3-2011

Generalized Construction of Scalable Concurrent Data Structures via Relativistic Programming

Josh Triplett
Portland State University

Paul E. McKenney

Philip W. Howard

Jonathan Walpole
Portland State University

Citation Details

Triplett, J., Howard, P. W., McKenney, P. E., & Walpole, J. (2011). Generalized construction of scalable concurrent data structures via relativistic programming. Tech. Rep. 14, Portland State University.

Generalized Construction of Scalable Concurrent Data Structures via Relativistic Programming

Pre-Publication Draft

Josh Triplett
Portland State University
josh@joshtriplett.org

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

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

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

Abstract

We present *relativistic programming*, a concurrent programming model based on shared addressing, which supports efficient, scalable operation on either uniform shared-memory or distributed shared-memory systems. Relativistic programming provides a strong causal ordering property, allowing a series of read operations to appear as an atomic transaction that occurs entirely between two ordered write operations. This preserves the simple immutable-memory programming model available via mutual exclusion or transactional memory. Furthermore, relativistic programming provides *joint-access parallelism*, allowing readers to run concurrently with a writer on the same data.

We demonstrate a generalized construction technique for concurrent data structures based on relativistic programming, taking into account the natural orderings provided by reader traversals and writer program order. Our construction technique specifies the precise placement of memory barriers and synchronization operations. To demonstrate our generalized approach, we reconstruct the algorithms for existing relativistic data structures, replacing the algorithm-specific reasoning with our systematic and rigorous construction. Benchmarks of the resulting relativistic read algorithms demonstrate high performance and linear scalability: relativistic resizable hash-tables demonstrate 56% better lookup performance than the current state of the art in Linux, and relativistic red-black trees show 2.5x better lookup performance than transactional memory.

1 Introduction

Two common approaches exist for the construction of concurrent programs. The shared memory approach assumes a single underlying memory resource available to all concurrent processes, and relies on synchronization between these processes to maintain the consistency of data in memory. Distributed approaches avoid requiring a single shared memory, opting for local memories available to a subset of processes, and message passing to communicate between processes.

For shared memory synchronization, mutual exclusion techniques remain by far the most common. These techniques provide coherent access to shared memory by reducing concurrency and serializing access. Partitioning shared structures allows mutual exclusion to become more fine-grained, allowing *disjoint-access parallelism*: accesses to disjoint locations may proceed concurrently.

Concurrent programming via mutual exclusion introduces a number of potential correctness problems, such as deadlock and priority inversion. These problems have seen extensive study, and well-known solutions exist to avoid or mitigate them. The solutions establish a set of construction rules for practitioners to follow; for example, to avoid deadlock, concurrent algorithms establish a total order on simultaneously acquired locks. Despite these well-known solutions, algorithms based on mutual exclusion can never qualify as *lock-free* or *wait-free*.

Alternative approaches to shared-memory synchronization can avoid these problems, and provide lock-free and wait-free operation. *Transactional memory* provides a high degree of disjoint-access parallelism, equivalent to fine-grained locking, while avoiding the well-known problems of locking.

However, approaches based on shared memory tend to rely on strong memory consistency models. These models make concurrent programs far easier to write, but incur high communication costs that limit scalability. [1, 3, 13]

By contrast, distributed techniques can quite effectively utilize available processing power on a system without shared memory, avoiding the scalability limitations of coherent shared memory. Even modern shared-memory multiprocessor systems increasingly exhibit properties of distributed systems; projects such as Corey [7], fos [38], and the Barrelfish Multikernel [5] have demonstrated the effectiveness of distributed techniques on systems that support shared memory. Efficient distributed algorithms typically require partitioning, dividing data into separate partitions and using a separate process (or a separate system) to operate on each partition; this approach provides excellent disjoint-access parallelism. Many efficient shared-memory algorithms use partitioning for the same reason.

A large gulf remains between the shared-memory and distributed approaches. Despite their equivalence in expressive power [39], in practice code written for one model requires extensive modifications to perform well with the other. Most code written for existing operating systems depends on shared memory; programs designed for distributed operation typically consist of a collection of processes, each of which runs on a system using the shared memory abstraction. Furthermore, high-performance distributed systems running in a shared memory environment tend to implement message passing via shared memory, to take advantage of this highly optimized communication channel.

The *end-to-end principle* [30] argues that we should avoid implementing costly functionality in an abstraction layer that only a subset of higher-level systems built on that layer will benefit from, since all systems built on that layer must pay the cost. Forcing processes on a shared-memory system to communicate via message passing, and implementing the message-passing system on shared memory for efficiency, violates the end-to-end principle. Any programs which could directly run on shared memory can run on a message-passing system, but they will pay an unnecessary additional cost for the abstraction. Conversely, forcing a distributed system to implement the abstraction of coherent shared memory also violates the end-to-end principle; current parallel hardware finds itself in this position, as it has many of the architectural and performance characteristics of a distributed system but must continue to support a the shared-memory abstraction.

Ideally, a model for concurrent programming should avoid introducing unnecessarily costly abstractions, but should still support straightforward algorithms for concurrent programs. This model should provide the full potential scalability of distributed operation, but must still run efficiently on existing shared-memory hardware. We want to leverage the large existing codebase of shared-memory concurrent software, without requiring extensive modification. Finally, we wish to go beyond the limits of disjoint-access parallelism, and strive for *joint-access parallelism*, allowing readers to run concurrently with a writer on the same data.

To these ends, we propose *relativistic programming*, a distributed model for concurrent programming based on shared addressing, which does not require either coherent shared memory or explicit message passing. Relativistic programming provides full joint-access parallelism between readers and writers: any number of readers may simultaneously access the same data, together with a writer concurrently modifying that data. Furthermore, relativistic programming provides an ordering property that makes readers appear transactional, as though they occur at a single specific time between two ordered write operations.

We use the term *relativistic programming* by analogy with relativity: independent observers may observe unrelated events in different orders, but must agree on the order of causally related events.

This technique builds on an extensive body of previous research on scalable concurrent data structures, with particular focus on practical implementations widely adopted within the Linux kernel. This previous research utilized the same techniques presented here, but without a model for correctness or a generalized technique for construction. Previous work used ad-hoc reasoning to establish ordering requirements for operations, and the implementation of those ordering requirements in a given system. The relativistic programming model we present here systematizes and generalizes those techniques.

