3-2011

Scalable Correct Memory Ordering via Relativistic Programming

Josh Triplett
Portland State University

Philip William Howard
Portland State University

Paul E. McKenney

Jonathan Walpole
Portland State University

Scalable Correct Memory Ordering via Relativistic Programming

Josh Triplett
Portland State University
josh@joshtriplett.org

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

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

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

Abstract

We propose and document a new concurrent programming model, *relativistic programming*. This model allows readers to run concurrently with writers, without blocking or using expensive synchronization. Relativistic programming builds on existing synchronization primitives that allow writers to wait for current readers to finish with minimal reader overhead. Our methodology models data structures as graphs, and reader algorithms as traversals of these graphs; from this foundation we show how writers can implement arbitrarily strong ordering guarantees for the visibility of their writes, up to and including total ordering.

1 Introduction

Concurrent programming faces a difficult tradeoff between simplicity and scalability. Simple concurrent algorithms incur high synchronization costs, while highly scalable algorithms prove highly complex to develop and maintain. This dilemma hinges primarily on the ordering of memory operations, and the order in which they become visible to threads. Simple concurrent algorithms force more serialization to preserve sequential reasoning techniques, while scalable algorithms allow more reordering and force the programmer to think about concurrent behavior.

Existing concurrent programming models approach this dilemma by choosing a point on the sliding scale and attempting to incrementally reduce the drawbacks associated with that point. Transactional memory presents a simple programming model where transactions appear serialized, and strives to eliminate the expensive synchronization costs. Scalable algorithms seek simplicity by introducing reasoning models for weakly ordered memory operations, and using proof systems to verify correctness.

We propose *relativistic programming*, a new model for concurrent programming which achieves scalability without sacrificing the simplicity of strong ordering guarantees. This model allows readers to run concurrently with writers, without blocking or using expensive synchronization. Despite this concurrency, however, relativistic writers can strictly control the order their writes become visible to readers, allowing straightforward reasoning about relativistic algorithms.

Readers commonly perform multiple separate memory reads as they traverse a data structure. By allowing readers to run concurrently with writers, and avoiding any delays in readers, we allow readers to perform their reads interleaved with a writer's writes. It therefore becomes the writer's responsibility to prevent readers from observing inconsistent states. Without blocking readers, the writer cannot generally make a series of writes appear atomic, but our methodology allows the writer to maintain the order of its writes. With no write-side synchronization, readers could observe any possible permutation of writes; by adding write-side synchronization, writers may strictly preserve program order for any or all writes, allowing any ordering policy up to and including total order. The choice of ordering policies affects writer latency, but never impacts reader performance or scalability.

"Laws of Order" [1] documents a set of interlocked properties for concurrent algorithms, and proves that any algorithm with all those properties must necessarily rely on expensive synchronization instructions. These properties include "strong non-commutativity": multiple operations whose results both depend on their relative ordering. Relativistic readers, however, cannot execute strongly non-commutative operations—a relativistic reader can-

not affect the results of a write, regardless of ordering. This allows relativistic readers to avoid expensive synchronization while maintaining correctness.

Relativistic programming builds on several existing synchronization primitives; in particular, relativistic writers rely on a primitive that waits for all current readers to finish. Read-Copy Update (RCU) implementations [10, 9, 7] provide such a primitive, with little to no overhead for the corresponding read-side delineation [8]. Any synchronization primitive with the same semantics will work as well.

Section 2 documents the relativistic programming methodology. Section 3 presents the status of our ongoing research in this area. Section 4 summarizes our contributions.

2 Methodology

Consider a simple set of reader and writer algorithms based on mutual exclusion. Each algorithm begins by acquiring a lock, and ends by releasing that lock. Readers and writers run for a finite duration, delineated by these lock operations. As far as other threads can observe, all operations of a running thread occur between the lock and unlock operations; since other threads cannot examine the data structure without acquiring the lock themselves, the operations performed by a thread appear atomic. Thus, any set of reads and writes in this system will appear sequential.

Reader-writer locking does not change this property; readers cannot modify the memory observed by other readers, so running readers concurrently does not affect the sequential consistency of the system; to generate an equivalent sequential order, choose any arbitrary order for read operations. Similarly, common applications of non-blocking synchronization and transactional memory will roll back readers or writers to ensure that reader operations cannot observe intermediate states.

When using a relativistic programming technique such as RCU, readers and writers still have finite durations, but readers no longer exclude other readers, or even writers. Thus, any number of readers can run concurrently, together with a simultaneous writer, providing joint-access parallelism. Multiple writers continue to use disjoint-access parallelism, accessing separate data simultaneously through the use of locking or partitioned data structures.

