Scalable Concurrent Hash Tables via Relativistic Programming

Josh Triplett
Portland State University
josh@joshtriplett.org

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

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

ABSTRACT

This paper presents a novel concurrent hash table implementation which supports wait-free, near-linearly scalable lookup, even in the presence of concurrent modifications. In particular, this hash table implementation supports concurrent moves of hash table elements between buckets, for purposes such as renames.

Implementation of this algorithm in the Linux kernel demonstrates its performance and scalability. Benchmarks on a 64-way POWER system showed a 6x scalability improvement versus fine-grained locking, and a 1.5x improvement versus the current state of the art in Linux.

To achieve these scalability improvements, the hash table implementation uses a new concurrent programming technique known as *relativistic programming*. This approach uses a copy-based update strategy to allow readers and writers to run concurrently without conflicts, avoiding many of the non-scalable costs of synchronization, inter-processor communication, and cache coherence. New techniques such as the proposed hash-table move algorithm allow readers to tolerate the resulting weak memory-ordering behavior that arises from allowing one version of a structure to be read concurrently with updates to a different version of the same structure. Relativistic programming techniques provide performance and scalability advantages over traditional synchronization, as demonstrated through several benchmarks.

1. INTRODUCTION

Hash tables enjoy widespread use in many applications and operating systems, due to their O(1) average time for many operations [3, 11]. These users of hash tables have increasingly become concurrent, to adapt to concurrent hardware. Thus, hash table implementations must work concurrently, and should ideally scale linearly with additional CPUs. In particular, many hash table applications involve far more lookups than modifications, so we want to support fast, scalable lookups [10].

Existing hash table implementations, whether based on fine-grained locking or lock-free algorithms, still require expensive synchronization operations for lookups. Fine-grained locking implementations require some form of lock surrounding a lookup operation, while lock-free algorithms require either atomic operations or memory ordering primitives such as barriers.

In some applications, a hash table must support not only insertion and removal, but also moving entries due to changing hash keys. For instance, if a filesystem cache uses filenames as keys, renaming will require a move operation. A change to the key may require moving an entry between buckets.

We present a novel hash table move operation, which supports concurrent, linearly scalable, wait-free lookups. The key step in this algorithm entails cross-linking hash chains to make a single entry appear in two hash buckets simultaneously. This state allows a single operation to both change the hash key and move the entry to the appropriate bucket for that key. The remainder of the algorithm consists of preparation for cross-linking and cleanup after cross-linking.

Section 2 documents the semantics the move operation must satisfy, and the semantics of hash-table lookups that make the move operation possible. Section 3 provides the full algorithm for the new move operation, including step-by-step diagrams of the hash table structure. Section 4 outlines the methodology for performance analysis, and section 5 presents the results of this analysis. Section 6 presents the "relativistic programming" methodology which led to this solution, and some further implications of that methodology. Section 7 summarizes our conclusions, and section 8 suggests further work based on this result.

2. SEMANTICS

Subsection 2.1 lists required properties of the move operation. Subsection 2.2 documents some standard properties of hash-table lookups that support the new move operation.

2.1 Properties of Hash-Table Moves

The move operation changes the key associated with an entry, and moves the entry 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 entry under the new key, a subsequent lookup ordered after the first cannot find the entry under the old key.
- If a lookup does not find the entry under the old key, a subsequent lookup ordered after the first must find the entry 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" means either a lookup running in the same thread as the first but later in program order, or a lookup equivalently ordered after the first via some appropriate synchronization.

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.

2.2 Properties of Hash-Table Lookups

The new hash table move operation relies on two key properties 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. Note that this requirement means that the hash table cannot safely hold multiple distinct entries with the same key, such as in the implementation of a multimap.

The first two possible semantics violations from section 2.1 (entries 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, simultaneous writes to multiple locations, or hardware transactional memory, so the writer cannot simultaneously and atomically change more than one pointer or key. Those architectures that do, or software systems such as software transactional memory that simulate such capabilities, incur a high cost for such an operation [2, 21, 23, 5].

3. ALGORITHM

Subsection 3.1 describes the key step in the new move algorithm. Subsection 3.2 outlines the hash-table lookup operation. Subsection 3.3 walks through the new move algorithm step-by-step. Subsection 3.4 discusses the correctness of this algorithm in terms of the required semantics from section 2.1.