In section 2, we outline the minimal level of functionality we assume from the underlying shared-addressing system, and then construct our relativistic programming abstraction on top of that baseline, defining the properties we guarantee and the mechanisms by which we can guarantee those properties. We then propose in section 3 a set of rules for the construction of relativistic data structures. To demonstrate the effectiveness of these rules, we use them to reconstruct algorithms for existing data structures based on preliminary relativistic programming techniques, and compare the resulting algorithms with the originals; these reconstructions and comparisons appear in section 4.

We emphasize practical results, implemented on real systems and running on real hardware. Thus, in section 5, we present a specific, high-performance implementation of relativistic programming, based on primitives available on shared-memory systems. In section 6, we present performance results for our relativistic algorithms based on this implementation.

2 The Relativistic Programming Model

By presenting a concurrent programming model based on shared addressing, we necessarily invite comparisons to shared memory consistency models. We recognize the central role of a memory consistency model in reasoning about any concurrent program or programming methodology based on shared memory, as argued by Adve and Boehm [1]; their work showed that the choice of memory model determines fundamental properties of a concurrent programming environment, and makes a fundamental tradeoff between scalability and ease of programming. Concurrent hardware and software can scale most effectively with a weaker memory consistency model, while a stronger memory consistency model allows greater simplicity of implementation. [1, 3, 13]

In basing relativistic programming on shared addressing, we apply the end-to-end principle: we specify the minimal behavior expected from the underlying system, and then specify consistency properties as features provided by the programming model rather than demands placed on the target system. We then rely on the generalized construction technique for relativistic algorithms to separate and automate the task of enforcing those properties from the task of designing algorithms relying on those properties. This allows relativistic algorithms to assume the consistency properties we guarantee without requiring that the underlying system provide those properties for all programs.

By *shared addressing*, we mean that all processes agree on a set of addresses, which refer to locations holding arbitrary values of a specific fixed word size. Two different processes must assign the same meaning to the same address. However, an address does not necessarily refer to a single canonical backing storage for values, as in a shared-memory system. Rather, an address may refer to a copy of that location in memory local to the process making the reference, which may represent a cache of some more authoritative location, or an authoritative replica for which no master location exists.

Given an address, a process may perform a write or a read operation. Writing an address stores a new value to the writing process's storage location for that address. Reading an address returns the corresponding value according to the reading process's storage location for that address. We require that the value returned by a read must precisely match a value written by a previous write, and not an arbitrary value or a bitwise combination of previously written values. While synchronization algorithms without this requirement do exist, notably Lamport's bakery algorithm for mutual exclusion [16], supporting joint-access parallelism without this property would prove far more difficult, and existing systems all guarantee this property, at least for word-sized word-aligned operations.

We require a form of *eventual consistency* for all local storage: eventually all written values become accessible to all potentially interested readers [33, 37]. In particular, we assume a stronger form of eventual consistency, in which writes continue to propagate while writers continue to perform new writes. Multiple writes to the same address must still result in a single value eventually propagating to all readers, though determining the winning write requires synchronization between writers. Limiting the duration of "eventually", or controlling the order of read and write operations, requires explicit constraints; we introduce the necessary constraints and their semantics as part of our model.

The relativistic programming model enforces a causality property: a reader which reads two different addresses will not observe results inconsistent with the order of writes to those addresses. We will formally define this ordering property and the explicit ordering constraints required to achieve it in the remainder of this section; section 3 defines the construction technique which specifies the necessary placement of these ordering constraints.

We define readers and writers as consisting of a series of memory operations, $R_1, R_2, \ldots, R_m$ and $W_1, W_2, \ldots, W_n$ respectively.[1] A process in turn consists of a series of read or write operations, each of which begins at a given delineated point, performs its series of memory operations, and ends at a subsequent point. Note in particular that we define a single reader or writer as consisting of the set of

[1]Formal notations exist for memory models and consistency properties [4, 2]. We introduce a minimal notation here for greater clarity in subsequent definitions, without attempting to write the definitions in pure mathematical notation.

memory operations needed to implement a single operation on the abstract data type, such as an insert or delete; a reader or writer does not extend for the lifetime of a process.

Each read or write operation occurs at a specific address. As previously defined, a read at a given address returns a value written by some previous write to the same address. Given a read R_i at the same address as a write W_j, we say that R_i *observes* W_j if R_i returns the value written by W_j. (We refer specifically to the value written by W_j, distinct from any other equivalent value, to avoid the ABA problem.) Otherwise, we say that R_i *fails to observe* W_j.

Readers have a natural ordering of memory operations, as defined by their program order. That program order arises from the *traversal order* of the shared data structure accessed by the reader; for instance, walking a list from head to tail, or a tree from the root to a leaf. Writers imply a natural ordering as well, based on their own program order; writers may choose to relax the ordering requirements between two writes, defining a partial order. Without further constraints, however, those orderings remain entirely local to the individual readers and writers.

Given two reads R_i and R_j performed by a single reader, we say that $R_i \prec R_j$ if R_i occurs before R_j in the reader's program order. Similarly, given two writes W_i and W_j performed by a single writer, we say that $W_i \prec W_j$ if W_i occurs before W_j in the writer's desired partial order. These orderings hold transitively: if $R_i \prec R_j$ and $R_j \prec R_k$ then $R_i \prec R_k$, and the analogous law holds for writers.

We can now define the precise ordering property we desire for relativistic programming. Given a reader performing two reads R_i and R_j, and a writer performing two writes W_k and W_l at the same addresses as R_i and R_j respectively, if $W_k \prec W_l$, then we do not allow the case where R_i fails to observe W_k but R_j observes W_l. (Note in particular that we require this property regardless of the ordering of R_i and R_j within the reader.) This case would allow the reader to observe a result inconsistent with the writer's desired order. By avoiding this case, we ensure that the reader observes a state of memory consistent with a single point in the writer's order.

This property avoids the need to analyze complex interactions between multiple read and write operations. Instead, a reader appears to execute atomically between two ordered writes of the writer: the reader can observe some prefix of the writes performed by the writer, and cannot observe the remaining writes. The writer thus need only consider the state of memory (and the semantics of data structures in memory) after each write it performs, rather than every possible interleaving of read and write operations. In the case where the writer uses program order rather than some looser partial order, this makes the number of states to consider linear in the number of write operations, rather than exponential as in the case without this ordering property.