A reader may perform multiple distinct read operations on memory in the course of traversing a single data structure. For instance, in the course of looking up an item in a hash table, a reader may read the base address of the hash table, the head pointer of the desired hash bucket, and the various "next" pointers of the linked list in that bucket. These multiple reads may still represent a single semantic operation on some abstract data type, such as the lookup of an item in a map or set.

Similarly, a single writer may need to perform multiple modifications to implement a single semantic operation. For example, to implement removal from a linked list or hash table, a writer must modify the preceding next pointer (or head pointer) to point past the disappearing item, and then reclaim the memory associated with that node. To move a node in a linked list, a writer could insert the new node and remove the old, in some order.

In the absence of additional synchronization, if a writer with concurrent readers performs two memory operations on a data structure, a reader traversing the data structure and reading these memory locations at different times may observe four distinct states of memory: the state with neither operation performed, the state with both operations performed, or either of the two states with a single operation performed. (These same principles extend to more than two memory operations in the obvious manner, with 2^n possible reader observations for n memory operations; subsequent examples will initially focus on the case of two memory operations.)

With neither operation performed, the state of the data structure appears the same to a reader as if that reader completed before the writer began; thus, observing this state cannot do harm to a reader. Similarly, with both operations performed, the state of the data structure appears the same to a reader as if that reader began after the writer completed; thus, observing this state can do no harm either. Incorrect behavior can only occur if a reader observes one of the two possible intermediate states.

For an example of such harm, consider the removal of a node described earlier: first unlink the node, then free the memory. If a reader does not observe the node unlinked (and thus sees the node), the reader must not observe the memory reclamation; otherwise, the reader would access reclaimed memory.

Mutual exclusion takes the simplest approach to avoid this problem: by preventing readers from running concurrently with writers, readers cannot observe intermediate states at all, and the writer's modifications appear atomic. This simplifies reasoning about the writer's algorithm, making it only necessary to consider the initial and final states of the data structure. In the removal example, the reader would either observe the node in the data structure

and not reclaimed, or would not observe the node at all.

However, the writer often does not need to guarantee full atomicity of its changes. Of the four possible orderings of the two changes required to remove a node, only one causes a problem in practice: not observing the removal (thus finding the node) and subsequently observing the memory reclamation (reading reclaimed memory). The reverse situation for removal cannot arise: a reader cannot observe the removal and care about the reclamation one way or another. Thus, the writer need only ensure that the operations appear in a particular order to the readers, rather than making them appear atomic.

In the case of moving a node in a linked list or hash table, both orderings of operations present unusual results. If a reader observes the removal and not the insertion, it will fail to find the item at all. If a reader observes the insertion and not the removal, it will observe two copies of the item. However, the latter scenario might not violate the semantics of the data structure; if looking for that node, the reader will find one of the two identical copies, and if looking for another node, the reader will skip the duplicated items. Barring some additional semantic requirement, this ordering does not pose a problem. Thus, again, the writer need only preserve the order of its write operations. This tolerance of duplicates allows relativistic writers to copy part of a data structure and operate on that copy to avoid disturbing concurrent readers.

Without the ability to block readers, a writer must instead avoid allowing readers to observe any alternative orderings of write operations. Once the writer has performed the first operation, some concurrent readers will observe that operation, and others will not. The former readers cannot cause a problem, since whether they observe the second write or not, they will not violate the ordering. The latter readers, which have not seen the first write, constitute potential hazards, since if they subsequently observe the second write, they violate the ordering constraint.

The writer can achieve this ordering with a "wait for current readers" operation, such as those provided by RCU. A reader can only observe two operations in different orders if its traversal runs concurrently with both operations. Thus, if the writer can prevent readers from running concurrently with both operations, the writer has enforced the order of those operations. The semantics of the "wait for current readers" operation require that if the beginning of a reader occurs before the beginning of the wait operation, the end of the reader must occur before the end of the wait operation. Thus, if a writer inserts a "wait for current readers" operation between two other writes, no reader can run concurrently with both writes, because any such reader would necessarily begin before the wait operation begins and end after the wait operation ends, violating the wait semantics.

This introduces a key property of our methodology for correct relativistic programming: waiting for current readers acts as a strong ordering barrier between two write operations. This mechanism extends to ordering an arbitrary number of operations, as long as a wait operation occurs between each pair. Thus, from 2^n possible observations of n operations, a writer can eliminate all but n. The remaining possibilities consist of observing incrementally longer prefixes of the series of operations, up to observing the entire set.

