2-2011

A Comparison 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

Citation Details

Howard, Philip W., Josh Triplett, and Jonathan Walpole (2011). "A Comparison of Relativistic and Reader-Writer Locking Approaches to Shared Data Access," Portland State University Computer Science Department Technical Report "11-02"

A Comparison of Relativistic and Reader-Writer Locking Approaches to Shared Data Access

Philip W. Howard, Josh Triplett, and Jonathan Walpole

Portland State University

Abstract. This paper explores the relationship between reader-writer locking and relativistic programming approaches to managing accesses to shared data. It demonstrates that by placing certain restrictions on writers, relativistic programming allows more concurrency than reader-writer locking while still providing the same isolation guarantees. Relativistic programming also allows for a straightforward model for reasoning about the correctness of programs that allow concurrent read-write accesses.

1 Introduction

When multiple threads access shared data, there is a need for some form of synchronization in order to prevent corrupted data. A common form of synchronization is mutual exclusion where only one thread at a time is allowed to access the data. The problem with this method is that it restricts concurrency and thus scalability. Reader-writer locks separate critical sections into read-sections and write-sections. Read-sections are allowed to proceed concurrently because they can not corrupt the data (since they don't change it).

Since reader-writer locks allow read-side concurrency, they should allow read-only or read-mostly workloads to scale. However, most implementations of reader-writer locks have a global lock variable that acts as a contention bottleneck. Even implementations that are specifically designed to scale [7] require accessing a global lock variable with an atomic read-modify-write instruction. These atomic instructions are a point of serialization. For small read-sections, these atomic instructions so dominate the computation that little or no scalability is observed in practice.[4, 8, 9][1]

Relativistic programming is a synchronization mechanism which appears, superficially, to be similar to reader-writer locking. Relativistic programming makes a distinction between read-sections and write-sections. Read-sections are allowed to proceed concurrently. The difference is that with relativistic programming, readers don't explicitly synchronize with writers. As a result, most relativistic programming implementations don't require any atomic instructions

[1] Lev et. al. present a reader-writer lock which scales for read-only workloads but suffers significantly with even 1% writers. Their works shows that the Solaris reader-writer lock does not scale for read-only workloads. Triplett shows that the Linux reader-writer lock does not scale for read-only workloads.

in the read-side primitives. The other differences between relativistic programming and reader-writer locks are that relativistic programming allows read-write concurrency, and relativistic programming has weaker consistency guarantees than reader-writer locks. The weaker consistency guarantees allow relativistic programs to have both higher performance and better scalability than their reader-writer lock counterparts.[3, 8, 9]

In this paper, we show how, under certain conditions, relativistic programming can preserve the important properties of reader-writer locks even with read-write concurrency. We then examine the consistency guarantees relativistic programming supplies in the general case—that is, when these conditions are not met. Finally, we show that these consistency guarantees are sufficient to form a framework in which it is easy to reason about the correctness of relativistic programs.

2 Methods and Their Primitives

Both the synchronization mechanisms we are considering define critical sections of code that are bounded by *begin* and *end* primitives. There are two sets of primitives: one for read-sections and one for write-sections. Relativistic programming has two additional primitives which are used to control the ordering of events.

2.1 Reader-Writer Lock Primitives

Reader-writer locks bound write-sections with *write-lock* and *write-unlock* primitives. Read-sections are bounded by *read-lock* and *read-unlock* primitives. The primitives guarantee that write-sections are not concurrent with other write-sections, and write-sections are not concurrent with read-sections. The simplest implementation of the read-side primitives enforces the rule, “allow a reader to start if there are no active writers.” Unfortunately, this implementation will allow continually overlapping read-sections to lead to write starvation. There are other implementations that favor writers, but they can lead to read starvation. There are still others that are fair. Practitioners can choose which implementation to use based on expected work-load. The write-preference and fair implementations tend to have higher overhead. This can lead to lower performance and more serialization due to the need for additional atomic instructions to implement the more complicated semantics.