The property we have specified provides a stronger consistency model than PRAM consistency [17], as it enforces the ordering of writes from a single writer. If causally-related writers enforce ordering between their causally-related writes just as they do between writes within the same writer, then the property we have specified also provides a stronger consistency model than causal consistency [2]. Both PRAM consistency and causal consistency only require that a series of reads performed by a reader observe an ordered series of writes in a manner that does not regress: each read can observe strictly more writes than the previous read. Our consistency property eliminates the interleaving entirely, requiring all reads within a reader to observe the same prefix of writes performed by a writer. This creates a form of transaction for the reader, making its reads appear atomic with respect to the write ordering.

The underlying shared-addressing model does not ensure the ordering property we have defined. To enforce that property, we require explicit ordering constraints in readers and writers, expressed in the form of barrier operations; these operations promote certain local ordering properties within a process into global ordering properties across processes. We require three kinds of barriers: read barriers, write barriers, and wait-for-current-readers barriers.

A read barrier enforces a causal property between the preceding reads and the subsequent reads. Given a reader performing reads R_i and R_j, and a read barrier RB such that $R_i \prec RB \prec R_j$, if R_i observes a write W_k at the same address as R_i, then the read R_j will not fail to observe any write ordered before W_k. (However, see section 5 and the definition of these barriers on real hardware for why this requirement need not imply an expensive hardware barrier.)

A write barrier enforces a causal property between the preceding writes and the subsequent writes. Given a writer performing writes W_i and W_j, and a write barrier WB such that $W_i \prec WB \prec W_j$, if a reader performs read operations R_k and R_l at the same addresses as W_i and W_j respectively, and $R_l \prec R_k$, then if R_l observes W_j, R_k will observe W_i. In other words, if the earlier read observes the later write, the later read will observe the earlier write.

Notice that the definitions of read and write barriers interact. The definition of a read barrier uses the partial order of writes as defined by writers, and the the definition of a write barrier uses the order of reads as defined by readers.

A wait-for-current-readers barrier enforces ordering between write operations and an entire reader. Given a writer performing writes W_i and W_j, and a wait-for-current-readers barrier B such that $W_i \prec B \prec W_j$, if any read operation R_k at the same address as W_i fails to observe W_i, no other read operation in the same reader as R_k may observe W_j.

3 Generalized Construction Technique

Having defined the underlying shared-addressing model, the desired ordering property of relativistic programming, and the barrier operations we can use to enforce that ordering property, we now present our generalized construction technique for relativistic data structure algorithms.

Our ordering property allows us to separate the construction of relativistic algorithms into three components: the construction of reader and writer algorithms based on our desired memory model, the placement of barriers to properly enforce that memory model, and the synchronization between writers.

Relativistic data-structure write algorithms require some adaptation from the normal assumptions of sequential algorithms. While a relativistic writer need not cope with arbitrarily interleaved readers, it must not block readers at any time, and thus it must assume that readers may run to completion between any pair of ordered writes. Therefore, each write performed by a relativistic writer must leave the data structure in a consistent state for readers (though not necessarily for other writers, depending on writer synchronization).

Since the shared-addressing model allows a writer to assume that a single write operation will appear atomic to readers, a relativistic writer may manipulate pointers within a data structure and assume that readers will see either the old or new pointer value. If a writer needs to modify a larger value atomically, it may do so by constructing a new value (including any outbound pointers), and then atomically changing the pointer to that node.

Since the relativistic programming model makes readers appear to run to completion between two ordered write operations, without interleaving with concurrent writers, relativistic data-structure read algorithms need not differ from the sequential algorithms usable with mutual exclusion or transactions.

Given appropriate reader and writer algorithms based on the relativistic programming model, we can mechanically insert barriers to construct algorithms designed to run natively on the weakly ordered system memory model. The read barriers prove simplest: any pair of read operations within a reader must have a read barrier between them to enforce that ordering. (Again, see section 5 for why this requirement need not imply an expensive hardware barrier.)

To determine where to insert barriers in a relativistic writer, consider a writer which performs writes W_i and W_j, with $W_i \prec W_j$. Per our ordering requirement, we must prevent the case where a reader performs a read R_k that fails to observe W_i and a read R_l that observes W_j. We consider three independent cases:

1. If no reader exists that performs reads R_k and R_l at the same addresses as W_i and W_j, respectively, then no reader may violate the ordering constraint. Thus, the writer need not perform any barrier operation.

2. If readers perform reads R_k and R_l such that $R_l \prec R_k$, then the writer need only execute a write barrier between its writes. Per our definition of a write barrier, if R_l observes W_j, then R_k will observe W_i.

3. If any reader performs reads R_k and R_l such that $R_k \prec R_l$, a write barrier does not suffice. However, a wait-for-current-readers barrier will ensure the desired ordering. Per our definition of a wait-for-current-readers barrier, if R_k fails to observe W_i, then no read operation in the same reader, including R_l, can observe W_j.

These cases cover the most general scenario of reads and writes to arbitrary memory addresses. In the case of a shared data structure, the three cases have natural interpretations based on the reader traversal order in the data structure. If a writer performs writes in independent parts of the data structure, not reachable in the same reader traversal, the writer need not execute any barrier between those writes. If a writer performs writes in the reverse of the order that readers traverse the data structure, the writer need only execute a write barrier between those writes. If a writer performs writes in the same order that a reader traverses the data structure, the writer must wait for current readers between those writes.

Note that if a single reader reads the same address multiple times, none of the ordering barriers can prevent that reader from potentially seeing different values for each such read. For the purposes of our construction technique, we limit ourselves to data structures with acyclic read traversals. In practice, this does not significantly restrict the set of data structures usable with relativistic programming; note in particular that it does not prohibit cyclic accesses used by writers only, such as previous pointers in a doubly-linked list or parent pointers in a tree. Shared cyclic data structures prove difficult to handle via fine-grained locking as well. In section 7, we reference some approaches to handling cyclic data structures and traversals.

Relativistic programming does not define a specific method of synchronization between writers. In the simplest case, writers may synchronize via coarse-grained locking; if writes occur sufficiently infrequently compared to reads, this may suffice. To improve concurrency, writers may opt for fine-grained locking instead, partitioning the data structure into independent pieces and applying a lock to each. Typically, these pieces will remain independent for readers as well, making any ordering concerns between writers moot; however, if readers may potentially read addresses written simultaneously by multiple writers, and the ordering of those write operations matters, the writers must arrange an appropriate barrier between their write operations.

As an alternative approach for allowing write concurrency, writers may use a software transactional memory (STM) system to perform writes. Howard [15] demonstrated a system combining relativistic programming with STM, such that relativistic readers can exist outside the transactional system while still preserving the isolation guarantees of the transactional system for writers. This preserves the low overhead scalable performance of the readers while also allowing for scalable writers within the limitations of the STM system.