3.1 Atomic Rename via Cross-Linking

The new hash-table move operation arises primarily from a single key insight: 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 approach satisfies the key semantics 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. When removing the cross-link, the writer must avoid disturbing any reader 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.

To safely resolve the cross-link, the algorithm makes use of a deferred destruction technique such as Read-Copy Update (RCU); specifically, the algorithm assumes an operation which waits for all current readers to finish. Deferred destruction removes one source of conflicts between readers and writers by separating memory reclamation from writers, and deferring that reclamation until readers have finished, but not deferring the writers. Writers can thus focus on maintaining higher-level semantics such as those in section 2.1, rather than on preventing readers from crashing. Implementations of deferred destruction exist that avoid all expensive synchronization instructions for readers, including multiple implementations of RCU in both the Linux kernel and standard POSIX userspace [4].

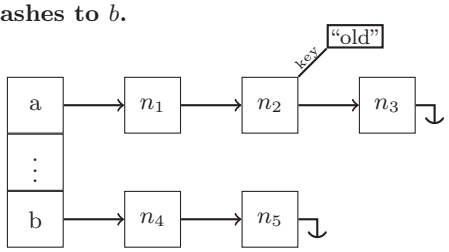
3.2 Hash-Table Lookup

The lookup operation consists of a standard uniprocessor hash-table lookup, except that it makes use of the appropriate primitives to support deferred destruction, and runs on commodity multiprocessor architectures:

1. Start deferring write-side destruction.
2. Hash the given key to determine the corresponding hash bucket.
3. Traverse the linked list in that hash bucket, comparing the given key to the key in each node.[1]
4. If a node has the given key, do the computation that required the node.

[1]On architectures such as DEC Alpha which do not automatically guarantee memory ordering for dependent reads [6], this traversal requires appropriate barriers for such reads, such as `smp_read_barrier_depends` on Linux, used in `rcu_dereference`. However, almost all current multiprocessor architectures provide dependent read ordering by default, making such barriers unnecessary.
Aggressive compiler optimizations, particularly those regarding local caches of global data, can also lead to problems in this step [1]. This may necessitate compile-time barriers to locally prevent such optimizations.

Figure 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.



5. If the traversal reaches the end of the list without finding a node with the given key, assume the node does not exist in the table, and proceed accordingly.

6. Stop deferring write-side destruction.

Note that all of these steps allow implementations to avoid expensive synchronization operations such as locks, atomic operations, or memory barriers. The deferred destruction steps can use a synchronization-free deferred destruction mechanism, as discussed earlier in section 3.1. The list traversal in step 3 relies on the property that reads and writes to word-sized word-aligned locations such as pointers will occur atomically, retrieving either the old or the new value but not a mix of the two; we do not know of any architectures for which this property does not hold.

Furthermore, this algorithm involves no helping, rollback, or retry code, making it deterministic.

3.3 Hash-Table Move

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

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

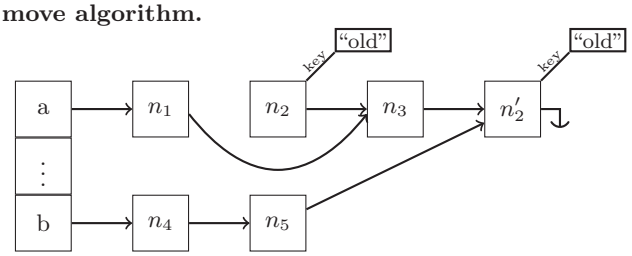
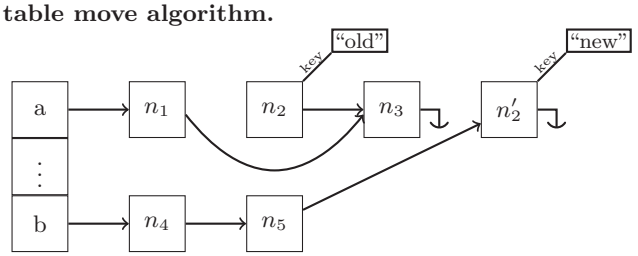


Figure 3: State of the hash table after un-cross-linking hash chains in step 12 of the relativistic hash table move algorithm.



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.

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

While the last step defers some memory reclamation until after readers have finished, the remainder of the algorithm should have little to no performance degradation from concurrent readers. Furthermore, because writers need not wait for concurrent readers, writers publish new data immediately, and new readers may immediately observe this new data.