In theory, a relativistic writer can order its operations with this approach alone and achieve correctness. However, the "wait for current readers" operation adds significant latency to the writer, and using it multiple times replicates that latency. Thus, for practical reasons, the writer needs alternative means of ordering write operations that does not always require a wait operation.

To implement a lightweight ordering mechanism, writers can take the reader algorithms into account. Readers, when traversing a data structure, introduce a natural ordering corresponding to the order they traverse nodes. Almost all modern CPU and memory architectures guarantee that if a CPU reads a pointer value, then dereferences that pointer to read the target memory (a *dependent read*), the fetch of the pointer will occur before the fetch of the target memory.[1] Thus, dependent reads occur in the reader's program order.

Common data structures storable in shared memory will look like some subset of a directed graph: allocated memory corresponds to a graph node, and a pointer corresponds to a directed edge, from the node containing the pointer to the node containing the target address. Because dependent reads occur in program order, the reader has a well-defined node position at any given point in its traversal; call this the reader's *cursor*. A writer cannot easily observe

[1] On architectures such as DEC Alpha which do not automatically guarantee memory ordering for dependent reads [5], this traversal requires appropriate barriers for such reads, such as `smp_read_barrier_depends` in the Linux kernel, used in `rcu_dereference`.

Aggressive compiler optimizations, particularly those regarding local caches of global data, can also lead to problems in this step [2]. This may necessitate compile-time barriers to locally prevent such optimizations.

a reader's cursor directly, but it can nonetheless rely on these cursors moving according to the reader's traversal algorithm.

Similarly, write operations occur at a well-defined node position. Readers whose cursors will reach that node position, but have not yet done so, must necessarily observe that write operation. In an acyclic data structure, readers whose cursors have already passed that node position cannot observe that write operation.

To avoid excess complexity in this reasoning, consider only those structures isomorphic to an acyclic graph. This excludes structures in which the read algorithms may revisit the same node more than once in a single traversal, which significantly simplifies reasoning about the progression of read cursors. This does not prohibit structures in which cycles exist, only structures in which the readers may perform cyclic traversals. For example, a doubly linked list would support cyclic traversals, but if we prohibit readers from following the "previous" pointers, reader traversals will remain acyclic. Writers may still use those previous pointers for their operations.

With this conceptual framework in mind, consider again the algorithm for moving a node via insertion and removal. The writer must insert the new node, and subsequently remove the original node, without allowing any reader to observe the second operation and not the first. The writer's insertion occurs at a particular position in the list. After the writer has inserted the new node, reader cursors before that position will observe the node, and cursors past that position will not. An ordering violation will occur if the reader cursors past the insert position can subsequently observe the removal; this would contradict the writer's program order, and cause the reader to miss the node entirely.

If the newly inserted node appears in the list before the original node, read cursors between the two positions might observe an ordering violation: they have already missed seeing the inserted node, and might potentially miss seeing the original node as well, making the node appear missing. The writer can preserve ordering by waiting for those readers to finish; this will "flush" all existing cursors from the structure, and any new readers must start from the well-defined start locations for traversals—in the case of a linked list, the head of the list. Any new read cursors must pass through the inserted node, and thus cannot observe the ordering violation.

However, if the newly inserted node appears in the list after the original node, no readers can possibly observe a problematic ordering violation. Readers between the two locations must have observed the original node, and may subsequently observe the new node, but as previously described this duplication need not violate the data structure semantics. Thus, the writer need not flush all cursors from the structure. As long as the writer preserves the program order of its writes via appropriate system write memory barriers, the readers will observe the writes in the same order. Such a write memory barrier can occur implicitly as part of a relativistic "publish" primitive for pointers.

This observation provides a general rule for efficiently ordering writes: writers may rely on reader traversal order to provide a natural order for observations, and need only wait for current readers to finish if the natural order does not imply the desired order.

Finally, while we have provided the tools by which a relativistic writer can preserve the ordering of arbitrary write operations, a given write algorithm need not preserve every possible ordering constraint. A writer may omit the synchronization between two writes if readers will not care about the order in which those writes become visible; subsequent synchronization will still order that entire set of writes with resepect to later writes. In particular, a writer may choose to batch a series of updates with a single synchronization. Similarly, independent writers may not need to order their updates with respect to each other, despite ordering the individual steps of each update to preserve correctness. Whether a writer may weaken ordering in this way depends on the semantics required by the corresponding read algorithms; however, this choice cannot affect the performance of readers, only that of writers.

Summarizing the properties of correct relativistic algorithms:

1. Writers need only enforce ordering of their operations, not atomicity of an entire block of operations.

2. Writers can wait for current readers between two write operations to enforce the order of those operations for concurrent readers.