As previously suggested in section 2, writers may choose to ignore some write ordering constraints that their program order would otherwise imply, if the order of those write operations does not actually matter to readers; for instance, a writer initializing a structure before publishing it need not order individual writes in the initialization. The rules for the placement of barriers still hold, but requirements to place barriers between two write operations with an ordering relationship no longer specify an exact location in program order. Instead, the writer need only place barriers to satisfy all the ordering constraints. Note that a wait-for-current-readers barrier satisfies a requirement for a write barrier. Finding a correct barrier placement will always prove trivial; future work will present an algorithm for the optimization of finding a minimal barrier placement for partially ordered writes.

4 Reconstructing Relativistic Algorithms

To demonstrate our construction technique for relativistic algorithms, we will revisit the algorithms previously presented in published papers on relativistic data structures. Ignoring the prior placement of barriers in these algorithms, we will apply the systematic construction presented in section 3 to place appropriate barriers. We will then compare the resulting algorithms and systematic reasoning with the original algorithms and the algorithm-specific reasoning required in the original papers.

4.1 Linked Lists

We begin with one of the simplest data structures commonly used with relativistic programming techniques: linked lists. [25] Linked-list readers have precisely one operation: traversing the list from the head to the end via the next pointers of each node, possibly stopping after encountering a node meeting some criteria. This algorithm defines the traversal order a writer must take into account when placing barriers.

The most common writer operations consist of insertion and deletion of an individual list node. Insertion of a node involves allocating and initializing that node (including its next pointer), and then publishing that node to readers by pointing some existing node's next pointer (or the head pointer) to that node. The writer must ensure that the initialization of the node occurs before the pointer manipulation that makes the node accessible to readers. Since readers read the pointer first, and then the node, these writes occur in reverse traversal order, and therefore the writer only needs a write barrier. This result agrees with the established algorithm for relativistic linked-list insertion.

Removing a node from a linked list also consists of two write operations: routing the list pointers around the node to make it inaccessible to all future readers, and reclaiming the memory associated with the node. These operations occur in traversal order, and thus writers must execute a wait-for-current-readers barrier after making the node inaccessible but before reclaiming the memory. This barrier placement agrees with the established algorithm for relativistic linked-list removal.

4.2 Hash Table Resize

Triplett [36] proposed an extension to relativistic hash tables to support resizing the table while allowing concurrent readers. These resizable hash tables supported both shrink and grow operations, in addition to the usual hash-table lookup operation. We analyze both the shrink and the grow algorithms here.

To support resizing, the algorithms refer to a hash table via a pointer to a structure, which includes both the hash buckets and the table size. Both resize algorithms allocate a new table structure, and later change the table pointer to point to the new structure.

Ignoring the originally specified barriers, the algorithm to shrink a hash table performs the following write operations:

1. Allocate the smaller hash table.
2. Link each bucket in the new table to the first bucket in the old table that contains entries which will hash to the new bucket.
3. Link the end of each such bucket to the beginning of the next such bucket; each new bucket will thus chain through as many old buckets as the resize factor.
4. Set the table size.
5. Point the table pointer to the new table.
6. Reclaim the old hash table.

Steps 1, 2, and 4 initialize a structure readers cannot observe yet. The writes in step 3 have no ordering requirements. Step 5 makes this structure accessible, making it a write in reverse traversal order, and requiring a write barrier. Step 6 performs a write in traversal order, necessitating a wait-for-current-readers barrier before reclamation. The resulting shrink algorithm with barriers matches the algorithm documented by the original paper.

Again ignoring existing barriers, the algorithm to expand a hash table performs the following operations:

1. Allocate the larger hash table.
2. Link each new bucket to the first node in the corresponding old bucket that hashes to the new bucket. This results in a valid hash table, with sets of new buckets zipped together into one list.
3. Set the table size.
4. Publish the new table pointer.
5. Loop through the buckets of the old table
    (a) Advance the pointer for that old bucket until it reaches a node that doesn't hash to the same bucket as the previous node. Call the previous node p.
    (b) Find the subsequent node which hashes to the same bucket as p, or NULL if no such node exists.
    (c) Link p to that subsequent node, bypassing nodes which do not hash to p's bucket.
6. If any changes occurred in this pass, repeat the loop in step 5.
7. Reclaim the old hash table.

Steps 1, 2, and 3 initialize a structure published in step 4; as before this requires a write barrier before the publish step. Step 4 must occur before step 5; these writes occur in traversal order, requiring a wait-for-current-readers after step 4. One iteration of the loop in step 5 has no internal ordering requirements, but each iteration for a given old bucket must complete before the next iteration for that bucket. The iterations occur in traversal order, requiring a wait-for-current-readers barrier between each pass. (Each pass operates on many old buckets, but one wait-for-current-readers barrier suffices to separate that pass from the previous.) Since the algorithm uses the old table for temporary storage, the reclamation in step 7 cannot occur until after all passes have completed; one final wait-for-current-readers barrier guarantees this ordering. The resulting expand algorithm with barriers matches the algorithm documented by the original paper.

4.3 Red-Black Trees

Howard [14] proposed a relativistic implementation of red-black trees. These red-black trees store sorted key-value pairs, and support various operations, of which we will consider the lookup, insert, and delete operations.

Readers perform lookups to find the value associated with a given key. A lookup always begins at the root of the tree and progresses toward a leaf. This order defines the natural traversal order for readers, which writers may use to determine the appropriate barriers between write operations.

Writers may perform insert operations to add a given key-value pair, and delete operations to remove the value associated with a given key. In general, following an insert or delete, the tree may require rebalancing to preserve the partial balancing property of red-black trees. Rebalancing consists of a series of tree rotations. Thus, we consider the write algorithms required for insert and delete, as well as

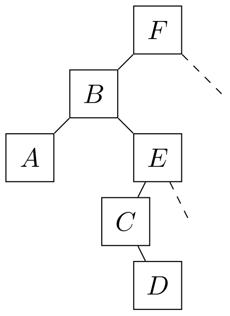

Figure 1: Tree before deletion of node B

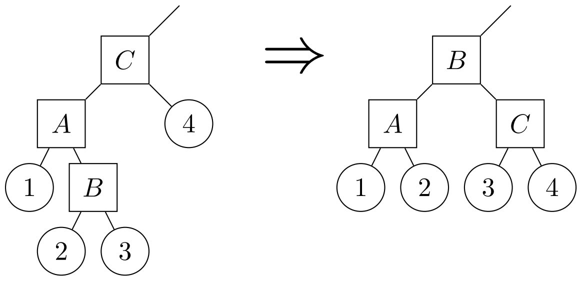

Figure 2: Subtree before and after a double rotation of nodes A, B, and C.

for the rotations required to restore a tree to its balanced state.