Atomic read-modify-write instructions are used to implement reader-writer locks because each of the four primitives needs to communicate with other threads. In order to preserve atomicity, these instructions form a serialization point. Not only do the atomic instructions execute serially, but they are also many times slower than non-atomic instructions. As a result the serial portion of the computation may be longer than the parallel portion of the computation (the work inside a read-section). According to Amdahl’s law, this will have a direct negative impact on scalability.

The lack of read-write concurrency means that readers will never see partially completed updates. This makes it easy to reason about the correctness of programs developed with reader-writer locks. Provided each writer leaves the data structure in a valid state prior to leaving its write-section, then readers will only encounter the data structure in a valid state. Also, writers will be guaranteed that the state of the data structure is valid when they begin their write-section. As a result, programmers only need to reason about whole operations and do not need to be concerned with concurrency.

2.2 Relativistic Programming Primitives

Relativistic write-sections are delimited by *write-lock* and *write-unlock*. Most relativistic programming implementations use a mutex to implement these primitives, but note that the mutual exclusion is only between writers—readers are not excluded. Relativistic readers are delimited by *begin-read* and *end-read* primitives. There are times when the presence of a read-section can delay a write-section, but write-sections never delay read-sections. This lack of symmetry allows the read-side primitives to be implemented using only local operations—atomic instructions are not required. It is this lack of atomic instructions which means that relativistic readers can proceed without any serialization.

There are times when a relativistic writer needs to delay an operation until all current readers have completed their read-sections. It is only readers that exist at the time the delay begins that will affect the duration of the delay. Any read-section that starts later will not be considered. As a result, continually overlapping read-sections will not create an unbounded delay in the writer.

There are two primitives that writers can use when they need to delay an operation. The *wait-for-all-readers* primitive forms a synchronous wait. The primitive will not return until all readers that existed at the time the primitive was called have finished their read-sections. The *defer-until-all-readers* primitive allows a writer to schedule an a operation to complete asynchronously some time after all current read-sections have completed. We will give examples of why these are necessary and how they are used later.

The implementation of the primitives mentioned in this section is beyond the scope of this paper. There are implementations freely available both in kernel and in user mode that are described elsewhere.[1, 5, 6] We have chosen generic names for the primitives presented in this paper rather than tying them to any particular implementation. The correspondence between our names and those of particular implementations should be obvious.

3 Synchronizing Linked List Operations

In this section, we show how both reader-writer locking and relativistic programming are used to perform the following linked list operations: insert, remove, and move. In each case, we look at what consistency guarantees can be made by each method. From those consistency guarantees, we show how to reason about the correctness of the implementation.

3.1 Insert

Figure 1 shows a linked list with nodes A, B, C, and E. Node D is inserted between nodes C and E. The gray arrow shows the pointer out of C prior to the insert. The insertion happens in two steps: node D must be initialized (including the pointer to E), and it must be linked into the list by changing the pointer out of C. With reader-writer locks, these two steps can happen in any order because readers can not see the intermediate states. With relativistic programming, it is important that the node be initialized and then linked into the list. Otherwise, a concurrent reader may access node D before it is initialized.

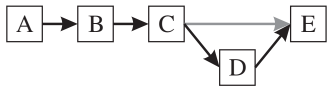

Fig. 1. Linked list after adding node D. The gray arrow shows the original pointer out of node C.

With both reader-writer locks and relativistic programming, there are only two visible states—before and after the insert. With reader-writer locks, reads are serialized with respect to writes. This means that any particular reader came either before or after the insert. Those that came before will see the list without D, those that came after will see the list with D. Relativistic programming allows reads to proceed concurrently with writes. But since on all modern processors an aligned pointer assignment is an atomic operation, when a reader follows the pointer out of C, it will either encounter D or E based on whether the writer has updated the pointer or not. So in both cases, the correctness of the writer can be determined by considering the state of the list before and after the update. No intermediate states need to be considered.