3. Writers may rely on reader traversal order to provide a natural order for their observations, and need only wait for current readers to finish if the writer might overtake a concurrent reader traversing in the same direction; otherwise, a standard write memory barrier suffices.

4. Writers may choose to leave a series of writes unordered and omit synchronization between them, while still using synchronization before or

after those writes to order other writes with respect to the entire series. This allows writers to reduce write latency by relying on the semantics of the reader algorithm, without affecting reader scalability.

3 Ongoing Research

Our current methodology for relativistic programming focuses on the scalability of concurrent readers, by pushing the costs of synchronization and correctness onto writers. This approach proves ideal for many common read-mostly workloads. The converse case of write-mostly workloads demands an approach which allows writers to proceed without synchronization, with readers bearing the synchronization costs. We can achieve that goal through partitioned data structures, which allow writers to perform CPU-local or thread-local writes without synchronization; readers then observe all relevant local structures to obtain a result. Simple cases of partitioned data structures include sloppy counters [3] or approximate counters [4], which partition a single integer counter. We plan to provide a more comprehensive treatment of write-optimized concurrent algorithms in the future.

Most existing relativistic algorithms synchronize between writers using mutual exclusion, with various degrees of granularity. However, to simplify the construction of relativistic writer algorithms, we have also developed a technique to implement relativistic writers using transactional memory, without disrupting or rolling back concurrent relativistic reads.

Scalable relativistic implementations of various data structures already exist based on Read-Copy Update, including linked lists, radix trees, and hash tables. We recently developed relativistic algorithms for resizable hash-tables [11] and balanced trees [6], based heavily on the new general methodology we've presented here.

In section 1, we discussed "Laws of Order", the properties it introduces which require expensive synchronization instructions, and how relativistic readers can avoid these properties and the corresponding cost. We have completed an extensive analysis of "Laws of Order" and its implications for scalable concurrent programming, including the remaining properties which imply expensive synchronization instructions, and the various ways to avoid those properties and the associated cost. We plan to document this analysis in future work.

4 Conclusion

Our proposed relativistic programming methodology supports the construction of concurrent algorithms for arbitrary acyclic data structures. Relativistic readers provide excellent scalability and performance by never blocking or using expensive synchronization instructions. The corresponding relativistic writers may enforce a total ordering of write operations based on writer program order, or weaken that ordering to decrease write latency, without affecting reader scalability. Relativistic programming thus allows concurrent algorithms to achieve scalability without sacrificing the simplicity of strong memory ordering.

5 Acknowledgments

Funding for this research provided by two Maseeh Graduate Fellowships, and by the National Science Foundation under Grant No. CNS-0719851. Thanks to Dr. Fariborz Maseeh and the National Science Foundation for their support.

References

[1] Attiya, H., Guerraoui, R., Hendler, D., Kuznetsov, P., Michael, M. M., and Vechev, M. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot be Eliminated. In *Proceedings of the ACM POPL'11* (2011).

[2] Boehm, H.-J. Threads cannot be implemented as a library. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (2005), ACM, pp. 261–268.

[3] Boyd-Wickizer, S., Clements, A. T., Mao, Y., Pesterev, A., Kaashoek, M. F., Morris, R., and Zeldovich, N. An Analysis of Linux Scalability to Many Cores. In *OSDI 2010: Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation* (2011).

[4] Corbet, J. The search for fast, scalable counters. Linux Weekly News. `http://lwn.net/Articles/202847/`, February 2006.

[5] Digital Equipment Corporation. Shared Memory, Threads, Interprocess Communication, August 2001.

[6] Howard, P. W., and Walpole, J. Relativistic red-black trees. Tech. Rep. 10-06, Portland State University, 2010. `http://www.cs.pdx.edu/pdfs/tr1006.pdf`.

[7] McKenney, P. E. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[8] McKenney, P. E. RCU vs. locking performance on different CPUs. In *linux.conf.au* (Adelaide, Australia, January 2004).

[9] McKenney, P. E., Sarma, D., Arcangeli, A., Kleen, A., Krieger, O., and Russell, R. Read-copy update. In *Ottawa Linux Symposium* (June 2002), pp. 338–367.

[10] McKenney, P. E., and Slingwine, J. D. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems* (October 1998), pp. 509–518.

[11] Triplett, J., McKenney, P. E., and Walpole, J. Resizable, scalable, concurrent hash tables. Tech. Rep. 11-01, Portland State University, 2011. `http://www.cs.pdx.edu/pdfs/tr1101.pdf`.