Scalable Concurrent Hash Tables via Relativistic Programming

Josh Triplett[1], Paul E. McKenney[2], and Jonathan Walpole[3]

[1] Portland State University
josh@joshtriplett.org
[2] IBM
paulmck@linux.vnet.ibm.com
[3] Portland State University
walpole@cs.pdx.edu

Abstract. Existing approaches to concurrent programming often fail to account for synchronization costs on modern shared-memory multiprocessor architectures. A new approach to concurrent programming, known as relativistic programming, can reduce or in some cases eliminate synchronization overhead on such architectures. This approach avoids the costs of inter-processor communication and memory access by permitting processors to operate from a relativistic view of memory provided by their own caches, rather than from an absolute reference frame of memory as seen by all processors. This research shows how relativistic programming techniques can provide the perceived advantages of optimistic synchronization without the useless parallelism caused by rollbacks and retries.

Descriptions of several implementations of a concurrent hash table illustrate the differences between traditional and relativistic approaches to concurrent programming. An analysis of the fundamental performance bottlenecks in existing concurrent programming techniques, both optimistic and pessimistic, directly motivates the key ideas of relativistic programming. Relativistic techniques provide performance and scalability advantages over traditional synchronization, demonstrated through benchmarks of concurrent hash tables implemented in the Linux kernel. The demonstrated relativistic hash table makes use of a novel relativistic hash table move operation. The paper concludes with a discussion of how the specific techniques used in the concurrent hash table implementation generalize to other data structures and algorithms.

1 Introduction

Moore's Law predicts that the density of transistors in an integrated circuit will double every two years. [1] Popular interpretations of Moore's Law take it as a statement on processor clock frequencies, which have previously experienced exponential growth. However, clock frequencies have reached physical limitations that make this exponential growth unsustainable.

The speed of light provides a hard limit on how far information can travel per unit time. In a 3 GHz processor, information can travel no more than about 10 cm per clock cycle, and this does not take into account delays caused by actual computation. A faster processor must therefore occupy a smaller space.

Furthermore, higher clock frequencies consume disproportionately more power, a problem known as the *Power Wall*. This increase in power consumption also increases heat production. Miniaturization further exacerbates this problem by making it more difficult to dissipate heat. Miniaturization faces its own set of limitations as well, such as quantum tunneling effects and the sizes of atoms.

Yet Moore's Law continues to hold: the growth in the number of transistors continues apace. Many of those transistors now implement hardware parallelism. [2] Current processors often ship with more than one processing core on a single processor die, known as a *multicore* processor. As the state of the art advances, processors will most likely increase the number of cores per die while holding clock frequency and eventually transistor density constant.

The same physical limitations that apply to clock frequency also define the characteristics of present and future parallel systems. The communication latency between processors grows due to increased distance. Processors continue to consume data faster than memory and busses can supply it, a problem known as the *Processor-Memory Gap* [3] or the *Memory Wall* [4]. To prolong the growth of individual processor performance, processor designers have provided many innovative ways to hide these costs, such as caching, speculative and out-of-order execution, and pipelining. However, operations that synchronize across multiple processors incur the true cost of the increasing communication latency, and thus run much more slowly than operations that can run locally on a single processor.

Sequential software does not automatically take advantage of hardware parallelism as it does of higher clock speeds. To take advantage of hardware parallelism, operating systems and application software must provide multiple threads of execution, and must efficiently share resources amongst these threads. Both of these goals require scalable concurrent programming techniques.

Scalable refers to code which can do more work in the same amount of time by taking advantage of larger numbers of threads and processor cores; ideal concurrent code will scale linearly with the number of processors. *Concurrent programming techniques* allow multiple concurrent threads to coordinate their accesses to shared resources, such as data structures. *Scalable concurrent programming techniques* coordinate shared resources across many threads running on many processors, minimizing the amount of overhead above the time required for accessing the shared resource itself.

This work presents a new approach to scalable concurrent programming, dubbed *relativistic programming*. It also presents a new algorithm for a hash table move operation, which demonstrates the methodology and advantages of the relativistic programming approach.

This work describes concurrent programming in terms of threads. However, the same principles apply (with some adaptation) to other forms of concurrent code running with shared memory, such as processes with limited amounts of

shared memory, interrupt handlers which access the same memory as the code they interrupt, coroutines, or thunks in a lazy functional language.

This work illustrates synchronization techniques by applying them to the specific shared resource of a concurrent hash table. Hash tables provide one of the standard data structures applied to many problems in many programs, and form part of the standard computer science toolbox. [5, 6] They provide attractive performance characteristics, with an ideal best-case of constant time for operations on a well-tuned hash table. Many concurrent programs use hash tables, along with some means to manage concurrent accesses and provide necessary semantics. In particular, operating system kernels make use of concurrent hash tables for many performance-critical data structures, such as caches, network connection tables, and routing tables.

Section 2 defines the specific semantics required for the hash table implementations analyzed in this paper. Section 3 provides an overview and comparison of existing concurrent programming techniques and the performance problems associated with each. Section 3.4 introduces the Relativistic Programming approach to concurrent programming, including the general methodology and existing examples. Section 4 introduces a new hash table implementation which forms one of the original contributions of this research. Section 5 presents the benchmarking methodology and the test harness module. Section 6 presents the results of this benchmark. Section 7 analyzes the results and draws conclusions. Section 8 describes future directions for research.

2 Hash Table Semantics

An accurate definition of concurrent programming techniques in terms of hash tables requires an accurate definition of a hash table, along with a set of operations and the semantics of those operations.

Consider a standard open hash table, using chaining within buckets. The hash table consists of an array of buckets, each containing a pointer to the head of the linked list for that bucket. Each bucket contains zero or more items in its linked list chain. An item present in the hash table will exist in the bucket corresponding to its hash value. [5, 6]

A hash table can support many different operations, and any given application may need some subset of these. Common hash table operations include insertion, deletion, replacement, resizing, lookup, and moving an item to a new key. This work will focus on two of those operations: lookup and move. Lookup provides the only read-only operation, and thus a comparison of concurrent programming techniques that differentiate readers and writers must use the lookup operation in the readers.

Of the write operations, the move operation seems particularly worthy of interest. For a chained hash table, the operations of insertion and deletion reduce to the equivalent operations on a linked list, and the obvious implementations of these operations do not involve problematic intermediate states.[7] The obvious implementations of a move operation, as insertion then removal or vice versa,

involve multiple steps with intermediate states that a reader could observe. This paper showcases an original contribution in the form of a relativistic hash table move operation, which avoids exposing the problematic intermediate states while avoiding expensive synchronization techniques in the reader.

The lookup operation will check if an item exists in the hash table with the specified key. If so, the lookup will return it; if not, the lookup will indicate that no such item exists in the hash table. To allow for concurrent programming techniques that permit concurrent modification and deletion, a reader using the lookup operation may only use the returned item while in a block appropriately delimited using read-side concurrency primitives, and may not hold an item for later use. Furthermore, the lookup operation will not guarantee that a subsequent lookup will return the same results, even in the same read-side synchronization block.

