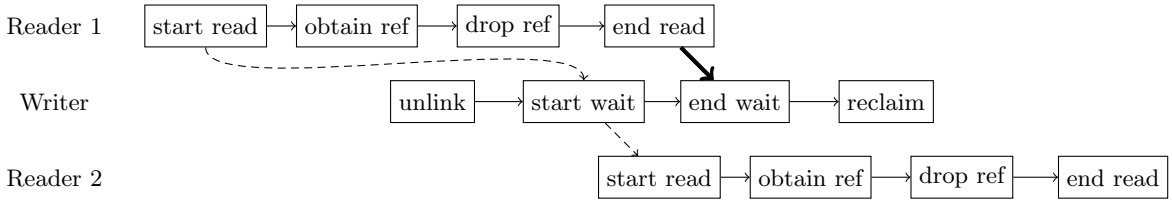


Figure 5: Relativistic programming enforces the ordering using the *wait-for-readers* primitive. Any reader that began before the wait started must finish before the wait completes.

is made visible to readers by changing the pointer out of C. These two steps must take place in this order to prevent a reader from seeing a partially initialized node.

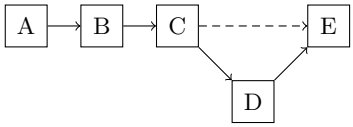


Figure 6: Linked list after adding node D. The dashed arrow shows the original pointer out of node C.

Figure 7 shows the timing relationships for adding a node to a linked list. The operations are as follows:

obtain ref	a reader obtains a reference out of C. This could be a reference to D or E.
deref	the reader accesses the contents of a the node whose reference was obtained by *obtain ref*. This includes dereferencing the pointer the next node.
init	the writer initializes node D.
link	the writer links D into the list

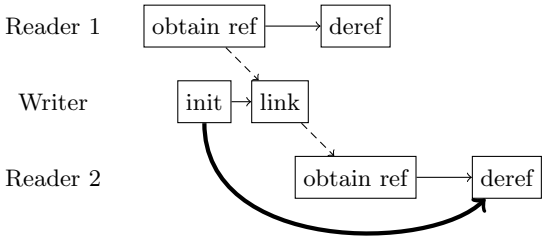


Figure 7: Happens-before relationships for inserting a node in a linked list. Reader 2 obtains a reference to the new node, so the initialization of that node must be visible to Reader 2 before Reader 2 accesses that node.

In Figure 7, Reader 1 has *obtain ref* $\rightarrow$ *link*, so Reader 1 will not see node D. There are no further happens-before relationships with Reader 1. Reader 2 has *link* $\rightarrow$ *obtain ref* so it obtains a reference to D. To prevent Reader 2 from seeing unitialized data, *init* $\Rightarrow$ $deref_{Reader\ 2}$.

The relationship between *init* $\Rightarrow$ $deref_{Reader\ 2}$ can be enforced by a memory barrier between *init* and *link*. In relativistic programming, pointers are updated using the *rp-publish* primitive. *rp-publish* includes the necessary compiler and memory barriers to guarantee that any prior initialization is visible to readers before the published pointer is visible. The *rp-publish* primitive is used any time a change is made to a reader-visible pointer.

4.2 Move Down

We have introduced the two primitives used by relativistic writers: *wait-for-readers* and *rp-publish*. Their use was illustrated using linked list delete and insert operations. These are the only two primitives required to enforce ordering. However, there are other aspects of program order that are important to relativistic algorithms. We illustrate this with a Move Down operation in the form of a red-black tree (RBTree) rotation.

One of the operations used to balance an (RBTree) following an insert or delete is a rotation [6, 16]. Figure 8 shows one type of rotation. In this rotation, node F is moved to a location lower in the tree, so this qualifies as a Move Down operation.

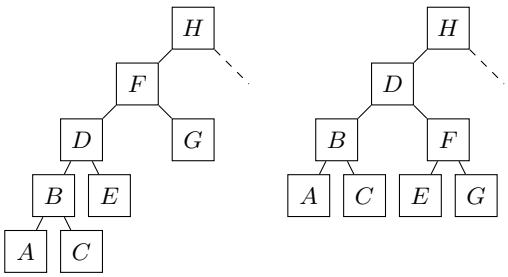
The following pointers need to be updated to accomplish the rotation:

1. The left child of H
2. The left child of F
3. The right child of D

Reader-writer locking uses mutual exclusion to prevent readers from seeing intermediate steps in the rotation so the order in which these pointers are updated does not matter. Readers either see the state before the rotation or after the rotation.