Even though reasoning about the correctness of the two approaches is the same, there is a subtle timing difference that must be considered. The timeline in Figure 2 shows the writer performing the insert as concurrent with two readers. The diamond on the write line marks when the pointer out of C is updated. The diamonds on the read lines mark when the pointer out of C is followed. Both readers begin before the writer and both follow the pointer out of C before it is updated. Reader R1 finishes its read-section prior to the update of the pointer. 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. If this presents a problem (see section 4.1 for a discussion of when this does and does not present a problem), the writer can call *wait-for-all-readers* prior to releasing the *write-lock*. This will extend the write-section until reader R1 completes meaning that another write can not begin until R1 completes. But notice that the delay caused by *wait-for-all-readers* is no worse than the delay in the reader-writer lock version. With reader-writer locks, there is an implicit *wait-for-all-readers* in the *write-lock*. The difference between the two approaches is that reader-writer locks insert the delay before the write-section, the relativistic programming approach inserts the delay at the end of the write-section.

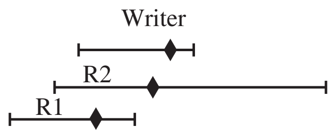

Fig. 2. Timeline showing reader $R2$ which observes the state of the data structure prior to the write, but extends well after the write.

Figure 3 shows the timing of four readers and a writer performing the insert described above. The circles show the external events that trigger the reading and writing. The diamonds show the time when the writer updated the pointer out of C and when the readers observed the pointer out of C. In the relativistic programming approach, reads are allowed to run concurrently with the write. Readers R1 and R2 observe the list prior to the insert. Readers R3 and R4 observe the list after the insert. 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 list prior to the insert. 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 write and two of the readers are delayed. This type of delay has the potential to significantly reduce scalability. The relativistic programming approach did not have these delays, but the outcome of the computations was the same.

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.

3.2 Remove

Removing a node from a linked list has many similarities to inserting. Removing the node can be done by a single pointer change. This means that the ordering and relative timing is the same as for insert.

There is one significant difference between an insert and a remove. Removing a node has two distinct steps: unlinking the node and reclaiming memory used

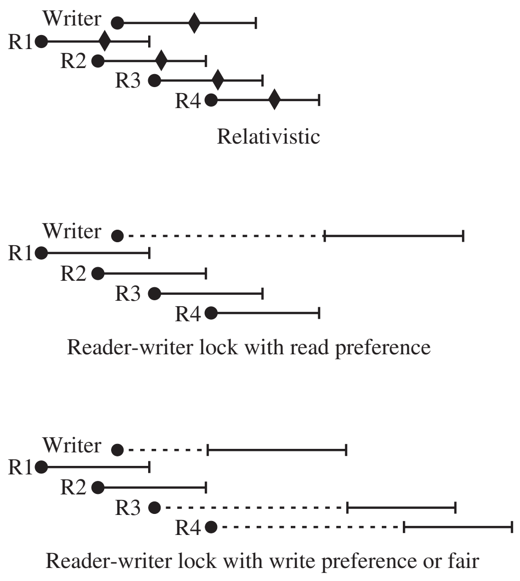
Writer
R1
R2
R3
R4
Relativistic
Writer
R1
R2
R3
R4
Reader-writer lock with read preference
Writer
R1
R2
R3
R4
Reader-writer lock with write preference or fair

Fig. 3. 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.

by the removed node. With relativistic programming, there is the possibility that a reader holds a reference to the removed node even after the writer unlinked the node. If the writer reclaimed the node while a reader still had a reference, the reader could access the memory after it gets reused for some other purpose. For this reason, the reclamation must be deferred until no readers hold a reference to the node.