The following pseudocode implements a sequential lookup:

1. Hash the given key to determine the corresponding hash bucket.
2. Traverse the linked list in that hash bucket, comparing the given key to the key in each node.
3. If a node has the given key, return that node.
4. If the traversal reaches the end of the list, return a lookup failure.

The move operation changes the key associated with an item, and moves the item to the hash bucket corresponding to the new key. The move operation guarantees a certain degree of atomicity with respect to concurrent lookups, by satisfying three requirements:

- If a lookup finds the item under the new key, a subsequent lookup ordered after the first cannot find the item under the old key.
- If a lookup does not find the item under the old key, a subsequent lookup ordered after the first must find the item under the new key.
- A move operation must not cause unrelated lookups to fail when they otherwise would have succeeded.

"Subsequent lookup ordered after the first" here means either a lookup running on the same processor but later in program order, or a lookup ordered after the first via a memory barrier.

The first two requirements originally arose through reasoning about the use of concurrent hash tables for directory entry lookups in an operating system kernel, and the observable effects this would have for userspace programs. The first requirement guarantees that during a move operation, a concurrent directory listing cannot show both the old and the new files simultaneously. The second requirement guarantees that the concurrent directory listing will always show either the old or the new file.

The following pseudocode implements a sequential move operation:

1. Hash the old key and the new key to determine the corresponding hash buckets.
2. Look up the node in the old bucket

3. Remove the node from the old bucket
4. Change the node's key to the new key
5. Insert the node into the new bucket

3 Concurrent Programming Techniques

As shown in section 2, a single-threaded hash table implementation can trivially provide the desired operations and semantics. The resulting hash table, if used without modification as a shared data structure in a concurrent program, would at best fail to provide the required semantics, and at worst encounter a error when attempting to dereference an invalid pointer. For example, unrelated lookups may fail if a concurrent move operation runs on the current element in a lookup traversal, because the lookup may traverse the wrong hash bucket. Furthermore, due to lack of memory ordering, a lookup or move may see an inserted item before its pointers or data become valid.

Several techniques exist for making a data structure safe for concurrent access; however, all of these techniques themselves represent sources of overhead compared to the single-threaded implementation. Before a concurrent hash table implementation can demonstrate a performance benefit, it must first overcome the overhead added by its concurrent programming techniques.

3.1 Mutual Exclusion

Mutual exclusion represents the most commonly applied technique for concurrent programming. Dijkstra's *semaphore* [8] and Hoare's *monitor* [9] provide the archetypal examples of the *locking* form of mutual exclusion. Several approaches exist for mutual exclusion, many of them dependent on features of the underlying hardware and instruction set, but all achieve the same effect: they provide a *lock* which many threads may simultaneously attempt to acquire, one of which will succeed and the rest of which will wait. The term *critical section* refers to the section of code between lock acquisition and lock release, during which no other code using the same lock can run.

Concurrent algorithms using mutual exclusion can safely violate data structure semantics in a critical section as long as they restore valid semantics before ending the critical section. Thus, the sequential lookup and move algorithms presented in section 2 will work in a concurrent program if wrapped in appropriate critical sections.

A global lock covering all accesses of the hash table provides a simple approach to make the hash table work correctly. However, this solution provides no concurrency at all: at most one hash table operation can occur at a time. Depending on the amount of work the program has to do other than manipulate the hash table, the concurrent program using a global lock may still run faster, but all the hash table operations will run serially; this puts a hard minimum on the best-case execution time of the concurrent program based on the number of hash table operations in the program. Adding more processors will not

shorten that minimum execution time. This provides an example of one common performance problem in concurrent programming: lock contention.

Several approaches exist to reduce lock contention. Reader–writer locks provide one such approach, without relying on any properties specific to hash tables. A *reader–writer lock* (sometimes written as *readers–writer lock*) provides two different sets of locking primitives: one for use by writers, which may modify the data structure, and one for use by readers, which may only inspect the data structure. A writer blocks both readers and other writers, making it the only critical section running, as with mutual exclusion locking. However, since readers do not affect the data structure, more than one reader may run at once; readers need only block writers. Mellor-Crummey and Scott [10] offer many forms of reader–writer synchronization.

However, while a reader–writer lock allows some degree of parallelism, it encounters another problem in addition to lock contention: critical section overhead. The critical section has a certain fixed time needed to implement the locking primitive, and at least part of this time does not parallelize, due to the need to coordinate among processors. Reader–writer locks have more state to track than plain locks, such as reader counts, and thus incur more overhead. Given a sufficiently short critical section body, the critical section overhead can dominate the execution time, nullifying the effect of the additional parallelism, and potentially making a reader–writer lock less efficient than a standard lock. The benchmark results in section 6 demonstrate this. (The global lock approach suffers from critical section overhead as well, but it has no parallelism to begin with.) Furthermore, a naive implementation of reader–writer locks allows a series of readers to indefinitely delay a writer; more advanced reader–writer locks that avoid this problem have higher critical section overhead.

McKenney [7] describes four additional sources of overhead in concurrent programs, other than lock contention: instruction execution overhead, pipeline stall overhead, memory latency, and contention for resources such as memory. Critical section overhead can arise from any combination of these sources. The synchronization instructions used to implement critical sections have high overhead compared to normal instructions, due to the high communication latency mentioned in section 1. [7, 11] These synchronization instructions can also lead to pipeline stalls. Synchronization instructions may decrease the effectiveness of caching, and may thus incur the cost of memory latency. Finally, critical section implementations tend to compete for the same resources, such as locks. Several approaches exist to mitigate the last effect [12, 13], but these approaches do not mitigate the remaining three sources.

A much larger reduction in contention comes from *data partitioning*. If a data structure consists of multiple substructures, and an algorithm need only access some subset of those substructures to work, the algorithm can acquire *fine-grained locks* on the substructures it needs, rather than locking the entire structure. In the case of a hash table, the global lock can become one lock per hash bucket. A lookup or move operation can then lock only the bucket or buckets it accesses. This approach allows multiple operations on different

buckets to proceed concurrently. Given a sufficiently large and evenly accessed hash table, this can almost entirely eliminate contention as a source of overhead. However, data partitioning and fine-grained locking still do not eliminate the other sources of critical section overhead.

3.2 Non-Blocking Synchronization

Mutual exclusion introduces several potential pitfalls which can lead to poor performance or incorrect behavior. These include deadlock, priority inversions in the scheduler, and relatively long windows of data-structure inconsistency. These problems occur due to blocking acquisition of locks. For these reasons, several researchers have proposed the use of *non-blocking synchronization* instead, which addresses all of these problems. [14–16]

In non-blocking synchronization, algorithms to modify a shared data structure begin by observing necessary values from the structure, such as a node pointer within a linked list. The algorithms prepare their modification separately, using these values; they then apply the changes using atomic instructions, such as compare-and-swap or load-linked and store-conditional, to ensure that the data structure has not changed between observation and modification.

While non-blocking synchronization eliminates many problems with mutual exclusion, it does not fully address scalability. Non-blocking synchronization does not have lock and unlock operations, but it has a corresponding set of operations with almost exactly the same effect on scalability. The read or load-linked instruction corresponds to the lock, and the compare-and-swap or store-conditional instruction corresponds to the unlock; the region between the two thus corresponds to the critical section. If multiple threads execute the critical section simultaneously, only one can succeed, and the others must retry. This effectively serializes executions of the critical section.