Relativistic programming allows read-write concurrency which means that readers can see intermediate states. Care must be taken in the order of the updates so that readers will always see a valid state. In this instance, a valid state means that all of the nodes in the tree must be reachable by any reader that is looking for them even if the tree changes during the lookup.

Most hardware architectures do not have an operation that can update three pointers atomically. If the pointers are



a) Initial subtree b) After rotation

Figure 8: Nodes B-D-F need to be rotated so that D is the root of the subtree.

updated one at a time, an invalid intermediate step will be visible as shown in Figure 9. Even a three-way atomic update would not be sufficient to prevent concurrent readers from observing an invalid state. A reader at F looking for B at the time of the update would fail to find B. Howard [9] showed how this rotation can be performed by making a copy of node F. Figure 10 shows the intermediate step following the insertion of the copy of F (labeled F' in the figure).

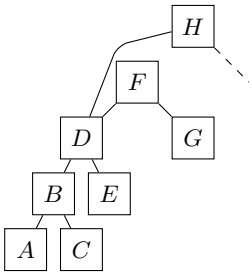
Figure 11 shows the happens-before relationships for three different orderings of this rotation. The event *deref(X)* means reading the contents of node X including following a pointer to the left or right child of X. Readers 1 and 2 have *deref(H)* $\rightarrow$ *unlink(F)* so these readers observe F. Any reader that observes F has *deref(F)* $\Rightarrow$ *reclaim(F)*. The required-before relationship can be satisfied by placing a *wait-for-readers* between *unlink(F)* and *reclaim(F)*. This case is identical to removing a node from a linked list as discussed in Section 3.2.

Readers 2 and 3 have *link(F')* $\rightarrow$ *deref(D)* so these readers observe F'. Observing F' means there is a required-before relationship between *init(F')* and *deref(F')*. This relationship is satisfied using *rp-publish* to perform the *link(F')* operation. This situation is the same as adding a node to a linked list as discussed in Section 4.1.

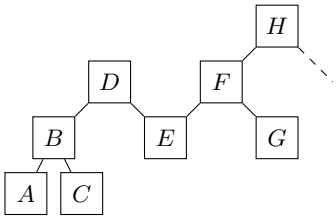
Reader 3 has *unlink(F)* $\rightarrow$ *deref(H)* meaning that the reader did not observe F. If this reader is looking for the key in F, it must observe F' (since it did not observe F). The relationship *link(F')* $\Rightarrow$ *deref(D)* captures this dependency. The *rp-publish* primitive guarantees that all writes in program order prior to the publish will be visible to readers before the published value. If *rp-publish* is used to *unlink(F)*, then any reader that observed the unlink, must also observe any previous writes, namely, *link(F')*. This is precisely the condition we need.

All of the required-before relationships can be satisfied by using *wait-for-readers* prior to *reclaim(F)* and by using *rp-publish* to update the pointers between nodes. These two primitives are sufficient to guarantee correctness for this rotation operation.

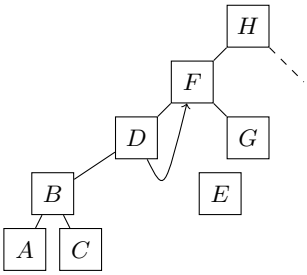
4.3 Move Up



a) Left child of H updated first. F and G unreachable



b) Left child of F updated first leaving A, B, C, and D unreachable



c) Right child of D updated first leaving E unreachable and a loop through D and F.

Figure 9: Possible intermediate steps in rotation of nodes B-D-F.

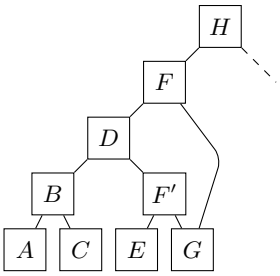


Figure 10: A relativistic programming rotate inserts a copy of F (labeled F') below D before removing the original F.