Relativistic readers are required to explicitly delimit their read-sections. The definition of a read-section is any region of code where the reader holds a reference to the data structure. This means that when a reader exits their read-section they do not hold any references to the data structure. Therefore, if a writer waits until all current read-sections terminate then it is safe to reclaim the memory for the removed node. This is precisely what *wait-for-all-readers* and *defer-until-all-readers* are for. The writer can unlink the node then call *wait-for-all-readers* and then safely reclaim the node. Alternatively, the writer can unlink the node and call *defer-until-all-readers* to schedule the reclamation asynchronously. The advantage of the second method is that the write-section can complete sooner because the delay happens outside the write-section.

3.3 Move

Moving a node from one location to another within a linked list is more complicated than either an insert or a remove because it involves updates to multiple pointers. Figure 4 shows a linked list before and after moving node D. The grayed arrows in the after view show the pointers which must be changed to effect the move. With reader-writer locking there is no read-write concurrency so no reader will observe an intermediate state of the write. Only the state before and the state after the write is visible. This is not the case with relativistic programming.

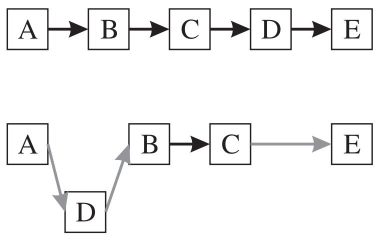

Fig. 4. Linked list before and after moving node D. The grayed arrows show which pointers must be changed to effect the move.

There are three pointers involved in moving a node. This means that there are $2^3 = 8$ states that a reader could observe. These states include all-of-the-move and none-of-the-move which are clearly safe, but there are six intermediate states that may or may not be safe. Relativistic programming can reduce the number of intermediate states down to one. Consider the move as shown in Figure 5. Rather than moving node D, a copy of that node is made (labeled D'). This allows the outbound pointer from D' to be changed to point to B before D' is visible to any readers. This eliminates four of the possible intermediate states. The two pointers labeled a and b in the figure allow two intermediate states: a but not b or b but not a. However, if the writer does *wait-for-all-readers* between the two writes, then only one intermediate state is visible—the state after the first write.

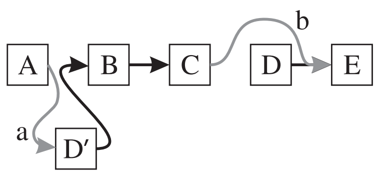

Fig. 5. Relativistic move of node D. The order in which pointers a and b are written determine which single intermediate state is visible.

If pointer b is updated before pointer a, then a reader might scan the entire list and never see D. If a is changed first, then readers will always encounter node D, but may encounter it twice (once as D' and once as D). Triplett [8] exploits this in his relativistic hash table move operation. Hash table lookups compare a search key to each key in the bucket list until they find the item being searched for. If the lookup was searching for the key in node D, then it would not matter if the search found node D or D' because both nodes contain the same data. If the lookup was searching for any other key, both nodes D' and D would be skipped because the key would not match the search key.

By inserting a *wait-for-all-readers* between each write, relativistic writers can reduce the number of visible states from 2^n down to n where n is the number of reader-visible writes to memory. In addition, the states become visible in the program order of the writer. In other words, readers are able to see any prefix of writes performed by the writer. This allows the programmer to reason about far fewer states. There is a single question, "Is the data structure in a consistent state following this particular memory write?" that the programmer needs to reason about.

4 Generalized Relativistic Programming

The previous section introduced relativistic programming through examples using linked list operations and comparing the details of the relativistic programming implementation with that of a reader-writer lock implementation. In this section we extend these concepts to the general case.

In Section 3.1 we observed that for a linked list insert operation both reader-writer locks and relativistic programming provide the same ordering and correctness guarantees. This is because with a linked list insert operation both the the insert and the observation of the insert are atomic operations. Any time a write can be made by a single atomic operation and the effects of that write can be observed by a single atomic operation, the following two properties will apply to the relativistic implementation provided a *wait-for-all-readers* occurs between each write:

1. Reads are totally ordered with respect to writes
2. Reads will not observe partially complete writes

Relaxing either the *wait-for-all-readers* requirement or the atomicity requirements means that these properties will not be guaranteed.