Non-blocking synchronization does have the advantage of providing similar scalability to fine-grained mutual exclusion without the particular complexity of lock ordering and deadlock avoidance. However, the use of non-blocking synchronization cannot improve scalability beyond that point. While non-blocking synchronization allows many threads to run in parallel, that parallelism may consist of threads which conflict with each other but have not yet rolled back. This gave rise to the term *useless parallelism*. Useless parallelism, in addition to taking up processor time on work that will not complete, can harm the performance of other threads by causing memory contention and cache misses.

3.3 Software Transactional Memory

Michael and Scott [17, 15], among others, have proposed various non-blocking algorithms. Unlike mutual exclusion, which supports a broad class of operations with a few primitives, each non-blocking algorithm tends to involve application-specific or data-structure-specific functionality, and these algorithms tend toward higher complexity for more intricate data structures. Herlihy [18] proposed

a generic approach to transform lock-based algorithms to use non-blocking synchronization, but this approach requires copying a full data structure to make any modification to it, and thus does not perform well. Achieving good performance requires a data-structure-specific non-blocking algorithm, and as Michael and Scott [17] describe, "Good data-structure-specific multi-lock and non-blocking algorithms are sufficiently tricky to devise that each has tended to constitute an individual publishable result."

For these reasons, Herlihy [19] proposed the abstraction of *software transactional memory*. This abstraction provides *transactions*—separate groupings of operations that must occur atomically. Since their proposal, much research on synchronization has focused on software transactional memory [20].

Fundamentally, software transactional memory works the same way that other forms of non-blocking synchronization do, by atomically checking for other modifications to the data structure before committing its own modifications. If other modifications have occurred, the transaction can roll back and retry. Unlike other forms of non-blocking synchronization, software transactional memory sometimes runs optimistically, modifying the data structure before the transaction has finished; thus, transactions must avoid changes that cannot easily roll back, and must avoid catastrophic failures even if another transaction temporarily violates data-structure invariants.

The abstraction of the transaction substantially reduces the complexity of non-blocking synchronization by removing the need for data-structure-specific synchronization, at the cost of performance. However, software transactional memory has no positive impact on scalability over non-blocking synchronization or mutual exclusion. As with other forms of non-blocking synchronization, software transactional memory still has an equivalent to lock and unlock operations that delimit a critical section (the transaction), and any parallelism between transactions does not help, as either the transactions reference independent memory (and thus fine-grained mutual exclusion would work at least as well), or all but one of the transactions must roll back. Furthermore, software transactional memory still requires expensive synchronization instructions, and its critical section involves significant overhead.

Proponents of transactional memory have proposed that hardware implementations can overcome the performance disadvantages of software transactional memory. Rossbach et al [21] presented an implementation of Linux based on hardware-assisted transactional memory with various enhancements for practical use. However, this work built upon a simulation of the hardware transaction mechanism, with various assumptions such as all instructions completing in a uniform duration of 1 cycle. Without an implementation designed for physical hardware, we cannot perform a meaningful comparison of hardware transactional memory on Linux, and thus we omit it from our benchmarks in section 5.

3.4 Relativistic Programming

Consider a reader and writer coordinating via a reader–writer lock. Assume the reader has already taken the read lock and started a lookup operation, when

the writer wishes to acquire the write lock. The reader–writer lock will block the writer until the reader finishes. The equivalent situation using non-blocking synchronization or transactional memory may follow the same pattern by rolling back the writer, or may allow the writer to run and then roll back the reader.

However, a third possibility exists: why not allow the writer to run without rolling back the reader? Furthermore, why not allow readers to proceed concurrently with a running writer? Writers can keep the data structure in a consistent state at all times, either by using atomic operations, or by copying (parts of) data structures to new versions and leaving the old versions undisturbed. With the structures perpetually consistent, readers can always safely proceed without waiting. Thus, readers need not use any form of critical section, and thus incur no critical section overhead.

This approach provides an example of a broader class of concurrent programming techniques and data structures, which share the common theme of allowing additional parallelism by permitting concurrent access to shared data without a critical section. By avoiding critical sections and minimizing or eliminating the use of expensive synchronization instructions, each processor can take full advantage of the technologies described in section 1 to hide memory latency. As a result, each processor may see a different view of memory as presented by its own cache and access order. For example, a thread may walk a linked list concurrently with a sequence of insertions, and observe a set of items which do not correspond to any state the list passed through as a result of those insertions: it may see items inserted later in time (from the perspective of the thread performing the insertions) without seeing items inserted earlier.

We refer to this property as *relativity*, by analogy with physics; the avoidance of these instructions allows processors to see a relative view of memory, rather than an absolute reference frame. Furthermore, actions taken by one processor may appear to other processors at different times. We refer to parallel code that has this property as *relativistic*, and to the concurrent programming techniques associated with it as *relativistic programming*.

Relativistic programming techniques provide the potential for greater parallelism than fine-grained mutual exclusion by allowing accesses to a shared data structure to run concurrently even when one or more of those accesses includes a modification. By extension, these techniques provide greater parallelism than either non-blocking synchronization or software transactional memory, since neither of those permits any greater parallelism than fine-grained mutual exclusion. Benchmarks of code implemented via relativistic programming provide some highly compelling scalability results. [22–24]

Relativistic approaches to concurrent programming share a common methodology. Modifications to a shared data structure consist of only one semantically significant operation at a time, which may become visible at any time, including immediately. Multiple modifications may become visible in different orders to different processors. Modifications which need to become visible in a particular order to satisfy the semantic requirement of a higher-level operation must use memory barriers to constrain the potential ordering of these modifications. How-

ever, rather than forcing the programmer to deal with the complexity of directly using memory barriers, relativistic programming techniques provide higher-level primitives which provide memory barriers as needed.

Several existing concurrent programming techniques make use of relativity. As a simple example, the common technique of splitting numeric variables across CPUs or threads can take advantage of relativity by accumulating these values without synchronization. In exchange for performance, this approach may accumulate snapshots of the values from slightly different times in each thread. Liskov hints at this approach in [25]: "Conflicts with deposits and withdrawals are necessary if the reported total is to be up to date. They could be avoided by having total return a sum that is slightly out of date."

More generally applicable relativistic techniques include those based on deferred destruction. Deferred destruction addresses one of the problems associated with concurrent modifications: how to free memory without disrupting a concurrent thread reading that memory. Deferred destruction allows a writer to wait until no readers hold references to the removed item before reclaiming and reusing its memory.

Several techniques exist for deferred destruction [11], including epoch-based reclamation [26], hazard-pointer-based reclamation [27], and quiescent-state-based reclamation [7, 22, 28]. Epoch-based reclamation divides execution into explicit epochs, and allows memory reclamation after an epoch has passed. Hazard-pointer-based reclamation requires readers to indicate their references explicitly as hazard pointers, and allows reclamation of any memory not pointed to by a hazard pointer. Quiescent-state-based reclamation notes the passage of quiescent states in which readers cannot run, and uses these quiescent states to wait until all existing readers have finished before reclaiming memory. Of those techniques, implementations of epoch-based reclamation and quiescent-state-based reclamation exist which do not make use of synchronization instructions.