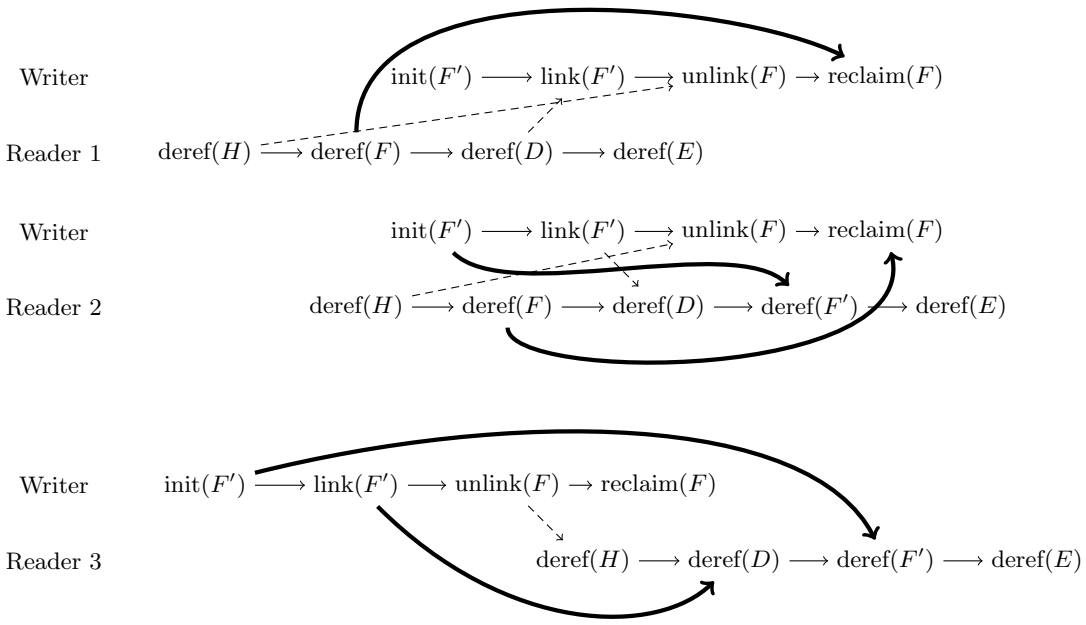


Figure 11: Happens-before relationships for various orderings of a writer performing a rotate and a reader. Note: the same writer was repeated for clarity in drawing each of the readers.

Our final example shows how *wait-for-readers* can be used to order the visibility of steps in an update process. This example is also taken from Howard's RBTree [9]. When internal nodes are deleted from a tree, they are swapped with a leaf node so the removal can happen at the bottom of the tree. The swapped node moves to a location higher in the tree, so this qualifies as a Move Up operation.

Figure 12 shows the removal of node B. A copy of C (labeled C') gets swapped up into the place of B. If C was removed from the bottom of the tree at this point, any readers at B which were looking for C would complete their lookup without finding C. Any readers above C' (including any new readers) can find the key for C at C'. A *wait-for-readers* between adding C' and removing C will allow any readers at or below B to complete their lookup before C is removed. Any new readers will see C' and not need to see C.

Figure 13 shows the happens-before relationships for a swap and two readers. Both readers are looking for the key in C. In Reader 1, *link(C')* → *deref(F)*, so the reader finds the key at C' and stops its search. Since this reader accessed C', there is a required-before relationship between *init(C')* and *deref(C')*. This required-before relationship is guaranteed by using *rp-publish* to perform the *link(C')*. This is the same as inserting a node in a linked list as discussed in Section 4.1.

unlink(C).

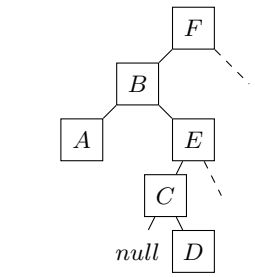
In Reader 2, *deref(F)* → *link(C')* so this reader does not observe C' and therefore must observe C. The relationship *deref(E)* ⇒ *unlink(C)* captures this dependency.. A memory barrier is not sufficient to enforce this relationship. If a reader has read past F prior to *link(C')*, then a memory barrier can not cause the reader to "back up" and observe C'. Similarly, a memory barrier attached to *unlink(C)* will not stall until the reader performs *deref(E)*. The solution proposed by Howard is to place a *wait-for-readers* between *link(C')* and *unlink(C)*. The *wait-for-readers* will allow all readers for which *deref(F)* → *link(C')* to complete prior to *unlink(C)* thus enforcing *deref(E)* ⇒ *unlink(C)*.