4.1 Total Order Of Reads With Respect To Writes

If two writes happen without a *wait-for-all-readers* between them, it is possible for a single read to be concurrent with both writes. This is illustrated in Figure 6. Operation R is a read. Operations $W1$ and $W2$ are writes. Both of the writes are concurrent with the read. When operations are concurrent, there is no implied happens-before relationship between the two operations.[2] This means that R

can be thought of as happening before or after $W1$ as well as before or after $W2$. This further means that the order $W2 \Rightarrow R \Rightarrow W1$ is possible (where $\Rightarrow$ reads "happens-before") even though this would appear to violate the implied happens-before relationship of the non-concurrent $W1$ and $W2$.

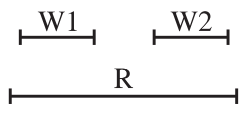

Fig. 6. Operation R can see operations W1 and W2 in any order.

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 reads with respect to writes. This is done by delaying certain operations through mutual exclusion (see section 3.1). Relativistic programming has the *wait-for-all-readers* primitive which can be used to delay writes to preserve this total ordering. Performance and scalability will be better without these delays.

It is our contention that many writes do not need to be ordered. Consider a phone company. One customer cancels their service. Another customer signs up for service. Provided the new customer is not given the old customer's phone number, these two operations are independent. There is no causal relationship between them. A phone book that was printed containing either, neither, or both updates would be equally valid. When a previously used phone number gets reassigned to a new customer, there is a causal relationship between the two writes: the phone number was available because the previous customer relinquished it. If relinquished phone numbers are placed into the pool of available numbers using *defer-until-all-readers* then there is no need for *wait-for-all-readers* between writes. The causal order of updates will be preserved without it.

4.2 Partially Completed Updates

Reader-writer locking protects readers from seeing partially complete writes. If relativistic writes can be made visible via a single atomic operation and if readers observe the change via a single atomic operation, then relativistic readers will not see partially complete writes. The need for atomicity on the write side is obvious. To illustrate the need for atomicity on the read side, consider the move operation depicted in Figure 4. The move required three pointers to be updated.

Even if these three pointers could be updated as an atomic unit, reads do not observe the change via a single atomic operation. Instead, readers observe the change by reading the pointer out of A and the pointer out of C. Consider a reader that progressed to node B prior to the move, then the move happens atomically, then the reader traverses the rest of the list. The reader observed the removal of node D from between nodes C and E but did not observe node D in its new location between nodes A and B. In other words, the reader observed a partially completed write.

One of the difficulties in reasoning about the correctness of a concurrent program is the number of states generated by every possible interleaving of operations. The linked list move operation described in Section 3.3 required three pointers to be changed. This led to a total of $2^3 = 8$ states. Reader-writer locking reduces the number of states that need to be reasoned about to two: the state before and after the move. Relativistic programming reduces the number of states to three: before and after the move and one intermediate state. In general, relativistic programs can reduce the number of states from 2^n down to n by placing a *wait-for-all-readers* between each reader-visible memory write.

If a write operation requires a *wait-for-all-readers* following each write to memory, write operations could be quite slow. However, there is an optimization that often applies that can eliminate most of these. The optimization applies whenever there is a traversal order that readers follow. For example, linked lists read from the head to the tail of the list; tree lookups progress from the root to a leaf; hash table lookups access the bucket list and then the list associated with a particular bucket.

In the linked list move example given in Figure 5, there were two visible writes. One of the writes removed D from the list, the other inserted D'. The read traversal order follows the arrows in the figure. If the order of the writes is the opposite of the traversal order, namely removing D and then inserting D', then any reader that observes the second write will also observe the first write. This can be seen to be true by considering the traversal order. Any reader that observes D' did not read past C until after D was removed from the list. So when the write order is opposite of the traversal order, the program order of the writes is preserved for all readers even without the use of *wait-for-all-readers*.