Many common data structures have relativistic implementations which use deferred destruction. These include linked lists, radix trees, and tries. A relativistic hash table implementation exists [29, 30], but this implementation does not supply a relativistic move operation; instead, the move operation makes use of the Linux *sequence lock*, which provides a means for a reader to check whether it raced with a move operation and retry if so. While semantically correct, it still requires the use of synchronization instructions on the read side, and it can potentially delay a reader indefinitely.

4 Relativistic Hash Tables

As a first approximation, since hash chains consist of a linked list, a relativistic hash table could simply use relativistic linked lists as hash buckets. However, a simple combination of the linked list insert and delete operations in series, in any order, cannot satisfy the required move semantics; inserting first will violate the first semantic by allowing a reader to see both items, and deleting first will violate the second semantic by allowing a reader to see neither item.

This necessitates a new relativistic move operation specific to hash tables, such as the one presented here.

The new relativistic hash table move operation relies on two key behaviors of a hash table lookup.

First, after using the hash of the search key to find the appropriate bucket, a reader must compare the individual keys of the nodes in the list for that bucket to the actual search key. Thus, if a node shows up in a bucket to which its key does not hash, no harm befalls any reader who comes across that node while searching that bucket, apart from a marginal amount of extra time spent traversing the hash chain for that bucket.

Second, when traversing the list for a given hash bucket, a reader will stop when it encounters the first node matching the search key. If a node occurs twice in the same bucket, the search algorithm will simply return the first such node when searching for its key, or ignore both nodes if searching for a different key. Thus, multiple nodes with the same key can safely appear in a given hash bucket.

Both of the possible requirements violations, appearing in neither bucket or appearing in both buckets, occur when the writer does not simultaneously remove the node from the old bucket and add it to the new bucket with the new key. Most modern architectures do not feature memory-to-memory swaps or simultaneous writes to multiple locations, so the writer cannot simultaneously and atomically change more than one pointer or key.

However, if the writer can make the moving node appear in both buckets simultaneously, it can in one operation remove the node from the old bucket and add it to the new bucket, by atomically changing the key. Before the change, searches in the old bucket using the old key will find the node, and searches in the new bucket using the new key will always skip over it; after the change, searches in the old bucket with the old key will always skip over the node, and searches in the new bucket with the new key will find it. This satisfies both requirements for the move operation.

Because nodes can safely appear in buckets to which their keys do not hash, the writer can make the node appear in both buckets by cross-linking one hash chain to the other. The writer can then change the node's key to the new value, and must then un-cross-link the chains. However, when removing the cross-link, the writer must ensure that it does not disturb any writer currently traversing the old hash bucket, even if that reader currently references the node getting moved. The remainder of the algorithm consists of safely resolving the cross-linking via deferred destruction.

The lookup operation consists of a standard hash table lookup, except that it makes use of the appropriate primitives to support deferred destruction:

1. Hash the given key to determine the corresponding hash bucket.
2. Start deferring write-side destruction.
3. Traverse the linked list in that hash bucket, comparing the given key to the key in each node.
4. If a node has the given key, do the computation that required the node.
5. If the traversal reaches the end of the list, indicate a lookup failure.

6. Stop deferring write-side destruction.

Fig. 1. Initial hash table configuration used to illustrate move algorithm. $n_1.key$, $n_2.key$, and $n_3.key$ hash to a. $n_4.key$ and $n_5.key$ hash to b. The move operation will change $n_2.key$ from "old" to "new". "new" hashes to b.

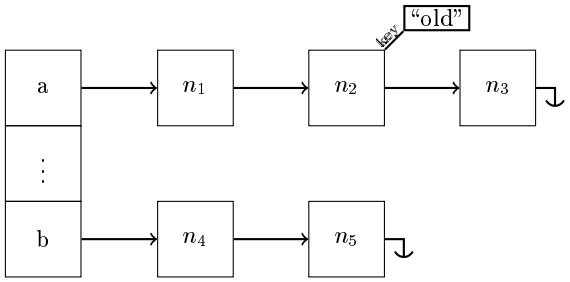


Figure 1 shows a sample configuration of a hash table, used to illustrate the move algorithm. The following steps illustrate the move algorithm on this hash table.

1. Perform the appropriate synchronization to modify hash buckets a and b. For instance, obtain the locks for hash buckets a and b, in hash bucket order to avoid deadlocks. Note that this step only exists to synchronize with other concurrent moves, not with lookups.
2. Make a copy of the target node n_2; call the copy n'_2.
3. Set $n'_2.next$ to NULL.
4. Execute a write memory barrier to ensure that the new value of $n'_2.next$ will become visible to other processors before n'_2 does.
5. Set $n_3.next$ to n'_2.
6. Execute a write memory barrier to ensure that n'_2 will become visible to other processors before n_2 disappears.
7. Remove n_2 from a by pointing $n_1.next$ to n_3. a now has the target node n'_2 at the end.
8. Point the tail of bucket b ($n_5.next$) to the new target node (n'_2). Both hash bucket chains now include n'_2. Figure 2 shows the state of the hash table after this step.
9. Execute a write memory barrier to ensure the removal of n_2 and the cross-linking will appear before $n'_2.key$ changes.
10. Atomically change $n'_2.key$ to "new".
11. Execute a write memory barrier to ensure that $n'_2.key$ will change before n'_2 disappears from bucket a.
12. Point $n_3.next$ to null, un-cross-linking the chains. Figure 3 shows the state of the hash table after this step.
13. Release the write-side synchronization for hash buckets a and b.

Fig. 2. State of the hash table after cross-linking hash chains in step 8 of the relativistic hash table move algorithm.

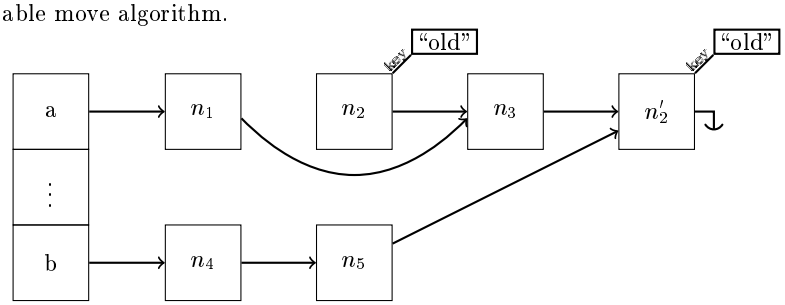
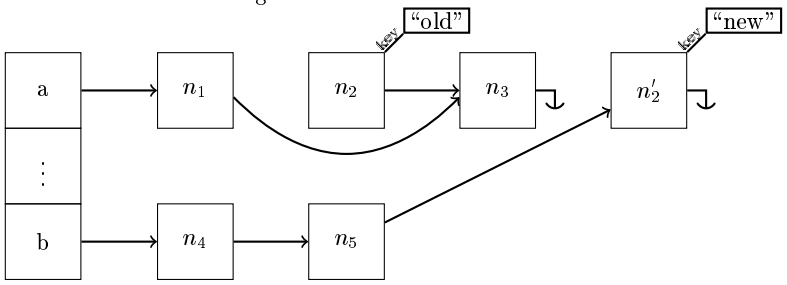


Fig. 3. State of the hash table after un-cross-linking hash chains in step 12 of the relativistic hash table move algorithm.