The final required-before relationship in Reader 2 which exists between *deref(C)* and *reclaim(C)* can be satisfied by a *wait-for-readers* between *unlink(C)* and *reclaim(C)*. This is the same a removing a node from a linked list as discussed in Section 3.2

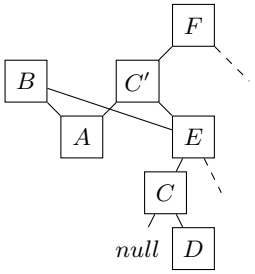
4.4 The General Case

The previous sections gave examples of the use of *rp-publish* and *wait-for-readers*. In this section we give general guidelines in their use. We have published a technical report [17] that gives a fuller treatment of these guidelines.

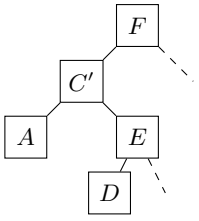
Nodes in relativistic data structures contain two types of data: immutable and mutable. Generally, the immutable data is the payload of the nodes and the mutable data is the pointers used to connect the nodes. Writers must guarantee that once a reader obtains a reference to a node, the immutable portion of the node will not change. If a writer needs to change an immutable portion of a node, it must make a copy of that node, change the node in its private memory, and then replace the old node with the new one by changing a mutable pointer. If multiple changes to the mu-



a) Initial subtree



b) After insertion of C'



c) Final subtree after removal of C

Figure 12: Deletion of node B through a swap with C

table portion of a node need to be made to appear atomic, the same copy-on-update proceedure can be used.

There are two rules for the placement of the *rp-publish* and *wait-for-readers* barriers. The first one is simple and deals with *rp-publish*. Any time mutable data is changed, it must be changed using *rp-publish*. Doing so guarantees that all changes to mutable data from a single writer will be seen in order by all readers.

The second rule deals with *wait-for-readers*. If readers have a traversal order (e.g. from the root of a tree to a leaf), then if two causally related changes to mutable data are made in the same direction as the traversal, they must be separated by a *wait-for-readers*. An example of this is the Move Up described in Section 4.3. The Move Up required two changes to mutable data: The copy of the swapped node was placed high in the tree then the swapped node was removed from the bottom of the tree. These changes are causally related and they occurred in traversal order so a *wait-for-readers* was required between them.

In the Move Down example described in Section 4.2 there were two changes. The first one involved placing a copy of F lower in the tree and then removing the original F from higher in the tree. Since these operations happen in the reverse of traversal order, *wait-for-readers* was not needed between them.

The use of *wait-for-readers* for safe memory reclamation also fits the traversal order rule. There are two changes in a reclamation. The first one is to remove the node from the data structure, the second is the reclamation of the node's memory. Removing the node from the data structure involves changing the pointer which points at the node. This pointer must be earlier in the traversal order than the contents of the node which get reclaimed, so these two changes happen in traversal order and thus need a *wait-for-readers* between them.

Just as there are two rules for writers, there are two for readers: *rp-read* must be used when reading mutable data and each mutable data item must only be read once. The *rp-read* accomplishes two things: it forces a read from memory (instead of allowing a cached copy to be used) and it enforces dependent read consistency. These actions are the same as for the C++0x *memory_order_consume* load.

If a mutable data item is read multiple times, the value of that item might change between each read. Listing 2 illustrates what might happen if a mutable item is read more than once. The code snippet shows a condition for terminating a linked list traversal. The pointer `node->next` may change between the condition in line 1 and the return statement in line 2. If this happens, the data returned will not come from the same node that was examined.

```
1 if (node->next->key == key) {
2     return node->next->value;
3 }
```

Listing 2: Invalid coding pattern because `node->next` may change between lines 1 and 2

5. CONSIDERING MULTIPLE WRITES

The previous sections examined the ordering constraints between a single reader and a single writer. There are other considerations when considering a reader and multiple writers. We examine these considerations in this section.

The timeline in Figure 14 shows a writer performing a delete and two readers. Using the terminology from Section 3 the diamond on the write line marks the *unlink*. The diamonds on the read lines mark the *deref*. Both readers begin before the writer and in both cases *deref* $\rightarrow$ *unlink*. Reader R1 finishes its read-section prior to the *unlink*. Our claim is that this case is indistinguishable from a reader-writer locking reader that executed just prior to the write. But reader R2 continues well after the writer completed its operation. This leaves open the possibility that reader R2 will be concurrent with another write operation and it may see the results of the subsequent write without observing the results of the first one.