3.4 Discussion

These operations meet the required semantics described in section 2.1. 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 complete their lookup, 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.

4. PERFORMANCE ANALYSIS METHODOLOGY

The lookup algorithm requires no synchronization instructions, and runs wait-free, even when running concurrently with a move operation. Thus, it should allow significantly more lookups per unit time than a 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 a lock-based move operation. To test these hypotheses, we needed a new benchmark framework for concurrent hash tables.

The new move operation requires a deferred destruction mechanism. Read-Copy Update (RCU) provides a mature and popular implementation of deferred destruction. 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, and a fast concurrent memory allocator. Thus, a Linux kernel module provided the most practical and straightforward target for a benchmark.

The benchmark module implemented for this paper, `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. `rcuhashbash` includes the following hash table implementations:

- The move and lookup algorithms presented in this paper, as described in sections 3.2 and 3.3. This implementation uses per-bucket spinlocks to synchronize with other writers. The lookup operation contained no synchronization operations of any kind.
- Multiple variants of mutual exclusion: whole-table spinlocks, whole-table reader–writer locks, per-bucket spinlocks, and per-bucket reader–writer locks.
- The RCU-based algorithm currently used for the Linux directory entry cache (dcache) [17, 13]. This algorithm uses an RCU-based linked list to allow concurrent insertions and deletions without disrupting readers. However, the lookup operation uses an optimistic *sequence lock* [12] to detect concurrent moves, and retries a lookup if it raced with a move; this sequence lock entails some expensive synchronization operations.

`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 randomly chooses an old key and a new key from the range of 0 to twice the maximum initial value ($[0, 8191]$ for this experiment), and attempts to move the item with the old key to the item with the new key. The lookup operation randomly chooses a key from the same range and performs a lookup. The lookup and the move operation each increment a thread-local count of the number of operations completed.

The machine used for testing had 16 IBM POWER6 physical processors at 4.7GHz, each with two cores of two logical threads each, for a total of 64 hardware-supported threads (henceforth referred to simply as "CPUs"). This machine ran the Linux 2.6.28 kernel, compiled for the 64-bit powerpc architecture, using the "classic" RCU implementation [16] and no preemption. To observe scalability, the benchmark ran each hash table implementation on 1, 2, 4, 8, 16, 32, and 64 CPUs. 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.

5. PERFORMANCE ANALYSIS

Subsection 5.1 gives the performance results for hash-table lookups, and subsection 5.2 gives the results for hash-table moves. Subsection 5.3 summarizes the results.

5.1 Hash Lookup Performance

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.

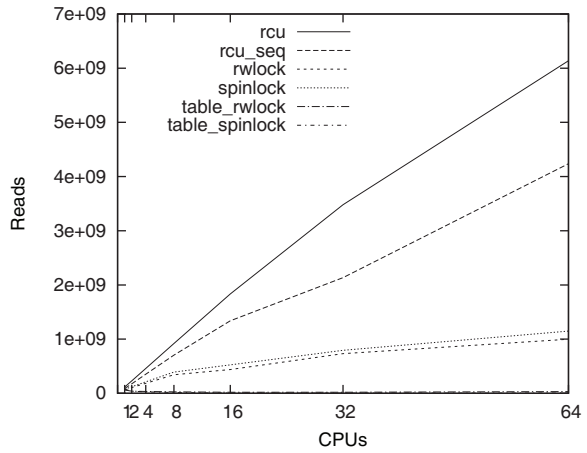


Figure 4: Average lookups in 30 seconds by number of CPUs with 999999:1 read:write ratio

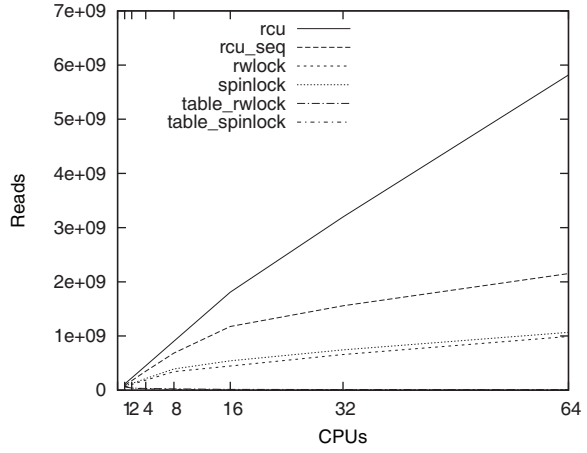


Figure 5: Average lookups in 30 seconds by number of CPUs with 999:1 read:write ratio