14. Use deferred destruction to remove the original n_2 and the old key "old" after all current readers have finished.

This operation meets the required move semantics. First, "If a lookup finds the item under the new key, a subsequent lookup ordered after the first cannot find the item under the old key." Suppose a reader finds the item under the new key. It must find n'_2, because $n_2.key$ never changes. The writer writes the new key in step 10, so the reader must observe the result of this step. To subsequently find an item under the old key, the reader must find n_2, because n'_2 no longer has the old key. To find n_2, the reader must not see the change to $n_1.next$ in step 7 removing it. However, the write memory barrier in step 9 ensures that a reader cannot see the result of step 10 and not step 7.

Second, "If a concurrent lookup does not find the item under the old key, a subsequent lookup ordered after the first must find the item under the new key." Suppose a reader does not find the item under the old key. It must not see n_2, and it must not see n'_2 before its key changes. Since it does not see n_2, it must see the result of step 7. Since it does not see n'_2, it must either see the result of step 10 or not see the result of step 5. Since the reader saw the result of step 7, the memory barrier in step 6 ensures that the reader must see the result of step 5, and therefore the reader must see the result of step 10. However, if the

reader sees the result of step 10, it will find n_2' with the new key on a subsequent lookup.

Finally, "A move operation must not cause unrelated lookups to fail when they otherwise would have succeeded." For a lookup to fail, a reader must fail to see an item that it otherwise would have seen. Placing n_2' at the end of buckets a and b, and removing it from bucket a, cannot cause a reader to miss an item, which leaves only the removal of n_2. This removal can only affect a reader traversing bucket a. The removal of n_2 does not free n_2 until existing readers exit, so a reader can only notice the change of $n_1.next$ to n_3. This change does not prevent a reader traversing bucket a from seeing the other items, n_1 and n_3. Thus, a reader will never fail to see an item it would otherwise have seen, so unrelated lookups will not fail.

Comparing these relativistic lookup and move algorithms to the locking algorithms described in section 3.1 suggests several likely performance differences to test via benchmark. The relativistic lookup algorithm involves no synchronization instructions, and it does not block at all, even when running concurrent moves. Thus, it should allow significantly more lookups per unit time than the lock-based lookup operation. The corresponding move algorithm performs four extra write memory barriers, a memory allocation, and a deferred destruction operation, as well as various additional non-synchronizing operations. This should result in fewer moves per unit time than the lock-based move operation.

However, since the relativistic move operation only uses locking to synchronize with other moves, a single writer thread can dispense with locking synchronization entirely. This should result in more moves per unit time than the single-writer case of either the locking move algorithm or the relativistic move algorithm with locking between writers.

Section 5 defines the benchmark methodology used to test these hypotheses. Section 6 presents the results of this benchmark.

5 Benchmark Methodology

Read-Copy Update (RCU) provides the most mature and popular concurrent programming framework that supports relativistic programming techniques. The Linux kernel contains several mature and widely used implementations of RCU, as well as implementations of all of the standard forms of mutual exclusion. Thus, a Linux kernel module provided the most practical and straightforward target for a benchmark.

I created the rcuhashbash benchmark module to benchmark relativistic and non-relativistic hash implementations. rcuhashbash consists of two main components: a set of concurrent hash table implementations implementing a defined hash table interface, and a test harness which runs the hash table operations and tracks statistics. For each hash table reader and writer implementation, the benchmark contains a structure with a set of pointers to functions implementing the hash table interface.

As mentioned in section 2, this comparison of concurrent hash table implementations will focus on two operations: a read-only lookup, and a move operation. Execution time represents the only relevant performance metric for these operations. Executing one of these operations takes very little execution time, so following common practice, the benchmark will execute these operations as quickly as possible over a longer time period and record the number of operations performed. Measuring several hash table implementations over equal-length periods of time will provide operation counts in the same proportion as the average execution times of those operations. To avoid extraneous synchronization, the accumulation of statistics occurs via per-thread counters summed up at the end of the benchmark run.

rcuhashbash begins by constructing a hash table of a specified size, and loading it with integer values from 0 to a specified maximum. The experiments in this paper used a hash table with 1024 buckets and 4096 entries. rcuhashbash then spawns a specified number of threads at startup. Each thread goes into a continuous loop, randomly choosing to lookup or move based on a specified reader/writer ratio. The move operation chooses an old key and a new key from the range of 0 to twice the maximum initial value ($[0, 8192)$ for this experiment), and attempts to move the item with the old key to the item with the new key. The lookup operation chooses a key from the same range and performs a lookup.

rcuhashbash includes hash table implementations based on each of the synchronization techniques available in the Linux kernel: whole-table spinlocks and reader–writer locks, per-bucket spinlocks and reader–writer locks, readers based on deferred destruction and sequence locking (retrying a lookup if it raced with a move), and readers based solely on deferred destruction with writers using the novel move algorithm proposed in section 4. The algorithms using deferred destruction make use of the Linux kernel's implementation of Read-Copy Update (RCU).

The machine used for testing has 16 IBM POWER6 physical processors at 4.7GHz, each with two cores of two logical threads each, for a total of 64 threads. On the software side, the machine ran the Linux 2.6.28 kernel, compiled for the 64-bit powerpc architecture, using the "classic" RCU implementation. To observe scalability, the benchmark ran each hash table implementation with 1, 2, 4, 8, 16, 32, and 64 threads. To obtain enough samples for statistical analysis, the benchmark ran each implementation 10 times, for 30 seconds each time. To observe the effect of a varying read to write ratio, each implementation ran with the read to write ratio set to 999999:1, 999:1, and 1:1.

In the absence of any systematic variation caused by system interference, each measurement of the same operation for the same hash table implementation should represent an independent sample from the same statistical distribution. Thus, the central limit theorem applies, and the measurements should approximate a normal distribution. [31]

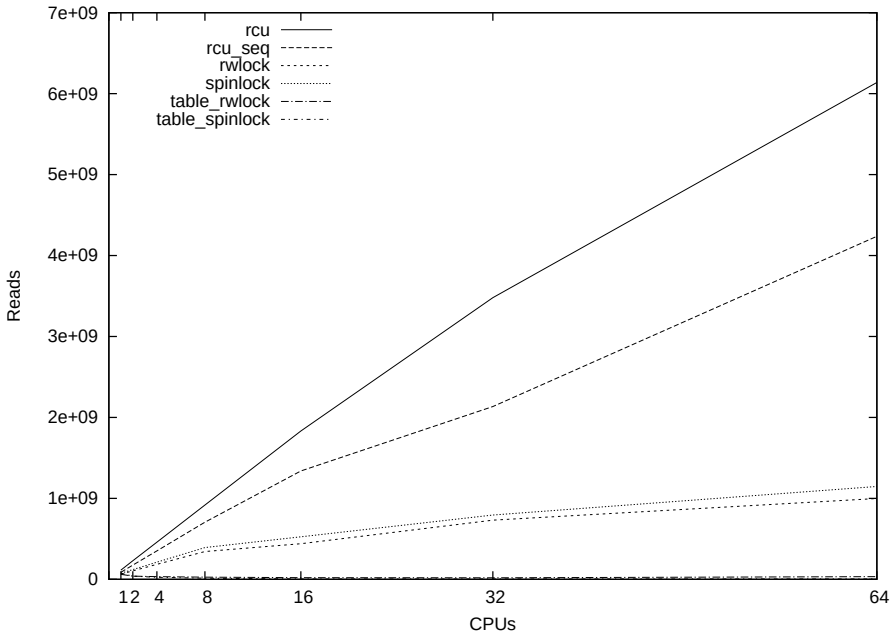


Fig. 4. Average lookups in 30 seconds by number of CPUs with 999999:1 read:write ratio