Figure 15 shows a reader concurrent with two writers. Our

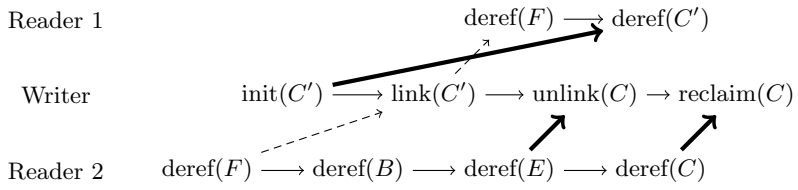


Figure 13: Happens-before relationships for various orderings of a writer performing a delete with swap and a reader looking for the key in C.

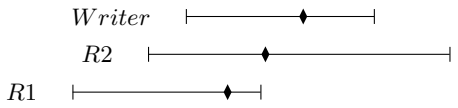


Figure 14: Timeline showing reader $R1$ which completes before the write takes effect and $R2$ which observes the state of the data structure prior to the write, but extends well after the write.

correctness criteria requires that a reader concurrent with a write can see the state immediately prior to the write or immediately after the write, but no other state. The case presented in Figure 15 requires a clarification of this rule. A reader concurrent with multiple writes can see the state prior to any of the writes, or the state generated by any combination of the concurrent writes being applied to that initial state. If I is the initial state, then the reader in Figure 15 can see one of the states $I, I + W1, I + W2, I + W1 + W2$.

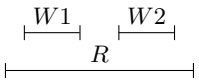


Figure 15: Operation R can see operations W1 and W2 in any order.

If the second write is dependent on the first, then observing $W2$ without observing $W1$ could potentially lead to an invalid observation. To prevent this from happening, a *wait-for-readers* could be inserted between the two writes (for example by calling *wait-for-readers* prior to releasing the lock used for mutual exclusion between writers). The presence of the *wait-for-readers* would guarantee that no reader was concurrent with both $W1$ and $W2$. Any reader concurrent with $W2$ would be guaranteed to see the state after $W1$.

The state observed by the reader is specific to that reader. If Figure 15 had shown two readers both concurrent with $W1$ and $W2$, then each reader could have observed different states. One could have observed $I + W1$ and the other $I + W2$. If this were so, the two readers would disagree on the order of the updates. The first reader would claim that $W1$ happened first (because it saw $W1$ but not $W2$). The second reader would claim that $W2$ happened first. The order of observations is local to (or relative to) a particular reader. There is no "global observer" that defines the correct order. Each reader is allowed to observe writes in their own reference frame. The potential reordering is limited in scope to the length of any concurrent reader. If there are two writes A and B such that there is no reader that is concurrent with both A and B, then all readers will agree on the order of A and B—that is to say, A and B will be totally ordered with respect to all reads.

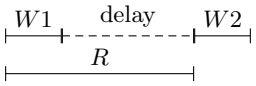
Reader-writer locking provides a total order of writes by delaying certain operations through mutual exclusion. If a relativistic solution requires a total order on writes, then *wait-for-readers* can be used to delay writes to preserve this total ordering.

The delays to totally ordered relativistic writes are no worse than the delays imposed by mutual exclusion in reader-writer locking. This is shown in Figure 16. For both relativistic programming and reader-writer locking, the delay can be up to the length of a reader. If the reader-writer implementation favors readers, then the delay can be any arbitrary length—the writer could starve. With relativistic programming writer starvation can not happen.

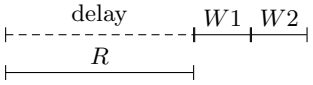
Read-sections and write-sections do not just happen spontaneously. They happen in response to external or internal events. When considering timing and ordering, it is useful to not only look at the timing and ordering of the read-sections and write-sections, but also the timing in relation to the events that triggered the read-sections and write-sections. Figure 17 shows the timing of four readers and a writer. The circles show the external events that trigger the reading and writing. The diamonds show the time when the writer updated a pointer and when the readers observed that pointer.

In the relativistic programming approach, reads are allowed to run concurrently with the write. Readers R1 and R2 observe the data structure prior to the update. Readers R3 and R4 observe the data structure after the update. This is evident by the order of the diamonds.