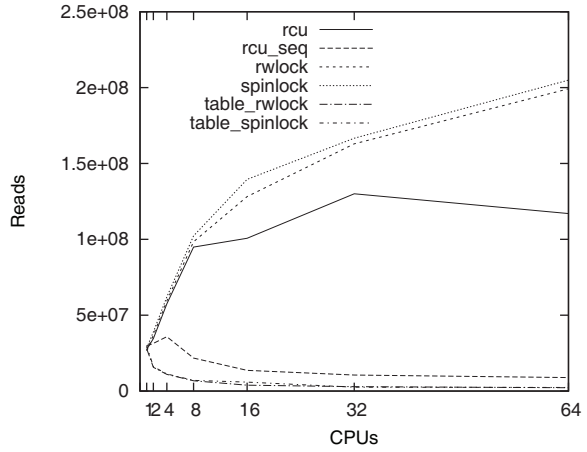


Figure 6: Average lookups in 30 seconds by number of CPUs with 1:1 read:write ratio

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 on the full 64 CPUs. Reader–writer locks have a higher critical section overhead than ordinary spinlocks, and for small critical sections this overhead nullifies the benefits of concurrent readers.

5.2 Hash Move Performance

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: 999999:1, 999:1, and 1:1.

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, likely due to retries or contention for the sequence lock. 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.

5.3 Performance Summary

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.

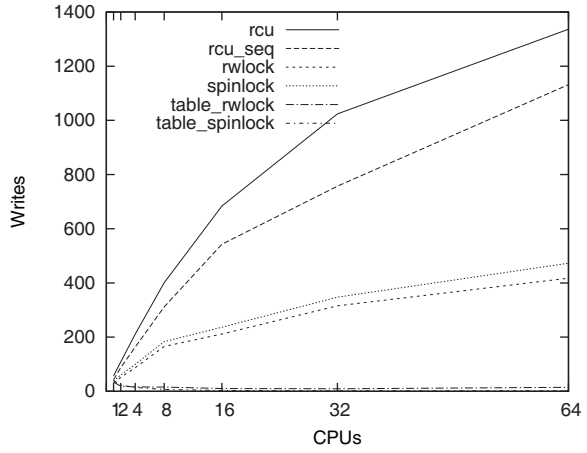


Figure 7: Average moves in 30 seconds by number of CPUs with 999999:1 read:write ratio

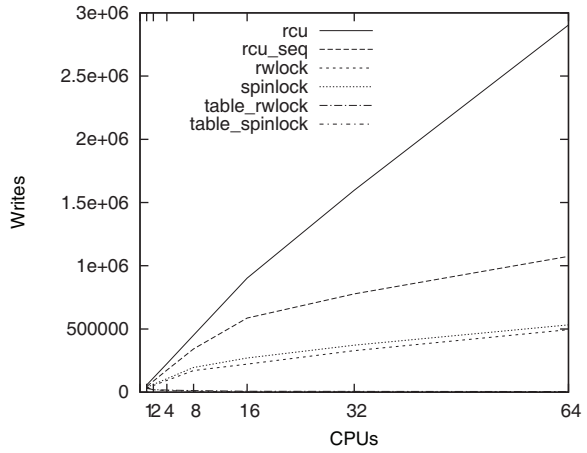


Figure 8: Average moves in 30 seconds by number of CPUs with 999:1 read:write ratio

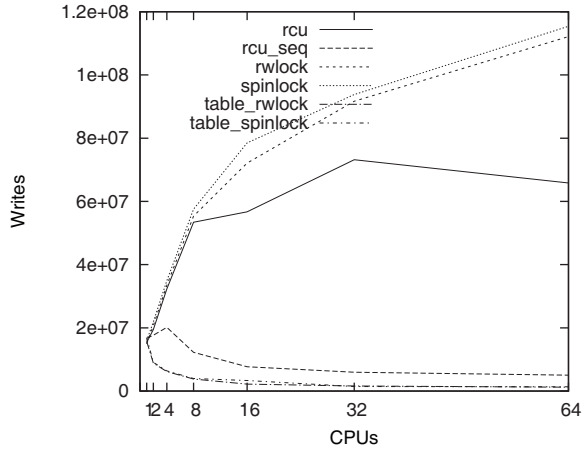


Figure 9: Average moves in 30 seconds by number of CPUs with 1:1 read:write ratio