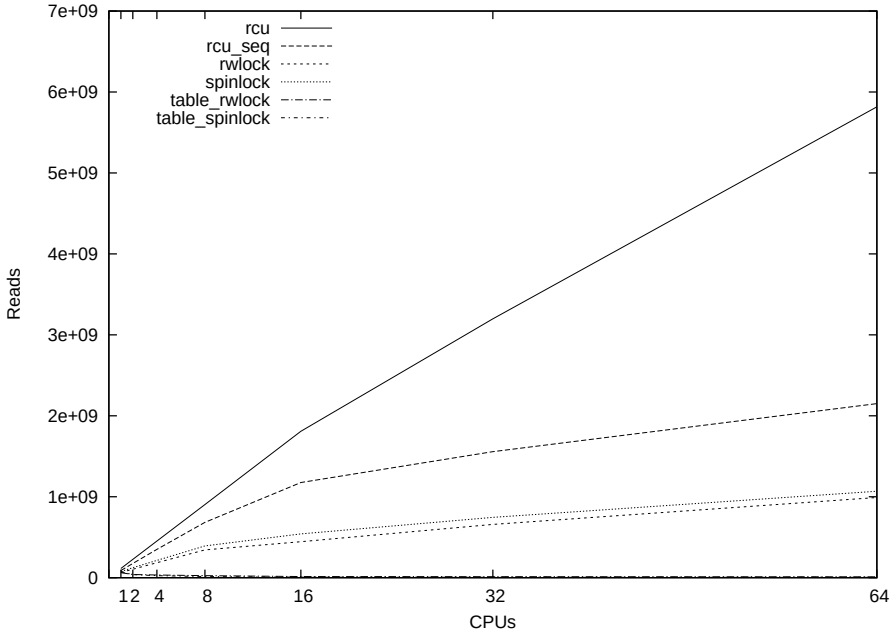


Fig. 5. Average lookups in 30 seconds by number of CPUs with 999:1 read:write ratio

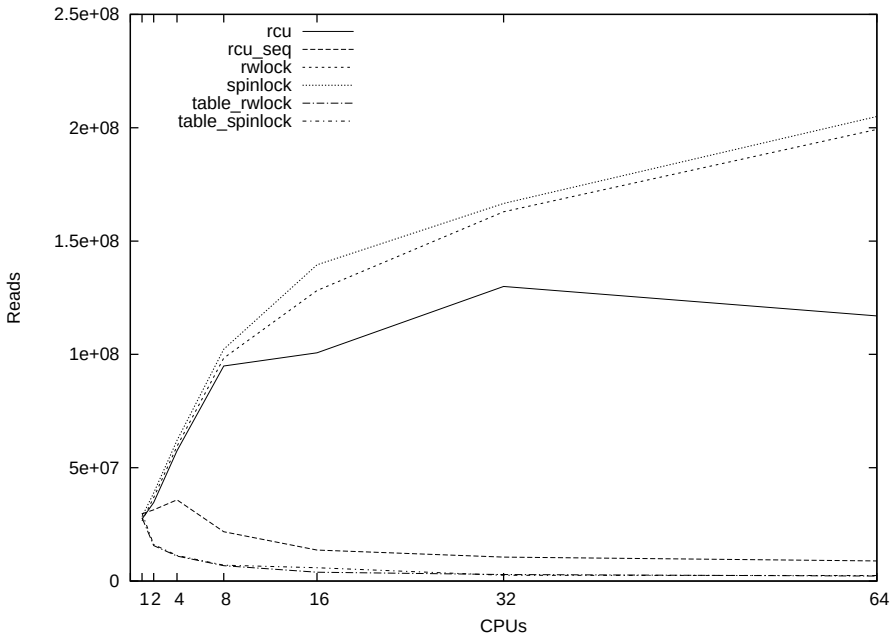


Fig. 6. Average lookups in 30 seconds by number of CPUs with 1:1 read:write ratio

6 Benchmark Results

Figures 4, 5, and 6 show the average number of lookups in 30 seconds for each hash table implementation as the number of CPUs used increases; the three figures depict the three decreasing read to write ratios.

The results show clear separation into groups. For the two read-biased workloads, the proposed hash table algorithm (labeled "rcu") proves the clear winner, scaling better than the Linux kernel's current approach based on sequence locks ("rcu_seq") by a significant margin. The algorithms based on per-bucket mutual exclusion ("spinlock" and "rwlock") follow at a distance with barely positive scalability, and the algorithms based on whole-table mutual exclusion ("table_spinlock" and "table_rwlock") scale so badly that they remain barely distinguishable from the axis.

At the lower 999:1 read to write ratio, the rcu_seq algorithm scales much worse for larger numbers of CPUs, likely due to retries or contention for the sequence lock; the proposed algorithm suffers only a minor scalability degradation with the decreased read to write ratio. With the balanced 1:1 read to write ratio, per-bucket mutual exclusion outperforms the deferred destruction approaches as expected; however, the proposed algorithm still scales far better than the sequence-lock-based algorithm used in the Linux kernel when used with the non-read-biased workload.

At all three read to write ratios, per-bucket spinlocks outperform per-bucket reader-writer locks, even for the full 64 threads, supporting the claim in 3.1 that the overhead of reader-writer locks can outweigh their additional parallelism.