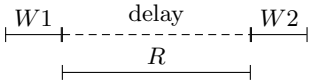
The reader-writer lock read preference case shows how the write is delayed until all readers have finished. The delay is shown by the dashed line. In this case, all four readers observe the data structure prior to the update. Note that this violates the order of the events that triggered the read and write operations. The reader-writer lock write preference or fair case shows a delay in both the writer and readers R3 and R4. In this case, the readers make the same observations as in the relativistic case, but the writer and two of the readers are delayed. This type of delay has the potential



a) Relativistic approach

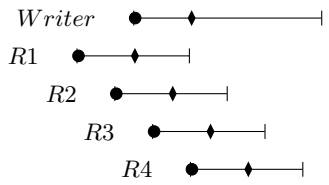


b) Reader-Writer approach where R goes first

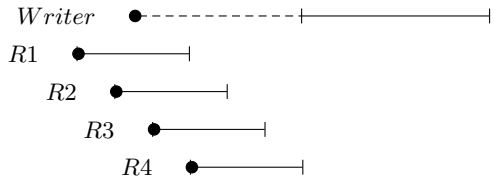


c) Reader-Writer approach where $W1$ goes first

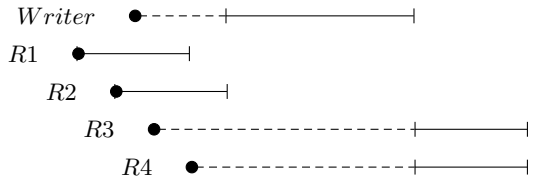
Figure 16: Timelines showing different delays. In a) $W2$ is delayed because $W1$ holds the write-lock until R completes. In b) $W1$ is delayed because R acquires the reader-writer lock first. In c) $W1$ acquires the lock first but $W2$ is still delayed for the full duration of R. The relativistic version suffers the minimum delay.



Relativistic Programming



Reader-writer lock with read preference



Reader-writer lock with write preference or fair

Figure 17: Timing of a writer and four readers using the relativistic approach and reader-writer locks with read preference and write preference or fair. The relativistic approach allows both the earliest possible writes and earliest possible reads.

to significantly reduce scalability. The relativistic programming approach did not have these delays, but the outcome of the computations was the same as for the reader-writer lock with writer preference case.

The reader-writer locking approach obtains consistency by delaying operations. However, this delay has the potential to cause operations to be ordered differently than the events that triggered these operations. Relativistic programming on the other hand allows the earliest possible completion of operations.

6. PERFORMANCE

We do not intend to show the overall performance benefits of relativistic programming in this paper. Papers on specific relativistic algorithms [9, 10, 18, 19] have made the case that relativistic algorithms are both high performance and highly scalable. In this section will will examine some of the specific performance claims made earlier in the paper.

It is our claim that reader-writer lock readers will not scale for sufficiently small critical sections because of the contention for the lock variable, and that relativistic programing readers should scale for any size critical section because there is no shared variable to contend for. To test this, we created a micro benchmark using linked lists. The data in the list was sorted and operations included lookups, inserts, and deletes. The length of the critical section can be varied by changing the length of the linked list. The tests were run on a four processor quad-core Intel Xeon system (16 hardware threads) running Linux 2.6.32. The relativistic programming implementation was supplied by Usermode RCU developed by Desnoyers et. al. [5, 4]. The following synchronization mechanisms were benchmarked:

nolock No synchronization was used. This is valid for read-only work loads but could lead to data corruption in the presence of updates. *nolock* is included as a potential upper bound on performance.

rp Relativistic programming using the Usermode RCU implementation. The *write-lock* is a pthread mutex.

torp Totally Ordered Relativistic programming using the Usermode RCU implementation. *torp* is the same as *rp* except that *wait-for-readers* is called before releasing the write-lock.

rwlr Reader-writer locks that favor readers. The implementation is based on Mellor-Crummey and Scott [13].

rwlw Reader-writer lock that favors writers. The implementation is based on from Mellor-Crummey and Scott [13].

Figure 18 shows read performance for a linked list of size one—the shortest meaningful critical section. Both *nolock* and *rp* scale linearly. There is overhead associated with the *rp* read-side primitives, so the performance of *rp* is less than that of *nolock*, however the overhead is both small and fixed—additional threads do not increase the overhead. Both of the reader-writer locks do not scale because additional threads mean more contention for the lock variable. The additional contention for the lock variable exceeded any benefit of parallelism so that the best performance was with a single thread.