Many updates meet the traversal order requirement for eliminating the *wait-for-all-readers*. To illustrate the types of updates that meet this requirement, a relativistic hash table move operation was developed that did not require any *wait-for-all-readers*.[8] With both red-black trees and AVL trees, it is possible to do an insert or delete, including rebalancing, with not more than one *wait-for-all-readers*.[3]

4.3 A Prefix of Program Order

We have shown that relativistic programming can be used to reduce the number of states that a reader can see. This reduces the number of states that must be considered when reasoning about the correctness of a program. There is one additional point that needs to be made concerning this. Writers can guarantee that readers will only see a prefix of the memory writes as performed in program order. If a writer makes five changes to memory labeled $A..E$, then the writer can guarantee that the only combination of changes a reader can see are: *none*, A, AB, ABC, $ABCD$, $ABCDE$. The mechanisms for guaranteeing this are either to use *wait-for-all-readers* or to rely on the traversal order and update order.

The prefix property makes it much easier to reason about the correctness of a relativistic program. Programmers tend to think of their code as executing sequentially. Sometimes concurrent threads appear to see the effects of another thread non-sequentially. This can lead to programming errors. By eliminating this possiblity, relativistic programming means that the programmer only needs to consider, "is the data consistent following this particular memory write?" If the answer to this question is "yes" for every reader visible update, then the program can be proven correct.

5 Conclusion

We have introduced relativistic programming and compared it to reader-writer locking. Relativistic programs have the following advantages over programs written using reader-writer locks:

1. Relativistic programs allow more concurrency and thus should scale better
2. Relativistic programs allow writes to happen sooner and allow readers to more quickly observe writes

We also discussed properties of relativistic programs that can be used to reason about their correctness. In particular, relativistic programs can be written such that the following are true:

1. The number of observable states is n where n is the number of reader visible changes made to memory
2. Readers can only observe a prefix of the program order of reader-visible updates made to memory

By maintaining these two properties, relativistic programming makes it easy to reason about the correctness of a program even though that program allows read-write concurrency.

Acknowledgements

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

References

1. Desnoyers, M.: Low-Impact Operating System Tracing. Ph.D. thesis, École Polytechnique de Montréal (December 2009), [Online]. Available: `http://www.lttng.org/pub/thesis/desnoyers-dissertation-2009-12.pdf`.
2. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Language Systems 12(3), 463–492 (1990)
3. Howard, P.W., Walpole, J.: Relativistic red-black trees. Tech. Rep. 1006, Portland State University (2011), `http://www.cs.pdx.edu/pdfs/tr1006.pdf`
4. Lev, Y., Luchangco, V., Olszewski, M.: Scalable reader-writer locks. In: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures. pp. 101–110. SPAA '09, ACM, New York, NY, USA (2009), `http://doi.acm.org/10.1145/1583991.1584020`
5. McKenney, P.E.: Kernel korner: using RCU in the Linux 2.5 kernel. Linux J. 2003(114), 11 (2003)
6. McKenney, P.E.: 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: `http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf` [Viewed October 15, 2004]
7. Mellor-Crummey, J.M., Scott, M.L.: 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. pp. 106–113. ACM, New York, NY, USA (1991)
8. Triplett, J., McKenney, P.E., Walpole, J.: Scalable concurrent hash tables via relativistic programming. SIGOPS Oper. Syst. Rev. 44, 102–109 (August 2010), `http://doi.acm.org/10.1145/1842733.1842750`
9. Triplett, J., McKenney, P.E., Walpole, J.: Resizable, scalable, concurrent hash tables. Tech. Rep. 1101, Portland State University (2011), `http://www.cs.pdx.edu/pdfs/tr1101.pdf`