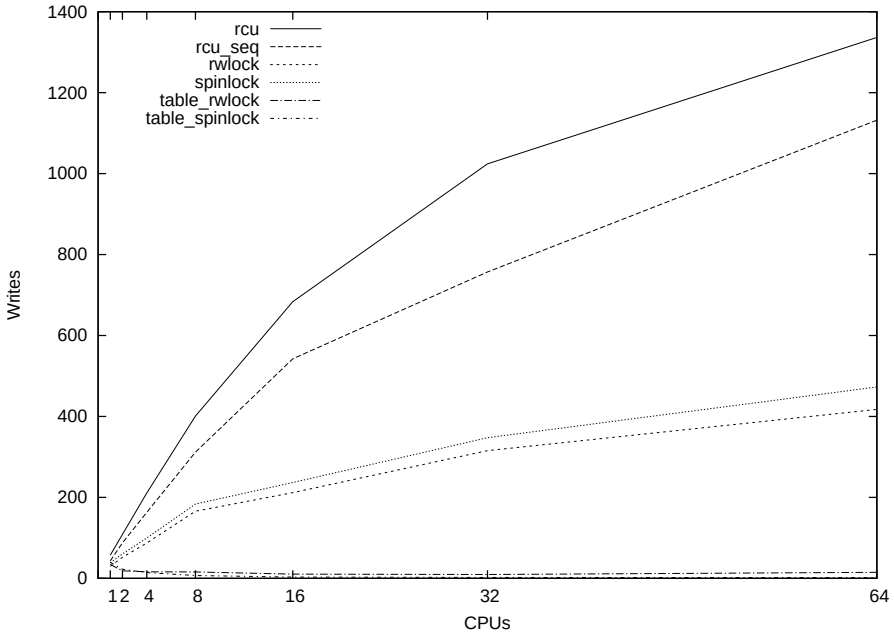


Fig. 7. Average moves in 30 seconds by number of CPUs with 999999:1 read:write ratio

Figures 7, 8, and 9 show the average number of moves in 30 seconds for each hash table implementation as the number of CPUs used increases; the three figures again depict the three decreasing read to write ratios.

Unexpectedly, for read-biased workloads, the deferred destruction approaches actually outperform per-bucket mutual exclusion for writes, despite their higher overhead. We speculate that the write side of these algorithms may benefit from decreased contention with readers. Again, the proposed algorithm significantly outperforms the sequence-lock-based algorithm, with the performance difference increasing at the less read-biased 999:1 read to write ratio. Per-bucket mutual exclusion follows at a distance, with spinlocks still outperforming reader–writer locks, and whole-table mutual exclusion remains at the bottom.

For the balanced 1:1 read to write ratio, per-bucket mutual exclusion takes a healthy lead, with spinlocks still winning over reader–writer locks. However, the proposed algorithm again manages to scale far better than the sequence-lock-based algorithm used in the Linux kernel when used with the non-read-biased workload.

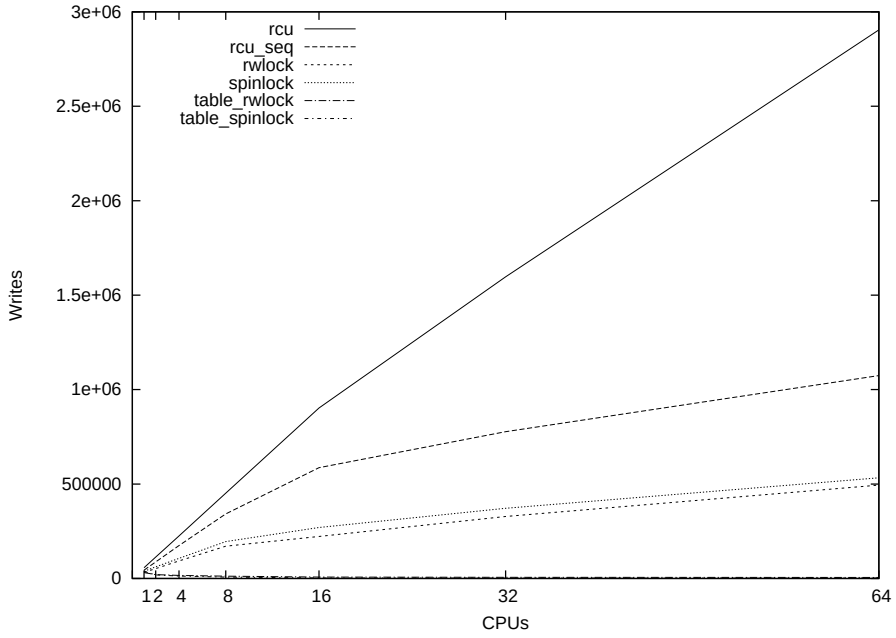


Fig. 8. Average moves in 30 seconds by number of CPUs with 999:1 read:write ratio

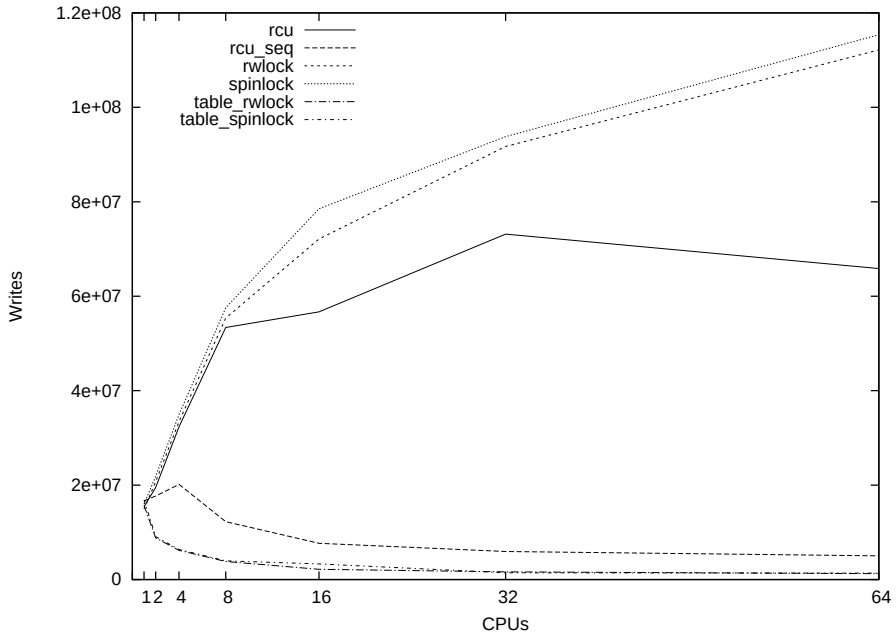


Fig. 9. Average moves in 30 seconds by number of CPUs with 1:1 read:write ratio

Thus, we conclude that the proposed hash table algorithm provides marked performance and scalability advantages compared to the current state of the art used in the Linux kernel. It proves the clear winner for read-biased workloads, and degrades gracefully for balanced workloads.

7 Conclusions

This work presented a new class of techniques for scalable concurrent programming, dubbed *relativistic programming*. Relativistic programming techniques share a common methodology: to avoid the use of expensive synchronization instructions and allow processors to see a relative view of memory, rather than an absolute reference frame. Modifications to a relativistic data structure consist of only one semantically significant operation at a time, which may become visible at any time, including immediately; such modifications must therefore keep the data structure in a consistent state at all times.

Section 2 introduced the hash table structure used to illustrate this new methodology, including the specific lookup and move operations considered, and the required semantics of those operations. Section 3 surveyed existing concurrent programming techniques, and compared the limitations and performance bottlenecks of these techniques, motivating the introduction of relativistic programming in section 3.4. Section 4 provided a relativistic algorithm for implementing the hash table in section 2, including an original move algorithm, along with an informal argument for the correctness of this algorithm. Section 5 presented a new benchmark, rcuhashbash, designed to compare relativistic and non-relativistic implementations of concurrent hash tables. Section 6 documents the results of this benchmark.

The results in section 6 show that the proposed relativistic hash table implementation provides marked performance and scalability advantages compared to the current state of the art used in the Linux kernel, proving the clear winner for read-biased workloads and a strong contender even for balanced workloads.