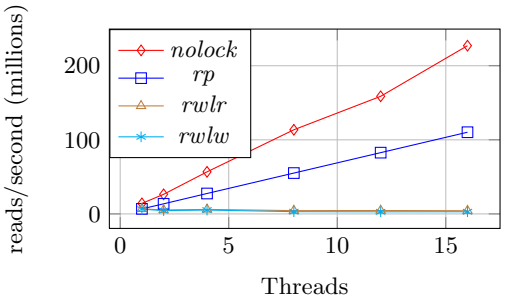


Figure 18: Read performance with a list size of 1. Reader-write locks do not scale because of the short critical sections.

Figure 19 shows read performance with a list of size 1,000. All synchronization mechanisms show good scalability because the critical section is large with respect to the cost of synchronization. However, even for lists of size 1,000, the performance of *rwlw* (and to a lesser extent *rwlr*) is noticeably less than linear for higher thread counts. The effects of lock contention can be seen even for critical sections that involve thousands of instructions.

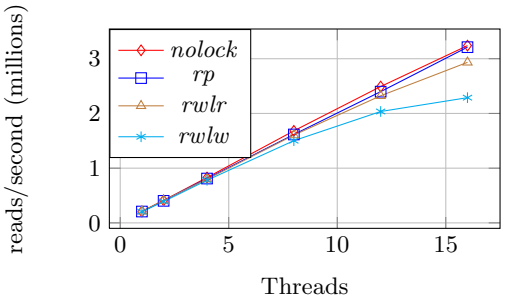


Figure 19: Read performance with a list size of 1000. All synchronization mechanisms scale because the size of the critical section is large with respect to the cost of synchronization.

Figure 20 shows the update performance of *rp* and *torp* across a range of list sizes. There were 15 concurrent readers. Since *torp* has a *wait-for-readers* between each update, and since the length of *wait-for-readers* should be proportional to the length of read-sections (that is *wait-for-readers* is O(N) where N is the length of the list), longer lists should have a proportionally larger impact on *torp*. However, both read-sections and write-sections are O(N), so both *rp* and *torp* are affected by the list size in the same way.

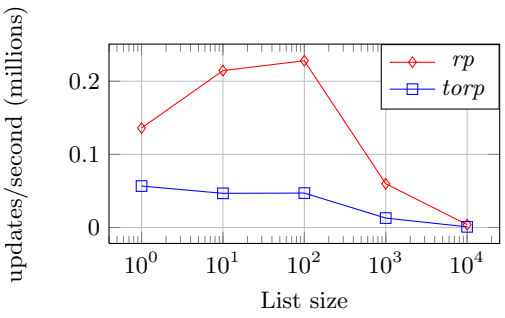


Figure 20: Update performance with 15 concurrent readers. Larger list sizes do not adversely affect *torp* despite the longer *wait-for-readers* because both the wait and the write performance are O(*list size*).

In order to see the effect of *wait-for-readers* on *torp*, the list update routines were changed so that updates were always made to the first element of the list. The list was no longer sorted, so lookups had to scan the entire list looking for the desired element. With this configuration, writes were O(1) with small constant and reads (and thus *wait-for-readers*) were O(N). Update performance for this configuration is shown in Figure 21. For larger lists, *rp* clearly outperforms *torp*. The performance of *rp* is better for larger lists because there is less contention for the first element of the list (the one that gets updated) because readers spend a higher percentage of their time on later list elements with longer lists. The performance of *torp* does not improve for longer lists because the *wait-for-readers* takes longer.

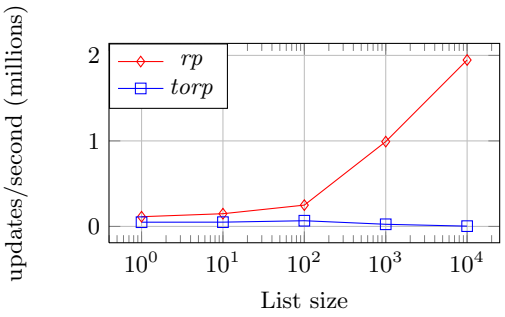


Figure 21: Update performance with 15 concurrent readers using a O(1) update and O(N) read. The difference between *rp* and *torp* gets progressively bigger with larger lists because of the duration of *wait-for-readers*.

In Section 5 we claimed that the delay in releasing the lock for *torp* was no worse than the delay for acquiring the lock in *rwl*. Figure 22 demonstrates this. The update performance of *torp* tracks fairly closely with *rwlw*. However, as Figure 23 shows, *torp* has significantly better read performance than *rwlw*. The read performance of *torp* is better than *rwlr* for small list sizes, but about the same for larger list sizes.