6. RELATIVISTIC PROGRAMMING

Our hash table move operation allows fast scalable lookups by making it possible for those lookups to avoid all expensive synchronization operations. We removed conflicts between readers and writers by making updates appear semantically consistent after every visible step, and by deferring destruction to maintain referential integrity.

Furthermore, the lookup algorithm involves no helping, rollback, or retry code. In addition to improving performance, this gives lookups a deterministic response time, making them more suitable for use in code requiring real-time response. This deterministic behavior assumes an appropriate real-time implementation of deferred destruction, such as a real-time RCU implementation [8]. Similarly, the move algorithm does not wait on concurrent readers; memory reclamation must wait until readers have completed, but this can occur asynchronously as long as the system has sufficient memory.

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. In general, we can use a copy-based update strategy to allow readers and writers to run concurrently without conflicts, avoiding many of the non-scalable costs of inter-processor communication, cache coherence, and synchronization.

However, by allowing one version of a structure to be read concurrently with updates to a different version of the same structure, we may permit weaker memory-ordering behavior than normally expected by readers. 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 approach as *relativistic programming*, by analogy with physics, because it allows threads to see a relative view of memory, rather than an absolute reference frame. Furthermore, actions taken by one thread may appear to other threads in different orders.

Algorithms designed to work with relativistic programming must either tolerate this weakened memory ordering, or take steps to ensure that the data structure appears to change directly from one semantically consistent state to another without inconsistent intermediate states. The hash-table move operation described in this paper implements the latter approach to preserve the semantics described in section 2.1.

Relativistic programming techniques provide the potential for greater scalability 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 [8, 15, 20].

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. This approach relies on the commutativity of the accumulating operations such as addition, just as relativistic linked lists rely on the commutativity of most linked list manipulations: if the order of operations does not matter, the algorithms need not enforce any order.

More generally applicable relativistic programming 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. Writers can thus focus on maintaining higher-level semantics such as those in section 2.1, rather than on preventing readers from crashing.

Several techniques exist for deferred destruction [9], including epoch-based reclamation [7], hazard pointers [19], and quiescent-state-based reclamation [14, 18]. 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 these techniques, implementations free of synchronization instructions exist for epoch-based reclamation and quiescent-state-based reclamation.

Many common data structures have relativistic implementations which use deferred destruction. These include linked lists, radix trees, and tries. Previous work introduced semi-relativistic hash tables based on sequence locking and retries [17, 13], as described in section 4; while semantically correct, this approach still requires the use of synchronization instructions on the read side, and it can potentially delay a reader indefinitely. The algorithm presented in this paper allows for a fully relativistic hash table lookup.

7. CONCLUSIONS

We presented novel algorithms for a concurrent hash table, supporting a semantically atomic move operation based on cross-linking hash chains. Benchmarks of the proposed concurrent hash table implementation demonstrated a 6x scalability improvement for lookups versus fine-grained locking, and a 1.5x improvement versus the current state of the art in Linux. Read-biased workloads provided the highest scalability and performance, but the algorithm remained competitive even for balanced workloads. We plan to enhance this concurrent hash table implementation further to add support for resizing, and then incorporate it into the Linux kernel as a replacement for many existing uses of less-flexible hash-table implementations.

We further proposed some general principles of *relativistic programming* which motivated this algorithm. These principles provide a framework for future development of scalable concurrent data structures and programming techniques.

8. FUTURE WORK

To maintain its O(1) performance, a hash table must remain appropriately sized for the amount of data it contains; without resizing, the performance of a hash table will degrade to linear in the number of entries. Maintaining an appropriate size generally requires resizing the table at runtime, to fit additional entries or reclaim space after removing many entries. The bulk of resizing involves moving entries between buckets. The algorithm presented in this paper provides a hash table move operation which permits scalable, wait-free lookups, by using the key change operation to simulate an atomic move between buckets. However, the move operation needed in a hash table resize does not involve a key change. Thus, resizing a hash table requires a different move algorithm, ideally optimized for moving many elements at once. We have created such an algorithm, with research in progress on its performance.