Inserts always take place at the leaves. Thus, inserting a new node requires only two writes: allocating and initializing the node, and changing a child pointer in a leaf node to point to the new node. These writes occur in reverse-traversal order, so the writer need only execute a write barrier between them. This barrier placement matches the insert algorithm presented by Howard.

Deletion potentially involves more complexity, depending on the position of the deleted node. Deleting a leaf node only requires changing a single child pointer and then reclaiming the node's memory; since these writes occur in reader traversal order, the writer must execute a wait-for-current-readers barrier between them. However, deleting an internal node requires additional steps. Consider the tree shown in figure 1, in which we want to delete the internal node B. Howard's algorithm does so by replacing B with a copy of C, the left-most node on the right branch of B. This eliminates node B, and leaves the original leaf node C, which we can then unlink and delete using the procedure for leaf nodes.

The algorithm performs the following write operations:

1. Create and initialize a copy of node C, with the same key/value pair as C, and the color and pointers (parent, left, and right) of node B.
2. Link the copy of C into the tree in place of node B, by updating the left pointer of node F
3. Delete the original node C from the tree, by replacing the left pointer of node E.
4. Reclaim the memory of nodes B and C.

The individual writes required to accomplish step 1 can occur in any order, because readers can not yet reach the new node. Steps 1 and 2 occur in reverse traversal order, requiring a write barrier between them. Steps 2 and 3 occur in traversal order, so the algorithm must execute a wait-for-current-readers barrier between them; otherwise, a reader could miss both copies of node C. Finally, steps 3 and 4 occur in traversal order, requiring a second wait-for-current-readers barrier between them. The resulting algorithm with barriers included matches the algorithm originally defined by Howard.

Rebalancing the tree following an insert or delete may require recoloring some nodes. Since readers never observe the color of the nodes, recoloring need not take readers into account, and need not execute any barriers. Rebalancing may also require restructuring the tree using a rotation or double rotation operation. We examine the double rotation here; an analogous procedure holds for other rotations.

Consider the subtree shown in figure 2. The double rotation can occur in one of two ways: by placing a copy of node B above nodes A and C, or by placing copies of nodes A and C below node B. To determine the optimal alternative, notice that moving a node involves copying the original node to the new location and removing the original. The writer must preserve the order of these two steps, to prevent any reader from missing both the original and the copy. If the new location occurs after the old in traversal order, the writer can order them with a write barrier rather than the more expensive wait-for-current-readers barrier. Moving nodes A and C below node B produces this result. Even though this alternative copies two nodes rather than one, benchmarks have demonstrated that this method performs better than moving B by avoiding the additional wait-for-current-readers barrier. Thus, the double rotation proceeds as follows:

1. Allocate a copy of A, with children 1 and 2.
2. Allocate a copy of C, with children 3 and 4.
3. Set B's left child to the copy of A.
4. Set B's right child to the copy of C.

5. Remove the original node A and node C by pointing the parent of the original C to B.
6. Reclaim the unlinked nodes.

This algorithm presents an example of a writer that requires only partial ordering. Steps 1 and 2 have no ordering requirements, because the nodes have not yet become visible to readers. Steps 3 and 4 also have no ordering requirements between them. However, steps 3 and 4 must both occur after steps 1 and 2; since in both cases the writes occur in reverse traversal order, a write barrier suffices. The write in 5 occurs in reverse-traversal order from the writes in steps 3 and 4, requiring a second write barrier before step 5. Finally, as always, the writer must wait for current readers before reclaiming the unlinked nodes in step 6. The resulting algorithm, including barrier placement, matches the original algorithm as defined by Howard.

Note that both the deletion and rotation algorithms insert copies of nodes into the tree, and readers may potentially observe both the original and the copy. If the RBTree represents a set or a map, a lookup will simply return the first value found, so this temporary duplication does not present a problem. If the RBTree represents a multiset, multimap, or some other abstract data type that may have multiple values associated with the same key, this temporary duplication changes the semantics of the data structure; such a structure would need to use a different set of algorithms.

5 Implementation

In section 2, we defined the ordering property for relativistic programming, and a set of barriers we can use to enforce those properties on top of the underlying shared-addressing system. To implement the relativistic programming model and relativistic algorithms using it on real hardware, we must implement the barrier operations in terms of those available on that hardware. We present here an implementation of relativistic programming via shared memory, without assumptions about consistency. (Section 8 outlines a preliminary alternative formulation based on message passing.)

On a shared-memory multiprocessor system, the local memories or replicas we described correspond to the caching layers present between processors and system memory. Various multiprocessor systems implement differing coherence protocols to maintain some consistency property between processors, though none implement the precise ordering property we specified for relativistic programming. These caching layers, as well as other properties of the architecture, potentially allow reordering of memory accesses in violation of the relativistic ordering property. In particular, writes may remain in a writer's store buffer or cache before reaching memory, and reads may return data from a reader's cache without refreshing that cache from memory. All such caching may have different durations for different memory addresses, leading to potential reordering of writes as observed by readers. [12]

In this model, we can describe the behavior of a read or write barrier as forcing preceding operations to interact with memory before subsequent operations. (In practice, a barrier may instead force an operation to become visible to other processors by way of a cache coherence protocol without necessarily forcing it to reach memory.)

Thus, a read barrier forces any read operations following the barrier to obtain values from memory at least as up to date as those returned by reads prior to the barrier, with respect to each writer. A read barrier does not, however, interact with the caching layers of processors other than the one executing the barrier. This matches the semantics of various common read memory barrier instructions provided by current processors, as well as portable abstractions such as the `smp_rmb` function in the Linux kernel or the `_read` ordering operations provided by libatomic-ops.

In the most general case, a reader performing a series of unrelated independent reads would need to execute a read barrier operation between each pair of reads. However, in practice, readers typically perform a series of related reads to traverse some data structure in shared memory. In particular, readers often perform *dependent read* operations, in which subsequent reads depend on the results of previous reads. For instance, reading a pointer and subsequently dereferencing that pointer constitutes a dependent read. All current processors used in shared-memory multiprocessor systems automatically preserve the ordering of dependent reads without any explicit barrier. Thus, a reader performing a data structure traversal that consists entirely of dependent reads need not execute any expensive hardware memory barrier instructions.