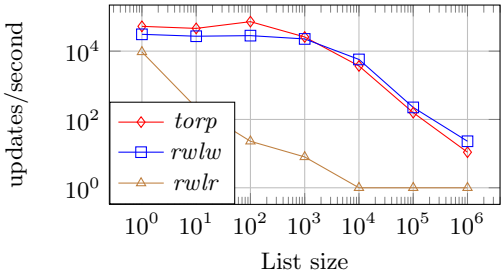


Figure 22: Update performance with 15 concurrent readers. The performance of *torp* and *rwlw* because *torp* has to wait for all current readers before releasing the write lock and *rwlw* has to wait for all current readers before acquiring the write lock.

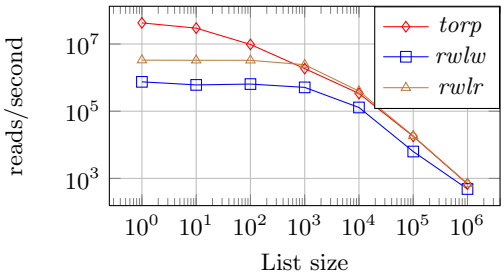


Figure 23: Read performance with 15 reader threads and one writer.

7. CONCLUSION

We have argued that the correctness of a concurrent implementation can be maintained by limiting what order operations occur. We showed the ordering constraints required by reader-writer locking and those required by relativistic programming. The two relativistic programming primitives, *rp-publish* and *wait-for-readers* can be used by relativistic writers to limit what orderings are visible to concurrent readers. The rules for using these primitives are simple and easy to understand. When these primitives are used correctly, only orderings which yield correct reads are allowed. The burden for correctness is placed on writers so that readers can proceed without the need for heavy weight synchronization. The different ordering requirements of relativistic programs allows them to have higher performance and better scalability than their reader-writer locking counterparts.

Acknowledgements

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

8. REFERENCES

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29:66–76, 1996.

[2] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli. The semantics of power and arm multiprocessor machine code. In *Proceedings of the 4th workshop on Declarative aspects of multicore programming*, DAMP '09, pages 13–24, New York, NY, USA, 2008. ACM.

[3] H.-J. Boehm. Threads cannot be implemented as a library. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 261–268, New York, NY, USA, 2005. ACM.

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

[5] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints), 2011.

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

[7] M. Herlihy, N. Shavit, and O. Waarts. Linearizable counting networks. *Distrib. Comput.*, 9:193–203, February 1996.

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

[9] P. W. Howard and J. Walpole. Relativistic red-black trees. Technical Report 1006, Portland State University, 2010. `http://www.cs.pdx.edu/pdfs/tr1006.pdf`.

[10] P. W. Howard and J. Walpole. A relativistic enhancement to software transactional memory. In *HotPar'11: Proceedings of the 3rd USENIX Workshop on Hot Topics in Parallelism*, HotPar'11, Berkeley, CA, USA, 2011. USENIX Association.

[11] W. C. Hsieh and W. E. Weihl. Scalable reader-writer locks for parallel systems. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 216–230, Beverly Hills, CA, USA, March 1992.

[12] Y. Lev, V. Luchangco, and M. Olszewski. Scalable reader-writer locks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 101–110, New York, NY, USA, 2009. ACM.

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

[14] D. Mosberger. Memory consistency models. *SIGOPS Oper. Syst. Rev.*, 27:18–26, January 1993.

[15] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 478–503, Berlin, Heidelberg, 2010. Springer-Verlag.

[16] B. Schneier. Red-black trees. *Dr. Dobb's J.*, 17(4):42–46, 1992.

[17] J. Triplett, P. W. Howard, P. E. McKenney, and J. Walpole. Generalized construction of scalable concurrent data structures via relativistic

programming. Technical Report 14, Portland State University, Mar. 2011. http://www.cs.pdx.edu/pdfs/tr1104.pdf.

[18] J. Triplett, P. E. McKenney, and J. Walpole. Scalable concurrent hash tables via relativistic programming. *SIGOPS Oper. Syst. Rev.*, 44:102–109, August 2010.

[19] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, scalable, concurrent hash tables. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, Berkeley, CA, USA, 2011. USENIX Association.