The benchmarking approach used a Linux kernel module, to take advantage of its implementation of the best-of-class RCU deferred destruction algorithm, as well as its other highly optimized mutual-exclusion primitives. However, since the original implementation of `rcuhashbash`, Mathieu Desnoyers has produced a userspace implementation of RCU, `liburcu` [4]. We plan to port `rcuhashbash` to run as a Linux userspace process using `liburcu`.

This userspace port will make it possible to include and compare hash table implementations based on concurrent programming techniques not available in the Linux kernel, such as software transactional memory (STM). Given the synchronization-free lookup our move operation supports, we expect to observe significantly better performance and scalability results from our algorithm than from an STM-based implementation. Furthermore, our algorithm already outperforms the current Linux approach based on optimistic sequence locks, and we do not expect a far more optimistic STM-based approach to offer improvements. We eagerly anticipate the opportunity to test these hypotheses.

We hypothesize that impacts to scalability arise primarily from communication between processors for coordination purposes, as described in section 6. Cache coherence protocols, such as the classic MESI protocol and its many variations, require coordination between processors, and thus imply additional communication. A cache-incoherent system, such as the Intel Single-chip Cloud Computer, would turn this implicit communication into explicit communication, allowing algorithms to reduce or eliminate it when not necessary for correctness. We believe that our relativistic programming approach would provide a strong basis for a programming model on such platforms, and thus provide a path towards even more scalable systems.

The algorithm documented in this paper required novel algorithmic steps, and required special-case reasoning to argue for its correctness. The development of relativistic algorithms for other use cases currently requires the same level of novel development and special-case reasoning. To ease the adoption of Relativistic Programming techniques, we have work in progress to provide a full reasoning model for the correctness of relativistic algorithms, and a generalized construction technique for broad classes of data structures.

Disclaimer

The authors have patents and pending patents on some of the algorithms presented in this paper, including U.S. patent 7,668,851 [22]. However, the authors have also released implementations of these algorithms as Free and Open Source Software under the GNU General Public License, which anyone may use or adapt.

Acknowledgments

Thanks to Larry Kessler, 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 review and feedback, and for discussions leading to the realization that typical reader–writer locks force writers to delay publication of updated data. Thanks to Jamey Sharp for review, feedback, and advice on drafts of this paper.

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

9. REFERENCES

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

[2] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter Chapter 11: Hash Tables. MIT Press, second edition, 2001.

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

[5] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commericial hardware transactional memory implementation. In *Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, Washington, DC, USA, March 2009.

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

[7] K. Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge Computer Laboratory, 2004. (accessed April 28, 2008).

[8] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal*, 47(2), April 2008.

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

[10] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*, chapter Chapter 11: Concurrent Hashing. Morgan Kaufmann Publishers, 2008.

[11] D. Knuth. *The Art of Computer Programming*, chapter Section 6.4: Hashing. Addison-Wesley, second edition, 1998.

[12] C. Lameter. Effective synchronization on Linux/NUMA systems. In *Gelato Conference*, May 2005.

[13] H. Linder, D. Sarma, and M. Soni. Scalability of the directory entry cache. In *Ottawa Linux Symposium*, pages 289–300, June 2002.

[14] P. E. McKenney. *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.

[15] P. E. McKenney. RCU vs. locking performance on different CPUs. In *linux.conf.au*, Adelaide, Australia, January 2004. (accessed April 28, 2008).

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

[17] P. E. McKenney, D. Sarma, and M. Soni. Scaling dcache with RCU. *Linux Journal*, 2004(117), 2004.

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

[19] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[20] J. Morris. SELinux scalability and analysis patches, November 2004. (accessed April 28, 2008).

[21] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and managing hardware transactional memory in an operating system. In *SOSP'07: Twenty-First ACM Symposium on Operating Systems Principles*. ACM SIGOPS, October 2007.

[22] J. Triplett. Lockless hash table lookups while performing key update on hash table element. US Patent 7668851, February 2010.

[23] H. Volos, N. Goyal, and M. M. Swift. Pathological interaction of locks with transactional memory. In *3rd ACM SIGPLAN Workshop on Transactional Computing*, New York, NY, USA, February 2008. ACM.