As a notable exception, the DEC Alpha processor did not order dependent reads; concurrent algorithms which require portability to Alpha must use an explicit barrier between dependent reads. [9] The Linux kernel provides an `smp_read_barrier_depends` operation with precisely these semantics, which uses an appropriate barrier on Alpha and compiles to nothing on all other architectures. Linux also provides a function for the common case of dereferencing a pointer as a dependent read (having previously read the pointer value itself), and this function includes the necessary barrier implicitly.

For writers, a write barrier follows the same model of bypassing caching layers between the writer and memory. Executing a write barrier does not necessarily guarantee that preceding write operations have reached memory; however, it does guarantee that subsequent write operations will not reach memory any sooner than preceding write operations. This matches the semantics of common write memory barrier instructions provided by current processors, as well as portable abstractions such as the `smp_wmb` function in the Linux kernel or the `_write` ordering operations provided by libatomic-ops.

In addition to the reordering possible in hardware, compilers and language runtimes may also reorder operations, either directly or by providing another layer of caching behavior. For instance, a compiler may read a value from memory into a processor register, and reuse that register for a subsequent read from the same address without re-reading memory, unless explicitly instructed to do otherwise. [6, 12]

To address any reordering provided by the compiler or language runtime, we make use of built-in ordering primitives. For example, the C language provides the `volatile` keyword, which prohibits the compiler from caching the value of a variable. The GCC compiler additionally provides a "memory clobber" constraint which forces the compiler to invalidate all cached references it holds and re-fetch values from memory. All of the barrier implementations we have defined must additionally make use of such primitives to ensure that the compiler does no more reordering than the underlying hardware.

To implement the wait-for-current-readers barrier, we require a mechanism that allows writers to wait for current readers to complete, ideally without incurring synchronization costs in readers. For that purpose, we turn to Read-Copy Update (RCU). [18, 25] RCU provides lightweight processor-local operations processes can use to mark the start and end of a reader, and a `synchronize_rcu` operation which blocks waiting for all currently running readers to finish. Existing applications of RCU have used this wait-for-current-readers operation primarily to implement safe memory reclamation; writers may remove elements from a data structure, making them inaccessible to current readers, and defer garbage collection until all existing readers have finished.

RCU provides joint-access parallelism, allowing readers to run concurrently with writers. Many algorithms exist for scalable data structures based on RCU, including linked lists [18, 11], hash tables [24, 35, 36, 19], balanced trees [14], radix trees, and various other data structures [26]. Because the RCU read-side primitives avoid expensive synchronization operations, applications of RCU produce consistently excellent performance and scalability for readers. [24, 28, 34]

The Linux kernel includes several implementations of RCU, which have encouraged growing adoption by kernel developers, with the current Linux 2.6.38 kernel including over 5000 calls to the RCU API [23]. In addition, URCU provides a portable userspace implementation of RCU. [8]

We can use the RCU `synchronize_rcu` primitive to implement the wait-for-current-readers barrier we require for relativistic programming. `synchronize_rcu` provides the semantics we require: if a writer executes a `synchronize_rcu` between two writes, any reader which fails to observe the earlier write will complete before `synchronize_rcu` completes, and therefore cannot observe the writes after the `synchronize_rcu`. RCU preserves our scalable readers by not adding any expensive synchronization primitives to those readers; RCU's read-side delineation incurs little to no cost.

RCU additionally provides an asynchronous mechanism, `call_rcu`, to wait for current readers and execute a callback when they have all completed. Implementations of relativistic algorithms could choose to use this primitive as an alternative to `synchronize_rcu`, splitting the write operations that must occur after the wait-for-current-readers barrier into a callback invoked by `call_rcu`.

Note that implementations of the wait-for-current-readers primitive based on RCU will wait for all current readers to complete before performing subsequent write operations. The alternative message-passing formulation outlined in section 8 instead performs subsequent write operations immediately; current readers ignore those writes, and new readers can see those writes immediately, without waiting for all other reader processes to complete.

6 Performance Results

Given a concrete implementation of our relativistic programming model, we wish to evaluate the scalability and performance of algorithms designed and implemented via this model. To this end, we implemented the reconstructed hash-table resize and red-black tree algorithms from sections 4.2 and 4.3, to compare them to other algorithms for the same data structures. (The linked-list algorithm from section

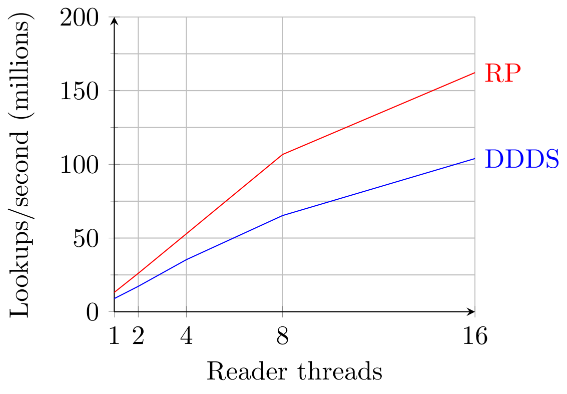

Figure 3: Lookups/second by number of reader threads for the relativistic hash-table resize algorithm versus the optimistic DDDS hash-table resize algorithm. The hash table size resized continuously between 8192 and 16384 buckets, in both cases containing a fixed 65536 entries.

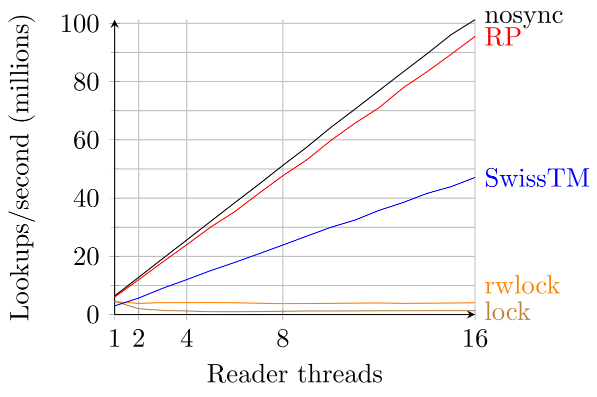

Figure 4: Lookups/second by number of reader threads in the absence of writers, for the relativistic red-black tree algorithm (RP), a synchronization-free baseline (nosync), locks, reader-writer locks, and SwissTM [10] transactional memory.