8 Future Directions

The performance of the relativistic hash table shows promise for the relativistic programming approach to scalable concurrency. Further research will expand the breadth of relativistic programming into new areas.

As mentioned in section 3.3, we would like to include some performance comparisons with transactional memory, perhaps with an implementation designed for hardware assistance. The limitations of existing implementations prevent us from including such a comparison at this time, but we plan to do so in the future. Furthermore, we believe some hybrid of relativistic techniques and transactional memory could potentially provide some of the performance and scalability advantages of relativistic programming while retaining some of the simplicity of the transactional memory programming model.

Several uses of relativistic hash tables in operating system kernels such as Linux would benefit from the ability to resize such a hash table while allowing concurrent reads. Given the typical approach of a hash function with a larger output range than the size of the hash table, shrinking a hash table simply requires coalescing buckets by concatenating linked lists, which does not require reader synchronization. A variation on the move operation proposed in this paper may support growing a hash table dynamically without reader synchronization.

The proposed move algorithm entails allocating a copy of the target node and freeing the original once readers no longer reference it. This approach does not permit readers to retain references beyond their delimited read-side blocks, even if using techniques such as reference counting. Some uses of concurrent hash tables do rely on retaining references, and providing this capability would avoid the need to rewrite these uses to remove this assumption, making the proposed algorithm more widely applicable.

The existing implementations of relativistic programming via deferred destruction focus primarily on read-mostly data structures. Many applications would benefit from data structures optimized for more balanced workloads, or for write-mostly data such as logs.

Other data structures could potentially benefit from relativistic implementations, including closed hash tables, heaps, priority heaps, balanced trees, B-trees, skip lists, and Judy arrays. Furthermore, the requirement to invent a novel algorithm for each new data structure limits the adoption of relativistic techniques other than those implemented in existing code. Relativistic programming would greatly benefit from a set of more general building blocks which make the implementation of data structures a largely mechanical task.

Section 4 provided an informal argument for the correctness of the proposed hash table algorithm with respect to the specified semantics. A formal proof would give a higher degree of confidence in the correctness of the algorithm. Furthermore, future formal proofs of relativistic algorithms may benefit from a library of relativistic programming constructs for use with an automated proof engine.

Relativistic programming imposes several constraints on the correct implementation of readers and writers. Extending static analysis tools to check these constraints would ease the task of implementing relativistic algorithms.

9 Acknowledgments

Thanks to Ray Harney, Darren Hart, Scott Nelson, and the IBM Linux Technology Center for the opportunities to build multiple internships around RCU and Linux. Thanks to IBM for access to the 64-way system used for benchmarking. Thanks to Phil Howard for discussions leading to the realization that typical reader-writer locks permit stale data. Thanks to Jamey Sharp for review, feedback, and advice on drafts of this paper.

Funding for this research provided by a Maseeh Graduate Fellowship and by the National Science Foundation under Grant No. CNS-0719851.

References

1. Moore, G.E.: Cramming more components onto integrated circuits. Electronics **38**(8) (April 1965)
2. Sutter, H.: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal **30**(3) (March 2005)
3. Hennessy, J.L., Patterson, D.A.: Computer Architecture: a Quantitative Approach. Fourth edn. Morgan Kaufmann (2007)
4. Wulf, W.A., McKee, S.A.: Hitting the memory wall: Implications of the obvious. Computer Architecture News **23**(1) (March 1995) 20–24
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Chapter 11: Hash Tables. In: Introduction to Algorithms. Second edn. MIT Press (2001)
6. Knuth, D.: Section 6.4: Hashing. In: The Art of Computer Programming. Second edn. Addison-Wesley (1998)
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. Dijkstra, E.W.: The structure of the "THE"-multiprogramming system. Communications of the ACM **11**(5) (1968) 341–346
9. Hoare, C.A.R.: Monitors: an operating system structuring concept. Communications of the ACM **17**(10) (1974) 549–557
10. Mellor-Crummey, J.M., Scott, M.L.: Scalable reader-writer synchronization for shared-memory multiprocessors. In: Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM Press (April 1991) 106–113
11. Hart, T.E., McKenney, P.E., Brown, A.D., Walpole, J.: Performance of memory reclamation for lockless synchronization. Journal of Parallel and Distributed Computing **67**(12) (2007) 1270–1285
12. Anderson, T.E.: The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Transactions on Parallel and Distributed Systems **1**(1) (1990) 6–16
13. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems **9**(1) (1991) 21–65
14. Greenwald, M., Cheriton, D.: The synergy between non-blocking synchronization and operating system structure. In: Second Symposium on Operating Systems Design and Implementation, USENIX Association (1996) 123–136
15. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the Symposium on Principles of Distributed Computing. (1996) 267–275
16. Massalin, H., Pu, C.: A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Computer Science Department, Columbia University (1991)
17. Michael, M.M., Scott, M.L.: Relative Performance of Preemption-Safe Locking and Non-Blocking Synchronization on Multiprogrammed Shared Memory Multiprocessors. In: Proceedings of the 11th International Symposium on Parallel Processing, IEEE Computer Society (1997) 267–273
18. Herlihy, M.: A methodology for implementing highly concurrent data objects. ACM Transactions on Programming Languages and Systems **15**(5) (November 1993) 745–770

19. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the Twentieth Annual International Symposium on Computer Architecture. (1993)
20. Marathe, V.J., Scott, M.L.: A qualitative survey of modern software transactional memory systems. Technical report, Department of Computer Science, University of Rochester (June 2004)
21. Rossbach, C.J., Hofmann, O.S., Porter, D.E., Ramadan, H.E., Bhandari, A., Witchel, E.: TxLinux: using and managing hardware transactional memory in an operating system. In: SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. (2007) 87–102
22. Guniguntala, D., McKenney, P.E., Triplett, J., Walpole, J.: The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. IBM Systems Journal **47**(2) (April 2008)
23. McKenney, P.E.: RCU vs. locking performance on different CPUs. In: linux.conf.au, Adelaide, Australia (January 2004) (accessed April 28, 2008).
24. Morris, J.: SELinux scalability and analysis patches (November 2004) (accessed April 28, 2008).
25. Liskov, B.: Distributed programming in Argus. Communications of the ACM **31**(3) (1988) 300–312
26. Fraser, K.: Practical Lock-Freedom. PhD thesis, University of Cambridge Computer Laboratory (2004) (accessed April 28, 2008).
27. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Transactions on Parallel and Distributed Systems **15**(6) (June 2004) 491–504
28. McKenney, P.E., Slingwine, J.D.: Read-Copy Update: Using Execution History to Solve Concurrency Problems. In: Parallel and Distributed Computing and Systems. (October 1998) 509–518
29. McKenney, P.E., Sarma, D., Soni, M.: Scaling dcache with RCU. Linux Journal **2004**(117) (2004)
30. Linder, H., Sarma, D., Soni, M.: Scalability of the directory entry cache. In: Ottawa Linux Symposium. (June 2002) 289–300
31. Walpole, R.E., Myers, R.H., Myers, S.L., Ye, K.: Probability & Statistics for Engineers & Scientists. Seventh edn. Prentice Hall (2002)