4.1 has already gone through extensive demonstrations of its scalability and performance via previous research on RCU-based linked lists.)

To evaluate the hash-table resize algorithm, we present results from the `rcuhashbash-resize` benchmark. [36] This benchmark uses the RCU implementation in the Linux kernel. It runs a configurable number of threads performing lookups on a hash table, along with a single thread continuously resizing that hash table. We compared the lookup performance of the relativistic resize implementation to the two other implementations available in `rcuhashbash-resize`: a baseline implementation based on reader-writer locking, and Nick Piggin's *dynamic dynamic data structures* (DDDS) [27], the current best-known method available for the Linux kernel. (DDDS uses an optimistic approach, copying from a primary to a secondary hash table, and forcing lookups to check the secondary table if the primary lookup fails due to a conflict with a writer.)

Figure 3 shows the results. (The graph omits the results from the implementation using reader-writer locks due to limits of scale; reader-writer locking maxed out at under 1.3 million lookups/second on 16 CPUs, a factor of 126 slower than the relativistic implementation.) In the presence of concurrent resizes, the relativistic resize algorithm achieves far better scalability than the implementation based on reader-writer locking. Furthermore, though DDDS scales reasonably well, the relativistic algorithm provides better performance.

Figures 4 and 5 present performance results for the red-black tree implementation. Figure 4 presents

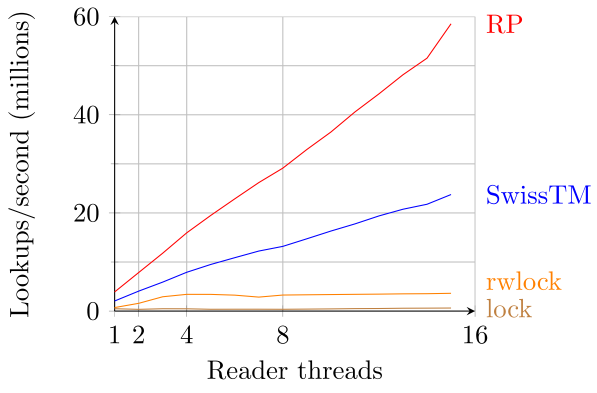

Figure 5: Lookups/second for 1–15 reader threads in the presence of a writer thread, for the relativistic red-black tree algorithm (RP), locks, reader-writer locks, and SwissTM [10] transactional memory.

lookup performance for 1–16 reader threads in the absence of a writer. Figure 5 presents lookup performance for 1–15 reader threads in the presence of a writer thread.

With and without a concurrent writer, the relativistic red-black trees produce significantly better results than an implementation based on SwissTM [10] transactional memory, and both surpass the complete lack of scalability exhibited by the locking techniques. In the no-writer case, we present a variation with no synchronization at all as a theoretical limit; the relativistic implemenation remains very close to this bound even for 16 CPUs.

7 Related Work

An early use of writer synchronization to avoid read barriers comes from work by McKenney, introducing a stronger write memory barrier that forced a barrier on all processors via inter-processor interrupts [20]; this avoids the overhead of a read barrier in the common case, with no writers running. McKenney later made specific use of the RCU wait-for-current-readers operation to implement a more efficient write barrier with batching, as part of the implementation of Sleepable Read-Copy Update (SRCU) [21].

Much transactional memory research has investigated *invisible reads*, a technique allowing readers to maintain consistency while not incurring the overhead of maintaining globally visible transaction metadata. The relativistic programming model inherently allows invisible reads, and readers incur minimal overhead. Relativistic programming provides a transaction-like consistency model for readers, though readers may run between ordered writes rather than only between entire write transactions.

TxLinux [29] demonstrated a transactional memory system integrated in the Linux kernel. Their work explored interactions between transactional memory and other forms of synchronization, including mutual exclusion and RCU. TxOS [28] presented a similar case study for a transactional system-call interface between the Linux kernel and userspace, the implementation of which built heavily on RCU.

The relativistic programming model we have documented focuses on improving scalability for readers by allowing joint-access parallelism for readers and a single writer at a time. Alternative approaches exist to allow joint-access parallelism for multiple writers, based on data partitioning and replication. These approaches require readers to perform extra steps, but allow writers to run concurrently using only local replicas.

Shapiro, Preguiça, Baquero, and Zawirski [32] proposed an approach for scalable replicated data types based on commutative operations. Their methodology supports joint-access parallelism for writers, reconciling replicas by relying on the commutativity of operations to guarantee an identical result for a set of operations regardless of ordering. The data structures they propose would also prove amenable to our current relativistic programming model, and an adaptation of their methodology could potentially add support for concurrent writers.

Our construction technique for relativistic data structures requires readers to perform only acyclic traversals of data structures. This requirement prohibits a small but significant set of algorithms, most notably those involving general graph traversal; such algorithms also prove exceedingly difficult for fine-grained locking. Some research exists suggesting approaches for the use of Read-Copy Update on cyclic data structures, including graphs, with additional reader overhead required to maintain consistency. [22] Adaptation of this research to relativistic programming could extend the relativistic ordering property to cyclic data structures.

"Laws of Order" [3] 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 cannot affect the results of a write, regardless of ordering. This allows relativistic readers to avoid expensive synchronization without contradicting the results shown in "Laws of Order".

8 Future Work

As previously discussed, the relativistic programming model we have presented focuses primarily on reducing the cost of readers and allowing them to run concurrently. This model pushes expensive synchronization into the writer to avoid placing any in the reader. Many workloads depend most heavily on read performance, but write-centric workloads exist as well, and will not necessarily perform optimally with this approach to relativistic programming. Write-centric alternatives based on partitioning do not currently have a well-defined ordering property analogous to that presented in section 2. Furthermore, no generalized construction technique exists to support building the write and read algorithms. Our future work will examine write-centric relativistic programming with the same rigor we have applied to read-centric approaches.

The implementation in section 5 mapped the barrier primitives we specified for relativistic programming to typical memory barriers available on shared-memory processor hardware, as abstracted by systems such as the Linux kernel. We plan to perform more extensive analysis of the interactions between the relativistic programming semantics and the semantics of specific processor hardware, such as those specified by the x86-TSO model [31].

The relativistic programming implementation documented in section 5 uses shared memory. We also have a preliminary algorithm for relativistic programming based on distributed message-passing. This approach removes the need for the writer to wait for all readers to finish before performing a write that existing readers should not see yet. Instead, writers may perform writes immediately, and supply messages which inform current readers to ignore new writes to a data structure until they complete their current read. As a result, rather than waiting for all current readers to finish before allowing any reader to observe the write, this approach allows each new reader to see the write as soon as possible. Future research will explore this alternative implementation thoroughly, and compare it to other systems based on message-passing.

9 Conclusion

Concurrent programming faces a harsh tradeoff between scalability and ease of implementation, exemplified by the wide range of consistency models for shared data. Even the most scalable models struggle to go beyond disjoint-access parallelism. We presented relativistic programming, a model for concurrent programming based on shared-addressing, which does not require message-passing or coherent shared memory. Our model supports joint-access parallelism, allowing readers to run concurrently with a writer on the same data.

We defined the key ordering property for this model, which provides a transactional behavior for reads, allowing them to appear atomic with respect to concurrent write operations. We presented a generalized construction technique for relativistic data structure algorithms, and demonstrated its application and performance. We have shown that the relativistic programming model provides excellent scalability, while maintaining an easier programming model. We believe these results will allow more widespread adoption of relativistic programming techniques as a means for building scalable data structures, and support future research on concurrent programming models which allow both ease of programming and scalability.

References

[1] Adve, S. V., and Boehm, H.-J. Memory models: a case for rethinking parallel languages and hardware. *Communications of the ACM 53* (August 2010), 90–101.

[2] Ahamad, M., Neiger, G., Burns, J., Kohli, P., and Hutto, P. Causal memory: definitions, implementation, and programming. *Distributed Computing 9* (1995), 37–49.

[3] 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 38th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (2011), POPL 2011.

[4] Batty, M., Owens, S., Sarkar, S., Sewell, P., and Weber, T. Mathematizing C++ concurrency. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (2011), POPL 2011, pp. 55–66.

[5] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles* (2009), SOSP 2009, pp. 29–44.

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

[7] Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., and Zhang, Z. Corey: An operating system for many cores. In *8th USENIX Symposium on Operating System Design and Implementation* (2008), OSDI 2008, pp. 1–16.

[8] Desnoyers, M. *Low-Impact Operating System Tracing.* PhD thesis, École Polytechnique Montréal, 2009.

[9] Digital Equipment Corporation. Shared Memory, Threads, Interprocess Communication. `http://h71000.www7.hp.com/wizard/wiz_2637.html`, August 2001.

[10] Dragojevic, A., Felber, P., Gramoli, V., and Guerraoui, R. Why STM can be more than a research toy. *Communications of the ACM* (2011).

[11] Hart, T. E., McKenney, P. E., Brown, A. D., and Walpole, J. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing 67*, 12 (2007), 1270–1285.

[12] Herlihy, M., and Shavit, N. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.

[13] Herlihy, M., Shavit, N., and Waarts, O. Linearizable counting networks. *Distributed Computing 9* (1996), 193–203.

[14] Howard, P. W., and Walpole, J. Relativistic red-black trees. Tech. Rep. 10-06, Portland State University, 2010.

[15] Howard, P. W., and Walpole, J. A relativistic enhancement to software transactional memory. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar 2011)* (2011).

[16] Lamport, L. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM 17* (August 1974), 453–455.

[17] Lipton, R. J., and Sandberg, J. S. PRAM: A scalable shared memory. Tech. Rep. 180-88, Princeton University, 1988.

[18] 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.

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

[20] McKenney, P. E. Software implementation of synchronous memory barriers. US Patent 6996812, February 2006.

[21] Mckenney, P. E. Sleepable read-copy update. Linux Weekly News. `http://lwn.net/Articles/202847/`, 2008.

[22] McKenney, P. E. Efficient support of consistent cyclic search with read-copy-update. US Patent 7814082, October 2010.

[23] McKenney, P. E. RCU Linux usage. `http://www.rdrop.com/users/paulmck/RCU/linuxusage.html`, March 2011.

[24] McKenney, P. E., Sarma, D., and Soni, M. Scaling dcache with RCU. *Linux Journal 1*, 118 (January 2004), 38–46.

[25] 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.

[26] Olsson, R., and Nilsson, S. TRASH: A dynamic LC-trie and hash data structure. In *Workshop on High Performance Switching and Routing* (May 2007), HPSR 2007.

[27] Piggin, N. ddds: "dynamic dynamic data structure" algorithm, for adaptive dcache hash table sizing. Linux kernel mailing list. `http://news.gmane.org/find-root.php?message_id=<20081007064834.GA5959@wotan.suse.de>`, October 2008.

[28] Porter, D. E., Hofmann, O. S., Rossbach, C. J., Benn, A., and Witchel, E. Operating system transactions. In *Proceedings of the twenty-second ACM SIGOPS Symposium on Operating Systems Principles* (2009), SOSP 2009, pp. 161–176.

[29] Rossbach, C. J., Hofmann, O. S., Porter, D. E., Ramadan, H. E., Aditya, B., and Witchel, E. TxLinux: using and managing hardware transactional memory in an operating system. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles* (2007), SOSP 2007, pp. 87–102.

[30] Saltzer, J. H., Reed, D. P., and Clark, D. D. End-to-end arguments in system design. *ACM Transactions on Computer Systems 2* (November 1984), 277–288.

[31] Sewell, P., Sarkar, S., Owens, S., Nardelli, F. Z., and Myreen, M. O. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM 53* (July 2010), 89–97.

[32] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. A comprehensive study of convergent and commutative replicated data types. Tech. Rep. RR-7506, INRIA, January 2011.

[33] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the fifteenth ACM Symposium on Operating Systems Principles* (1995), SOSP 1995, pp. 172–182.

[34] Torvalds, L. Linux 2.6.38-rc1. Linux Weekly News. `https://lwn.net/Articles/423623/`, January 2011.

[35] Triplett, J., McKenney, P. E., and Walpole, J. Scalable concurrent hash tables via relativistic programming. *ACM Operating Systems Review 44*, 3 (July 2010).

[36] Triplett, J., McKenney, P. E., and Walpole, J. Resizable, scalable, concurrent hash tables. In *USENIX Annual Technical Conference* (2011).

[37] Vogels, W. Eventually consistent. *Communications of the ACM 52* (January 2009), 40–44.

[38] Wentzlaff, D., and Agarwal, A. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Operating Systems Review 43* (April 2009), 76–85.

[39] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (1987), SOSP 1987, pp. 63–76.